SPEEDING UP PATH PLANNING ON STATE LATTICES AND GRID GRAPHS BY EXPLOITING FREESPACE STRUCTURE

by

Tansel Uras

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Science)

December 2019

Acknowledgements

First and foremost, I would like to thank my advisor, Sven Koenig, for the many hours he has spent showing me how to write good papers and his support throughout my PhD.

I would like to thank Nathan Sturtevant and Carlos Hernandez for helping me take my first steps towards the research presented in this dissertation, as well as their continued support throughout my PhD.

I would like to thank David Kempe for spending a weekend teaching me how to be more precise in mathematical writing and reminding me that infinity is not a number.

I would like to thank my other dissertation and proposal committee members, John Gunnar Carlsson, Micheal Zyda, and Wei-Min Shen, for their helpful comments.

I would like to thank Daniel Harabor for the long and stimulating discussions, the friendly competition, and, most importantly, his friendship. We finally wrote a paper together. I would also like to thank Peter Stuckey for making the writing of this paper so enjoyable.

I would like to thank Maxim Likhachev for helping me adapt my research to state lattices and Ben Strasser for helping me better understand contraction hierarchies.

I would like to thank my brother-in-arms and flatmate, Liron Cohen, for his friendship, for our long debates and discussions, for keeping me grounded, and for putting up with me during the writing of this dissertation. A special shout out to his dog, Paz, for reminding me that running around in the house is still fun.

I would like to thank my colleagues and collaborators, Satish Kumar, Ariel Felner, Marcello Cirillo, Giuseppe Caggianese, Hang Ma, Hong Xu, Jiaoyang Li, Masabumi Furuhata, Alex Nash, William Yeoh, and Xiaoxun Sun for their friendship and the fun discussions.

Last but not least, I would like to thank my family and friends, who have made my much-needed breaks from work so enjoyable.

The research presented in this dissertation was supported by ARL/ARO under contract/grant number W911NF08-1-046; ONR in form of a MURI under contract/grant number N00014-09-1-1031; and NSF under grant numbers 1319966, 1409987, 1724392, 1817189, and 1837779.

Abstract

Path planning is the problem of finding a sequence of waypoints that an agent can follow through a continuous environment to reach its goal location while avoiding obstacles. A typical approach to solving the path-planning problem in many real-world applications is to discretize the configuration space of the agent into a graph, that can then be searched for a path from a given start vertex to a given goal vertex. The choice of discretization depends mostly on the agent, the environment, and the application. For instance, one can use road networks for navigation systems, grid graphs for video games, or state lattices for agents with kinematic constraints. In some applications of path planning, the environment is static and known in advance, allowing one to analyze its associated graph in a preprocessing phase to generate auxiliary information, which can then be used to answer path-planning queries faster. A number of preprocessing-based path-planning algorithms have been developed that speed up path-planning queries by several orders of magnitude on road networks. These algorithms are applicable to any graph and provably efficient on graphs that have small *highway dimensions*, such as road networks. Grid graphs, on the other hand, have larger highway dimensions than road networks. When some of these algorithms are applied to grid graphs, the speed-up is smaller, and the memory requirements, relative to the original graph, are higher.

State lattices and grid graphs (which are instances of state lattices) have other properties that can be exploited, which can be collectively referred to as their *freespace-structure*. *Translation invariance of freespace distances and freespace-canonical paths:* On state lattices constructed on obstacle-free and infinitely extending environments, called *freespace state lattices*, translating shortest paths by changing the x- or y-coordinates of all their vertices by the same integral amount produces shortest paths, and translating *canonical* paths produces canonical paths. *Octile property*: On *freespace grid graphs*, shortest paths consist of moves in at most two directions, a diagonal one and an associated cardinal one. The hypothesis of this dissertation is as follows: *One can develop preprocessing-based path-planning algorithms for state lattices and grid graphs that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.*

To validate this hypothesis: (1) We introduce the *subgoal graph* framework, a framework that can be specialized to exploit structure in different types of graphs through the use of a *reachability relation*. We introduce a hierarchical variant of subgoal graphs, called *N-level subgoal graphs*, and a suboptimal but complete variant of subgoal graphs, called *strongly connected subgoal graphs*. (2) We apply the subgoal graph framework to state lattices by using variants of *freespace-reachability* as the reachability relation. We experimentally demonstrate that answering queries using strongly connected subgoal graphs

constructed with respect to variants of freespace-reachability achieves *dominating query-time/path-suboptimality trade-offs* compared to answering queries using strongly connected subgoal graphs constructed with respect to *bounded-distance-reachability*; and *non-dominated query-time/path-suboptimality trade-offs* compared to weighted A* searches. (3) We apply the subgoal graph framework to grid graphs by using *safe-freespace-reachability* as the reachability relation. We show that jump-point search, an online path-planning algorithm that also exploits structure in grid graphs, can be understood as a search over a subgoal graph constructed on the *direction-extended canonical grid graph*. We show that contraction hierarchies, a preprocessing-based path-planning algorithm that has been developed for road networks with a non-dominated query-time/memory trade-off in the Grid-Based Path-Planning Competition, can be augmented to exploit structure in grid graphs by augmenting it with reachability relations in various ways. We experimentally demonstrate that answering queries using contraction hierarchies constructed on subgoal graphs achieves a *dominating query-time/memory trade-off* compared to several state-of-the-art path-planning algorithms on grid graphs, and can answer queries *2.42 times faster* than single-row compression, the fastest entry in the Grid-Based Path-Planning Competition, while requiring *139.06 times less memory*.

Contents

Acknowledgements ii

Abstract iii

List of Figures ix

List of Tables xii

List of Algorithms xiii

1 Introduction 1
- 1.1 Problem Definition . . . 3
  - 1.1.1 Graphs and Paths . . . 3
  - 1.1.2 The Path-Planning Problem . . . 3
  - 1.1.3 State Lattices and Grid Graphs . . . 4
  - 1.1.4 Assumptions . . . 4
  - 1.1.5 Preprocessing-Based Path Planning . . . 5
  - 1.1.6 Applications and Related Work . . . 6
- 1.2 Hypothesis . . . 7
- 1.3 Contributions . . . 8
  - 1.3.1 The Subgoal Graph Framework . . . 8
  - 1.3.2 Exploiting the Freespace Structure of State Lattices . . . 10
  - 1.3.3 Exploiting the Freespace Structure of Grid Graphs . . . 11
- 1.4 Dissertation Structure . . . 12

2 Path Planning 16
- 2.1 Discretizing Configuration Spaces into Graphs . . . 16
  - 2.1.1 Environments, Agents, and Configuration Spaces . . . 16
  - 2.1.2 State Lattices . . . 18
  - 2.1.3 Grid Graphs . . . 19
  - 2.1.4 Road Networks . . . 20
- 2.2 Search Algorithms . . . 21
  - 2.2.1 A Common Framework for Search Algorithms . . . 21
  - 2.2.2 Breadth-First and Depth-First Searches . . . 24
  - 2.2.3 Dijkstra Search . . . 24

2.2.4 Bidirectional Dijkstra Search . . . 25
2.2.5 A* Search . . . 26
2.2.6 Weighted A* Search . . . 27
2.3 Preprocessing-Based Path Planning . . . 27
2.3.1 Limitations and Applications . . . 27
2.3.2 Related Work: Road Networks . . . 29
2.3.3 Related Work: Grid Graphs . . . 35
2.4 Conclusions . . . 41

3 The Subgoal Graph Framework 43
3.1 Introduction . . . 43
3.2 Overlay Graphs . . . 44
3.2.1 Overlay Graphs . . . 45
3.2.2 Extended Overlay Graphs . . . 48
3.2.3 Optimality and Minimality of Extended Overlay Graphs . . . 50
3.2.4 Answering Queries Using Overlay Graphs . . . 54
3.2.5 Identifying Direct-Reachable Edges from a Source Vertex . . . 56
3.2.6 Contractions and Heavy Contractions . . . 60
3.3 Subgoal Graphs . . . 65
3.3.1 Reachability Relations . . . 66
3.3.2 Subgoal Graphs . . . 67
3.3.3 Shortest-Path Covers and Highway Dimension . . . 69
3.3.4 Answering Queries Using Subgoal Graphs . . . 72
3.3.5 Heavy R Contractions . . . 74
3.3.6 Incrementally Constructing an R-SPC . . . 78
3.4 N-level Overlay and Subgoal Graphs . . . 80
3.4.1 N-Level Overlay Graphs . . . 80
3.4.2 Subclasses of N-Level Overlay Graphs . . . 85
3.4.3 Constructing N-Level Overlay Graphs . . . 85
3.4.3.1 N-Level Overlay Graphs and Heavy Contractions . . . 87
3.4.3.2 R N-level Subgoal Graphs and Heavy R Contractions . . . 88
3.4.3.3 Contraction Hierarchies and Contractions . . . 88
3.4.3.4 R Contraction Hierarchies and R Contractions . . . 90
3.4.4 Answering Queries Using N-Level Overlay Graphs . . . 90
3.4.4.1 Connection . . . 90
3.4.4.2 Search . . . 91
3.4.4.3 Refinement . . . 93
3.5 Strongly-Connected Subgoal Graphs . . . 94
3.5.1 Strongly-Connected Subgoal Graphs . . . 95
3.5.2 Answering Queries Using Strongly-Connected Subgoal Graphs . . . 95
3.5.3 Constructing Strongly-Connected Subgoal Graphs . . . 96
3.5.3.1 Identifying Access Subgoals . . . 96
3.5.3.2 Strongly Connecting Access Subgoals Using R-Reachable Edges . . . 97
3.6 Conclusions . . . 101

4 Exploiting the Freespace Structure of State Lattices **103**
4.1 Introduction . . . 104
4.2 Preliminaries and Notation . . . 105
4.3 Assumptions . . . 106
4.4 Freespace Reachability . . . 107
4.4.1 Freespace State Lattice . . . 108
4.4.2 Translation Invariance of Freespace Distances . . . 111
4.4.3 Freespace Reachability (F-Reachability) . . . 112
4.4.4 F-Connect . . . 113
4.4.5 F-Refine . . . 118
4.5 Canonical Freespace Reachability . . . 120
4.5.1 Symmetries and Canonical Orderings . . . 120
4.5.2 Properties of Freespace-Canonical Paths . . . 123
4.5.3 Canonical Freespace Reachability (CF-Reachability) . . . 125
4.5.4 CF-Connect . . . 126
4.5.5 CF-Refine . . . 131
4.6 Experimental Evaluation . . . 131
4.6.1 Benchmarks . . . 131
4.6.2 Algorithms . . . 133
4.6.3 Implementation Details . . . 135
4.6.4 R-Connect . . . 138
4.6.4.1 Setup . . . 138
4.6.4.2 Number of R-Reachable Vertices from a Given Vertex . . . 141
4.6.4.3 Average Expansion Time . . . 145
4.6.4.4 Avoiding Expansions of Non-Direct-Reachable Vertices . . . 148
4.6.4.5 Direct-R-Reachable Expansion Rate . . . 151
4.6.5 Subgoal Graphs . . . 153
4.6.5.1 Setup . . . 153
4.6.5.2 Results . . . 155
4.6.6 Strongly-Connected Subgoal Graphs . . . 158
4.6.6.1 Setup . . . 158
4.6.6.2 Results . . . 160
4.7 Conclusions and Future Work . . . 171

5 Exploiting the Freespace Structure of Grid Graphs **173**
5.1 Introduction . . . 174
5.2 Preliminaries and Notation . . . 175
5.3 Subgoal Graphs on Grid Graphs . . . 177
5.3.1 Freespace Structure of Grid Graphs . . . 177
5.3.2 F-Reachability and CF-Reachability on Grid Graphs . . . 178
5.3.3 Safe-Freespace-Reachability (SF-Reachability) . . . 180
5.3.4 Using Convex Corner Cells as Subgoals . . . 180
5.3.5 SF-Connect . . . 186
5.3.6 Linear Time Preprocessing . . . 190
5.4 Jump-Point Graphs . . . 191

5.4.1 Jump-Point Search . . . 191
5.4.2 Direction-Extended Canonical Grid Graph . . . 194
5.4.3 Jump-Point Graph . . . 196
5.5 Experimental Evaluation . . . 199
5.5.1 Algorithms and Implementation Details . . . 199
5.5.2 Benchmarks . . . 203
5.5.3 Result Presentation . . . 206
5.5.4 Subgoal Graphs . . . 208
5.5.5 Jump-Point Graphs . . . 213
5.5.6 Contraction Hierarchies . . . 218
5.5.7 Contraction Hierarchies on Subgoal Graphs . . . 222
5.5.8 Contraction Hierarchies on Jump-Point Graphs . . . 228
5.5.9 R-Refining R-Reachable Shortcut Edges . . . 233
5.5.10 R Contraction Hierarchies . . . 236
5.5.11 N-Level Subgoal Graphs . . . 241
5.5.12 Grid-Based Path-Planning Competition . . . 246
5.5.12.1 Competition Benchmarks . . . 246
5.5.12.2 Competition Results . . . 249
5.6 Conclusions . . . 256

6 Conclusions **257**

Bibliography **261**

List of Figures

2.1 Environment and configuration space of a translating agent 17
2.2 A video game environment represented as a grid 18
2.3 Primitives used for the Boss entry in the DARPA Urban Challenge 19
2.4 Grid graphs 20
2.5 International E-road network 21
2.6 Search trees and paths found by various search algorithms 25
2.7 An overlay graph constructed from a vertex separator 31
2.8 Contraction hierarchies 32
2.9 Overlay graph used by rectangular symmetry reduction. 39
2.10 Operation of jump-point search 40
3.1 Overlay graphs 46
3.2 Extended overlay graphs 49
3.3 Answering queries using the Connect-Search-Refine algorithm and overlay graphs 55
3.4 Variants of the Overlay-Connect algorithm 58
3.5 Contractions and heavy contractions 62
3.6 Bounding Overlay-Connect by using a reachability relation 67
3.7 A BD4 subgoal graph and an overlay graph that is not a BD4 subgoal graph 68
3.8 Constructing R-SPCs by heavy R contractions 76
3.9 R-SPCs constructed with different orderings of heavy R contractions . . . 77
3.10 Incrementally constructing an R-SPC 79
3.11 N-level overlay graphs 81
3.12 Subclasses of N-level overlay graphs 86
3.13 Searching N-level overlay graphs 93
3.14 Unpacking shortcut edges 94
3.15 Identifying access subgoals 98
3.16 Strongly connecting access subgoals with R-reachable edges 100
4.1 Unicycle and Urban primitives and their corresponding freespace-shortest paths 108
4.2 Shortest paths on state lattices 109
4.3 F4-Connect on a 4-neighbor grid graph 114
4.4 F50-Connect using Unicycle primitives 114
4.5 F-Refine 118
4.6 Canonical orderings 121
4.7 Symmetric freespace-shortest paths on state lattices 122
4.8 CF4-Connect on a 4-neighbor grid graph 128

4.9 Experimental setup: Primitives and grids . . . 132
4.10 Storage of freespace information . . . 137
4.11 Search trees of BD50-, F50-, and CF50-Connect . . . 139
4.12 Number of R-reachable vertices . . . 142
4.13 Number of R-reachable vertices (cube root) . . . 142
4.14 Percentage of BDb-reachable vertices that are also Fb- and CFb-reachable from a source vertex . . . 143
4.15 Average R-connect expansion times . . . 146
4.16 Average numbers of successors processed per expansion by R-connect . . . 146
4.17 R-connect statistics, Unicycle-Arena2 . . . 149
4.18 R-connect statistics, Urban-Aurora . . . 149
4.19 Direct-R-reachable expansion rate of R-connect . . . 152
4.20 Comparison of subgoal graphs (Unicycle-Arena2) . . . 157
4.21 Comparison of strongly-connected subgoal graphs (Unicycle-Arena2) . . . 162
4.22 Number of R-reachable vertices . . . 166
4.23 Percentage of subgoals in strongly-connected subgoal graphs . . . 166
4.24 Connection times when answering queries using strongly-connected subgoal graphs . . . 167
4.25 Search times when answering queries using strongly-connected subgoal graphs . . . 167
4.26 Percentage of query times spent in the search phase when answering queries using strongly-connected subgoal graphs . . . 168
4.27 Speed-up over A* searches when answering queries using strongly-connected subgoal graphs . . . 168
4.28 Path suboptimality when answering queries using strongly-connected subgoal graphs . . . 170
4.29 Query-time/path-suboptimality Pareto frontiers for answering queries using strongly-connected subgoal graphs . . . 170
5.1 A grid graph without corner-cutting diagonal moves . . . 176
5.2 Freespace-shortest paths on grid graphs . . . 178
5.3 Canonical orderings on grid graphs . . . 179
5.4 A convex corner cell . . . 181
5.5 A query subgoal graph on grid graphs . . . 181
5.6 All possible 2-subpaths on grid graphs . . . 182
5.7 Replacing diagonal-to-cardinal turns with cardinal-to-diagonal turns . . . 183
5.8 Freespace-taut paths that are not freespace-shortest . . . 184
5.9 Placement of subgoals on other types of grid graphs . . . 186
5.10 Operation of SF$^{\rightarrow}$-Connect . . . 188
5.11 Operation of jump-point search . . . 192
5.12 Operation of jump-point search without diagonal jump points . . . 194
5.13 Scanning procedure of jump-point search using clearance values . . . 195
5.14 Straight jump points and subgoals . . . 198
5.15 Searches over jump-point and subgoal graphs . . . 198
5.16 Relationships between variants and combinations of subgoal graphs, jump-point graphs, and contraction hierarchies . . . 203

5.17 MovingAI grid maps. 206
5.18 Staircase and random patterns of subgoal graphs on grid graphs. 208
5.19 Jump-point and subgoal graphs on grids with staircase and random patterns. 215
5.20 Searches over subgoal graphs and jump-point graphs on street maps . . . 217
5.21 Search trees on street maps 226
5.22 Shortest paths on jump-point graphs that are not shortest paths on G . . 228
5.23 Symmetric paths on jump-point graphs with different topologies 231
5.24 R contraction hierarchies on maze maps 240
5.25 R N-level subgoal graphs and staircase patterns 244
5.26 Query-time/memory trade-off on GPPC benchmarks 251

List of Tables

1.1 Terminology, arbitrary graphs 13
1.2 Types of overlay and subgoal graphs 14
1.3 Terminology, state lattices and grid graphs 15
3.1 Covering notation. 47
3.2 Variants of contractions 87
3.3 Classes of hierarchies 87
3.4 Answering queries using different subclasses of N-level overlay graphs . . 91
4.1 Reachability relations on state lattices 134
4.2 R-connect statistics 154
4.3 Comparison of subgoal graphs constructed by pruning or growing an R-SPC, Unicycle-Arena2. 156
4.4 Comparison of strongly-connected subgoal graphs (Unicycle-Arena2) . . . 161
5.1 Algorithms on grid graphs evaluated in this dissertation 200
5.2 MovingAI benchmarks 205
5.3 Subgoal graphs on grid graphs 210
5.4 Jump-point graphs 214
5.5 Contraction hierarchies on grid graphs 219
5.6 Contraction hierarchies on subgoal graphs 224
5.7 Contraction hierarchies on jump-point graphs 230
5.8 Trade-offs of R-refining R-reachable edges 234
5.9 Trade-offs of R contraction hierarchies 237
5.10 Trade-offs of R contraction hierarchies on subgoal graphs 238
5.11 Trade-offs of R N-level subgoal graphs 243
5.12 GPPC benchmarks 246
5.13 GPPC results 248

List of Algorithms

1 A common framework for search algorithms. 22
2 Overlay-Connect (Conservative) 57
3 Contract and HeavyContract 60
4 Connect-Search-Refine 73
5 Heavy$R$Contract 74
6 Prune (R, T)-SPC. 75
7 Grow (R, T)-SPC 78
8 IdentifyAccessSubgoals 97
9 StronglyConnectSubgoals 99
10 $F^{\rightarrow}$-Connect 115
11 F-Refine 118
12 $CF^{\rightarrow}$-Connect 127
13 CF-Refine 131
14 $SF^{\rightarrow}$-Connect. 187
15 SF-Connect. 189

Chapter 1

Introduction

Path planning is the problem of finding a sequence of waypoints that an agent can follow through a continuous environment to reach its goal location while avoiding obstacles. A typical approach to solving the path-planning problem in many real-world applications is to discretize the configuration space of the agent into a graph, that can then be searched for a path from a given start vertex to a given goal vertex. The choice of discretization depends mostly on the agent, the environment, and the application. For instance, one can use road networks for navigation systems, grid graphs for video games, or state lattices for agents with kinematic constraints. In some applications of path planning, the environment is static and known in advance, allowing one to analyze its associated graph in a preprocessing phase to generate auxiliary information, which can then be used to answer path-planning queries faster. A number of preprocessing-based path-planning algorithms have been developed that speed up path-planning queries by several orders of magnitude on road networks. These algorithms are applicable to any graph and provably efficient on graphs that have small *highway dimensions*, such as road networks. Grid graphs, on the other hand, have larger highway dimensions than road networks. When some of these algorithms are applied to grid graphs, the speed-up is smaller, and the memory requirements, relative to the original graph, are higher.

State lattices and grid graphs (which are instances of state lattices) have other properties that can be exploited, which can be collectively referred to as their *freespace-structure*. *Translation invariance of freespace distances and freespace-canonical paths:* On state lattices constructed on obstacle-free and infinitely extending environments, called *freespace state lattices*, translating shortest paths by changing the x- or y-coordinates of all their vertices by the same integral amount produces shortest paths, and translating *canonical* paths produces canonical paths. *Octile property*: On *freespace grid graphs*, shortest paths consist of moves in at most two directions, a diagonal one and an associated cardinal one. The hypothesis of this dissertation is as follows: *One can develop preprocessing-based path-planning algorithms for state lattices and grid graphs that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.*

To validate this hypothesis:

(1) We introduce *subgoal graphs*, which are overlay graphs that can be specialized to different types of graphs to exploit their structure. Similar to overlay graphs, subgoal graphs can be used to answer shortest-path queries in three phases, by first *connecting*

the given start and goal vertices to them, *searching* the resulting *query subgoal graphs* for shortest paths, and finally *refining* these paths by replacing their edges with corresponding shortest paths on the original graphs. Unlike overlay graphs, all edges of (query) subgoal graphs satisfy a binary relation R defined over pairs of vertices, called a *reachability relation*. Subgoal graphs can thus be specialized to different types of graphs by first capturing their structures with reachability relations, and then exploiting them in various ways, such as using specialized algorithms to perform connection and refinement. We introduce a hierarchical variant of subgoal graphs, called *N-level subgoal graphs*, and discuss their relationship to contraction hierarchies. We also introduce a suboptimal but complete variant of subgoal graphs, called *strongly connected subgoal graphs*.

(2) We prove the translation invariance of freespace distances and freespace-canonical paths on state lattices and show that it can be exploited to efficiently compute and compactly store freespace information, such as distances or shortest path trees on freespace state lattices. We introduce *freespace-reachability* and *canonical-freespace-reachability* as reachability relations to capture the pairs of vertices on state lattices between which the freespace information is "accurate", and introduce efficient connection and refinement operations that exploit this freespace information. We experimentally show that subgoal graphs constructed with respect to various reachability relations on state lattices can be used to answer path queries faster than A* searches on state lattices, but only by a small factor. We experimentally show that answering queries using strongly connected subgoal graphs constructed with respect to freespace-reachability and canonical-freespace-reachability have *dominating query-time/path-suboptimality trade-offs* compared to answering queries using strongly connected subgoal graphs constructed with respect to *bounded-distance-reachability*, and a *non-dominated query-time/path-suboptimality trade-off* compared to weighted A* searches with various heuristics.

(3) We show that the Octile property of freespace grid graphs can be exploited to (a) construct small subgoal graphs with respect to freespace reachability that use only the convex corners of unblocked cells on grid graphs as vertices, (b) perform connection efficiently during queries by using precomputed *clearance* values, and (c) perform preprocessing in time linear in the size of the underlying grid. We show that jump-point search, an online path-planning algorithm that also exploits structure in grid graphs, can be understood as a search over a subgoal graph constructed on the *direction-extended canonical grid graph*. We show that contraction hierarchies, a preprocessing-based path-planning algorithm that has been developed for road networks with an undominated query-time/memory trade-off in the Grid-Based Path-Planning Competition, can be augmented to exploit structure in grid graphs by augmenting it with reachability relations in various ways. We experimentally demonstrate that answering queries using contraction hierarchies constructed on subgoal graphs achieves a *dominating query-time/memory trade-off* compared to several state-of-the-art path-planning algorithms on grid graphs, and can answer queries *2.42 times faster* than single-row compression, the fastest entry in the Grid-Based Path-Planning Competition, while requiring *139.06 times less memory*.

Our work on using subgoal graphs to exploit freespace structure in state lattices and grid graphs has resulted in preprocessing-based path-planning algorithms that achieve non-dominated query-time/path-suboptimality trade-offs on state lattices and non-dominated query-time/memory trade-offs on grid graphs. At the time of the publication

of this dissertation, subgoal graphs on grid graphs have been applied to moving target search (Nussbaum & Yorukcu, 2015), adapted to 2^k-neighbor grid graphs (Hormazábal, Díaz, Hernández, & Baier, 2017), and used for planning high-level paths for agents maneuvering in continuous and uncertain environments (Zeng, Qin, Hu, Hu, & Yin, 2019). The clearance-based scanning that we use during the connection phases of queries answered using subgoal graphs on grid graphs has been adapted and used in preprocessing-based variants of jump-point search, previously an online path-planning algorithm on grid graphs (Harabor, Grastien, et al., 2014; Rabin & Sturtevant, 2016).

1.1 Problem Definition

In this section, we formally define the path-planning problem and provide a brief overview on preprocessing-based path planning. These subjects are covered in more depth in Chapter 2.

1.1.1 Graphs and Paths

A graph is a tuple $G = (V, E, c)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $c : E \to \mathbb{R}_{>0}$ is a function that assigns a positive length to each edge. G is *undirected* if and only if, for every $(u, v) \in E$, there exists $(v, u) \in E$ with $c(u, v) = c(v, u)$, and *directed* otherwise. An edge $(u, v) \in E$ is an *out-edge* of u and an *in-edge* of v. For any edge (u, v), v is a *successor* of u, and u is a *predecessor* of v.

A path on G is a sequence of vertices $\pi = \langle p_0, \ldots, p_k \rangle$ such that, $\forall i \in \{1, \ldots, k\}$, $(p_{i-1}, p_i) \in E$. $p_0, \ldots, p_k$ are the vertices and $(p_0, p_1), \ldots, (p_{k-1}, p_k)$ are the edges of π. The length of π, $l(\pi)$, is the sum of its edge lengths, that is, $l(\pi) = \sum_{i \in \{1,\ldots,k\}} c(p_{i-1}, p_i)$. We refer to π as an *s-t* path if $v_0 = s$ and $v_k = t$. Given two paths $\pi_1 = \langle u_0, \ldots, u_m \rangle$ and $\pi_2 = \langle v_0, \ldots, v_n \rangle$ with $u_m = v_0$, we denote their *concatenation* as the path $\pi_1 \cdot \pi_2 = \langle u_0, \ldots, u_m = v_0, \ldots v_n \rangle$.

An *s-t* path on G is a shortest path if and only if no other *s-t* path π' exists on G such that $l(\pi') < l(\pi)$. The *s-t* distance on G, $d_G(s, t)$ is the length of a shortest *s-t* path on G. An *s-t* path π is w-suboptimal if and only if $l(\pi) \leq w d_G(s, t)$, and optimal if and only if it is a shortest path.

1.1.2 The Path-Planning Problem

Path planning is the problem of finding a sequence of waypoints that an agent can follow through a continuous environment to reach its goal location while avoiding obstacles. In this dissertation, we consider the discrete version of the path-planning problem. Namely, we assume that the configuration space of the agent is discretized into a graph $G = (V, E, c)$, and that we want to find an *s-t* path on G from a given start vertex $s \in V$ to a given goal vertex $t \in V$. Throughout this dissertation, we use G to denote the graph on which we perform path planning. We refer to s and t as the *query vertices*, and the problem of finding an *s-t* path on G as an *s-t path query*.

A solution to an *s-t* path query, a path π, is optimal if and only if π is a shortest *s-t* path on G, w-suboptimal (bounded suboptimal) if and only if π is a w-suboptimal *s-t*

path on G, and feasible if and only if π is an s-t path on G. A path-planning algorithm is optimal if and only if it is guaranteed to find an optimal solution (for every $s, t \in V$), w-suboptimal if and only if it is guaranteed to find a w-suboptimal solution, and complete if and only if it is guaranteed to find a feasible solution.

We refer to the problem of finding the s-t distance on G as an s-t distance query. We collectively refer to s-t path and distance queries as s-t queries, or simply as queries if the specific s and t are irrelevant.

1.1.3 State Lattices and Grid Graphs

In this dissertation, we are interested in solving the path-planning problem on two specific types of graphs, namely state lattices and grid graphs.

State lattices are constructed by systematically discretizing the environment into a grid of square cells, discretizing various features about the state of an agent, such as its orientation or velocity, into a finite set of integers, and discretizing the motions available to the agent into a finite set of *motion primitives* (Pivtoraiko & Kelly, 2005b; Likhachev & Ferguson, 2009; Kushleyev & Likhachev, 2009). State lattices are typically used for planning paths for agents with kinematic constraints. In this dissertation, we consider state lattices that discretize the location and the orientation of the agent. That is, each vertex specifies the x- and y-coordinates of the location of the agent on the grid and the orientation θ of the agent. We describe state lattices in more detail in Sections 2.1.2 and 4.2.

Grid graphs can be considered as state lattices that discretize only the location of the agent (that is, each vertex specifies only the x- and y-coordinates of the location of the agent on the grid). They are typically constructed with respect to eight motion primitives, each corresponding to a straight-line motion in one of the four cardinal or four diagonal directions. We describe grid graphs in more detail in Sections 2.1.3 and 5.2.

Freespace state lattices and grid graphs are state lattices or grid graphs that are constructed by assuming that the environment is obstacle-free and infinitely extending. The systematicity in the construction of state lattices and grid graphs gives rise to certain properties on freespace state lattices and grid graphs, which we refer to as their *freespace structure*. Namely, on freespace state lattices, translating shortest paths by changing the x- or y-coordinates of all their vertices by the same integral amount produces other shortest paths, and translating *canonical* shortest paths (that is, shortest paths that are the lexically smallest between the vertices they connect, with respect to a canonical ordering on the primitives) produces other canonical shortest paths. We refer to this property as the *translation invariance of freespace distances and freespace-canonical paths*, and prove that it holds for state lattices in Sections 4.4.2 and 4.5.2. On *freespace grid graphs*, which are arguably more "structured" than freespace state lattices, shortest paths consist of moves in at most two directions, a diagonal one and an associate cardinal one. We refer to this property as the *Octile property*, and describe it in further detail in Section 5.3.1.

1.1.4 Assumptions

We make the following assumptions on $G = (V, E, c)$.

Assumption 1.1. $|V| < \infty$.

Assumption 1.2. *G is strongly connected (for every $s, t \in V$, $d(s,t) < \infty$).*

Assumption 1.3. *Every edge $(u,v) \in E$ is the unique shortest u-v path on G.*

We make these assumptions to simplify our definitions and proofs, and think that they are reasonable assumptions to make in many applications of the path-planning problem. Assumption 1.1 is automatically satisfied for state lattices and grid graphs if the underlying grid (the environment) is finite. Assumption 1.2 can be satisfied by assuming that G corresponds to the largest strongly connected component of a given state lattice or grid graph, and can be useful to make in practice if one does not want the agent to move into a vertex from which other vertices become inaccessible. Assumption 1.3 can be satisfied by removing edges (u,v) from G that are not unique shortest u-v paths, without increasing distances on G.

1.1.5 Preprocessing-Based Path Planning

Preprocessing-based path-planning algorithms are a class of path-planning algorithms that operate in two phases: In a *preprocessing phase*, they analyze G and compute auxiliary information about G. In a *query phase*, they find s-t paths (or distances) quickly by using the auxiliary information. In this dissertation, we use the following criteria to compare preprocessing-based path-planning algorithms. *Preprocessing time:* The time it takes to compute auxiliary information about G. *Memory:* The memory required to store auxiliary information about G. *Query time:* The time it takes to find an s-t path on G. *Path suboptimality:* The ratio of the lengths of s-t paths found compared to the lengths of shortest s-t paths.

A preprocessing-based path-planning algorithm with shorter query times is not necessarily "better" than another one with longer query times but shorter preprocessing times. For instance, if a specific application can only accommodate short preprocessing times, it might not be able to use the former algorithm but use the latter one instead, despite its longer query times. In this dissertation, we therefore compare preprocessing-based path-planning algorithms with respect to their *trade-offs*. An Algorithm A *dominates* an Algorithm B with respect to the *trade-off* of N different criteria if and only if Algorithm A is strictly better than Algorithm B with respect to one of the N criteria, and better than or equal to with respect to the other $N-1$ criteria. An Algorithm A is *non-dominated* by a set of K algorithms (with respect to the trade-off of N different criteria) if and only if none of the K algorithms dominate Algorithm A (with respect to the trade-off of N different criteria). Among a set of algorithms, the subset of algorithms with non-dominated trade-offs (with respect to the trade-off of N different criteria) forms the Pareto frontier of that set. We say that an Algorithm A *improves* the Pareto frontier of a set of algorithms if and only if including Algorithm A to the set changes the Pareto frontier of the set. That is, if and only if no algorithm in the set dominates Algorithm A or has a trade-off equivalent to that of Algorithm A.

Observe that, to show that an algorithm improves the Pareto frontier of a set of algorithms with respect to the trade off of N different criteria, it is sufficient to show that it improves the Pareto frontier with respect to only M of the N different criteria. For

instance, we can show that an Algorithm A improves the query-time/memory trade-off of a set of K algorithms by simply showing that Algorithm A has shorter query times than any of the K algorithms. In this dissertation, we therefore compare the trade-offs of algorithms with respect to two criteria at a time, for simplicity. We compare optimal algorithms with respect to their *query-time/memory* trade-offs, but also report their preprocessing times. We compare suboptimal algorithms with respect to their *query-time/path-suboptimality* trade-offs, but also report their preprocessing times and memory requirements.

We provide a detailed description of the limitations of applications of preprocessing-based path-planning as well as an extensive overview of related work in Section 2.3.

1.1.6 Applications and Related Work

Preprocessing-based path-planning algorithms have been studied extensively on road networks, with possible applications in GPS navigation devices and web services, such as Google Maps, that represent real-life road networks as graphs and find shortest paths on them to provide driving directions to their users. *Transit node routing* (Bast, Funke, & Matijević, 2006; Arz, Luxen, & Sanders, 2013) and *hub labeling* (Abraham, Fiat, Goldberg, & Werneck, 2010; Abraham, Delling, Goldberg, & Werneck, 2011, 2012) can answer distance queries 6–7 orders of magnitude faster than Dijkstra searches, by precomputing distances between some pairs of vertices on the road network and answering distance queries with a small number of distance look-ups. *Contraction hierarchies* (Geisberger, Sanders, Schultes, & Delling, 2008; Geisberger, 2008) can answer distance or path queries 4–5 orders of magnitude faster than Dijkstra searches and use little extra memory, by forming a hierarchy among the vertices of a graph and performing searches over the hierarchy that ignore "unimportant" vertices. All three of these algorithms are non-dominated with respect to their query-time/memory trade-offs on road networks (Sommer, 2014) and are applicable to any graph. Abraham et al. show that many preprocessing-based path-planning algorithms that have been developed on road networks, including contraction hierarchies, transit node routing, and hub labeling, are provably efficient on graphs with small *highway dimensions*, such as road networks (Abraham et al., 2010). We provide a more detailed overview of preprocessing-based path-planning algorithms on road networks in Section 2.3.2 and describe highway dimension in Section 3.3.3.

Preprocessing-based path-planning algorithms have been studied less extensively on grid graphs, which are often used in video games or robotics. Video games are a suitable application for preprocessing-based path planning (Sturtevant & Geisberger, 2010) since path planning is performed frequently and thus needs to be fast. Furthermore, path planning in video games typically ignores other agents (Reynolds, 1999), which can help maintain the validity of the preprocessed auxiliary information. *Jump-point search* (Harabor & Grastien, 2011) is an online (that is, does not perform preprocessing) path-planning algorithm on grid graphs that uses *canonical orderings*, (partial) orderings on the moves available to the agent, in order to designate, among multiple symmetric paths that consist of the same set of moves executed in different orders, some as *canonical paths*. Namely, jump-point search uses the diagonal-first canonical ordering to designate the canonical paths as those paths where no cardinal-to-diagonal turn can be replaced with

a diagonal-to-cardinal turn. Jump-point search only explores canonical paths, prunes successors of expanded vertices that produce non-canonical paths, and, if an expanded vertex has a single unpruned successor, immediately expands the successor. Recursively expanding vertices with single unpruned successors allows jump-point search to perform "jumps" in the grid graph, significantly reducing the number of insertions and removals in its OPEN list. Jump point search has been adapted to use preprocessing to perform jumps faster (Harabor et al., 2014; Traish, Tulip, & Moore, 2016), and has been combined with bounding boxes to further improve its pruning (Rabin & Sturtevant, 2016). *Single-row compression* (Strasser, Botea, & Harabor, 2015; Botea, Strasser, & Harabor, 2015) computes all-pairs next-move tables, compresses them by combining multiple entries that share the same next move, and answers path queries with a series of next-move look-ups. Both jump point search and single-row compression have non-dominated query-time/memory trade-offs in the most recent Grid-Based Path-Planning Competition held in 2014 (Sturtevant, Traish, Tulip, Uras, Koenig, Strasser, Botea, Harabor, & Rabin, 2015), with jump point search having the lowest memory requirements and single-row compression having the shortest query times. The other two entries in the competition with non-dominated query-time/memory trade-offs are contraction hierarchies and our entry, N-level subgoal graphs. We provide a more detailed overview of preprocessing-based path-planning algorithms on grid graphs in Section 2.3.3.

State lattices are a relatively recent method of discretizing configuration spaces, and research on state lattices has focused mostly on the generation of motion primitives that are sparse yet representative of the motions available to the agent and the development of heuristics on state lattices (Pivtoraiko & Kelly, 2005b; Likhachev & Ferguson, 2009; Kushleyev & Likhachev, 2009). State lattices have been successfully used in the winning entry in the DARPA Urban Challenge for self-driving cars, Boss from Carnegie Mellon University, for navigation in unstructured and cluttered environments such as parking lots (as opposed to following lanes of roads) (Urmson, Anhalt, Bagnell, Baker, Bittner, Clark, Dolan, Duggins, Galatali, Geyer, et al., 2008). We think that navigation in parking lots is also a suitable application for preprocessing-based path planning since parking lots are mostly static or change in predictable ways. We explain this motivating application, as well as a motivating application on grid graphs, in more detail in Section 2.3.1.

1.2 Hypothesis

Preprocessing-based path-planning algorithms that achieve non-dominated query-time/ memory trade-offs on road networks have been shown to exploit their hierarchical structure and small highway dimensions (Abraham et al., 2010). State lattices and grid graphs, on the other hand, have larger highway dimensions than road networks (Abraham et al., 2010) and, when some of these algorithms are applied to grid graphs, they achieve smaller speed-ups relative to Dijkstra searches and require more memory relative to the size of the original graph. However, state lattices and grid graphs have other properties that can be exploited, namely, their freespace structure. The hypothesis of this dissertation is therefore as follows:

One can develop preprocessing-based path-planning algorithms for state lattices and grid

graphs that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.

1.3 Contributions

We make the following contributions in this dissertation to validate its hypothesis: In Chapter 3, we introduce the subgoal graph framework, which can be specialized to exploit structure in different types of graphs through the use of a reachability relation. We introduce a hierarchical variant of subgoal graphs, called N-level subgoal graphs, and a suboptimal but complete variant of subgoal graphs, called strongly connected subgoal graphs. In Chapter 4, we apply the subgoal graph framework to state lattices by using variants of freespace-reachability as reachability relations. We experimentally demonstrate that answering queries using strongly connected subgoal graphs constructed with respect to variants of freespace-reachability have *dominating query-time/path-suboptimality trade-offs* compared to those constructed with respect to bounded-distance-reachability and a *non-dominated query-time/path-suboptimality trade-off* compared to weighted A* searches. In Chapter 5, we apply the subgoal graph framework to grid graphs by using *safe-freespace-reachability* as the reachability relation. We discuss the similarities and differences between subgoal graphs and jump-point search, and N-level subgoal graphs and contraction hierarchies. We experimentally demonstrate that answering queries using contraction hierarchies constructed on subgoal graphs achieves a *dominating query-time/memory trade-off* compared to several state-of-the-art path-planning algorithms on grid graphs.

We now provide a more detailed summary of these contributions.

1.3.1 The Subgoal Graph Framework

We introduce the subgoal graph framework, which can be specialized to exploit structure in different types of graphs through the use of a *reachability relation*. The subgoal graph framework can be considered as the theoretical foundation of this dissertation, and provides us with the means to exploit freespace structure in state lattices and grid graphs. Specifically:

- We use overlay graphs (Holzer, Schulz, & Wagner, 2009) as the basis of the subgoal graph framework, which are graphs that correctly represent distances between a subset of the vertices of G, using the minimum set of shortcut edges to do so (that is, removing any shortcut edge (u, v) increases the u-v distance on the overlay graph). Overlay graphs have been used (either explicitly or implicitly) by various preprocessing-based path-planning algorithms, such as transit node routing. We introduce the three-phase Connect-Search-Refine algorithm, that can answer path queries optimally using overlay graphs: During the *connection phase*, it connects the start and goal vertices to the overlay graph to form a query overlay graph. During the *search phase*, it searches the query overlay graph for a shortest path between the start and goal vertices. During the *refinement phase*, it replaces the

edges of the path found on the query overlay graph with corresponding shortest paths on G.

We introduce a variant of overlay graphs, called *extended overlay graphs*, which can be considered as the combination of all possible extensions of an overlay graph into different query overlay graphs, and prove that the Connect-Search-Refine algorithm finds shortest paths. We show that vertex contractions, an operation used for constructing contraction hierarchies, can be used to construct overlay graphs and introduce a variant, called heavy contractions, that can be used to construct extended overlay graphs.

- We introduce reachability relations as binary relations defined over pairs of vertices of G. Reachability relations can be used to distinguish those pairs of vertices of G between which a certain property holds. For instance, a reachability relation can be used to distinguish those pairs of vertices between which the distance is less than a given bound, or between which shortest paths can be computed efficiently.

- We define subgoal graphs as overlay graphs constructed with respect to a specific reachability relation: For every pair of start and goal vertices, all edges of the corresponding query subgoal graph are guaranteed to connect pairs of vertices that satisfy the given reachability relation. Therefore, connection phases of queries answered using subgoal graphs only add edges between vertices that satisfy the reachability relation, and refinement phases only find shortest paths between vertices that satisfy the reachability relation. We define the subgoal graph framework as a framework that can be specialized to different types of graphs by choosing a reachability relation and developing efficient connection and refinement algorithms that exploit structure in that type of graph (for instance, the freespace structure in state lattices and grid graphs). We introduce a variant of heavy contractions, called heavy R contractions, and show that they can be used to construct subgoal graphs with respect to a specific reachability relation. We introduce *bounded-distance-reachability* as a reachability relation that distinguishes those pairs of vertices on G between which the distance is no more than a given bound, and show that, on graphs with small highway dimensions, it is possible to construct "locally sparse" subgoal graphs with respect to bounded-distance-reachability.

- We introduce N-level overlay and subgoal graphs, which can be constructed as a sequence of overlay or subgoal graphs, each one constructed on the previous one. We show that contraction hierarchies can be considered as an instance of N-level overlay graphs, and introduce a variant of contraction hierarchies, called R contraction hierarchies, whose edges connect only pairs of vertices that satisfy a given reachability relation R. We discuss how each of these hierarchies can be constructed by using a different variant of the vertex contraction operation, and discuss how each one can be searched for shortest paths by using bidirectional searches.

- We introduce strongly connected subgoal graphs, which can be used to answer path queries with the Connect-Search-Refine algorithm, but without the guarantee of

finding shortest paths. We introduce a simple and efficient algorithm for constructing strongly connected subgoal graphs with respect to a given reachability relation.

1.3.2 Exploiting the Freespace Structure of State Lattices

We apply the subgoal graph framework to state lattices by introducing freespace-reachability and canonical-freespace-reachability as reachability relations and developing connection and refinement algorithms for them that exploit the freespace structure of state lattices. Specifically:

- We introduce freespace state lattices as state lattices constructed on an obstacle-free and infinitely extending environment. We prove that translating shortest paths on freespace state lattices by offsetting the x- or y-coordinates of their vertices by the same amount produces shortest paths. We call this property the *translation invariance of freespace distances*, and show that it allows for the efficient computation and compact storage of pairwise distances on freespace state lattices. We introduce freespace-reachability as a reachability relation on state lattices, which distinguishes those pairs of vertices on state lattices between which the freespace distance is accurate. We introduce a connection algorithm that can identify freespace-reachable vertices from a given vertex efficiently, by performing a depth-first search that uses freespace distances to guarantee that it only explores shortest paths on the state lattice. We introduce a refinement algorithm that can find shortest paths between pairs of freespace-reachable vertices efficiently, by using freespace distances as a perfect heuristic.

- We enforce a canonical ordering on motion primitives that uniquely designates, among multiple shortest paths between two vertices on the corresponding freespace state lattice, one as the freespace-canonical path and ensures that all freespace-canonical paths that originate or terminate at a vertex are guaranteed to form a tree. We prove that translating freespace-canonical paths on freespace state lattices produces other freespace-canonical paths. We call this property the *translation invariance of freespace-canonical paths*, and show that it allows for the efficient computation and compact storage of pairwise freespace-canonical paths. We introduce canonical-freespace-reachability as a reachability relation on state lattices, which distinguishes those pairs of vertices on state lattices between which the canonical-freespace path is unblocked. We introduce a connection algorithm that can identify canonical-freespace-reachable vertices from a given vertex efficiently, by performing a depth-first search without duplicate detection that generates only those successors of expanded vertices that extend a freespace-canonical path into another freespace-canonical path. We introduce a refinement algorithm that can find shortest paths between pairs of canonical-freespace-reachable vertices efficiently, by simply generating the freespace-canonical path between them.

- We experimentally demonstrate that answering queries using subgoal graphs constructed with respect to bounded-distance reachability, freespace-reachability, or canonical-freespace-reachability have similar query times that are only slightly shorter (by a factor of $\sim$2) than A* search times on state lattices. We conjecture that

state lattices may have large highway dimensions, based on our theoretical result that it is possible to construct locally-sparse bounded-distance reachability subgoal graphs on graphs with small highway dimensions.

- We experimentally demonstrate that answering queries using strongly connected subgoal graphs constructed with respect to freespace-reachability or canonical-freespace-reachability have *dominating query-time/path-suboptimality trade-offs* compared to answering queries using strongly connected subgoal graphs constructed with respect to bounded-distance-reachability and a *non-dominated query-time/ path-suboptimality trade-off* compared to weighted A* searches with various heuristics.

1.3.3 Exploiting the Freespace Structure of Grid Graphs

We apply the subgoal graph framework to grid graphs by introducing safe-freespace-reachability as a reachability relation and developing connection and refinement algorithms for safe-freespace-reachability that exploit the freespace structure of grid graphs. We discuss the similarities and differences between subgoal graphs and jump-point search, and N-level subgoal graphs and contraction hierarchies. Specifically:

- We introduce safe-freespace-reachability as a reachability relation, which distinguishes those pairs of vertices on grid graphs between which *all* shortest freespace paths are unblocked. We show that subgoal graphs can be constructed on grid graphs with respect to safe-freespace-reachability by using the convex corners of blocked cells as subgoals. We introduce a connection algorithm for safe-freespace-reachability which uses precomputed clearance values to efficiently scan the grid for convex corners of blocked cells that are safe-freespace-reachable from a given vertex. We prove that this algorithm can be used to construct subgoal graphs in time linear in the size of the underlying grid.

- We introduce direction-extended canonical grid graphs, whose vertices correspond to pairings of cells with the eight cardinal and diagonal directions, and whose edges ensure that all paths on direction-extended canonical grid graphs are diagonal-first and taut. We show that jump-point search can be understood as a search on a subgoal graph that is constructed with respect to freespace-reachability on the direction-extended canonical grid graph. We introduce a variant of jump-point search that explicitly constructs this subgoal graph, which we call the jump-point graph.

- We experimentally evaluate how the query-time/memory trade-off of answering queries using contraction hierarchies changes when we augment contraction hierarchies on grid graphs with reachability relations in three different ways: 1) We show that constructing contraction hierarchies on subgoal graphs or jump-point graphs can decrease both query times and memory requirements. 2) We show that restricting the edges of contraction hierarchies to be freespace- or canonical-freespace-reachable can significantly decrease memory requirements but, on random and room maps, significantly increases query times. 3) We show that refining those

shortcut edges that satisfy a reachability relation with an appropriate refinement algorithm that exploits the freespace structure of grid graphs can decrease query times. Our results also show that answering queries using a combination of contraction hierarchies and subgoal graphs achieves a dominating query-time/memory trade-off compared to answering queries using a combination of contraction hierarchies and jump-point graphs. However, we currently do not understand well the combination of contraction hierarchies and jump-point graphs, and cannot rule out the possibility that combining them in a different way achieves better results.

- We compare answering queries using variants and combinations of subgoal graphs, jump-point graphs, and contraction hierarchies, with the state-of-the-art path-planning algorithms on grid graphs that have been evaluated in the Grid-Based Path-Planning Competition, by performing experiments using the same benchmarks and normalizing our results. Our results suggest that answering queries using contraction hierarchies on subgoal graphs and performing freespace-based refinement is *2.34 times faster* than single-row compression, the fastest entry in the Grid-Based Path-Planning Competition, while requiring *139.06 times less memory*.

At the time of the publication of this dissertation, subgoal graphs on grid graphs have been applied to moving target search (Nussbaum & Yorukcu, 2015), adapted to 2^k-neighbor grid graphs (Hormazábal et al., 2017), and used for planning high-level paths for agents maneuvering in continuous and uncertain environments (Zeng et al., 2019). The clearance-based scanning that we use during the connection phases of queries answered using subgoal graphs on grid graphs have been adapted and used in preprocessing-based variants of jump-point search, previously an online path-planning algorithm on grid graphs (Harabor et al., 2014; Rabin & Sturtevant, 2016).

1.4 Dissertation Structure

This dissertation is structured as follows: In Chapter 2, we provide an in-depth overview of path planning. In Chapter 3, we introduce the subgoal graph framework. In Chapter 4, we apply the subgoal graph framework to state lattices by using freespace-reachability and canonical-freespace reachability as reachability relations. In Chapter 5, we apply the subgoal graph framework to grid graphs by using safe-freespace-reachability as the reachability relation, introduce a variant of jump-point search that uses subgoal graphs, and augment contraction hierarchies with reachability relations. In Chapter 6, we summarize the contributions made in this dissertation. Tables 1.1 and 1.3 summarize the terminology that we use throughout this dissertation. Table 1.2 summarize the types of overlay and subgoal graphs discussed in this dissertation.

Terminology	Description	Ref.
$G = (V, E, c)$	The graph on which we want to find paths	
s, t	Typically used for the start and goal vertices	
u, v, n, p	Typically used for vertices	
S, T	Typically used for sets of vertices $S \subseteq T \subseteq V$	
G_S	Overlay graph	D3.3
$G_{S,T}$	Extended overlay graph	D3.4
$G_{\mathbb{L}}$	N-level overlay graph	D3.12
π	Typically used for paths	
$l(\pi)$	Length of π	
$d(s, t)$	s-t distance on G	
$n \sqsubset \pi \equiv n$ covers π	n is an internal vertex of π (that is, excluding the first and last vertices on π)	D3.1
$n \sqsubset (s, t)$	$n \sqsubset$ a shortest s-t path on G	D3.1
$S \sqsubset \pi$	$\exists n \in S$ such that $n \sqsubset \pi$	D3.1
$S \sqsubset (s, t)$	$\exists n \in S$ such that $n \sqsubset (s, t)$	D3.1
R	Typically used for a reachability relation $R \subseteq V \times V$	D3.7
D_S	Direct-reachability with respect to S	D3.2
R_S	Direct-R-reachability with respect to S	S3.3.1
$R^{A \Rightarrow B}(D_S^{A \Rightarrow B}, R_S^{A \Rightarrow B})$	R-reachable pairs in $A \times B \subseteq V \times V$	D3.2
$R^{A \to B}(D_S^{A \to B}, R_S^{A \to B})$	R-reachable pairs (u, v) in $A \times B \subseteq V \times V$ where $u \neq v$	D3.2
BDb	Bounded-distance reachability with bound b ($(s, t) \in$ BDb iff $d(s, t) \leq b$)	S3.3.1

Table 1.1: Terminology used for arbitrary graphs. The references (Ref.) specify the definition (D) or the section (S) that introduce the terminology.

Graph	Description	Ref.
Overlay graph (G_S)	Contains an edge (u, v) if and only if $u, v \in S$, $S \not\sqsubset (u, v)$, and $u \neq v$	D3.3
Extended overlay graph ($G_{S,T}$)	Contains an edge (u, v) if and only if $u, v \in T$, $S \not\sqsubset (u, v)$, and $u \neq v$	D3.4
R subgoal graph	An overlay graph with R-reachable edges only (S is an R-SPC on G)	D3.9
N-level overlay graph ($G_{\mathbb{L}}$)	Sequence of extended overlay graphs G_{S_{i+1},S_i} where S_i is the set of vertices with level i or higher according to the level function $\mathbb{L}$	D3.12
R N-level subgoal graph	N-level overlay graph with R-reachable edges only	D3.14
R contraction hierarchy	R N-level subgoal graph with no same-level edges except at the highest level	D3.14
Contraction hierarchy	N-level overlay graph with no same-level edges	D3.14
R strongly connected subgoal graph	A strongly connected graph with vertex set S where, $\forall n \in V$, $\exists u, v \in S$ with $(u, n) \in R$ and $(n, v) \in R$	D3.15

Table 1.2: Types of overlay and subgoal graphs. The references (Ref.) specify the definition that introduce the type of graph.

Terminology	Description	Ref.
$\mathcal{G}$, $\mathcal{M}$, $\mathcal{L}_{\mathcal{G},\mathcal{M}}$	Grid $\mathcal{G} \subseteq \mathbb{Z}^2$, set of primitives $\mathcal{M}$, state lattice $\mathcal{L}_{\mathcal{G},\mathcal{M}}$ induced by $\mathcal{M}$ on $\mathcal{G}$	S4.2
$\vec{m}$	Typically used for a primitive	
$\mathbb{Z}^2$	Freespace grid	D4.1
$\mathcal{F} = \mathcal{L}_{\mathbb{Z}^2,\mathcal{M}}$	Freespace state lattice	D4.1
freespace path	A path on $\mathcal{F}$	D4.1
freespace-shortest path	A shortest path on $\mathcal{F}$	D4.1
freespace-canonical path	For some s and t, the lexically smallest freespace-shortest s-t path	D4.4
$s + \vec{m} = t$	$\vec{m}$ induces the edge (s, t)	S4.2
$s + (x, y) = t$	Translating s by (x, y) results in t	S4.4.2
$\pi + (x, y) = \pi'$	Translating π by (x, y) results in π'	S4.4.2
F, Fb	F = freespace-reachability, Fb = F $\cap$ BDb $(s, t) \in$ F if and only if a freespace-shortest s-t path is unblocked on G	D4.2
CF, CFb	CF = canonical-freespace-reachability, CFb = CF $\cap$ BDb $(s, t) \in$ CF if and only if the freespace-canonical s-t path is unblocked on G	D4.5
SF	Safe-freespace-reachability $(s, t) \in$ SF if and only if all freespace-shortest s-t paths are unblocked on G	D5.1
$\vec{c}$, $\vec{d}$, $\vec{v}$	Typically used for a move in a cardinal, diagonal, or any direction	S5.2
$\vec{v}_1 \perp \vec{v}_2$	$\vec{v}_1$ and $\vec{v}_2$ are perpendicular	S5.2
$\vec{c}_1 + \vec{c}_2 = \vec{d}$	d is a combination of $\vec{c}_1$ and $\vec{c}_2$	S5.2
$s + \vec{v} \times k = t$	t is reached by moving k steps in direction $\vec{v}$ from s	S5.2
taut path	A path whose two-edge subpaths are shortest paths on G	S5.2
freespace-taut path	A path whose two-edge subpaths are shortest paths on $\mathcal{F}$	S5.2
freespace-diagonal-first path	Shortest path on $\mathcal{F}$ where diagonal moves appear before cardinal ones	S5.2
freespace-cardinal-first path	Shortest path on $\mathcal{F}$ where cardinal moves appear before diagonal ones	S5.2
locally-diagonal-first path	Taut path on G where no cardinal-to-diagonal turn can be replaced with a diagonal-to-cardinal turn	S5.4.1

Table 1.3: Terminology, state lattices and grid graphs. The references (Ref.) specify the definition (D) or the section (S) that introduce the terminology.

Chapter 2

Path Planning

The motion-planning problem is the problem of finding a continuous motion that an agent can follow in a continuous environment to reach a goal configuration, while avoiding the obstacles in the environment. Finding exact solutions to this problem while taking into account the kinematic constraints of the agent has been shown to be PSPACE-hard (Canny, 1988) and not practical for real-world applications (Choset, Lynch, Hutchinson, Kantor, Burgard, Kavraki, & Thrun, 2005). The path planning problem can be considered to be a simplified version of the motion-planning problem that discretizes the configuration space of the agent into a graph that can be searched for paths. The path-planning problem can be solved in time polynomial in the size of the graph by using search algorithms such as A*. In this chapter, we provide an overview of the path-planning problem.

This chapter is organized as follows: In Section 2.1, we discuss three types of graphs that can be generated by discretizing configuration spaces of agents, namely state lattices, grid graphs, and road networks. In Section 2.2, we discuss algorithms commonly used for searching graphs for paths. In Section 2.3, we discuss preprocessing-based path planning, its limitations, applications, and related work.

2.1 Discretizing Configuration Spaces into Graphs

In this section, we discuss three types of graphs that can be generated by discretizing the configuration space of an agent. We first formally define the concepts of environments, agents, and configuration spaces, and then discuss how configuration spaces can be systematically discretized into state lattices, grid graphs, or road networks.

2.1.1 Environments, Agents, and Configuration Spaces

An *environment* (work space) is a k-dimensional Euclidean space defined by the shapes of its boundary and the obstacles contained within that boundary. An *agent* operating in this environment can be defined by the shapes of its components, their kinematics (how they move with respect to each other and the environment), and kinematic constraints (the constraints on their movements, for instance, acceleration constraints, turning radii, or speed limits). The *configuration* of the agent in the environment may include its location in the environment, as well as other features of the agent, such as its orientation,

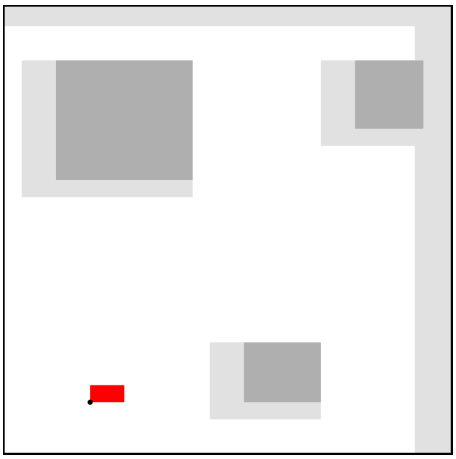


Figure 2.1: Environment and configuration space of a translating agent. Red rectangle: A translating agent (whose orientation is fixed). Black dot: The reference point on the agent to identify its location in the environment. Black lines: Boundary of the environment. Dark gray areas: Obstacles in the environment. Light gray areas: Inflation of obstacles to account for the configuration space treating the agent as a point rather than a rectangle. That is, the agent's footprint does not intersect with the obstacles or the boundary of the environment if and only if its reference point is within the white area. Image downloaded from `https://en.wikipedia.org/wiki/Motion_planning`.

joint angles, and velocity. The *configuration space* of the agent is the set of all possible configurations of the agent in the environment. A motion is a continuous curve in the configuration space. Motions that can be executed by the agent without violating its kinematic constraints are called *kinematically feasible*, and motions that avoid collisions with the boundary of the environment or the obstacles within that boundary are called *collision-free*. Motions that are both kinematically feasible and collision-free are called *feasible*. Motion planning is the problem of finding a *feasible* motion between two configurations in the configuration space of an agent.

Figure 2.1 shows an environment and its corresponding configuration space for a translating agent (that is, an agent that is only able to change its location but not its orientation) with a rectangular footprint. In this example, the environment and the configuration space are both 2-dimensional. Each point in the configuration space corresponds to a location for the agent with respect to the reference point on the bottom-left corner of its footprint. That is, in the configuration space, the agent is treated as a point rather than a rectangle, and obstacles are "inflated" to account for the "shrinking" of the agent into a point.

In this dissertation, we consider the path-planning problem as the discretized (and therefore simplified) version of the motion-planning problem, where the configuration space of the agent is discretized into a graph. Vertices of the resulting graph correspond to (sets of) configurations, and edges correspond to feasible motions that connect these configurations.

Figure 2.2: A video game environment represented as a grid. Image taken as a screenshot from the game Age of Empires II.

2.1.2 State Lattices

State lattices are constructed by discretizing the configuration space of an agent into a graph, where the environment is discretized into a grid, various features of the configuration of the agent are discretized into a finite set of integers, and the motions available to the agent are discretized into a finite set of motion primitives (Pivtoraiko & Kelly, 2005b; Likhachev & Ferguson, 2009; Kushleyev & Likhachev, 2009). We now describe each of these discretizations.

Grids are regular tessellations of n-dimensional Euclidean spaces. Regular triangles, squares, or hexagons can be used to tessellate 2-dimensional (2D) Euclidean spaces, and n-dimensional cubes can be used to tessellate n-dimensional Euclidean spaces. In this dissertation, we only consider tessellations of 2D environments using squares, and simply use the term *grid* to refer to this tessellation and *cell* to refer to each square in the tessellation. When a grid is overlaid on a 2D environment, cells that intersect with obstacles are called *blocked* cells, and cells that do not intersect with obstacles are called *unblocked* cells. Figure 2.2 shows an example from the video game Age of Empires II, which uses grids to represent environments. The cells that contain the town center, trees, or bushes are blocked and cannot be traversed by the agents.

The vertices of state lattices are called *states* (configurations), which correspond to "evenly spaced" discrete points in the configuration space of the agent. For an n-dimensional configuration space of an agent operating in a k-dimensional environment, each state in the corresponding state lattice is specified by n integers: The first k integers describe the location of (the reference point of) the agent on the grid, and the remaining integers describe other discretized features of the configuration of the agent, such as its orientation, velocity, or joint angles.

The edges of state lattices correspond to feasible motions between the states they connect, and are constructed with respect to a given set of *motion primitives* (primitives, for short). Each primitive represents a kinematically feasible motion for the agent, and may induce multiple edges in the state lattice. There are multiple factors that determine

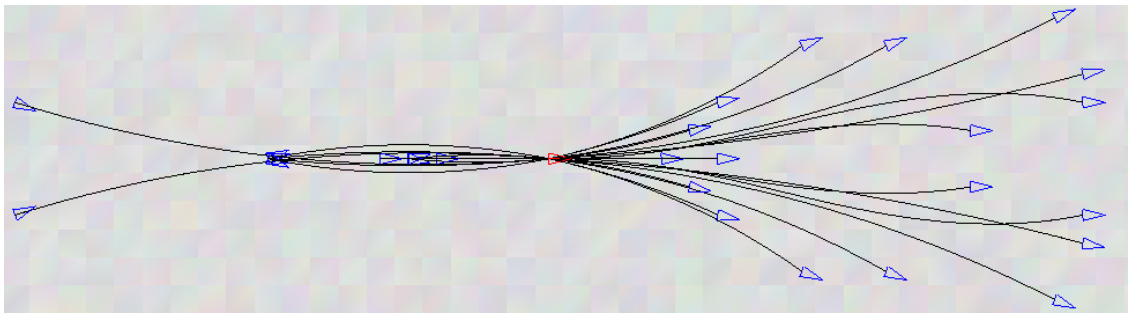


Figure 2.3: Primitives used for the Boss entry in the DARPA Urban Challenge. The red triangle shows the current location and orientation of the agent. Each blue triangle shows a location and orientation that can be reached by executing a motion primitive.

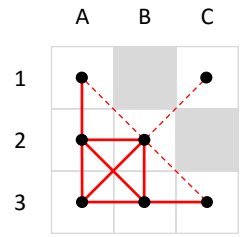
whether a primitive induces a particular edge (u, v) in the state lattice: 1) State u satisfies the "preconditions" of the primitive, which are defined over the discretized features of the configuration of the agent except for its location. For instance, a primitive that moves the agent in a straight line in a particular direction might be applicable only at those states where the agent is facing in that particular direction. 2) Executing the primitive from state u results in a collision-free motion. 3) Executing the primitive from state u results in state v.

Figure 2.3 shows the set of primitives that was used in Carnegie Mellon University's entry in the DARPA Urban Challenge (Urmson et al., 2008). There are 32 discrete orientations for the agent, and the figure shows all primitives available to the agent when it is facing East. Executing a primitive can change both the location and orientation of the agent. If the environment contains obstacles (the grid contains blocked cells), the agent might not be able to execute some of these primitives, namely those that result in a collision.

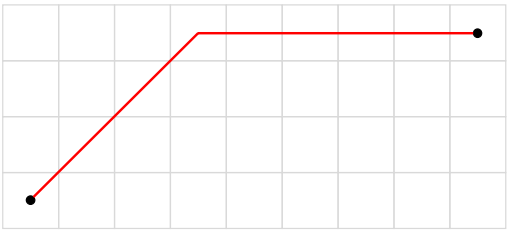
The systematic discretization of configuration spaces using a small set of primitives gives rise to structure in state lattices. Namely, state lattices typically have many translationally symmetric edges (since each primitive can induce multiple edges in the state lattice) and, therefore, many translationally symmetric paths (since paths are combinations of primitives). We describe this structure in further detail in Chapter 4 and discuss how it can be exploited to speed up path planning on state lattices.

2.1.3 Grid Graphs

Grid graphs can be considered to be instances of state lattices where states describe only the discretized locations of the agent, and primitives have no preconditions. The vertices of grid graphs are typically placed at the corners or centers of unblocked cells, and the primitives typically allow the agent to move in straight lines on the grid. That is, grid graphs typically do not consider the kinematic constraints of the agent, for instance, by allowing the agent to instantaneously turn in place. They are typically used in video games or when planning paths for holonomic robots.



(a) A grid graph. Dashed lines correspond to corner-cutting moves and are not included in the grid graph.



(b) Shortest paths on grid graphs with no blocked cells consist of moves in at most two directions: A diagonal one and an associated cardinal one.

Figure 2.4: Grid graphs.

The Grid-Based Path-Planning Competition defines grid graphs as follows: 1) The vertices are placed at the centers of unblocked cells. 2) The agent can move between cardinally or diagonally adjacent unblocked cells, but is not allowed to "cut corners". That is, it cannot move in a diagonal direction unless it can also move in both associated cardinal directions. 3) Cardinal moves cost 1, and diagonal moves cost $\sqrt{2}$. Figure 2.4a shows an example of a grid graph. Corner-cutting diagonal moves are shown as dashed lines and are not represented as edges in the grid graph. For instance, the diagonal move from A1 to B2 is a corner-cutting move since the agent cannot move from A1 to B1. In this dissertation, we also use *4-neighbor* grid graphs to illustrate the operations of our algorithms. A 4-neighbor grid graph is a grid graph without any edges that correspond to diagonal moves.

Since grid graphs are instances of state lattices, they inherit the structural properties of state lattices. However, since grid graphs are "very specific" instances of state lattices, they have other properties that are not necessarily present in an arbitrary state lattice. Namely, on a grid with no blocked cells, shortest paths consist of moves in at most two directions, namely a diagonal one and an associated cardinal one. Figure 2.4b shows an example. We refer to this property as the "Octile property", which is used in the literature to efficiently calculate "Octile distances", that is, distances between vertices assuming that the grid contains no blocked cells. We describe the Octile property of grid graphs in further detail in Chapter 5 and discuss how it can be exploited to speed up path planning on grid graphs.

2.1.4 Road Networks

Road networks can be used to represent how roads are connected within a city, country, or continent. Their edges typically represent road segments, and their vertices typically represent points of intersection of these road segments. Edge lengths typically represent either the length of the road segment (*travel-distance metric*) or the expected time to traverse the road segment by taking into account the type of the road segment and traffic (*travel-time metric*). Figure 2.5 shows the international European E-road network, where green lines represent road segments, and red lines represent borders between countries.

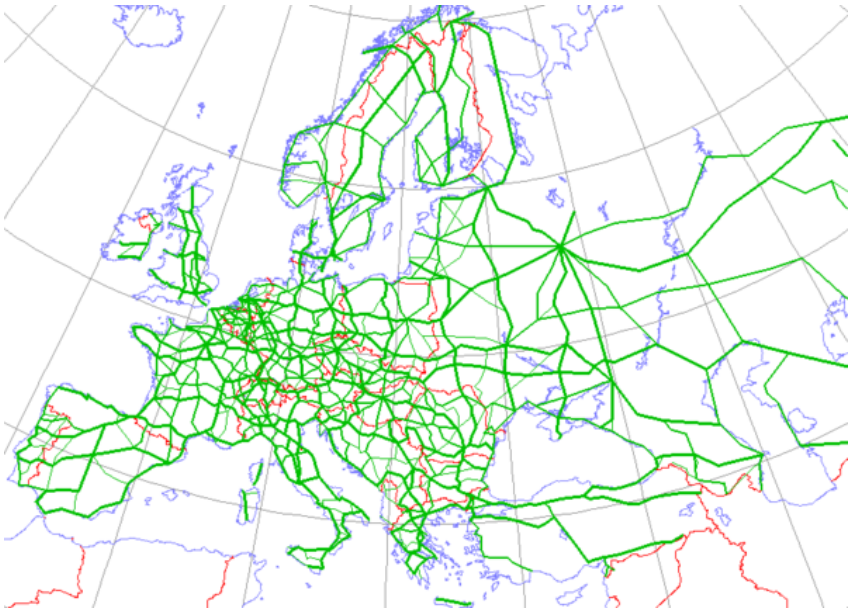


Figure 2.5: International E-road network. Image downloaded from `https://en.wikipedia.org/wiki/International_E-road_network`

Road networks that are used in real-world applications, such as GPS navigation devices or Google Maps, are typically very large since they not only represent highways between cities, but also the roads within cities. For instance, the USA road network used in the 9th DIMACS implementation challenge has 24 million vertices and 58 million edges (Demetrescu, Goldberg, & Johnson, 2009). However, these road networks also have a *hierarchical structure* (Kalapala, Sanwalani, Clauset, & Moore, 2006). For instance, paths between two locations in different cities can typically be found by considering only the highway network and the roads within those two cities; whereas the roads within other cities can be considered as "unimportant" and be ignored. As we discuss in Section 2.3.2, path-planning algorithms that operate on road networks typically exploit the hierarchical structure of road networks in order to achieve very short query times.

2.2 Search Algorithms

In this section, we first describe a common framework for search algorithms and discuss its properties, and then describe some well-known search algorithms within this framework.

2.2.1 A Common Framework for Search Algorithms

Given a graph G, a start vertex s, and a goal vertex t, a search algorithm can be considered to be a systematic exploration of paths on G until an s-t path is found. Initially, the set of explored paths contains only the trivial s-s path $\langle s \rangle$. This set is then grown by

Algorithm 1 Search

Input: A graph G, a start vertex s, and a goal vertex t.
Output: An s-t path π on G.

```
// Initialize
OPEN = {s}
CLOSED = ∅
for all u ∈ V do
    parent(u) ← undefined
// Grow search tree
while SelectNodeToExpand(OPEN) ≠ t do
    u ← SelectNodeToExpand(OPEN)
    OPEN ← OPEN \ {u}
    CLOSED ← CLOSED ∪ {u}
    for all successors v of u do
        if v ∉ OPEN ∪ CLOSED then
            OPEN ← OPEN ∪ {u}
            parent(v) ← u
        else if v ∈ OPEN ∧ ShouldUpdateParent(v, u) then
            parent(v) ← u
// Extract path
π ← ⟨t⟩
u ← t
while u ≠ s do
    π ← ⟨parent(u), u⟩ · π
    u ← parent(u)
return π
```

extending an s-u path in the set with a successor v of u to generate an s-v path, until a (shortest) s-t path is generated.

Algorithm 1 outlines a common framework for search algorithms, which differs from the "sketch" of search algorithms outlined above as follows: Algorithm 1 organizes the set of paths explored by the search into a *search tree* that maintains at most one copy of each vertex of G, uses *expansions* to extend an s-u path in the search tree into s-v paths for all successors v of u, and never expands a vertex more than once. We now discuss Algorithm 1 in more detail. We note that, since we assume that G is finite and is guaranteed to contain an s-t path for any $s, t \in V$ (Assumptions 1.1 and 1.2), Algorithm 1 does not consider the possibility that it might not be able to find an s-t path.

- **Search tree:** Algorithm 1 compactly represents the set of paths generated during the search as a search tree. Each node n in the search tree is labeled with a vertex $L(n) \in V$, the root node is labeled with the start vertex s, and each edge from a node n to a child node m corresponds to an edge $(L(n), L(m)) \in E$. Therefore, any path from the root node to a node n in the search tree corresponds to an s-$L(n)$ path on G.

Although it is possible to have different nodes n and m in the search tree with the same label $L(n) = L(m)$, Algorithm 1 avoids such *duplicate nodes* in its search tree, since they can negatively effect search times (Taylor & Korf, 1993). Since Algorithm 1 avoids duplicate nodes, each node in its search tree has a unique label. For simplicity, we refer to a vertex and the node it uniquely corresponds to with the same name (that is, $L(n) = n \in V$), and use the term vertex for both unless it results in ambiguity. Algorithm 1 maintains its search tree by maintaining a *parent* for every vertex except for the root vertex s (lines 4–5, 14, and 16). Since Algorithm 1 maintains a single parent for each vertex, each vertex can appear in the search tree at most once and, therefore, the search tree maintained by Algorithm 1 cannot have duplicate vertices.

- **Expansions, OPEN and CLOSED lists, and duplicate detection:** Algorithm 1 initializes the search tree to only contain s as its root (lines 4–5) and then grows it by repeatedly *expanding* vertices from the search tree (lines 6–16). It maintains two lists: The CLOSED list contains exactly the vertices in the search tree that have already been expanded, and the OPEN list contains exactly the vertices in the search tree that have not been expanded. Initially, the OPEN list contains only the start vertex s (line 2) and the CLOSED list is empty (line 3). Algorithm 1 always chooses the next vertex u to expand from the OPEN list (line 8), which removes u from the OPEN list (line 9) and places it in the CLOSED list (line 10). When a vertex u is expanded, for each successor v of u, Algorithm 1 considers extending the current s-u path in the search tree into an s-v path by using the edge (u, v) (lines 11–16): If v does not appear the search tree, it is added to the search tree as a child of u (lines 12–14). Otherwise, v is not added to the search tree for a second time to avoid duplication, and Algorithm 1 decides whether to update the current parent of v to u (lines 15–16).

- **Termination and completeness guarantees:** Algorithm 1 expands each vertex at most once: Each vertex is inserted into the OPEN list at most once, namely when it is first added to the search tree (line 13), and then removed from the OPEN list if it is expanded (line 9). Since G is finite (Assumption 1.1), and since Algorithm 1 expands each vertex at most once, it is guaranteed to expand a finite number of vertices. Furthermore, since an s-t path exists on G (Assumption 1.2), Algorithm 1 is guaranteed to select t for expansion at some point: Suppose that Algorithm 1 does not select t for expansion and, during the execution of line 7, OPEN $= \emptyset$. Let $\pi = \langle v_0, \ldots, v_k \rangle$ be any s-t path. Since $v_k = t$ is not selected for expansion, v_{k-1} could not have been selected for expansion either because, otherwise, $v_k = t$ would have been added to OPEN. We can recursively apply this argument to deduce that $v_0 = s$ could not have been selected for expansion. However, this results in a contradiction, since Algorithm 1 always selects $v_0 = s$ as the first vertex to expand (lines 2 and 8). Therefore, Algorithm 1 must select t for expansion before terminating.

- **Implementation and complexity:** Since Algorithm 1 expands each vertex at most once and, therefore, evaluates each edge at most once, it executes lines 8–10

at most $O(|V|)$ times and lines 12–16 at most $O(|E|)$ times. That is, it performs at most $O(V)$ insertions into and removals from the OPEN list, $O(V)$ insertions into the CLOSED list, $O(E)$ checks to see if a vertex is in the OPEN or CLOSED lists, and $O(E)$ decisions whether to update the parent of a vertex. Algorithm 1 can be implemented to execute each of these operations in $O(1)$ time, for a total of $O(E)$ runtime, using $O(V)$ memory, by maintaining the CLOSED list as a hash table, and maintaining the OPEN list as a queue or a stack.

The search algorithms that we describe in the remainder of this section mainly follow the framework we have outlined in Algorithm 1, and differ in how they select the next vertex to expand from the OPEN list (line 8), how they decide to update the parents of vertices v when multiple s-v path are found during the search (line 15), and whether they also perform a simultaneous search backwards from the goal, using predecessors rather than successors of expanded vertices. We discuss how their implementations differ from the one outlined above and how these differences affect their runtimes in their respective sections.

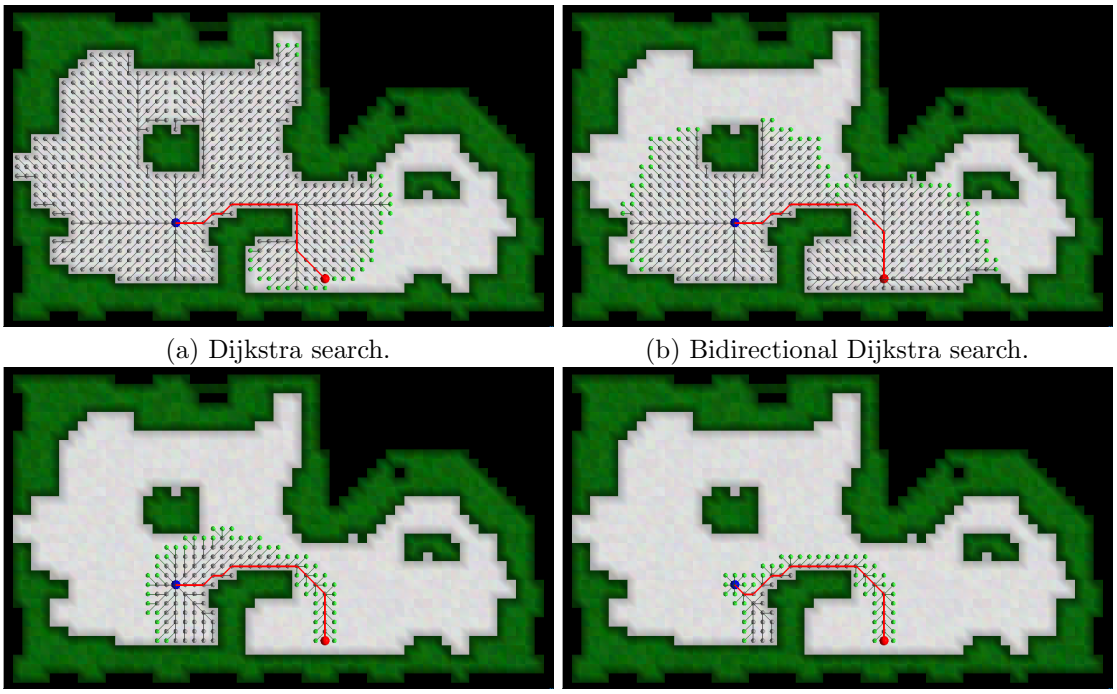
2.2.2 Breadth-First and Depth-First Searches

Breadth-first and depth-first searches select the next vertex to expand from the OPEN list as a vertex with the minimum or maximum depth in the search tree, respectively. Neither search updates the parents of vertices except when the vertices are first added to the search tree. Both searches can be implemented to run in $O(|E|)$ time and use $O(|V|)$ memory, as discussed in Section 2.2.1, by implementing the OPEN list of breadth-first search as a (first-in, first-out) queue and implementing the OPEN list of depth-first search as a stack (first-in, last-out queue). Neither breadth-first nor depth-first search is guaranteed to find shortest paths. However, breadth-first search is guaranteed to find minimum-hop paths (that is, paths with the smallest number of edges), and can therefore be used to find shortest paths on graphs where all edges have the same length. In this dissertation, we are mainly interested in breadth-first and depth-first searches for developing efficient connection and refinement algorithms for our reachability relations, since they allow for $O(1)$ time insertions into and removals from the OPEN list.

2.2.3 Dijkstra Search

Dijkstra search, also known as Dijkstra's algorithm or uniform-cost search, selects the next vertex to expand from the OPEN list as the vertex with the minimum *tentative distance* (g-value) from the start vertex, and is guaranteed to find shortest paths (Dijkstra, 1959). Dijkstra search is the basis for the rest of the search algorithms that we discuss in this section, as well as the basis for many of the preprocessing-based path-planning algorithms that we discuss in the next section.

Dijkstra search maintains a g-value $g(u)$ for every vertex u, such that, at any point during the search, $g(u)$ is equal to the s-u-distance in the search tree (or, if no such path exists, $g(u) = \infty$). Initially, $g(s) = 0$, and $g(u) = \infty$ for all $u \neq s$. When a vertex u is expanded, for each successor v of u, Dijkstra search checks whether $g(u) + c(u, v) < g(v)$, that is, if the g-value of v can be decreased by using the current s-u path and the



(a) Dijkstra search. (b) Bidirectional Dijkstra search.

(c) A* search. (d) Weighted A* search ($w = 2$).

Figure 2.6: Search trees and paths found by various search algorithms on a grid graph. Start and goal vertices are shown as blue and red disks, respectively. Vertices in the OPEN and CLOSED lists are shown as gray and green disks, respectively. The path found by the algorithm is shown as a red line.

edge (u, v). If so, $g(v)$ is updated to $g(u) + c(u, v)$ and the parent of v is updated to u. Dijkstra search maintains the invariant that, whenever a vertex u is selected for expansion, $g(u) = d(s, u)$. Therefore, when the goal vertex t is selected for expansion, $g(t) = d(s, t)$, and the search tree contains a shortest s-t path. Figure 2.6a shows the search tree of a Dijkstra search on a grid graph after it has found a shortest s-t path.

In the worst case, Dijkstra search performs $O(|E|)$ *decrease-key* operations (decrease the g-value of a vertex), and $O(|V|)$ *extract-min* operations (remove a vertex with the minimum g-value from OPEN). Using a binary heap implementation of the OPEN list, which supports $O(log|V|)$ decrease-key and extract-min operations, Dijkstra search runs in $O(|E|log|V|)$ time. Using a Fibonacci heap implementation of the OPEN list, which supports amortized $O(1)$ decrease-key and $O(log|V|)$ extract-min operations (Fredman & Tarjan, 1987), Dijkstra search runs in $O(|E| + |V|log|V|)$ time.

2.2.4 Bidirectional Dijkstra Search

Bidirectional Dijkstra search simultaneously runs a (forward) Dijkstra search from the start vertex and a backward Dijkstra search from the goal vertex, and terminates when the two searches "meet" at a vertex u, that is, when u is selected for expansion by both

searches. The s-u path is extracted from the forward search and the u-t path is extracted from the backward search, which are then combined into an s-t path which is guaranteed to be a shortest path. Figure 2.6b shows the search trees of a bidirectional Dijkstra search on a grid graph after it has found a shortest s-t path.

Bidirectional Dijkstra searches typically either alternate between expanding vertices in the forward and backward searches, or expand a vertex from the search whose *radius* is smaller, where the radius of a Dijkstra search is equal to the smallest g-value of a vertex in its OPEN list. A vertex is expanded at most once in bidirectional Dijkstra searches, either by the forward search or the backward search, since the search terminates when a vertex is about to be expanded for a second time.

The main reason to use a bidirectional Dijkstra search over a (unidirectional) Dijkstra search is that it typically expands fewer vertices. Whereas a Dijkstra search from s to t expands vertices with distances of at most $d(s,t)$ from s, a bidirectional Dijkstra search that meets at a vertex u expands vertices with distances of at most $d(s,u)$ from s on G, and vertices with distances of at most $d(u,t)$ from t on the reverse graph of G.

2.2.5 A* Search

A* search (Hart, Nilsson, & Raphael, 1968) extends Dijkstra search by using a *heuristic* h to "guide" the search towards the goal vertex.

A heuristic is a function $h : V \times V \rightarrow [0, \infty)$ that estimates the distance between any two vertices on G. h is *admissible* if and only if it never overestimates distances on G, that is, for any $u, v \in V$, it holds that $h(u,v) \leq d(u,v)$. h is *consistent* if and only if 1) for any $t \in V$, $h(t,t) = 0$, and 2) the heuristic distances obey the *triangle inequality*, that is, for any $(u,v) \in E$ and $t \in V$, it holds that $h(u,t) \leq c(u,v) + h(v,t)$. A* searches use a heuristic to estimate the distances of vertices to goal vertices only. Therefore, if a goal vertex t is available, we use $h(u)$ as a shorthand for $h(u,t)$ if doing so does not result in ambiguity.

A* search chooses the next vertex to expand from the OPEN list as the vertex u with the smallest f-value $f(u) = g(u) + h(u)$. If h is admissible but not consistent, A* search might need to expand some vertices more than once to be guaranteed to find shortest paths, which is disallowed in our framework. If h is consistent, then A* search is guaranteed to find shortest paths, and is *optimally efficient* when doing so. That is, no optimal unidirectional search algorithm that uses h can expand fewer vertices than A* (modulo tie-breaking) (Pearl, 1985). Figure 2.6c shows the search tree of an A* search using Octile distances as a consistent heuristic on a grid graph, after it has found a shortest s-t path.

If a consistent heuristic h_1 is no less informed than a consistent heuristic h_2 (that is, for any u, $h_1(u) \geq h_2(u)$), then an A* search that uses h_1 is guaranteed to expand no more vertices than an A* search using h_2 (modulo tie-breaking). Dijkstra search can be considered as an A* search that uses the least informed heuristic, namely the zero-heuristic h_0, where, for any $u, t \in V$, $h_0(u,t) = 0$.

2.2.6 Weighted A* Search

Weighted A* search (Pohl, 1970) extends A* search by using a parameter $w \geq 1$ to weigh the contribution of the heuristic to the f-values of vertices. Weighted A* search selects the next vertex to expand from the OPEN list as the vertex u with the minimum f-value $f(u) = g(u) + wh(u)$. If h is consistent, weighted A* search is guaranteed to find a w-suboptimal path. Increasing the value of w makes the search more "greedy", by steering it more aggressively towards the goal, which usually results in fewer expansions but also the search finding longer paths. Figure 2.6d shows the search tree of a weighted A* search with $w = 2$ using Octile distances as a consistent heuristic on a grid graph, after it has found a w-suboptimal s-t path.

2.3 Preprocessing-Based Path Planning

Preprocessing-based path-planning algorithms analyze G in a *preprocessing phase* to generate auxiliary data, which can then be used to answer s-t queries faster in a *query phase*. In this section, we discuss the limitations and applications of preprocessing-based path-planning algorithms and discuss related work.

2.3.1 Limitations and Applications

Consider a preprocessing-based path-planning algorithm that precomputes the next vertex u along a shortest s-t path for all pairs of vertices s and t. This computation can be done in $O(|V|^3)$ time using the Floyd-Warshall algorithm (Floyd, 1962; Warshall, 1962), and the auxiliary data can be stored using $O(|V|^2)$ memory. An s-t query can then be answered in time linear in the number of edges along an s-t path. Although the query phase of this algorithm runs in the minimum possible time to output a complete path description, its preprocessing time and memory requirements can be limiting factors in its applicability to real-world problems. For instance, it could require hours of preprocessing time and terabytes of storage for a graph with 1 million vertices. This example highlights the first limitation for preprocessing-based path-planning algorithms, namely their preprocessing time and memory requirements. Different applications can accommodate algorithms with different amounts of preprocessing time and memory requirements, and therefore, a preprocessing-based path-planning algorithm with shorter query times is not necessarily better than another one with shorter preprocessing times or smaller memory requirements.

Another limitation of preprocessing-based path-planning algorithms is that, in many real-world applications of path planning, the environment is not static. Changes in the environment, such as other agents moving in the environment and acting as obstacles, might invalidate the auxiliary data computed during preprocessing. Although recomputing the data might be feasible for some applications (for instance, if there is time before the next query and preprocessing is sufficiently fast), "repairing" the data can be faster. For instance, repairing a pairwise distance matrix when a single edge is added or removed from the graph is asymptotically faster than recompuing it (Demetrescu & Italiano, 2004).

Despite their limitations, preprocessing-based path-planning algorithms can be used to speed up path planning by several orders of magnitude for applications that can accommodate their preprocessing times and memory requirements. Below, we provide example applications for preprocessing-based path-planning algorithms that operate on road networks, grid graphs, or state lattices.

- **Road networks:** With the increasing popularity of GPS navigation devices, finding shortest routes within or between cities has become an increasingly popular application of path planning using road networks. Web services such as Google Maps answer thousands of queries per second on continental scale networks, where both the unidirectional and bidirectional versions of Dijkstra searches require several seconds to find a shortest path (Sommer, 2014). Using preprocessing-based path-planning can reduce query times to microseconds on road networks (Sommer, 2014) and these services can afford to store large amounts of auxiliary data in order to decrease the load on their servers by significantly improving query times. Although travel-times can to change throughout the day due to traffic, the overall structure of the road network mostly remains the same, allowing a two-step preprocessing approach that first preprocesses the road network without information on edge lengths, and then using a customization phase that adjusts the auxiliary data when information on edge lengths are available (Dibbelt, Strasser, & Wagner, 2014; Delling, Goldberg, Pajor, & Werneck, 2015).

- **Grid graphs:** Grid graphs have been used in many video games (such as StarCraft, Baldur's Gate, Age of Empires II, Warcraft III, Dragon Age: Origins, Dawn of War), where paths for agents need to be calculated within milliseconds due to real-time constraints. This can be particularly challenging for real-time strategy games such as StarCraft or Age of Empires II, which can have multiple players in the game each controlling up to 200 units. Furthermore, path planning often needs to share CPU time with other processes that run within the game, and can also be used as a subroutine by higher-level AI systems that make strategic decisions on how to move the units. To meet these demands, path-planning systems in video games often make compromises on solution quality (that is, the lengths of paths returned by path planning). Preprocessing-based path planning can be used as an alternative in order to achieve short query times (Sturtevant & Geisberger, 2010).

  Although video game maps are rarely static due to many agents moving around in the environment, these agents are typically ignored during path planning. The reason for this is two-fold: First, in order to avoid other agents whose current paths are known, path planning needs to consider them as moving obstacles, which requires a time component to be added to searches, making them slower. Second, even if a plan is found that avoids other agents' paths, the other agents' paths can frequently change, which can invalidate the current agent's path. A common practice to address this problem is to simply ignore other agents during path planning and use steering algorithms to avoid collisions with other agents (Reynolds, 1999), which can significantly reduce, or completely eliminate, the frequency of updates to the data calculated during preprocessing.

- **State lattices:** State lattices have been used for path planning for autonomous vehicles such as aerial vehicles (Thakur, Likhachev, Keller, Kumar, Dobrokhodov, Jones, Wurz, & Kaminer, 2013), automobiles (Likhachev & Ferguson, 2009), boats (Svec, Shah, Bertaska, Alvarez, Sinisterra, Von Ellenrieder, Dhanak, & Gupta, 2013), and all terrain vehicles (Pivtoraiko & Kelly, 2005a). Navigating autonomous vehicles through cluttered environments such as parking lots require complex maneuvers that need to be planned and executed very efficiently, even at human driving speeds (∼15 mph) (Likhachev & Ferguson, 2009). Therefore, similar to video games, path-planning systems for autonomous vehicles often make compromises on solution quality, for instance, by using weighted A* searches, in order to achieve real-time performance (Likhachev & Ferguson, 2009). However, unlike video games, these compromises often have real-world consequences, such as increased fuel consumption. Using preprocessing-based path-planning could help achieve a better query-time/solution-length trade-off to mitigate these consequences, or simply to meet the demands for the real-time nature of the application.

  The winning entry in the DARPA Urban challenge for self-driving cars, the Boss vehicle from Carnegie Mellon University, uses state lattices for path-planning in parking lots (Urmson et al., 2008). Given the imminent popularity of self-driving cars, it is not far-fetched to imagine automated parking lots that need to efficiently navigate cars in and out of the parking lot. We consider this as a possible application for preprocessing-based path planning. Although unoccupied parking spaces can become occupied and vice versa, preprocessing could assume that all of the parking spaces are occupied and that a single parking space (the target) would be considered to be unoccupied during queries. Although this precludes the vehicle from following shorter paths through unoccupied parking spaces, it might be undesirable for vehicles to follow such paths to begin with, both due to safety reasons and because, ideally, one would like to park the vehicle at the closest available parking space which can be accessed without going through other available parking spaces.

2.3.2 Related Work: Road Networks

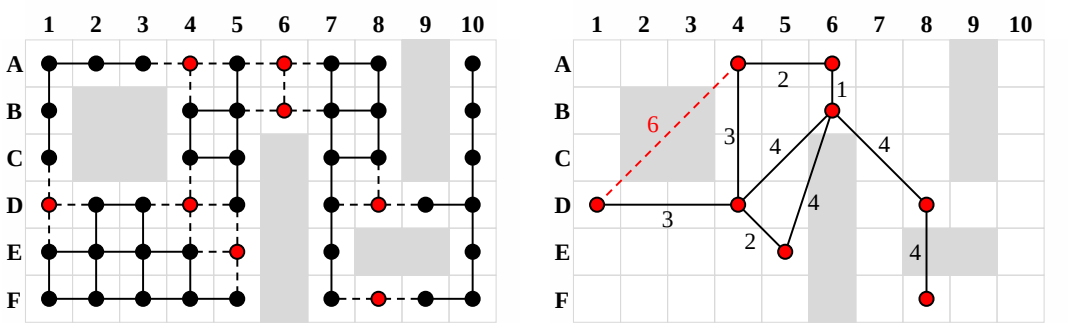
Path planning on road networks has recently attracted a considerable amount of interest in the research community, with the increasing popularity of GPS navigation devices and web services such as Google Maps that allow users to plan shortest routes within or between cities. Over the past two decades, a number of algorithms have been developed and a competition was held for preprocessing road networks (Demetrescu et al., 2009). These algorithms can be used to answer distance queries optimally, and most of them can also be used to answer path queries optimally. Although these algorithms are applicable to any type of graph, they typically achieve greater speed-ups on road networks compared to other types of graphs, such as grid graphs (Abraham et al., 2010). The survey article of Bast, Delling, Goldberg, Müller-Hannemann, Pajor, Sanders, Wagner, and Werneck groups these algorithms into five categories, based on the techniques that they use (Bast et al., 2016). Below, we provide an overview of preprocessing-based path-planning algorithms developed for road networks, by grouping them into the same five categories.

- **Goal-directed techniques:** Algorithms that use goal-directed techniques use the position of the goal vertex to prune vertices from searches. A very simple example of this technique is employed by A* search, which uses a heuristic to avoid expanding vertices that are "sufficiently distant" from the goal vertex. Although Euclidean distance (straight-line distance) or geodesic distance (distance on the surface of a sphere) can approximate distances on road networks pretty accurately under the travel-distance metric, heuristics based on these distances are much less informed under the travel-time metric (Goldberg & Harrelson, 2005).

  As we have discussed in Section 2.2.5, using more informed heuristics in A* searches may result in fewer expansions and thus faster searches. *True-distance heuristics* use precomputed distances on G to more accurately estimate heuristic distances during searches. One such example of a true-distance heuristic is the *landmark heuristic* (also known as the differential heuristic) (Goldberg & Harrelson, 2005). During preprocessing, distances from every vertex to a small set of landmark vertices $L \subseteq V$ are calculated and stored using $O(|L||V|)$ memory. During searches, the heuristic distance from a vertex s to a vertex t with respect to a landmark l is then calculated as $h_l(s,t) = |d(s,l) - d(t,l)|$, and the heuristic distance with respect to all landmarks L is calculated as $h_L(s,t) = \max_{l \in L} h_l(s,t)$. An A* search that uses the landmark heuristic is called the ALT (A*, landmarks, and triangle inequality) algorithm. We discuss more variants of true-distance heuristics in the context of grid graphs in Section 2.3.3.

  Another technique for goal-directed pruning is to label each edge (u,v) with a set of vertices $L(u,v)$ during preprocessing such that, for any vertex $t \in V$, $t \in L(u,v)$ if and only if (u,v) can appear as the first edge along a shortest u-t path. During queries, any edge (u,v) whose label does not contain the goal vertex t can be safely pruned from the search since it cannot be part of a shortest path to the goal. Computing and storing exact labels for each edge can be costly, however, as it requires an all-pairs shortest paths computation and $O(|V||E|)$ memory for storing the labels. Therefore, the two algorithms that we describe next instead use approximate labels, which overestimate the labels for each edge to guarantee optimality. *Geometric containers* (Schulz, Wagner, & Weihe, 2000; Wagner, Willhalm, & Zaroliagis, 2005) annotate each edge (u,v) with a region described by a geometric container, such as a bounding box, that contains all $t \in L(u,v)$. *Arc flags* (Lauther, 2004; Hilger, Köhler, Möhring, & Schilling, 2009) partition V into disjoint sets $V_1, \ldots, V_k$ that have similar numbers of vertices and small boundaries. Each edge (u,v) is then annotated with k bits where the ith bit is set if and only if $L(u,v)$ contains at least one vertex from V_i.

- **Separator techniques:** Algorithms that use separator techniques exploit the fact that road networks tend to have small separators (Eppstein & Goodrich, 2008; Delling, Goldberg, Razenshteyn, & Werneck, 2011). A vertex separator of G is a set of vertices $S \subset V$ whose removal decomposes G into multiple disjoint components, which preferably contain similar numbers of vertices. An edge separator of G is defined similarly, except that edges are removed to decompose G. Figure 2.7(a) shows a vertex separator of a 4-neighbor grid graph.



(a) A vertex separator (red disks) decomposing a 4-neighbor grid graph into 5 disjoint components.

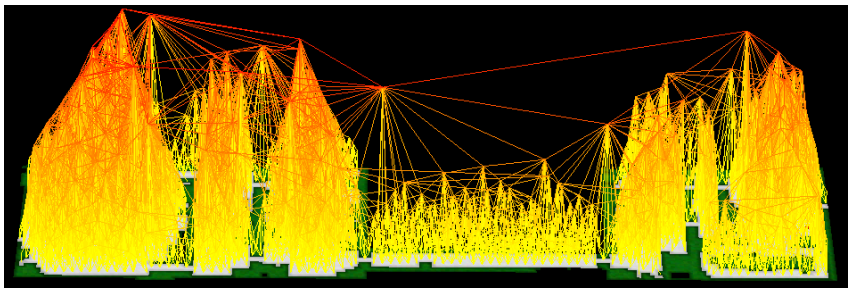
(b) An overlay graph constructed from the vertex separator from (a). The shortcut edge (D1, A4) is not included since it is redundant with the path ⟨ D1, D4, A4 ⟩.

Figure 2.7: An overlay graph constructed from a vertex separator.

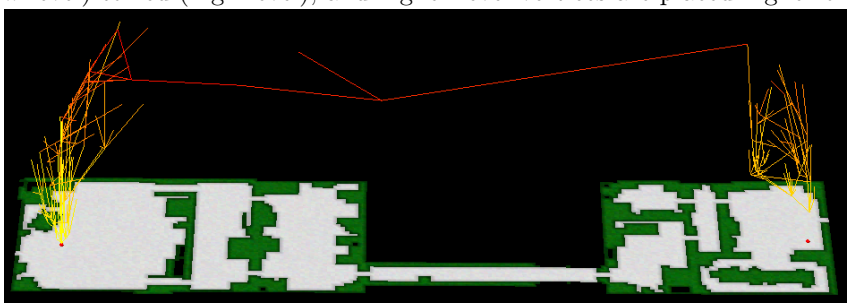
Schulz et al. use vertex separators to identify a set of "important" vertices $S \subseteq V$ and add shortcut edges between them to create an *overlay graph* G_S that preserves distances between vertices in S (Schulz et al., 2000). Figure 2.7(b) shows an overlay graph constructed from a vertex separator on a 4-neighbor grid graph. During queries, searches consider only the overlay graph and the components that contain the start and goal vertices, ignoring the rest of the vertices. This approach can be extended to multiple levels, by creating overlay graphs on top of previous ones (Schulz, Wagner, & Zaroliagis, 2002; Holzer et al., 2009), or by adding many more shortcut edges to the graph during preprocessing (Delling, Holzer, Müller, Schulz, & Wagner, 2009), which allows searches to ignore even more vertices. Other variants exists that use edge separators to construct the overlay graphs (Jung & Pramanik, 2002; Delling et al., 2015).

In this dissertation, we use overlay graphs as the basis of our subgoal graph framework. However, we do not require their vertices to be vertex separators, and we answer queries using them with the Connect-Search-Refine algorithm that first connects the start and goal vertices to them, then searches the resulting query overlay graph for a shortest path, and finally refines this path into a path on G by replacing its edges with corresponding shortest paths on G. We describe overlay graphs and how the subgoal framework uses them in greater detail in Chapter 3.

- **Hierarchical techniques:** Algorithms that use hierarchical techniques do not necessarily use hierarchies themselves, but exploit the inherent hierarchy of road networks to prune vertices from searches. Sufficiently long shortest paths eventually use one of a very small number of important road segments, such as highways. Once the search is "far away" from the start and goal vertices, it might be possible for it to consider only these important road segments and ignore the "less important" parts of the graph.



(a) A contraction hierarchy constructed on a grid graph. Vertices are colored on a scale from yellow (low level) to red (high level), and higher-level vertices are placed higher on the grid.



(b) Search trees of a bidirectional search on the contraction hierarchy from (a). Both frontiers explore the hierarchy in the "up" direction.

Figure 2.8: Contraction hierarchies.

One of the earlier algorithms that exploit this intuition is based on the notion of *reach* (Gutman, 2004). Intuitively, a vertex with a short reach is relevant only for queries over short distances, whereas a vertex with a long reach can be relevant for queries over longer distances. Specifically, the reach of a vertex u with respect to a shortest s-t path π that contains u is calculated as $\min(d(s,u), d(u,t))$, and the (global) reach of u, $r(u)$, is calculated as the maximum reach of u with respect to any shortest path that contains u. During preprocessing, the reach of every vertex is precomputed and stored. During an s-t query, u can be safely pruned from the search if it can be determined that $r(u) < \min(d(s,u), d(u,t))$. However, it might be difficult to determine $d(s,u)$ or $d(u,t)$ during queries. One possible method to use the notion of reach to prune vertices from searches is to answer queries using a bidirectional Dijkstra search: If u has been expanded in the forward or backward search, its g-value in that search is equal to $d(s,u)$ or $d(u,t)$, respectively. If u has not been expanded by the forward or backward search, the current radius of the forward or backward search provides a lower bound on $d(s,u)$ and $d(u,t)$, respectively, which can be used to prune u from the search. Adding shortcut edges to G has been shown to reduce the reaches of most vertices, speeding up both preprocessing and queries (Goldberg, Kaplan, & Werneck, 2009).

Another technique for exploiting the hierarchical structure of road networks is to actually construct hierarchies on road networks to capture their structure. Earlier examples of algorithms that use hierarchies include *multi-level overlay graphs* (Schulz

et al., 2002; Holzer et al., 2009), *highway hierarchies* (Sanders & Schultes, 2005, 2006) and *highway-node routing* (Schultes & Sanders, 2007). Answering queries using *contraction hierarchies* (Geisberger et al., 2008; Geisberger, 2008) builds on ideas from these earlier algorithms, but is faster and conceptually simpler than its predecessors. Contraction hierarchies are constructed by assigning levels to the vertices of G and adding shortcut edges between them to guarantee that, between any two vertices $s, t \in V$, the contraction hierarchy contains an *up-down path*, a path that visits vertices first in increasing then in decreasing orders of levels, with length $d(s, t)$. During queries, contraction hierarchies can be searched with a modified bidirectional Dijkstra search where the forward search constructs the up-part and the backward search constructs the down-part of a shortest up-down s-t path. That is, both searches search "upward" in the hierarchy, ignoring any vertices that cannot be reached from the start vertex with an up path or cannot reach the goal vertex with a down path, respectively. Figure 2.8a shows an example of a contraction hierarchy constructed on a grid graph, and Figure 2.8b shows the search trees of the bidirectional Dijkstra search over this contraction hierarchy. Essentially, contraction hierarchies can capture the hierarchical structure of road networks so that searches over them can avoid generating the "less important" (lower-level) vertices when expanding the "more important" (higher-level) vertices.

The name "contraction" hierarchy comes from the fact that contraction hierarchies are constructed by contracting the vertices of G one by one, that is, removing vertices from G and adding shortcut edges to preserve shortest paths between the remaining vertices. The contraction order plays an important role in determining the memory requirements and the preprocessing and query times of answering queries using contraction hierarchies, and is usually determined heuristically. We describe the contraction operation in greater detail in Section 3.2.6, the construction of contraction hierarchies in Section 3.4.3.3, and the modified bidirectional Dijkstra search that can be used to search contraction hierarchies in Section 3.4.4.

Answering queries using contraction hierarchies is one of three algorithms in Sommer's survey that has a non-dominated query-time/memory trade-off on road networks (Sommer, 2014), along with *transit-node routing* and *hub labeling* that we describe next. On the Western Europe road network with 18 million vertices and 42 million edges, queries can be answered using different variants of contraction hierarchies in 9–900 microseconds, using 100–600 megabytes of memory.[1] In comparison, variants of transit-node routing and hub labeling can answer queries within 0.2–4 microseconds, but require 1.8–18 gigabytes of memory. Furthermore, answering queries using contraction hierarchies has a non-dominated query-time/memory trade-off in the Grid-Based Path-Planning Competition. We experimentally demonstrate in Chapter 5 that answering queries on grid graphs by using a combination of subgoal graphs and contraction hierarchies achieves a dominating query-time/memory trade-off compared to answering queries using contraction hierarchies only.

[1] We estimate these values from Figure 4 in the survey.

- **Bounded-hop techniques:** Algorithms that use bounded-hop techniques add "virtual" shortcut edges to G to ensure that, between any two vertices, there exists a shortest path with no more than a fixed number of hops (edges). For instance, using pairwise distances can be considered as adding virtual shortcut edges between all pairs of vertices so that there exists a shortest path with at most one hop between any two vertices. Distance queries can then be answered in constant time by simply looking up the length of the corresponding 1-hop path in the pairwise distance table. Algorithms that we describe in this category add a much smaller number of virtual shortcut edges to guarantee that a shortest path with at most two or three hops exists between any two vertices. They can answer distance queries with only a small number of table lookups, avoiding search altogether, and arguably have more reasonable query-time/memory trade-offs compared to using pairwise distance tables.

  Hub labeling (Abraham et al., 2011, 2012) can be considered a 2-hop technique. During preprocessing, each vertex u is labeled with a set of vertices $L(u)$, called the *hubs* of u, along with their distances from u. These labels satisfy the property that, for any $s, t \in V$, a shortest s-t path contains a vertex $u \in L(s) \cap L(t)$. s-t distance queries can then be answered by scanning the labels of s and t for the vertex u that minimizes $d(s,u)+d(u,t)$. By sorting the labels of each vertex during preprocessing, these scans can be performed during queries in time linear in the size of the labels, by using an operation that is similar to the merging of two ordered lists. For directed graphs, hub-labeling distinguishes between forward labels $L_f(u)$ and backward labels $L_b(u)$ for each vertex u, and uses $L_f(s)$ and $L_b(t)$ to answer s-t distance queries. Hub labeling can also be used to answer s-t path queries, by recursively replacing the pair (s,t) with the pairs (s,u) and (u,t) for the vertex $u \in L_f(s) \cap L_b(t)$ that minimizes $d(s,u)+d(u,t)$, until all pairs correspond to edges of G.

  Transit-node routing (Bast et al., 2006; Arz et al., 2013) can be considered a 3-hop technique. During preprocessing, pairwise distances between a small set of *transit nodes* $T \subseteq V$ and distances from every vertex u to their *access nodes* $A(u) \subseteq T$ are computed and stored. A transit node $v \in T$ is an access node of a vertex $u \in V \setminus T$ if and only if no shortest u-v path on G passes through another transit node. (Similar to hub labeling, transit node routing distinguishes between forward and backward transit nodes on directed graphs, which we omit from our description for brevity). During an s-t distance query, transit-node routing first tries to prove that a shortest s-t path on G passes through a transit node, by using a *locality filter*, which is typically implemented by partitioning V into *cells* and checking whether s and t belong to different cells. If so, the query is treated as a *long-distance query* and is answered by identifying the vertices $u \in A(s)$ and $v \in A(t)$ that minimize the sum $d(s,u) + d(u,v) + d(v,t)$. Otherwise, the query is treated as a *local query* and is answered by using a different algorithm, for instance, by using contraction hierarchies.

  Transit-node routing can be considered as implicitly using the overlay graph G_T of transit nodes. Rather than storing the overlay graph, transit-node routing stores

pairwise distances on G_T to avoid searches. Rather than identifying the edges that connect query vertices to the overlay graph, transit-node routing stores those edges implicitly as access nodes. However, it does not store pairwise distances between vertices in $V \setminus T$ and, therefore, has to treat some queries as local queries, when the extension of the overlay graph with the query vertices does not contain a path that corresponds to a shortest s-t path on G.

- **Combinations:** The algorithms in this category combine techniques from the previous categories. REAL (Goldberg et al., 2009) combines reach with landmarks, by using the landmark heuristic to provide better lower-bounds for reach-based pruning. Core-ALT (Bauer, Delling, Sanders, Schieferdecker, Schultes, & Wagner, 2010; Delling & Nannicini, 2012) combines overlay graphs and landmarks, by computing landmark distances only for the vertices in the overlay graph. ReachFlags (Bauer et al., 2010) combines reach, arc flags, and overlay graphs, by computing an overlay graph that contains only vertices with sufficiently high reaches, and computing arc flags only for edges in the overlay graph. CHASE (Bauer et al., 2010) combines contraction hierarchies with arc flags, and computes arc flags only for edges between vertices with high levels in the hierarchy. TNR-AF (Bauer et al., 2010) combines transit-node routing with arc flags to reduce the number of distance look-ups.

Most of the algorithms that we have described in this section are applicable to any type of graph, but work especially well on road networks, which typically have small graph separators (Eppstein & Goodrich, 2008; Delling et al., 2011) and small *highway dimensions* (Abraham et al., 2010). We describe the notion of highway dimension in Section 3.3.3. Bast et al. remark that "proving better running time bounds than those of Dijkstra's algorithm is unlikely for general graphs; in fact, there are inputs for which most algorithms are ineffective" (Bast et al., 2016). Abraham et al. show that, on undirected graphs with highway dimension h and diameter D, after running a polynomial-time preprocessing routine, queries can be answered using contraction hierarchies or reach in $O((h \log h \log D)^2)$ time, hub labeling in $O(h \log h \log D)$ time, and long-range transit-node routing in $O(h^2)$ time (Abraham et al., 2010).

2.3.3 Related Work: Grid Graphs

Preprocessing-based path-planning algorithms have been studied less extensively on grid graphs than road networks. However, developments on road networks and the recent Grid-Based Path-Planning Competition (Sturtevant, 2012b) have spurred interest in the area over the past decade. Preprocessing-based path-planning algorithms on grid graphs typically use different techniques from their counterparts on road networks, due to the different structures and applications of grid graphs compared to road networks. Following Bast et al.'s example, we group preprocessing-based path-planning algorithms on grid graphs into several categories based on the techniques that they use.

- **Abstraction techniques:** Algorithms that use abstraction techniques construct an abstract graph G' of G whose vertices are either a subset of the vertices of G or represent groups of vertices of G. Unlike overlay graphs, abstract graphs do not

necessarily preserve distances on G. Therefore, abstract graphs can typically be constructed with fewer edges than overlay graphs, but cannot necessarily be used to find shortest paths.

*Hierarchical path-finding A** (HPA*) (Botea, Müller, & Schaeffer, 2004) partitions the vertices of G into *clusters* by using larger grid cells and selects, for each adjacent pair of clusters, a single edge $(u, v) \in E$ that connects them on G. These edges are called *inter-edges*, and their endpoints form the set of vertices S of the abstract graph. HPA* also adds *intra-edges* that connect vertices in S within the same cluster through shortest paths. That is, the abstract graph it uses can be considered to be multiple overlay graphs, one for each cluster, that are connected to each other through inter-edges. The abstract graph is then used as an overlay graph to answer queries, using the Connect-Search-Refine algorithm. However, since the abstract graph does not necessarily represent distances on G correctly, HPA* is not guaranteed to find shortest paths. HPA* can be extended to use multiple levels of abstraction, each level of abstraction created by using larger grid cells to generate larger clusters.

*Partial refinement A** (PRA*) (Sturtevant & Buro, 2005) uses a hierarchical abstraction, where the lowest level of the hierarchy corresponds to G, and the vertices at higher levels of the hierarchy correspond to groups of vertices at lower levels. Specifically, when generating the $(i + 1)$st level of the hierarchy, the vertices in the ith level are grouped into disjoint cliques of size at most 4. Each such clique becomes a vertex at level $i + 1$. An edge (U, V) is added to the $(i + 1)$st level of abstraction if and only if, for some $u \in U$ and $v \in V$, an edge (u, v) exists at the ith level of abstraction. The length of (U, V) is determined by calculating x- and y-coordinates for U and V by averaging the x- and y-coordinates of their constituent vertices, and then using these coordinates to determine the Euclidean (straight-line) distance between them. The preprocessing ends once the highest level contains a single vertex. During an s-t query, PRA* performs an A* search at every level of the hierarchy, starting at the highest level, from the abstract vertex S that contains s to the abstract vertex T that contains t. The path Π_i found at level i is then used to constrain the search at level $i-1$. Specifically, any vertex at level $i-1$ that is not contained in a higher-level vertex U that appears on Π_i is pruned from the search. Since the distances are not preserved in higher-level abstractions, PRA* is not guaranteed to find shortest paths. In order to find short paths in practice, PRA* can perform searches starting at a lower level of the hierarchy, which is determined heuristically as level $\lfloor l/2 \rfloor$, where l is the lowest level of the hierarchy that groups s and t into the same vertex.

Finally, the path-planning algorithm used in the video game Dragon Age: Origins uses abstractions (Sturtevant & Geisberger, 2010). During preprocessing, the grid is divided into clusters (sectors), similar to HPA*. Each cluster is further subdivided into *regions* that correspond to different connected components within the cluster. The abstract graph contains a single representative vertex from each region, and edges between the representative vertices of adjacent regions. Similar to HPA*, the

abstract graph is used as an overlay graph to answer queries, and can be extended with additional levels of abstraction.

- **Dead-end techniques:** Algorithms that use dead-end techniques identify regions (subsets of vertices) of G that are relevant only for searches between some pairs of start and goal vertices. These regions can be considered to be *dead-ends* for searches between other pairs of start and goal vertices and therefore be safely pruned.

  Dead-end heuristics (Björnsson & Halldórsson, 2006) decompose G into smaller components and construct a *component graph* of G. s-t queries are answered by first performing a depth-first search on the component graph, between the components S and T that contain the start and goal vertices, respectively, to identify the components that appear on *some* S-T path on the component graph. All other components are labeled as dead-ends, and are pruned from the subsequent A* search on G.

  Another idea for identifying dead-ends is based on the notion of *swamps* (Pochter, Zohar, Rosenschein, & Felner, 2010). A swamp is a set of vertices S whose removal from the graph does not change the distances between the remaining vertices. A set of (disjoint) swamps is called a *modular swamp* if and only if the union of any subset of them forms a swamp. During queries, the (individual) swamps (in a modular swamp) that do not contain the start or the goal vertex can be safely pruned from the search. Pochter et al. observe that some sets of vertices S' that are not swamps may become swamps if another set of vertices S is removed from the graph. They introduce *swamp hierarchies* to exploit this observation, in order to prune more vertices during searches. For brevity, we omit a more detailed description of swamp hierarchies.

- **True-distance heuristics:** Algorithms that use true-distance heuristics store precomputed (true) distances between some pairs of vertices, which are then used to calculate well-informed heuristic distances during A* searches. One such example of a true-distance heuristic is the landmark heuristic that we have discussed in Section 2.3.2, which is referred to as the *differential heuristic* by the heuristic search community (Sturtevant, Felner, Barrer, Schaeffer, & Burch, 2009). *Canonical heuristics* (Sturtevant et al., 2009) and *compressed differential heuristics* (Goldenberg, Sturtevant, Felner, & Schaeffer, 2011) can be considered to be variants of differential heuristics, where distances to only a small subset of (instead of all) the landmarks are stored for each vertex.

  Another class of true-distance heuristics are calculated by summing up true distances, rather than subtracting them. These calculations can be considered to be similar to the calculations made by the algorithms that use bounded-hop techniques discussed in Section 2.3.2. *Gateway heuristics* (Björnsson & Halldórsson, 2006) are calculated by decomposing G into smaller components and constructing a component graph of G, similar to dead-end heuristics. During preprocessing, pairwise distances are computed and stored between *gateways*, which correspond to boundaries between adjacent components. During queries, the heuristic distance between two vertices s and t is calculated by identifying gateways G_s and G_t associated

with the components containing s and t, respectively, that minimizes the value of the expression $h(s, G_s) + d(G_s, G_t) + h(G_t, t)$, where $h(u, G_u) = h(G_u, u)$ is equal to the minimum Octile distance between the vertex u and a vertex in the gateway G_u. *Border heuristics* (Felner, Sturtevant, & Schaeffer, 2009) are calculated using a similar idea, but use pairwise distances between the components, instead of between gateways. *Portal heuristics* (Goldenberg, Felner, Sturtevant, & Schaeffer, 2010) are also calculated using a similar idea, but use pairwise distances between the individual vertices in gateways, called *portals*. Although the ideas behind the calculation of these different heuristic distances are similar, they are calculated by using different decompositions of G into components. These differences are beyond the scope of this overview.

The calculation of portal heuristics can be considered to be implicitly using an overlay graph of the set of portals. Similar to the vertices of overlay graphs constructed on road networks (Schulz et al., 2002; Holzer et al., 2009), the set of portals also form vertex separators of G. Similar to transit-node routing, portal heuristics use pairwise distances on the overlay graph. Goldenberg et al. propose a portal-based search algorithm that simply uses the components that contain the start and goal vertices plus the portal distance between these components. This algorithm is very similar to transit-node routing, except that the access nodes of vertices are not precomputed but identified during queries.

- **First-move compression techniques:** Algorithms that use first-move compression techniques compress *first-move tables*. First-move tables are similar to pairwise-distance tables, except that each entry (s, t) in a first-move table specifies the first move, an edge (s, u), along a shortest s-t path on G. First-move tables can be used to find shortest s-t paths by repeatedly following first moves from s to t. Although first-move tables and pairwise-distance tables have the same number of $O(|V|^2)$ entries, each entry in a first-move table can be stored using only $\lceil \log b \rceil$ bits, where b is the maximum out-degree of a vertex of G. On an 8-neighbor grid graph, since the maximum out-degree of a vertex is 8, each entry in a first-move table can be stored using 3 bits. The algorithms that we describe in this category compress first-move tables by grouping together entries that specify the same first move.

  Copa (Botea, 2011; Botea & Harabor, 2013) compresses first-move tables by exploiting the observation that vertices that are "close together" on the grid graph can typically be reached from a vertex s with the same first move. During preprocessing, for each vertex $s \in V$, the grid is divided into non-overlapping rectangles such that the vertices contained in each rectangle can be reached from s with the same first move. During queries, the first move along an s-t path can be determined by iterating over the rectangles computed for s to find the rectangle that contains t. Copa also uses more sophisticated compression schemes, such as storing the most frequent first move for each vertex s as the default first move from s, and computing the rectangles for s accordingly.

  Single-row compression (SRC) (Strasser et al., 2015; Botea et al., 2015) orders the columns (which correspond to goal vertices) of first-move tables so that their rows (which correspond to start vertices) contain consecutive entries with the same first

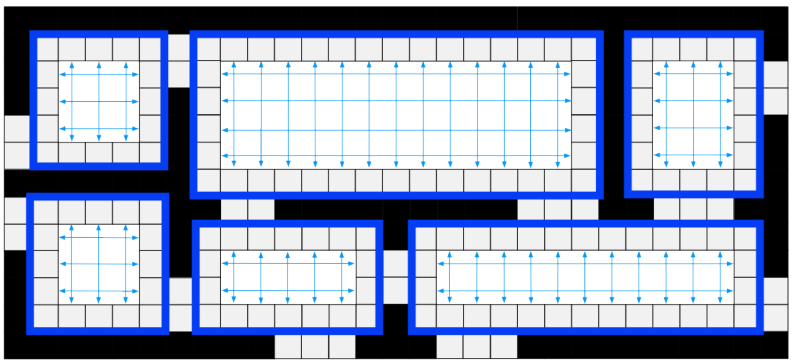
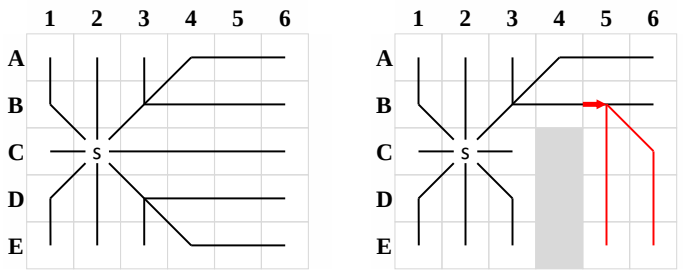


Figure 2.9: Overlay graph of rectangular symmetry reduction on a 4-neighbor grid graph. Vertices of the overlay graph are placed at the cells at the boundaries of the rectangles. The overlay graph also contains edges between adjacent cells, which are not shown. Image downloaded from `https://harablog.wordpress.com/2011/09/01/rectangular-symmetry-reduction/` .

move, and stores each row compactly by representing consecutive entries with the same next move as intervals. During queries, it can determine the first move along an s-t path by performing a binary search over the sequence of intervals stored for row s to find the interval that contains t. SRC identifies an ordering of columns in first-move tables heuristically as the order of vertices expanded by a depth-first search from a random vertex of G. If more than one possible first move exists from s to t, SRC picks the one that helps to minimize the size of the compressed table. An extension of SRC is *multi-row compression* (MRC), which also groups rows that have similar intervals, and stores their shared intervals only once. During queries, it can determine the first move along an s-t path by performing two binary searches, one over the sequence of shared intervals and one over the sequence of individual intervals stored for row s, to find the interval that contains t.

- **Symmetry reduction techniques:** Unlike road networks, grid graphs typically contain multiple shortest paths between two vertices. Algorithms that use symmetry reduction techniques try to avoid exploring all but one of these shortest paths.

  Rectangular symmetry reduction (RSR) (Harabor, Botea, & Kilby, 2011), although not described as such in the literature, uses overlay graphs to reduce symmetries on grid graphs. During preprocessing, RSR decomposes the grid into non-overlapping rectangular areas that do not contain blocked cells. Interior vertices of the rectangles are removed from the graph, and shortcut edges are added between boundary vertices to preserve shortest paths between them. This is equivalent to generating an overlay graph using vertex separators, where the removal of the boundary vertices of rectangles decomposes the grid graph into multiple disjoint (rectangular) components. Figure 2.9 shows an example of RSR's overlay graph on a 4-neighbor grid graph. Observe that not all paths on the grid graph are represented in RSR's overlay graph. For instance, the interiors of rectangles can only be traversed by



(a) Diagonal-first paths on a grid with no blocked cells.

(b) A jump point and its forced neighbors.

Figure 2.10: Operation of jump-point search.

going in a straight line, disallowing turns. During queries, RSR uses the Connect-Search-Refine algorithm, where the connection phases simply connect the start and goal vertices to the boundaries of the rectangles that respectively contain them, and the refinement phases simply replace shortcut edges with corresponding paths that are guaranteed to move in straight lines. That is, the connection and refinement phases of RSR exploit the fact that each rectangle can be considered as "freespace" (since the rectangles do not contain blocked cells). The subgoal graphs that we construct on grid graphs exploit a similar intuition, except that their vertices are placed at the corners of blocked cells and query vertices are connected to them using a different procedure.

Jump-point search (JPS) (Harabor & Grastien, 2011) is an online (as opposed to preprocessing-based) path-planning algorithm on grid graphs. Sturtevant and Rabin break down JPS into three components; namely an A* search, a canonical ordering, and a successor generation (jumping) policy (Sturtevant & Rabin, 2016).

Canonical orderings are (partial) orderings on the actions available to an agent. JPS uses the diagonal-first canonical ordering: A path is said to be diagonal-first if and only if all its 2-edge subpaths are shortest paths and no cardinal-to-diagonal turn on the path can be replaced with a diagonal-to-cardinal turn. JPS explores only diagonal-first paths, using three simple rules. The first two rules are as follows: 1) When expanding a vertex that is reached with a diagonal move, generate only those three successors that can be reached with moves in the same diagonal direction or its two associated cardinal directions. 2) When expanding a vertex that is reached with a cardinal move, generate only the successor that can be reached with a move in the same cardinal direction. Observe that these two rules are sufficient to explore all diagonal-first shortest paths that originate at a vertex on a grid with no blocked cells, as shown in Figure 2.10a. However, on grids with blocked cells, these two rules might fail to reach some of the vertices. Figure 2.10b shows an example where these two rules are insufficient to reach C5 from C2. To address this issue, JPS uses a third rule: 3) Also generate the *forced neighbors* of expanded vertices, that is, successors v of u where the only shortest path from the parent of u to v passes through u. For instance, in Figure 2.10b, C5 is a forced neighbor of B5 because the

only shortest path from the parent B4 of B5 to C5 is ⟨B4, B5, C5⟩. With these three rules, JPS is guaranteed to find shortest paths (Harabor & Grastien, 2011). Sturtevant and Rabin refer to A* search with these three rules as *canonical A* search*, and report that it runs ∼2 times faster than A* search on grid graphs used in the video game Dragon Age: Origins.

JPS extends canonical A* search with a *jumping policy*, which allows JPS to generate and expand only the eponymous *jump points* (plus the start and goal vertices). When the start vertex is expanded, JPS uses the first two rules described above to efficiently scan the grid, but does not use the third rule to generate forced neighbors. Instead, it marks those vertices with forced neighbors as jump points, annotates them with the direction from which they are reached, and inserts them into the OPEN list. When JPS expands a jump point, it performs a similar scan of the grid, using the direction of the jump point to limit the directions of its scans. Figure 2.10b shows an example, where JPS expands C2 by scanning the grid (black lines) and generates (B5, Right) as a jump point successor of C2. When JPS expands (B5, Right), it performs a similar scan (red lines) but only in two directions. Sturtevant and Rabin report that JPS runs ∼1.4 faster than canonical A* search on grid graphs used in the video game Dragon Age: Origins.

The subgoal graphs that we construct on grid graphs share several similarities with JPS. Namely, the vertices of our subgoal graphs are placed at *convex corner cells*, which are also the cells that JPS places jump points at. Furthermore, our connection algorithm explores diagonal-first paths, similar to JPS's scans of the grid. Our connection algorithm uses clearance values to perform these scans efficiently, which have been used in preprocessing-based variants of JPS (Harabor et al., 2014; Traish et al., 2016; Rabin & Sturtevant, 2016). We explore the similarities and differences between subgoal graphs on grid graphs and JPS in more detail in Section 5.4, where we show that JPS can be understood as a search over a *jump-point graph*, which is a subgoal graph constructed on the *direction-extended canonical grid graph*. We also experimentally evaluate algorithms that use subgoal graphs, jump-point graphs, and their combinations with contraction hierarchies in Section 5.5. We also compare these algorithms with several variants of JPS that have been evaluated in the Grid-Based Path-Planning Competition. Among these JPS variants, JPS+BB (Rabin & Sturtevant, 2016) combines JPS with bounding boxes (that is, the geometric containers discussed in Section 2.3.2), and achieves query times that are slightly longer than SRC query times, the shortest query times in the Grid-Based Path-Planning Competition, but uses significantly less memory.

2.4 Conclusions

In this chapter, we have introduced state lattices, grid graphs, and road networks as graphs generated by discretizing configuration spaces of agents, provided an overview of search algorithms that can be used to find paths on graphs, discussed the limitations and applications of preprocessing-based path planning, and discussed related work on road networks and grid graphs. We have not discussed related work on state lattices since, to

the best of our knowledge, no preprocessing-based path-planning algorithm exists that is specialized for state lattices. Our discussion of related work highlighted several algorithms that implicitly or explicitly use overlay graphs, which are also used by the subgoal graph framework that we introduce in the next chapter.

Chapter 3

The Subgoal Graph Framework

In this chapter, we introduce the subgoal graph framework by augmenting overlay graphs with reachability relations, extend this framework to hierarchies of subgoal graphs, called N-level subgoal graphs, and introduce a suboptimal but complete variant, called strongly connected subgoal graphs. We apply this framework to state lattices and grid graphs in Chapters 4 and 5, respectively.

This chapter is organized as follows. In Section 3.1, we motivate augmenting overlay graphs with reachability relations. In Section 3.2, we describe overlay graphs as they appear in the literature, introduce the three-phase Connect-Search-Refine algorithm to answer path queries using overlay graphs, discuss the relationship between overlay graphs and vertex contractions, and introduce extended overlay graphs and heavy contractions. In Section 3.3, we introduce the subgoal graph framework by augmenting overlay graphs with reachability relations, introduce bounded-distance reachability as a reachability relation, and prove that "locally sparse" subgoal graphs can be constructed with respect to bounded-distance reachability on graphs with small highway dimensions, introduce heavy R contractions that can be used for constructing subgoal graphs, and introduce an alternative method for constructing subgoal graphs. In Section 3.4, we introduce N-level overlay and subgoal graphs, discuss their relationship with contraction hierarchies, and introduce R contraction hierarchies by augmenting contraction hierarchies with reachability relations. In Section 3.5, we introduce strongly connected subgoal graphs, that can be used to answer path queries suboptimally, and introduce an algorithm for constructing them. In Section 3.6, we summarize our theoretical results.

3.1 Introduction

Several preprocessing-based path-planning algorithms on road networks and grid graphs use overlay graphs, either explicitly or implicitly, as discussed in Sections 2.3.2 and 2.3.3. An overlay graph of G is a graph G_S that contains only a subset $S \subseteq V$ of the vertices of G. The edges of an overlay graph are the minimum set of edges required to ensure that, for every $s, t \in S$, $d_G(s,t) = d_{G_S}(s,t)$. Overlay graphs can be used to answer s-t path queries optimally by using the three-phase Connect-Search-Refine algorithm: In the connection phase, s and t are connected to the overlay graph to form the query overlay graph. In the search phase, the query overlay graph is searched for a shortest s-t path Π. In the refinement phase, edges of Π are replaced with corresponding shortest paths on G.

The "benefit" of using overlay graphs is that they are typically smaller than G, and thus searches on them are faster. The "cost" of using overlay graphs is that one has to perform connection and refinement. Different algorithms that employ overlay graphs "pay" this cost in different ways. For instance, Schulz et al.'s overlay graphs (Schulz et al., 2000) and portal-based search pay the cost for connection by performing "local searches" during the connection phase. Transit-node routing, on the other hand, pays the cost for connection by precomputing and storing access nodes for every vertex and using a different path-planning algorithm to answer "local queries". Transit-node routing and Schulz et al.'s overlay graphs do not pay the cost for refinement and settle for answering distance queries only, whereas portal-based search pays the cost for refinement by performing A* searches between consecutive vertices on Π. Rectangular symmetry reduction arguably pays a much smaller cost for connection and refinement, by exploiting a rectangular decomposition of the underlying grid graph to perform these procedures more efficiently.

Subgoal graphs are overlay graphs that are constructed with respect to a *reachability relation* R, with the aim of finding a good trade-off between the cost and benefit of using overlay graphs. Intuitively, R is a list of pairs of vertices between which query subgoal (overlay) graphs are allowed to have edges. Although this requirement constrains how subgoal graphs can be constructed, and thus may result in longer search times, it can also result in shorter connection and refinement times since the connection phases of queries answered using subgoal graphs can operate under the assumption that they only need to identify edges that satisfy R, and the refinement phases can operate under the assumption that they only need to find shortest paths between vertices that satisfy R.

Schulz et al.'s overlay graphs, portal-based search, and rectangular symmetry reduction exploit this idea to various degrees, by constraining the vertices of their overlay graphs to be vertex separators of G. The vertex separators define disjoint connected components in G and, during connection, only the connected components that contain the start and goal vertices are considered. Rectangular symmetry reduction further requires that each connected component corresponds to a rectangular area of unblocked cells in the underlying grid, avoiding search altogether during connection. Subgoal graphs can be considered to be a generalization of these ideas into a framework that can be specialized to different types of graphs by choosing a reachability relation. For instance, Schulz et al.'s overlay graphs, portal-based search, and rectangular symmetry reduction can be considered to be instantiations of this framework, where the reachability relation distinguishes those pairs of vertices that belong to the same connected component with respect to some vertex separator of G.

The subgoal graph framework provides us with the means to exploit the freespace structure of grid graphs and state lattices, by capturing it with a reachability relation and exploiting it with efficient connection and refinement algorithms, as we discuss in Chapters 4 and 5.

3.2 Overlay Graphs

In this section, we discuss overlay and extended overlay graphs, how they can be constructed, and how they can be used to answer path queries optimally.

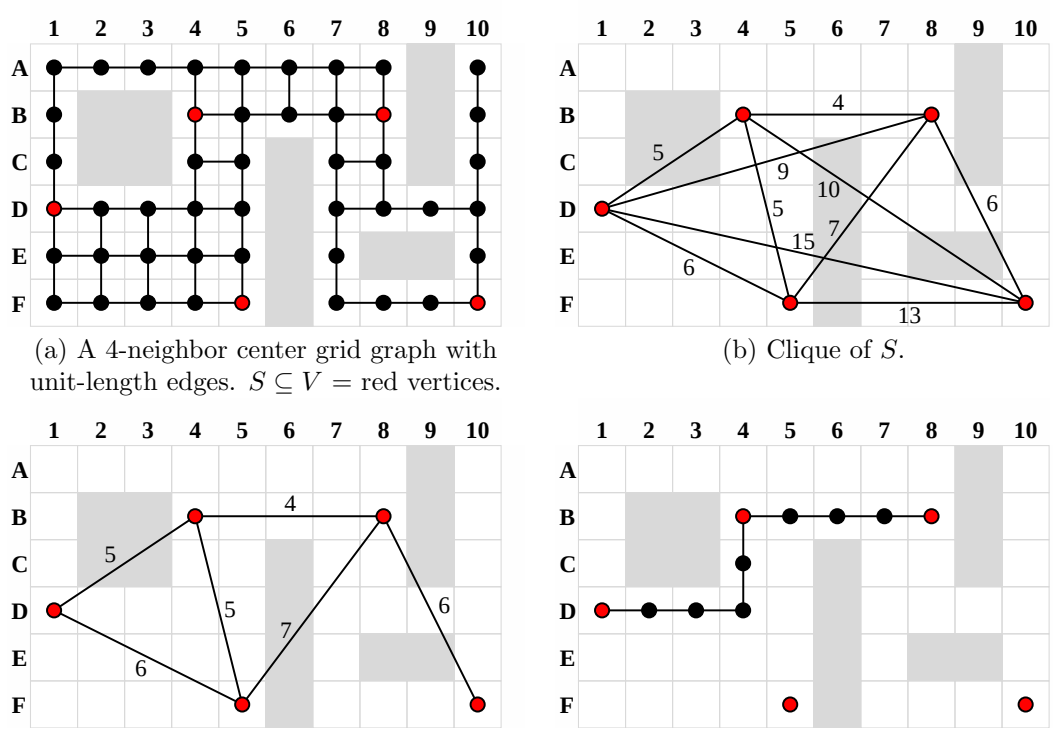
This section is organized as follows. In Sections 3.2.1 and 3.2.2, we establish concepts such as *covering*, *direct-reachability*, *overlay graphs*, and *extended overlay graphs*, which serve as a theoretical foundation for several data structures and algorithms that we discuss in this dissertation. Our definition of overlay graphs is equivalent to the definition by Holzer et al., and our definition of extended overlay graphs is inspired by, but slightly differ from, their definition of extended multi-level overlay graphs (Holzer et al., 2009). In Section 3.2.3, we prove the *optimality* and *minimality* of overlay and extended overlay graphs. In Section 3.2.4, we introduce the three-phase Connect-Search-Refine algorithm to answer path queries optimally using overlay graphs, and discuss how it differs from the approach used by Holzer et al.. In Section 3.2.5, we describe the Overlay-Connect algorithm as it appears in the literature (Holzer et al., 2009), and discuss its variants. The Overlay-Connect algorithm can be used for constructing the edges of overlay graphs or for connecting the start and goal vertices to the overlay graph when answering path queries. In Section 3.2.6, we describe "vertex contractions", which appear in the literature (Geisberger et al., 2008) and are used for constructing contraction hierarchies, and show that they can be used to construct overlay graphs as well. We also introduce a variant, called "heavy contractions", for constructing extended overlay graphs.

3.2.1 Overlay Graphs

An overlay graph G_S is a graph that preserves the distances between a set of vertices $S \subseteq V$ (that is, for every $s, t \in S$, $d_{G_S}(s,t) = d_G(s,t)$), using the minimum set of shortcut edges (that is, edges that do not necessarily appear in G). Consider the example in Figure 3.1[1]: Figure 3.1a shows $S \subseteq V$ in red, and Figure 3.1b shows a clique of S, where the length of each edge (u, v) in the clique is equal to the u-v distance on G. This clique preserves the distances between vertices in S but has *redundant* edges, that is, edges whose removal from the clique does not change the distances on the clique. For instance, the edge (D1, F10) has the same length as the path $\langle$D1, B4, B8, F10$\rangle$. Therefore, removing the edge from the clique does not change the D1-F10 distance on the clique. Removing all redundant edges from the clique results in an overlay graph, which has no redundant edges, as shown in Figure 3.1(c).

The (non-redundant) edges of overlay graphs can be characterized as *direct-reachable*: A pair of vertices (or, equivalently, an edge) $(s, t) \in V \times V$ is direct-reachable with respect to $S \subseteq V$ (equivalently, t is direct-reachable from s with respect to S) if and only if no shortest s-t path on G passes through an intermediate vertex $n \in S$. Every edge (s, t) in the clique of S that is not direct-reachable is redundant, since, if a shortest s-t path on G passes through an intermediate vertex $n \in S$, then the path $\langle s, n, t\rangle$ is an alternate shortest s-t path in the clique. Figure 3.1d shows an example where B8 is not direct-reachable from D1 (with respect to S) since a shortest D1-B8 path on G passes through B4 $\in S$. Therefore, (B8, D1) is redundant and thus excluded from the overlay graph of S.

[1]Although the techniques we discuss in this chapter are applicable to any directed graph, we use four-neighbor grid graphs as G in our examples since they are easy to visualize. The length of any edge (u, v) is equal to the u-v distance on G.



(a) A 4-neighbor center grid graph with unit-length edges. $S \subseteq V$ = red vertices.

(b) Clique of S.

(c) The overlay graph G_S preserves the distances between vertices in S using the minimum set of shortcut edges.

(d) (D1, B8) is not included in G_S because B4 $\in S$ covers a shortest D1-B8 path on G.

Figure 3.1: An overlay graph constructed on a 4-neighbor grid graph (G). The length of each shortcut edge (u, v) is equal the u-v distance on G.

In the remainder of this section, we formally define direct-reachability and overlay graphs, using the concept of *covering*: A vertex n covers an s-t path π if and only if n is a vertex of π with $n \notin \{s, t\}$ (Holzer et al., 2009). Definition 3.1 extends the definition of "covering" by Holzer et al..

Definition 3.1 (Covering a path/pair of vertices)**.** *A vertex n* covers *a path $\pi = \langle p_0, \ldots, p_k \rangle$ if and only if $n \in \{p_1, \ldots, p_{k-1}\}$. A vertex n covers a pair of vertices $(s, t) \in V \times V$ if and only if n covers at least one shortest s-t path on G.*

A set of vertices $S \subseteq V$ covers π if and only if, there exists $n \in S$ such that n covers π. A set of vertices $S \subseteq V$ covers (s, t) if and only if, there exists $n \in S$ such that n covers (s, t).

"Covering" is a fundamental concept in this dissertation which we frequently use in our theorems. We thus introduce the symbol "$\sqsubset$" to denote that a vertex or set of vertices covers a path or a pair of vertices. Table 3.1 summarizes our notation for "covering" and highlights some of the equivalences in our definitions. Namely, Definition 3.2 defines direct-reachability in terms of covering, and Lemma 3.1 proves that $n \sqsubset (s, t)$ if and only if $d(s, t) = d(s, n) + d(n, t)$ and $n \notin \{s, t\}$.

Notation	Meaning	Equivalent meaning
$n \sqsubset \pi$	n covers π	$n \in \pi$ and $n \notin \{s,t\}$, where π is an s-t path
$n \sqsubset (s,t)$	$n \sqsubset$ a shortest s-t path on G	$d(s,t) = d(s,n) + d(n,t)$ and $n \notin \{s,t\}$
$S \sqsubset \pi$	$\exists n \in S$ such that $n \sqsubset \pi$	
$S \sqsubset (s,t)$	$\exists n \in S$ such that $n \sqsubset (s,t)$	
$n \not\sqsubset \pi$	n does not cover π	
$n \not\sqsubset (s,t)$	n does not cover (s,t)	
$S \not\sqsubset \pi$	$\forall n \in S$, $n \not\sqsubset \pi$	
$S \not\sqsubset (s,t)$	$\forall n \in S$, $n \not\sqsubset (s,t)$	$(s,t) \in D_S^{V \Rightarrow V}$

Table 3.1: Covering notation.

Definition 3.2 (Direct-reachability)**.** *Let $S \subseteq V$. A pair of vertices (or an edge) (s,t) is* direct-reachable *with respect to S (equivalently, t is direct-reachable from s with respect to S) if and only if $S \not\sqsubset (s,t)$.*

We denote the set of all pairs of vertices in $V \times V$ that are direct-reachable with respect to S as the set D_S. Let $A \subseteq V$ and $B \subseteq V$. We denote the set of all pairs of vertices in $A \times B$ that are direct-reachable with respect to S as the set $D_S^{A \Rightarrow B} \subseteq A \times B$. We denote the set of all pairs of vertices in $A \times B$ that are direct-reachable with respect to S, except for pairs of the form (a,a), as the set $D_S^{A \to B} \subseteq A \times B$. That is:
$D_S = D_S^{V \Rightarrow V} = \{(a,b) \in V \times V : S \not\sqsubset (a,b)\}$,
$D_S^{A \Rightarrow B} = \{(a,b) \in A \times B : S \not\sqsubset (a,b)\}$, *and*
$D_S^{A \to B} = \{(a,b) \in A \times B : S \not\sqsubset (a,b)$ *and* $a \neq b\}$.

Definition 3.3 formally defines an overlay graph G_S *induced* by a set of vertices $S \subseteq V$, and is equivalent to the definition of an overlay graph by Holzer et al. (Holzer et al., 2009).

Definition 3.3 (Overlay graph)**.** *The overlay graph induced by $S \subseteq V$ on G is the graph $G_S = (S, E', c')$, where:*

1. $E' = D_S^{S \to S} = \{(u,v) \in S \times S : S \not\sqsubset (u,v)$ *and* $u \neq v\}$
2. $\forall (u,v) \in E'$, $c'(u,v) = d_G(u,v)$

Observe that, by Assumption 1.3, G does not contain any redundant edges and can therefore be considered as the overlay graph $G_V = G$ induced by the set of vertices V.

We now prove that, for every $n, s, t \in V$, $n \sqsubset (s,t)$ if and only if $d(s,t) = d(s,n) + d(n,t)$ and $n \notin \{s,t\}$, which we use in Section 3.2.3 to prove the *minimality* and *optimality* of overlay graphs.

Lemma 3.1. *For every $n, s, t \in V$, $n \sqsubset (s,t)$ if and only if $d(s,t) = d(s,n) + d(n,t)$ and $n \notin \{s,t\}$.*

Proof.

1. If $n \sqsubset (s,t)$, then $d(s,t) = d(s,n) + d(n,t)$ and $n \notin \{s,t\}$:
   1.1. Assume $n \sqsubset (s,t)$.

1.2. There exists $\pi = \langle p_0, \ldots p_k \rangle$ such that π is a shortest s-t path on G and $n \sqsubset \pi$ (Definition 3.1, since $n \sqsubset (s,t)$).

1.3. $n = p_i$ for some $0 < i < k$ (Definition 3.1, since $n \sqsubset \pi$).

1.4. Let $\pi_1 = \langle p_0, \ldots, p_i \rangle$, $\pi_2 = \langle p_i, \ldots, p_k \rangle$ with $\pi = \pi_1 \cdot \pi_2$ and $l(\pi) = l(\pi_1) + l(\pi_2)$.

1.5. $l(\pi) = d(s,t)$ (since π is a shortest s-t path).

1.6. $l(\pi_1) = d(s,n)$ (otherwise, π is not a shortest path).

1.7. $l(\pi_2) = d(n,t)$ (otherwise, π is not a shortest path).

1.8. $d(s,t) = d(s,n) + d(n,t)$ (since $l(\pi) = l(\pi_1) + l(\pi_2)$).

1.9. $n \notin \{s,t\}$ (Definition 3.1, since $n \sqsubset \pi$).

2. If $d(s,t) = d(s,n) + d(n,t)$ and $n \notin \{s,t\}$, then $n \sqsubset (s,t)$:

2.1. Assume $d(s,t) = d(s,n) + d(n,t)$.

2.2. Assume $n \notin \{s,t\}$.

2.3. Let π_1 be a shortest s-n path on G with $l(\pi_1) = d(s,n)$.

2.4. Let π_2 be a shortest n-t path on G with $l(\pi_2) = d(n,t)$.

2.5. Let $\pi = \pi_1 \cdot \pi_2$ with $l(\pi) = l(\pi_1) + l(\pi_2)$.

2.6. $l(\pi) = l(\pi_1) + l(\pi_2) = d(s,n) + d(n,t) = d(s,t)$.

2.7. π is a shortest s-t path on G (since $l(\pi) = d(s,t)$).

2.8. $n \sqsubset \pi$ (Definition 3.1, since $n \in \pi$, $n \neq s$, and $n \neq t$).

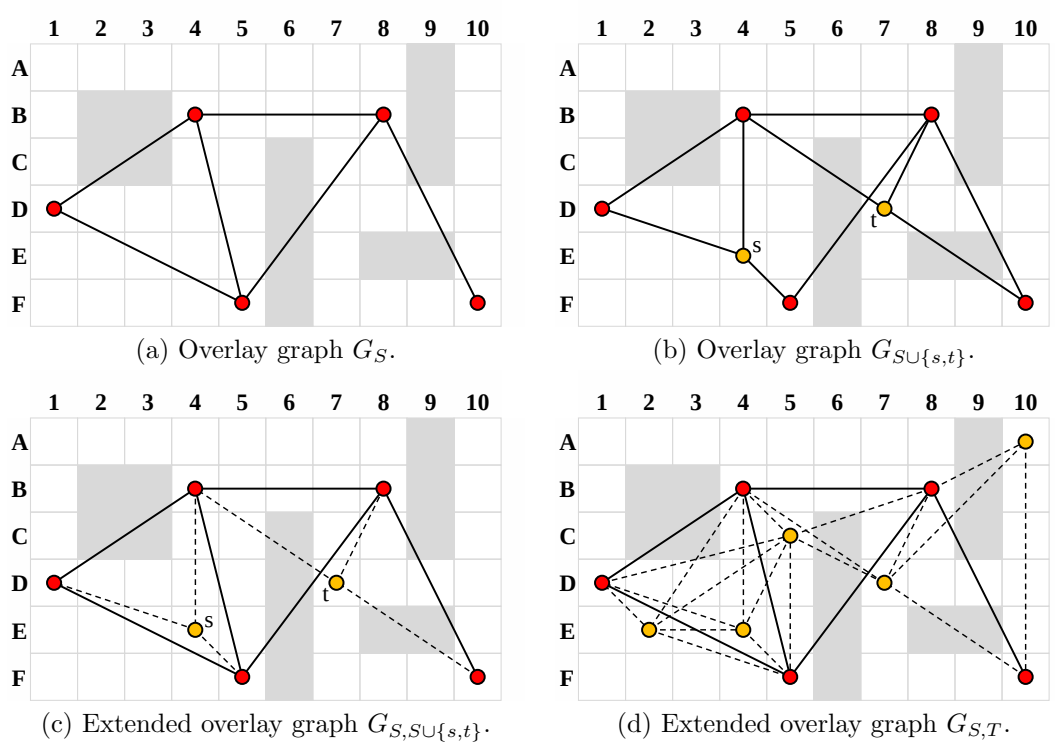
2.9. $n \sqsubset (s,t)$ (Definition 3.1, since $n \sqsubset \pi$ and π is a shortest s-t path on G).

□

3.2.2 Extended Overlay Graphs

Extended overlay graphs can be considered to be overlay graphs that are "extended" with additional vertices. These "extensions" add the necessary set of edges to preserve distances between the "extended" set of vertices, but never remove any of the existing edges. Therefore, extended overlay graphs may contain redundant edges. However, as we prove in Section 3.2.3, extended overlay graphs have the minimum set of edges necessary to preserve *overlay distances* between their vertices. We use extended overlay graphs for three different purposes in this dissertation: (1) When connecting start and goal vertices to overlay graphs during queries, we only add new edges to the overlay graph but never remove existing edges. Therefore, *query* overlay graphs can be considered to be extended overlay graphs (Section 3.2.4). (2) We use extended overlay graphs to prove that all possible extensions of subgoal graphs into query subgoal graphs use only edges that satisfy a reachability relation (Section 3.3.2). (3) We use extended overlay graphs as building blocks for constructing N-level overlay (and subgoal) graphs (Sections 3.4.1).

More formally, for every $S \subseteq T \subseteq V$, the extended overlay graph $G_{S,T}$ (G_S extended with $T \setminus S$) is a graph that contains all edges that can appear in an overlay graph G_U, for every U with $S \subseteq U \subseteq T$. Whereas the set of edges of an overlay graph G_S is



(a) Overlay graph G_S.

(b) Overlay graph $G_{S\cup\{s,t\}}$.

(c) Extended overlay graph $G_{S,S\cup\{s,t\}}$.

(d) Extended overlay graph $G_{S,T}$.

Figure 3.2: Overlay and extended overlay graphs on a 4-neighbor grid graph. The length of each shortcut edge (u, v) is equal the u-v distance on G.

characterized by the relation $D_S^{S\to S}$, the set of edges of an extended overlay graph $G_{S,T}$ is characterized by the relation $D_S^{T\to T}$. Therefore, for every $S \subseteq V$, $G_{S,S} = G_S$, and $G_{V,V} = G_V = G$. Definition 3.4 formally defines extended overlay graphs.

Definition 3.4 (Extended overlay graph)**.** *The extended overlay graph induced by $S \subseteq T \subseteq V$ on G is the graph $G_S = (T, E', c')$, where:*

1. $E' = D_S^{T\to T} = \{(u, v) \in T \times T : u \neq v \text{ and } S \not\sqsubset (u, v)\}$ *and*
2. $\forall (u, v) \in E'$, $c'(u, v) = d(u, v)$.

Figure 3.2 shows an example. When extending the overlay graph G_S (Figure 3.2a) with the vertices s and t, edges that appear in $G_{S\cup\{s,t\}}$ (or in $G_{S\cup\{s\}}$ or $G_{S\cup\{t\}}$) but not in G_S are added to G_S, resulting in the extended overlay graph $G_{S,S\cup\{s,t\}}$ (Figures 3.2b and Figures 3.2c). Figure 3.2d shows an example where $G_{S,S\cup\{s,t\}}$ is extended with three additional vertices. None of the extensions remove any existing edges.

By definition, $G_{S,T}$ contains a superset of the edges of G_T (that is, $D_S^{T\to T} \supseteq D_T^{T\to T}$ since $T \supseteq S$). Therefore, similar to G_T, $G_{S,T}$ preserves distances between the vertices in T, but does not necessarily have the minimum set of edges to do so. However, as we prove in the next section, $G_{S,T}$ has the minimum set of edges necessary to guarantee that, for every $s, t \in T$, at least one shortest s-t path π on $G_{S,T}$ is an *overlay path*, that is, a path that only uses vertices in $S \cup \{s, t\}$. Definition 3.5 formally defines overlay paths and overlay distances on extended overlay graphs.

Definition 3.5 (Overlay path/distance)**.** *An s-t path $\pi = \langle p_0, \ldots, p_n \rangle$ on $G_{S,T}$ is an overlay s-t path on $G_{S,T}$ if and only if $p_1, \ldots, p_{n-1} \in S$. The s-t overlay distance $d^{\mathcal{O}}_{G_{S,T}}(s,t)$ on $G_{S,T}$ is the length of a shortest overlay s-t path on $G_{S,T}$.*

As we prove in the next section, for every $s, t \in T$, $d^{\mathcal{O}}_{G_{S,T}}(s,t) = d_{G_{S,T}}(s,t) = d(s,t)$. Therefore, we use the notation "$d^{\mathcal{O}}$" only in the next section for proving this theorem, and simply use the notation "d" to represent (overlay) distances on (extended) overlay graphs throughout the dissertation.

Our definition of extended overlay graphs is influenced by the definition of *extended multi-level overlay graphs* by Holzer et al. (Holzer et al., 2009), and can be considered to be a slight variation of their "extended 2-level overlay graphs". Namely, in our definition, we allow an extended overlay graph $G_{S,T}$ to have edges between two vertices in T. Their definition of "extended 2-level overlay graphs" excludes such edges unless they are edges that already appear in G. As a result, N-level overlay (and subgoal) graphs, which we define as hierarchies of extended overlay graphs in Section 3.4.1, are different from extended multi-level overlay graphs, have different theoretical properties, and are searched using different search algorithms. A detailed discussion of these differences is beyond the scope of this dissertation since we do not consider N-level overlay (and subgoal) graphs as a main contribution of this dissertation (as we discuss in Section 5.5.11, there are other hierarchies that offer better query-time/memory trade-offs than N-level subgoal graphs).

3.2.3 Optimality and Minimality of Extended Overlay Graphs

We now prove the *optimality* and *minimality* of extended overlay graphs. The optimality of extended overlay graphs means that, for every $G_{S,T}$ and $s, t \in T$, $d^{\mathcal{O}}_{G_{S,T}}(s,t) = d_{G_{S,T}}(s,t) = d(s,t)$. The minimality of extended overlay graphs means that extended overlay graphs have the minimum set of edges necessary to guarantee their optimality. By the equivalence $G_{S,S} = G_S$, and the fact that every path on $G_{S,S}$ is an overlay path, these results extend to overlay graphs as well.

We first start with a useful lemma that proves that, for every $s, t \in T$, an overlay s-t path π exists on $G_{S,T}$ with $l(\pi) = d_G(s,t)$.

Lemma 3.2. *For every $s, t \in T$, there exists an overlay s-t path π on $G_{S,T}$ with $l(\pi) = d(s,t)$.*

Proof. We prove the lemma assuming that $s \neq t$. Otherwise, $\pi = \langle s \rangle$ trivially satisfies the lemma.
Let P = GenerateShortestOverlayPath be a procedure that is defined as follows:

```
1: π ← ⟨s, t⟩
2: while There exist (u, v) ∈ π and n ∈ S such that n ⊏ (u, v) do
3:     Replace (u, v) in π with ⟨u, n, v⟩
4: return π
```

We prove that P is guaranteed to return an overlay s-t path on $G_{S,T}$ with length $d(s,t)$:

1. Let $k \in \mathbb{N}$ be the number of times that line 2 is executed.
2. Let π_i be the value of π the ith time that line 2 is executed.

3. For every $(u, v) \in \pi_i$, $u \neq v$ (proof by induction on $i = 1, \ldots, k$):
   3.1. (base case) $\pi_1 = \langle s, t \rangle$ and $s \neq t$.
   3.2. (induction step) Each time line 2 is executed, $n \notin \{u, v\}$ (Definition 3.1, since $n \sqsubset (u, v)$).
4. $L(\pi_i) = \sum_{(u,v)\in\pi} d(u, v) = d(s, t)$ (proof by induction on $i = 1, \ldots, k$):
   4.1. (base case) $L(\pi_1) = d(s, t)$ (since $\pi_1 = \langle s, t \rangle$).
   4.2. (induction step) $L(\pi_i) = L(\pi_{i-1}) = d(s, t)$:
      4.2.1. $L(\pi_{i-1}) = d(s, t)$ (induction hypothesis).
      4.2.2. $d(u, v) = d(u, n) + d(n, v)$ (Lemma 3.1, since $n \sqsubset (u, v)$).
      4.2.3. $L(\pi_i) = L(\pi_{i-1}) - d(u, v) + d(u, n) + d(n, v) = L(\pi_{i-1})$.
5. P terminates ($k < \infty$):
   5.1. Let $\epsilon > 0$ be the minimum edge length in G.
   5.2. $\forall i = 1, \ldots, k$, $i \leq d(s, t)/\epsilon$:
      5.2.1. π_i has $i + 1$ edges (since π_1 has 1 and each iteration adds one more edge).
      5.2.2. $\forall (u, v) \in \pi$, $d(u, v) \geq \epsilon$ (since $u \neq v$).
      5.2.3. $L(\pi_i) \geq i\epsilon$ (since π_i has i edges).
      5.2.4. $d(s, t) \geq i\epsilon$ (since $L(\pi_i) = d(s, t)$).
      5.2.5. $d(s, t)/\epsilon \geq i$.
   5.3. k is bounded (since, $\forall i = 1, \ldots, k$, i is bounded).
6. π_k is an overlay s-t path with length $d(s, t)$:
   6.1. $\forall (u, v) \in \pi_k$, $S \not\sqsubset (u, v)$ (Definition 3.1, since the condition on line 2 of GenerateShortestOverlayPath fails for π_k).
   6.2. $\forall (u, v) \in \pi_k$, $(u, v) \in E'$ (Definition 3.4, since $S \not\sqsubset (u, v)$).
   6.3. Therefore, π_k is an s-t path on $G_{S,T}$.
   6.4. π_k is an overlay path (Definition 3.5, since every intermediate vertex $n \in S$).
   6.5. $\forall (u, v) \in E'$, $c'(u, v) = d(u, v)$ (Definition 3.4).
   6.6. $l(\pi) = \sum_{(u,v)\in\pi} c'(u, v) = \sum_{(u,v)\in\pi} d(u, v) = L(\pi) = d(s, t)$.

□

Theorem 3.3 (Optimality)**.** *For every $s, t \in T$, $d^{\mathcal{O}}_{G_{S,T}}(s, t) = d_{G_{S,T}}(s, t) = d(s, t)$.*

Proof.

1. Let $G_{S,T} = (T, E', c')$.
2. Let $s, t \in T$.

3. $d^{\mathcal{O}}_{G_{S,T}}(s,t) \geq d(s,t)$ and $d_{G_{S,T}}(s,t) \geq d(s,t)$ (since, by Definition 3.4, $\forall (u,v) \in E'$, $c'(u,v) = d(u,v)$).

4. $d^{\mathcal{O}}_{G_{S,T}}(s,t) \leq d(s,t)$ and $d_{G_{S,T}}(s,t) \leq d(s,t)$ (since, by Lemma 3.2, there exists an overlay s-t path π on $G_{S,T}$ with $l(\pi) = d(s,t)$).

□

To prove the minimality of extended overlay graphs, we first prove two useful lemmata. Lemma 3.4 extends Definition 3.1 to extended overlay graphs. Namely, Definition 3.1 states that, for every $n, s, t \in V$, $n \sqsubset (s,t)$ if and only if there exists a shortest s-t path π on G such that $n \sqsubset \pi$. Lemma 3.4 proves that, for every $n \in S$ and $s, t \in T$, $n \sqsubset (s,t)$ if and only if there exists a shortest overlay s-t path π on $G_{S,T}$ with $n \sqsubset \pi$. Lemma 3.5 proves that edges of extended overlay graphs are unique shortest overlay paths.

Lemma 3.4. *For every $n \in S$ and $s, t \in T$, $n \sqsubset (s,t)$ if and only if there is a shortest overlay s-t path π on $G_{S,T}$ with $n \sqsubset \pi$.*

Proof.

1. Let $n \in S$.

2. Let $s, t \in T$.

3. If $n \sqsubset (s,t)$, then there exists a shortest overlay s-t path π on $G_{S,T}$ with $n \sqsubset \pi$:

   3.1. Assume $n \sqsubset (s,t)$.

   3.2. Let π_1 be a shortest overlay s-n path on $G_{S,T}$ with $l(\pi_1) = d(s,n)$. Such π_1 exists (Lemma 3.2, since $s, n \in T$).

   3.3. Let π_2 be a shortest overlay n-t path on $G_{S,T}$ with $l(\pi_2) = d(n,t)$. Such π_2 exists (Lemma 3.2, since $n, t \in T$).

   3.4. Let $\pi = \pi_1 \cdot \pi_2$ with $l(\pi) = l(\pi_1) + l(\pi_2) = d(s,n) + d(n,t)$.

   3.5. π is an overlay path on $G_{S,T}$ (Definition 3.5, since π_1 is an overlay s-n path, π_2 is an overlay n-t path, $\pi = \pi_1 \cdot \pi_2$, and $n \in S$).

   3.6. $l(\pi) = d(s,n) + d(n,t) = d(s,t)$ (Lemma 3.1, since $n \sqsubset (s,t)$).

   3.7. π is a shortest s-t path on $G_{S,T}$ (since $l(\pi) = d(s,t)$).

   3.8. $n \notin \{s,t\}$ (Lemma 3.1, since $n \sqsubset (s,t)$).

   3.9. $n \sqsubset \pi$. (Definition 3.1, since $n \notin \{s,t\}$ and $n \in \pi$).

4. If there exists a shortest overlay s-t path π on $G_{S,T}$ with $n \sqsubset \pi$, then $n \sqsubset (s,t)$:

   4.1. Assume $\pi = \langle p_0, \ldots p_k \rangle$ is a shortest overlay s-t path on $G_{S,T}$.

   4.2. Assume $n \sqsubset \pi$.

   4.3. $n = p_i$ for some $0 < i < k$ (Definition 3.1, since $n \sqsubset \pi$).

   4.4. Let $\pi_1 = \langle p_0, \ldots, p_i \rangle$, $\pi_2 = \langle p_i, \ldots, p_k \rangle$ with $\pi = \pi_1 \cdot \pi_2$ and $l(\pi) = l(\pi_1) + l(\pi_2)$.

   4.5. $l(\pi) = d(s,t)$ (Theorem 3.3, since π is a shortest s-t path on $G_{S,T}$).

4.6. $l(\pi_1) = d(s, n)$ (otherwise, π is not a shortest path on $G_{S,T}$).

4.7. $l(\pi_2) = d(n, t)$ (otherwise, π is not a shortest path on $G_{S,T}$).

4.8. $d(s, t) = d(s, n) + d(n, t)$.

4.9. $n \notin \{s, t\}$ (Definition 3.1, since $n \sqsubset \pi$, which is an s-t path).

4.10. $n \sqsubset (s, t)$ (Lemma 3.1, since $d(s, t) = d(s, n) + d(n, t)$ and $n \notin \{s, t\}$).

□

Lemma 3.5. *Every edge (u, v) of $G_{S,T}$ is the unique shortest overlay u-v path on $G_{S,T}$.*

Proof.

1. Let $G_{S,T} = (T, E', c')$.
2. Let $(u, v) \in E'$.
3. $\langle u, v\rangle$ is a shortest overlay u-v path on $G_{S,T}$:
   - 3.1. $l(\langle u, v\rangle) = c'(u, v) = d(u, v)$ (Definition 3.4, since $(u, v) \in E$).
   - 3.2. $\langle u, v\rangle$ is an overlay path (Definition 3.5, since $\langle u, v\rangle$ has no internal vertices).
   - 3.3. $\langle u, v\rangle$ is a shortest path (Theorem 3.3, since $l(\langle u, v\rangle) = d(u, v)$).
4. If π is a shortest overlay u-v path on $G_{S,T}$, then $\pi = \langle u, v\rangle$:
   - 4.1. Assume $\pi = \langle p_0, \ldots, p_k\rangle$ is a shortest overlay u-v path on $G_{S,T}$.
   - 4.2. Assume (for contradiction) $\pi \neq \langle u, v\rangle$.
   - 4.3. $u \neq v$ (Definition 3.4, since $(u, v) \in E'$).
   - 4.4. $k \geq 2$ (since $u \neq v$ and $\pi \neq \langle u, v\rangle$).
   - 4.5. $p_1 \notin \{u, v\}$ (Since $u = p_0$ and $v = p_k$ with $k \geq 2$).
   - 4.6. $p_1 \in S$ (Definition 3.5, since π is an overlay path).
   - 4.7. $p_1 \sqsubset \pi$ (Definition 3.1, since $p_1 \in \pi$ and $p_1 \notin \{u, v\}$).
   - 4.8. $p_1 \sqsubset (u, v)$ (Lemma 3.4, since $p_1 \sqsubset \pi$ and π is a shortest overlay u-v path).
   - 4.9. $(u, v) \notin E'$ (Definition 3.4, since $p_1 \sqsubset (u, v)$ and $p_1 \in S$).
   - 4.10. $\perp$ ($(u, v) \in E'$ and $(u, v) \notin E'$).

□

Theorem 3.6. *An extended overlay graph $G_{S,T}$ has the minimum set of edges to ensure that, for every $s, t \in T$, an overlay s-t path π with $l(\pi) = d(s, t)$ exists on $G_{S,T}$.*

Proof. Since every edge (u, v) is the unique shortest overlay u-v path on $G_{S,T}$ (Lemma 3.5), removing (u, v) increases the length of a shortest overlay u-v path on $G_{S,T}$. □

3.2.4 Answering Queries Using Overlay Graphs

As we have shown in the previous section, for every $s, t \in V$ and $S \subseteq V$, an s-t path π exists on the extended overlay graph $G_{S,S\cup\{s,t\}}$ (Theorem 3.3). In order to find an s-t path on G using the overlay graph G_S, it is thus sufficient to extend G_S to $G_{S,S\cup\{s,t\}}$, find a shortest s-t path π on $G_{S,S\cup\{s,t\}}$, and then replace the edges of π with corresponding shortest paths on G. Every edge of the form (u, s) or (t, u) can never appear on a shortest s-t path on $G_{S,S\cup\{s,t\}}$. Therefore, we can instead extend G_S into an *s-t query overlay graph*. More formally, the edges of $G_{S,S\cup\{s,t\}}$ are characterized by the relation:

$$D_S^{S\cup\{s,t\}\to S\cup\{s,t\}} = D_S^{S\to S} \cup D_S^{s\to S\cup\{t\}} \cup D_S^{S\cup\{t\}\to s} \cup D_S^{t\to S\cup\{s\}} \cup D_S^{S\cup\{s\}\to t}.$$

Since the edges in $D_S^{S\cup\{t\}\to s} \cup D_S^{t\to S\cup\{s\}}$ can never appear on a shortest s-t path, they can be left out from the s-t query overlay graph, whose edges can then be characterized by the relation:

$$D_S^{S\to S} \cup D_S^{s\to S\cup\{t\}} \cup D_S^{S\cup\{s\}\to t}.$$

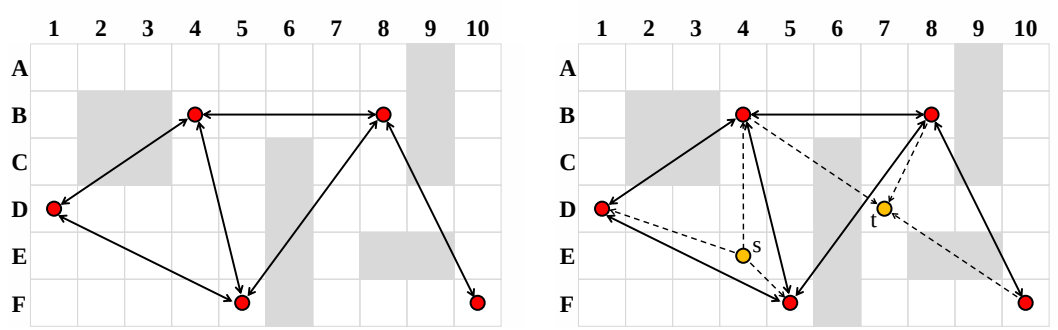
Definition 3.6 formally defines the s-t query overlay graph $G_S^{s,t}$.

Definition 3.6 (s-t query overlay graph)**.** *Let $G_S = (S, E', c')$ be an overlay graph. The s-t query overlay graph is the graph $G_S^{s,t} = (S \cup \{s, t\}, E'', c'')$ such that:*

1. $E^{\to} = D_S^{s\to S\cup\{t\}} = \{(s, u) : u \in S \cup \{t\},\ S \not\sqsubset (s, u),\ \text{and } u \neq s\}$;
2. $E^{\leftarrow} = D_S^{S\cup\{s\}\to t} = \{(u, t) : u \in S \cup \{s\},\ S \not\sqsubset (u, t),\ \text{and } u \neq t\}$;
3. $E'' = E' \cup E^{\to} \cup E^{\leftarrow}$; *and*
4. $\forall (u, v) \in E'',\ c''(u, v) = d(s, t)$.

In this dissertation, we consider a three-phase approach to answering s-t path queries using overlay graphs, called the Connect-Search-Refine algorithm. We describe its three phases below, but formally define it in Section 3.3.4 in the context of subgoal graphs. Our description assumes that we are using an overlay graph G_S. Figure 3.3 shows an example of the operation of the Connect-Search-Refine algorithm.

- **Connection phase:** In the connection phase, G_S is extended to the s-t query overlay graph $G_S^{s,t}$. The connection phase can be implemented as two searches on G, one in the forward direction from s to identify $E^{\to} = D_S^{s\to S\cup\{t\}}$ and one in the backward direction from t to identify $E^{\leftarrow} = D_S^{S\cup\{s\}\to t}$. These searches should correctly determine the lengths of edges in $E^{\to}$ and $E^{\leftarrow}$. We discuss how they can be performed in the next section and discuss variants that are allowed to identify a superset of the required edges for connection. An alternative to performing these searches during the connection phases of queries is performing them during the preprocessing phase and caching the results, essentially storing $G_{S,V}$ rather than G_S. However, this might incur a memory overhead that is impractical for most real-world applications. Transit-node routing, for instance, uses a similar strategy but avoids storing edges between vertices in $V \setminus S$ and uses local queries to identify



(a) Overlay graph G_S. (Edges of G_S are undirected since G is undirected.)

(b) Connection phase: Extend G_S to the s-t query overlay graph $G_S^{s,t}$.

(c) Search phase: Find a shortest s-t path π on $G_S^{s,t}$.

(d) Refinement phase: Replace edges of π with corresponding shortest paths on G.

Figure 3.3: Answering queries using the Connect-Search-Refine algorithm and overlay graphs.

them (Antsfeld, Harabor, Kilby, Walsh, et al., 2012). Speeding up the connection phase without an impractical memory overhead is one of the main motivations for augmenting overlay graphs with reachability relations. We discuss connection algorithms for our subgoal graphs that are specialized to exploit structure in state lattices and grids in Chapters 4 and 5, respectively.

- **Search phase:** In the search phase, $G_S^{s,t}$ is searched for a shortest s-t path π, typically by using an A* search or a bidirectional Dijkstra search. By Theorem 3.3, $l(\pi) = d(s,t)$. This result holds even if the connection phase is allowed to identify supersets of $E^{\rightarrow}$ and $E^{\leftarrow}$, as long as the length of each redundant edge (u,v) is greater than or equal to $d(u,v)$. The search phase can be sped up by further preprocessing the overlay graph, for instance, by using an N-level overlay graph or a contraction hierarchy (Section 3.4).

- **Refinement phase:** In the refinement phase, each edge $(u,v) \in \pi$ is replaced with a shortest u-v path on G. This can be achieved by performing an A* search on G for each such edge, or by performing these searches during preprocessing and caching the results. To the best of our knowledge, none of the preprocessing-based path-planning algorithms that use overlay graphs perform such caching, and instead

use A* searches (portal-based search), use a special refinement algorithm (rectangular symmetry reduction), or skip the refinement phase and settle for answering distance queries only (transit node routing). We discuss refinement algorithms for our subgoal graphs that are specialized to exploit structure in state lattices and grids in Chapters 4 and 5, respectively. The refinement phase is not necessary when answering distance queries rather than path queries.

Holzer et al. use a slightly different approach to answering queries using overlay graphs, that essentially combines the connection and search phases (Holzer et al., 2009): Their s-t query overlay graph corresponds to the overlay graph plus the connected components that contain s and t (induced by S, selected as a vertex separator of G), or simply a single connected component if it contains both s and t. This combination of the connection and search phases can be beneficial in some cases, for instance, by allowing the "connection phase" to terminate early if the "search phase" finds a solution. However, this argument only applies to a small set of queries, namely, when the s-t distance is smaller than the radius of the connected components that contain s and t. The separation of the two phases, on the other hand, allows us to have different implementations for the two phases, which is the case for the connection algorithms that we introduce in this dissertation.

3.2.5 Identifying Direct-Reachable Edges from a Source Vertex

In this section, we describe three variants of an algorithm called Overlay-Connect. Given a source vertex s, a set of vertices S that can cover paths, and a target set of vertices T, Overlay-Connect can identify the set of edges $D_S^{s \to T}$, along with their correct lengths (as distances on G). Overlay-Connect can be used during preprocessing to construct the edges of G_S given S (by running it for every $s \in S$ to identify $D_S^{s \to S}$), or during the connection phases of queries to connect the start and goal vertices to G_S (to connect the goal vertex, Overlay-Connect is run on the reverse graph of G, unless G is undirected). The "conservative" variant of Overlay-Connect (Algorithm 2) appears in the literature as a method for constructing the edges of overlay graphs (Schultes & Sanders, 2007; Holzer et al., 2009). It is *exact*, that is, guaranteed to return $D_S^{s \to T}$ along with the correct length of each edge. The "aggressive" and "stall-on-demand" variants of Overlay-Connect are *approximate*, that is, they may return a superset of $D_S^{s \to T}$, but correctly identify the length of each edge $(s, u) \in D_S^{s \to T}$ and never underestimate the lengths of redundant edges they return. These variants were first described in the context of highway-node routing (Schultes & Sanders, 2007), which answers queries using multiple levels of overlay graphs, and can be considered as a predecessor of contraction hierarchies.

- **Conservative variant:** This variant can be understood as a modified Dijkstra search that performs additional bookkeeping to identify whether each expanded vertex u is direct-reachable from s (with respect to S). It can identify $D_S^{s \to T}$ exactly, and is therefore called the "conservative" variant. Its differences from a Dijkstra search are highlighted in blue in Algorithm 2.

  Recall that a Dijkstra search expands a vertex u only after expanding all vertices v with $d(s, v) < d(s, u)$. In other words, a Dijkstra search expands a vertex u only

Algorithm 2 Overlay-Connect (Conservative)

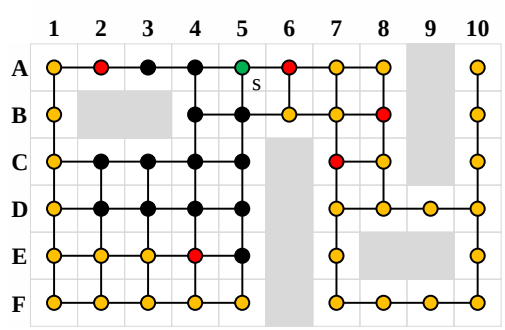
Blue text: Modifications to Dijkstra's algorithm.

Input: $G = (V, E, c)$, start vertex s, covering vertices $S \subseteq V$, target vertices $T \subseteq V$
Output: Edges $E^+ = D_S^{s \to T}$, edge lengths c^+ as distances on G

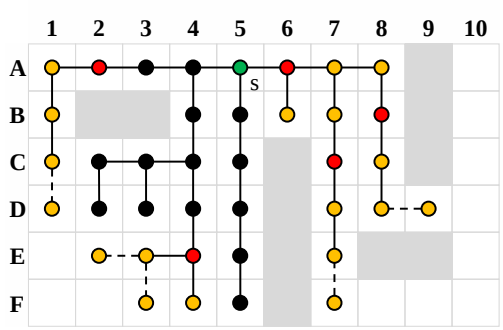
1: $E^+ \leftarrow \emptyset$
2: **for all** $n \in V$ **do**
3: $\quad g(n) \leftarrow \infty$
4: $\quad \text{covered}(n) \leftarrow \text{false}$
5: OPEN $\leftarrow \{s\}$
6: CLOSED $\leftarrow \{\}$
7: $g(s) \leftarrow 0$
8: **while** OPEN contains a vertex u with $\text{covered}(u) = \text{false}$ **do**
9: $\quad u \leftarrow$ vertex with minimum g-value in OPEN
10: $\quad$ Move u from OPEN to CLOSED
11: $\quad$ **if** $\text{covered}(u) = \text{false}$ and $u \in T \setminus \{s\}$ **then**
12: $\quad\quad$ Add (s, u) to E^+ with $c^+ = g(u)$
13: $\quad$ **for all** successors v of u such that $v \notin$ CLOSED **do**
14: $\quad\quad$ **if** $g(u) + c(u, v) < g(v)$ **then** ▷ A strictly shorter path to v is found
15: $\quad\quad\quad \text{covered}(v) \leftarrow \text{false}$
16: $\quad\quad$ **if** $g(u) + c(u, v) \leq g(v)$ **then**
17: $\quad\quad\quad g(v) \leftarrow g(u) + c(u, v)$
18: $\quad\quad\quad$ OPEN $\leftarrow$ OPEN $\cup \{v\}$
19: $\quad\quad\quad$ **if** $u \in S \setminus \{s\}$ or $\text{covered}(u) = \text{true}$ **then**
20: $\quad\quad\quad\quad \text{covered}(v) = \text{true}$
21: return E^+, c^+

after expanding all vertices v that can cover a shortest (s, u) path (Lemma 3.1). Algorithm 2 uses this fact to correctly propagate "covered" values for vertices: For every vertex u, covered(u) is set to true if and only if at least one of the (tentative) shortest s-u paths explored by the search is covered by S (lines 4, 14–15, and 19–20). Similar to g-values, covered-values are only tentative for vertices in OPEN, but exact for expanded vertices. Algorithm 2 terminates when all vertices in OPEN are covered (line 8) since, if the search were to continue, any paths to vertices not yet expanded by the search are guaranteed to be covered. An edge is added to E^+ for every expanded vertex $u \in T$ that is not covered (lines 1 and 11–12).

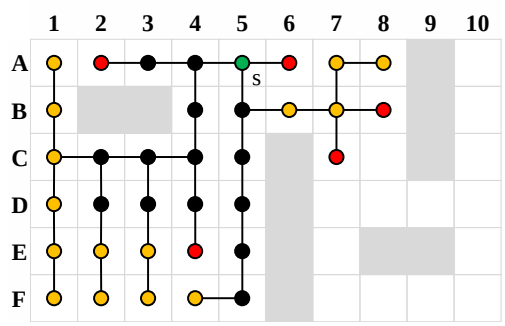
Schultes and Sanders point out that, in this variant, "if the shortest-path tree contains one path that is not covered for a long time, the tree can get very big even though other branches might have been covered very early" (Schultes & Sanders, 2007). Figure 3.4b shows an example where $S \not\sqsubset$ (A5, D2), causing the search to expand vertices within a radius of 6 and generate vertices within a radius of 7 from A5 (generated but not expanded vertices are connected to the search tree with dashed lines).



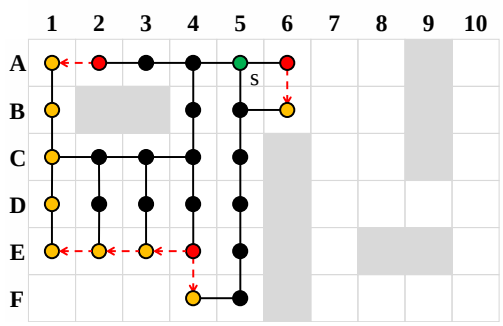
(a) Direct-reachable (black, A2, A6, E4) and non-direct-reachable (orange, B8, C7) vertices from s.



(b) Conservative variant: Propagate "covered", terminate when all vertices in OPEN are covered.



(c) Aggressive variant: Don't expand vertices in S.



(d) Stall-on-demand variant: Stall (don't expand) vertices in S. Stall vertices whose parents can be changed to a stalled vertex without increasing their g-values.

Figure 3.4: Variants of the Overlay-Connect algorithm. Vertices in S are shown in red. Black vertices are direct-reachable from s with respect to S, while orange vertices are not.

- **Aggressive variant:** This variant is simply a Dijkstra search that does not expand vertices in $S \setminus \{s\}$. That is, it "aggressively" prunes its search tree at vertices in $S \setminus \{s\}$. This pruning does not violate the correctness of this variant, since any paths explored through a vertex in $S \setminus \{s\}$ are, by definition, covered. Compared to the conservative variant, this variant can avoid the overhead of maintaining "covered" values for vertices, and can avoid expanding vertices that are "disconnected" from s if S were to be removed from G. Figure 3.4c shows an example, where the search, unlike the conservative variant, does not explore beyond C7 and B8 due to its aggressive pruning.

  Schultes and Sanders point out that the drawback of this approach is that the search might continue "around" the vertices in S and, in the worst case, explore the whole graph without the early termination of the conservative approach (Schultes & Sanders, 2007). That is, the search is guaranteed to expand every vertex u that is not "disconnected" from s if S were to be removed from G. For instance, in Figure 3.4c, removing S separates G into two components, and the aggressive

variant expands all vertices in the component that contains s. As a result: (1) It is no longer guaranteed that, for every expanded vertex u, the shortest s-u path in the search tree is a shortest path on G. For instance, in Figure 3.4c, A1 is reached with a path of length 8, where d(A5,A1) = 4. (2) The aggressive variant is not exact. For instance, in Figure 3.4, if the aggressive variant were used to identify $D_S^{s \Rightarrow S}$, it would identify (A5,B8) as an edge, even though A6 $\sqsubset$ (A5,B8).

- **Stall-on-demand variant:** This variant tries to address the problem of the aggressive variant by "stalling" the search (that is, not expanding a vertex) not just at vertices in S but also at vertices whose parents can be changed to a stalled vertex without increasing their g-values. Similar to the aggressive variant, the stall-on-demand variant does not expand vertices in S, but marks them as "stalled" instead. Similar to how the conservative variant propagates "covered" values, the stall-on-demand variant "lazily" propagates "stalled" values. That is, since a stalled vertex is not expanded, the fact that it is stalled is not communicated to its successors. Instead, whenever a vertex n is selected for expansion, a stall-on-demand check is performed to see if it can be stalled: n is stalled if and only if n has a predecessor p such that p is stalled and $g(p) + c(p, n) \leq g(n)$. If n is stalled due to a stalled predecessor p, its g-value is updated to $g(p) + c(p, n)$.

  Figure 3.4d shows an example. A6 is stalled when it is selected for expansion because A6 $\in S$. When the search selects B6 for expansion, the search checks if B6 can be stalled by scanning its predecessors. Since its predecessor A6 is stalled and since g(A6) + c(A6,B6) = 1 + 1 $\leq$ g(B6) = 2, the search stalls B6 as well. As a result, the search avoids exploring beyond A6 and B6, unlike both the aggressive and conservative variants. Similarly, E4 is stalled because E4 $\in S$. This stalling is then propagated to E3, E2, and E1, as each of these vertices are selected for expansion and "demand" to be stalled. The stalling does not violate the correctness of this variant since the stalling check for a vertex n essentially verifies either that n is reached with a suboptimal path or that an equal length path can be found that passes through a vertex in S (through a chain of stalled vertices).

  Although the stall-on-demand variant can mitigate the problem of the aggressive variant, it is not guaranteed to eliminate the problem completely. For instance, in Figure 3.4d, similar to the aggressive variant, the stall-on-demand variant expands B1 with g(B1) = 7. Unlike the aggressive variant, the g-value of A1 is updated to 4 (from 8) when A1 is stalled by A2. Although the stall-on-demand variant expands no more vertices than the aggressive variant, each expansion involves a stalling check that scans the predecessors of the vertex selected for expansion.

  Stall-on-demand is a technique that appears in various forms and contexts in the literature (Schultes & Sanders, 2007; Geisberger et al., 2008). The version that we have described is adapted from the contraction hierarchy implementation in the Grid-based Path-Planning Competition, which we discuss further in Section 3.4.4.2. It is similar to the stall-on-demand technique described by (Geisberger et al., 2008), which additionally performs a breadth-first search from a stalled vertex when its g-value is updated. The breadth-first search is run only on the search tree of the Dijkstra search in an attempt to correct g-values and stall more vertices. For

instance, in Figure 3.4d, when A1 is stalled by A2, its g-value is reduced from 8 to 4. A breadth-first search from A1 would also update the g-value of B1 from 7 to 5 and mark it as stalled.

The connection algorithms that we develop for our subgoal graphs on state lattices build upon the aggressive variant of Overlay-Connect, using reachability relations (rather than vertex separators or stall-on-demand) to "bound" the search. Our connection algorithms exploit the freespace structure of state lattices to replace the Dijkstra search with a depth-first search, and one variant avoids performing duplicate detection, resulting in significantly smaller average vertex-expansion times. We experimentally evaluate the three variants of Overlay-Connect on state lattices in Chapter 4, for a reachability relation that does not exploit the freespace structure of state lattices.

3.2.6 Contractions and Heavy Contractions

As discussed in the previous section, the edges of overlay graphs and extended overlay graphs can be constructed with the Overlay-Connect algorithm. In this section, we describe a different method for constructing their edges with *vertex contractions*; an operation that is used for constructing contraction hierarchies (Geisberger et al., 2008). The aim of this section is to understand vertex contractions in the context of (extended) overlay graphs and introduce a variant, called *heavy contractions*, to lay the groundwork for the variants of vertex contractions that we develop in Sections 3.3.5 and 3.4.2, which can be used to construct (N-level) subgoal graphs and augment contraction hierarchies with reachability relations, respectively. Algorithm 3 outlines the Contract (black and blue text) and HeavyContract (black and red text) operations.

Algorithm 3 Contract and HeavyContract

Blue text: Only for Contract.
Red text: Only for HeavyContract.

Input: Overlay graph $G_S = (S, E', c')$ / Extended overlay graph $G_{S,T} = (T, E', c')$, vertex $n \in S$ to remove from S
Output: Overlay graph $G_{S\setminus\{n\}}$ / Extended overlay graph $G_{S\setminus\{n\},T}$

```
1: E+ ← ∅
2: for all (p, n) ∈ E' do
3:     for all (n, s) ∈ E' do
4:         d ← length of a shortest p-s overlay path that does not pass through n
5:         if d > c'(p, n) + c'(n, s) then
6:             E+ := E+ ∪ {(p, s)}
7:             c'(p, s) = c'(p, n) + c'(n, s)
8: E− = {(u, v) ∈ E' : u = n or v = n}
9: return G_{S\{n},T} = (S \ {n}, T, (E' ∪ E+) \ E−, c')
```

- **Contract:** Contracting a vertex n from a graph removes it from the graph and adds the minimum set of shortcut edges to preserve the distances between the

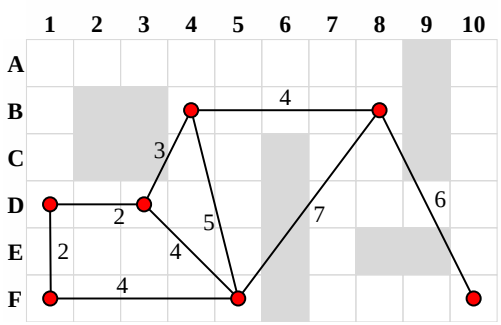
remaining vertices. Contracting a vertex $n \in S$ from an overlay graph G_S creates the overlay graph $G_{S\setminus\{n\}}$, as we prove at the end of this section (Theorem 3.9). To contract a vertex $n \in S$ from G_S, the Contract operation performs a *witness search* from every predecessor p to every successor s of n, to determine whether the path $\langle p, n, s\rangle$ is the unique shortest p-s path on G_S (lines 2–5). This search is typically performed by removing (the incident edges of) n from G_S and performing a p-s search on the remaining graph. If the p-s distance increases, that is, if a *witness path* of length equal to $l(\langle p, n, s\rangle)$ is not found, the shortcut edge (p, s) is added with length $l(\langle p, n, s\rangle)$ to preserve the p-s distance on $G_{S\setminus\{n\}}$ (lines 1, 6, 7, 9). After all necessary shortcut edges have been identified, n is removed from the graph along with its incident edges (lines 8–9).

Figure 3.5 shows an example of the Contract operation on an undirected graph. D3 has three neighbors in G_S (Figure 3.5a), namely D1, B4, and F5. To contract D3 from G_S, the Contract operation performs three witness searches for all pairings of these three vertices (if G_S were a directed graph, six witness searches would be performed). The witness search for a D1-B4 path that does not pass through D3 finds the path $\langle$D1, F1, F5, B4$\rangle$ with length 11, which is greater than $l(\langle$D1,D3,B4$\rangle) = 5$, verifying that $\langle$D1,D3,B4$\rangle$ is the unique shortest D1-B4 path on G_S. Therefore, the shortcut edge D1-B4 is added to $G_{S\setminus\{D3\}}$ with length 5 (Figure 3.5b). The witness search for a D1-F5 path that does not pass through D3 finds the path $\langle$D1, F1, F5$\rangle$ with length 6, which is smaller than or equal to $l(\langle$D1, D3, F5$\rangle) = 6$. Therefore, the shortcut edge (D1,F5) is not necessary to preserve the D1-F5 distance on $G_{S\setminus\{D3\}}$. The edge (B4, F5) already appears in G_S as the unique shortest B4-F5 path and is not added to the graph for a second time.

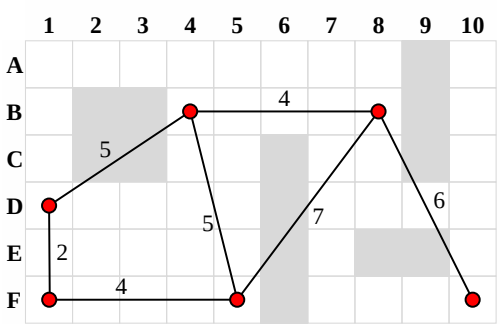
The Contract operation can be used to generate the overlay graph G_S from G, by starting with the overlay graph $G_V = G$ and repeatedly contracting all vertices $n \in V \setminus S$.

- **HeavyContract:** Heavy contracting a vertex $n \in S$ from the extended overlay graph $G_{S,T}$ creates the extended overlay graph $G_{S\setminus\{n\},T}$. The HeavyContract operation differs from the Contract operation in two ways: (1) Recall that, for every $s, t \in T$, an extended overlay graph $G_{S,T}$ is guaranteed to contain an s-t *overlay* path π with $l(\pi) = d(s, t)$. Therefore, when heavy contracting a vertex $n \in S$ from $G_{S,T}$, for every predecessor p and successor s of n, the shortcut edge (s, p) is required if and only if $\langle p, n, s\rangle$ is the unique shortest *overlay* path on $G_{S,T}$. As a result, witness searches in heavy contractions look for shortest *overlay* paths rather than shortest paths (line 4). (2) Since heavy contracting n from $G_{S,T}$ removes it only from S but not from T, its incident edges are not discarded (lines 8–9).

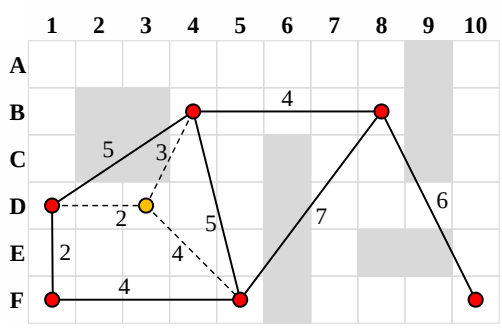
  Figure 3.5 shows an example. Similar to contracting D3 from G_S, heavy contracting D3 from $G_{S,S} = G_S$ requires three witness searches and adds the shortcut edge (D1,B4). However, it does not discard the edges incident to D3 (Figure 3.5c). In the resulting graph $G_{S\setminus\{D3\},S}$, B4 has one additional neighbor, D3, as opposed to the graph $G_{S\setminus\{D3\}}$ that results from contracting D3 from G_S. Therefore, heavy contracting B4 from $G_{S\setminus\{D3\},S}$ (Figure 3.5e) requires more witness searches than



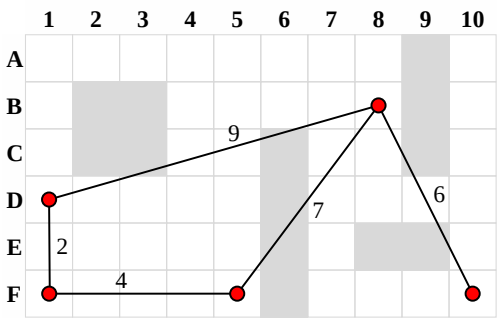
(a) Overlay graph G_S = extended overlay graph $G_{S,S}$.



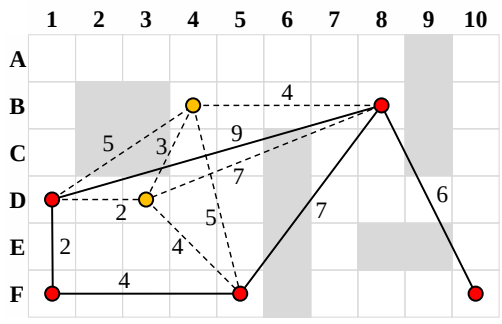
(b) Contracting D3 from G_S results in $G_{S\setminus\{D3\}}$.



(c) Heavy contracting D3 from $G_{S,S}$ results in $G_{S\setminus\{D3\},S}$.



(d) Contracting B4 from $G_{S\setminus\{D3\}}$ results in $G_{S\setminus\{D3,B4\}}$.



(e) Heavy contracting B4 from $G_{S\setminus\{D3\},S}$ results in $G_{S\setminus\{D3,B4\},S}$.

Figure 3.5: Using contractions and heavy contractions to generate overlay and extended overlay graphs, respectively.

contracting B4 from $G_{S\setminus\{D3\}}$ (Figure 3.5d), and results in the additional shortcut edge (D3,B8). Heavy contracting B4 from $G_{S\setminus\{D3\},S}$ adds the shortcut edge (D1,B8), similar to contracting B4 from $G_{S\setminus\{D3\}}$. Even though an alternate shortest D1-B8 path $\pi = \langle$D1, D3, B8$\rangle$ exists, it is not an overlay path (since D3 $\sqsubset \pi$ but D3 $\notin S \setminus \{D3\}$) and therefore ignored by the witness search.

The HeavyContract operation can be used to generate the extended overlay graph $G_{S,T}$ from the overlay graph G_T, by starting with the extended overlay graph $G_{T,T} = G_T$ and repeatedly heavy contracting all vertices $n \in T \setminus S$. Since heavy contractions do not discard existing edges but may add new ones, repeatedly heavy contracting vertices from $G_{T,T}$ typically increases the average number of predecessors and successors of other vertices. As a result, subsequent heavy contractions typically require witness searches between a greater number of predecessor-successor pairs and identify more shortcut edges than contractions would do after repeatedly contracting (rather than heavy contracting) a number of vertices from $G_T = G_{T,T}$, which we think justifies the term "heavy" to describe this variant of contractions.

We now prove that contractions and heavy contractions can be used to construct overlay and extended overlay graphs, respectively. In Lemma 3.7, we prove that an edge (u, v) appears in the extended overlay graph $G_{S\setminus\{n\},T}$ but not in $G_{S,T}$ if and only if $\langle u, n, v\rangle$ is the unique shortest overlay u-v path on $G_{S,T}$. That is, we characterize the edges that should be added when heavy contracting n from $G_{S,T}$ as those edges (u, v) where $\langle u, n, v\rangle$ is the unique shortest overlay u-v path on $G_{S,T}$. In Lemma 3.8, we prove that heavy contractions correctly determine those edges. In Theorem 3.9, we combine the two lemmata to prove that heavy-contacting $n \in S$ from $G_{S,T}$ results in $G_{S\setminus\{n\},T}$. In Corollary 3.10, we use Theorem 3.9 to show that contracting $n \in S$ from G_S results in $G_{S\setminus\{n\}}$.

Lemma 3.7. *For every $n \in S$ and $u, v \in T$, $\langle u, n, v\rangle$ is the unique shortest overlay u-v path on $G_{S,T}$ if and only if $n \sqsubset (u, v)$ and $S \setminus \{n\} \not\sqsubset (u, v)$.*

Proof.

1. Let $G_{S,T} = (T, E', c')$.
2. Let $u, v \in T$.
3. Let $n \in S$.
4. If $n \sqsubset (u, v)$ and $S \setminus \{n\} \not\sqsubset (u, v)$, then $\langle u, n, v\rangle$ is the unique shortest overlay u-v path on $G_{S,T}$:
   - 4.1. Assume that $n \sqsubset (u, v)$.
   - 4.2. Assume that $S \setminus \{n\} \not\sqsubset (u, v)$.
   - 4.3. Let $\pi = \langle p_0, \ldots, p_k\rangle$ be a shortest overlay u-v path on $G_{S,T}$ with $n \sqsubset \pi$. Such π exists (Lemma 3.4, since $n \sqsubset (u, v)$ with $n \in S$ and $u, v \in T$).
   - 4.4. $\pi \neq \langle u, v\rangle$:
     - 4.4.1. $(u, v) \notin E'$ (Definition 3.4, since $n \sqsubset (u, v)$).

4.5. $S \setminus \{n\} \not\sqsubset \pi$ (Lemma 3.4, since $S \setminus \{n\} \not\sqsubset (u, v)$ with $S \setminus \{n\} \subset S$ and $u, v \in T$).

4.6. $p_1, \ldots, p_{k-1} \notin S \setminus \{n\}$ (Definition 3.1, since $S \setminus \{n\} \not\sqsubset \pi$).

4.7. $p_1, \ldots, p_{k-1} \in S$ (Definition 3.5, since π is an overlay path).

4.8. $p_1, \ldots, p_{k-1} \in \{n\}$ (since $p_1, \ldots, p_{k-1} \in S$ and $p_1, \ldots, p_{k-1} \notin S \setminus \{n\}$).

4.9. $\pi = \langle u, n, v \rangle$ (since $\pi \neq \langle u, v \rangle$ and $p_1, \ldots, p_{k-1} \in \{n\}$).

5. If $\pi = \langle u, n, v \rangle$ is the unique shortest overlay u-v path on $G_{S,T}$, then $n \sqsubset (u, v)$

5.1. Assume that $\pi = \langle u, n, v \rangle$ is the unique shortest overlay u-v path on $G_{S,T}$.

5.2. $n \sqsubset \pi$ (Definition 3.1, since $n \in \pi$ and $n \notin \{u, v\}$).

5.3. $n \sqsubset (u, v)$ (Lemma 3.4, since $n \sqsubset \pi$ and π is a shortest overlay u-v path on $G_{S,T}$).

6. If $\pi = \langle u, n, v \rangle$ is the unique shortest overlay u-v path on $G_{S,T}$, then $S \setminus \{n\} \not\sqsubset (u, v)$:

6.1. Assume that $\pi = \langle u, n, v \rangle$ is the unique shortest overlay u-v path on $G_{S,T}$.

6.2. Assume (for contradiction) that there exists $w \in S \setminus \{n\}$ such that $w \sqsubset (u, v)$.

6.3. Let π' be a shortest overlay u-v path on $G_{S,T}$ with $w \sqsubset \pi'$. Such π' exists (Lemma 3.4, since $w \sqsubset (u, v)$ with $w \in S$ and $u, v \in T$).

6.4. $\pi' \neq \pi$ (since $w \neq n$ and $w \sqsubset \pi'$).

6.5. $\bot$ (π is the unique shortest overlay path and $\pi' \neq \pi$ is a shortest overlay path).

□

Lemma 3.8. *After heavy contracting a vertex $n \in S$ using Algorithm 3, for every $u, v \in T$, it holds that $(u, v) \in E^+$ if and only if $\langle u, n, v \rangle$ was the unique shortest overlay u-v path on $G_{S,T}$.*

Proof. Let $u, v \in T$ and $n \in S$.

1. If $\langle u, n, v \rangle$ is the unique shortest overlay u-v path, then $(u, v) \in E^+$:

1.1. $(u, n), (n, v) \in E'$ (since (u, n, v) is a path).

1.2. Lines 4–6 will be executed with $p = u$ and $s = v$ (since $(u, n), (n, v) \in E'$).

1.3. On line 4, $d > d(u, v) = c'(u, n) + c'(n, v)$ (since $\langle u, n, v \rangle$ is the unique shortest overlay u-v path on $G_{S,T}$).

1.4. Therefore, the check on line 5 succeeds and (u, v) is added to E^+.

2. If $(u, v) \in E^+$, then $\langle u, n, v \rangle$ is the unique shortest overlay u-v path.

2.1. Let $\pi = \langle p_0, \ldots, p_k \rangle$ be a shortest overlay u-v path.

2.2. Lines 4–6 are executed with $p = u$ and $s = v$ (since $(u, v) \in E^+$).

2.3. The check on line 5 is successful (since $(u, v) \in E^+$).

2.4. $(u, n), (n, v) \in E'$ (since lines 4–6 are executed with $p = u$ and $s = v$).

2.5. All shortest overlay u-v paths use n as an intermediate vertex (since the check on line 5 is successful).

2.6. $\langle u, n\rangle$ and $\langle n, v\rangle$ are unique shortest overlay u-n and n-v paths, respectively (Lemma 3.5, since $(u, n), (n, v) \in E'$).

2.7. Therefore, $\pi = \langle u, n, v\rangle$.

□

Theorem 3.9. *Heavy contracting a vertex $n \in S$ from $G_{S,T}$ returns the extended overlay graph $G_{S\setminus\{n\},T}$.*

Proof.

1. Let $u, v \in T$ be arbitrary vertices with $u \neq v$.
2. Let $n \in S$.
3. (u, v) is an edge of $G_{S\setminus\{n\},T}$ and not an edge of $G_{S,T}$ if and only if $n \sqsubset (u, v)$ and $S \setminus \{n\} \not\sqsubset (u, v)$:
   - 3.1. (u, v) is not an edge of $G_{S,T}$ if and only if $S \sqsubset (u, v)$ (Definition 3.4).
   - 3.2. (u, v) is an edge of $G_{S\setminus\{n\},T}$ if and only if $S \setminus \{n\} \not\sqsubset (u, v)$ (Definition 3.4).
   - 3.3. If $S \setminus \{n\} \not\sqsubset (u, v)$, then $S \sqsubset (u, v)$ if and only if $n \sqsubset (u, v)$ (Definition 3.1).
   - 3.4. Therefore, (u, v) is an edge of $G_{S\setminus\{n\},T}$ and not an edge of $G_{S,T}$ if and only if $n \sqsubset (u, v)$ and $S \setminus \{n\} \not\sqsubset (u, v)$.
4. $(u, v) \in E^+$ if and only if $n \sqsubset (u, v)$ and $S \setminus \{n\} \not\sqsubset (u, v)$ (Lemmata 3.7 and 3.8).
5. Therefore, $(u, v) \in E^+$ if and only if (u, v) is an edge of $G_{S\setminus\{n\},T}$ but not an edge of $G_{S,T}$.

□

Corollary 3.10. *Contracting a vertex n from G_S returns the overlay graph $G_{S\setminus\{n\}}$.*

Proof. From Theorem 3.9, heavy contracting $n \in S$ from $G_{S,S}$ results in $G_{S\setminus\{n\},S}$. Contracting n from G_S adds the same set of shortcut edges as heavy contracting n from $G_{S,S}$ since $G_S = G_{S,S}$ and since all paths on $G_{S,S}$ are overlay paths. Contracting n from G_S removes all edges incident to n, which are precisely the edges that appear in $G_{S\setminus\{n\},S}$ but not in $G_{S\setminus\{n\}}$ (Definitions 3.3 and 3.4). □

3.3 Subgoal Graphs

As discussed in Sections 3.1 and 3.2.5, several preprocessing-based path-planning algorithms that (explicitly or implicitly) use overlay graphs to answer queries constrain the vertices of their overlay graphs in various ways to speed up the connection and refinement phases of their queries. Subgoal graphs can be considered to be a generalization of this idea: Subgoal graphs are overlay graphs constructed with respect to a given *reachability*

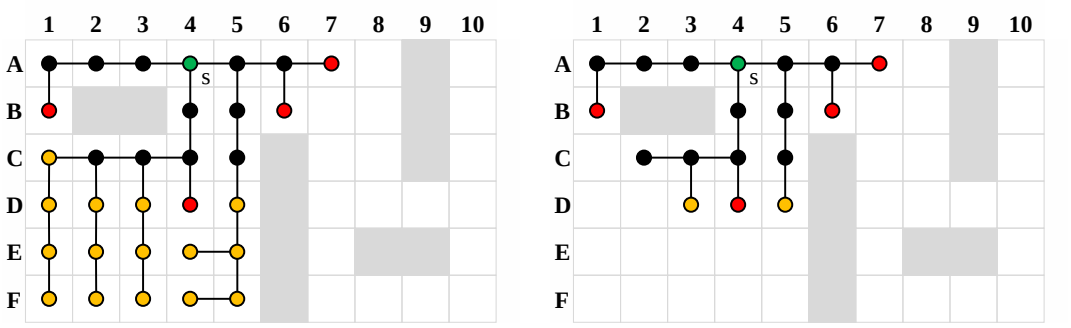
relation R, that is, a binary relation defined over the pairs of vertices $V \times V$ of G. For every $s, t \in V$, s-t query subgoal (overlay) graphs contain only R-reachable edges, that is, edges $(u, v) \in R$. Therefore, the connection phases of queries answered using subgoal graphs only need to identify R-reachable edges, and the refinement phases only need to find shortest paths between R-reachable pairs of vertices. In this dissertation, we use subgoal graphs to exploit the freespace structure in state lattices and grid graphs, by using a reachability relation to capture this structure and using specialized connection and refinement algorithms during queries to exploit this structure.

This section is organized as follows. In Section 3.3.1, we formally define reachability relations and introduce *bounded-distance reachability* as a reachability relation, which we use as a running example in this chapter and as a reachability relation to compare against on state lattices in Chapter 4. In Section 3.3.2, we formally define R shortest-path covers and subgoal graphs, and prove that, if the vertices of a subgoal graph form an R shortest-path cover, then every extension of this subgoal graph into a query subgoal graph contains R-reachable edges only. In Section 3.3.3, we discuss the similarities and differences of our definition for shortest path covers with those that appear in the literature, discuss how they relate to the notion of highway dimension, and prove that it is possible to construct "locally sparse" bounded-distance reachability shortest-path covers on graphs with low highway dimensions. In Section 3.3.4, we introduce a class of algorithms that can be used for connection and refinement during queries, called R-connect and R-refine algorithms, and characterize how they should operate so that they can be used for finding shortest paths. In Section 3.3.5, we introduce heavy R contractions, which can be used to construct subgoal graphs by *pruning* an R shortest-path cover. In Section 3.3.6, we introduce an alternative method of constructing subgoal graphs by *growing* an R shortest-path cover.

3.3.1 Reachability Relations

In this section, we formally define reachability relations, introduce notation for reachability relations, and introduce *bounded-distance reachability* as a reachability relation that we use as an example throughout the remainder of this chapter and as a reachability relation to compare against on state lattices in Chapter 4.

As we have discussed in Section 3.2.5, Overlay-Connect can identify all direct-reachable vertices from a given vertex s (with respect to a set of vertices S). Consider the search tree of the aggressive variant of Overlay-Connect shown in Figure 3.6a. The furthest direct-reachable vertices from s, B1 and C2, have a distance of 4 from s, but the aggressive variant of Overlay-Connect expands many more vertices, up to a distance of 8 from s, by going "around" the vertices in S. However, if it is known that "the furthest direct-reachable vertex from s has a distance of 4 from s", we could augment the aggressive variant of Overlay-Connect to avoid generating vertices that have a distance of more than 4 from s (Figure 3.6b). Reachability relations allow us to represent notions such as "distance of no more than 4 away", so that we can construct overlay graphs with respect to them, allowing us to implement connection and refinement algorithms that can operate under certain assumptions, such as "every edge needed to connect a vertex to a subgoal graph has a length of no more than 4". Definition 3.7 formally defines reachability relations.



(a) Aggressive Overlay-Connect. (b) Aggressive Overlay-Connect with bound 4.

Figure 3.6: Bounding the aggressive variant of Overlay-Connect by using BD4 as reachability relation. Vertices in S are shown in red. Black vertices are direct-reachable from s with respect to S, while orange vertices are not.

Definition 3.7 (R-reachability)**.** *A reachability relation is a relation $R \subseteq V \times V$ that satisfies:*

1. $\forall n \in V$, $(n, n) \in R$ *and*
2. $\forall (u, v) \in E$, $(u, v) \in R$.

By definition, a reachability relation R includes every edge of G (that is, $E \subseteq R$). As we discuss in the next section, this requirement guarantees that, for every R, we can construct a subgoal graph on G with respect to R.

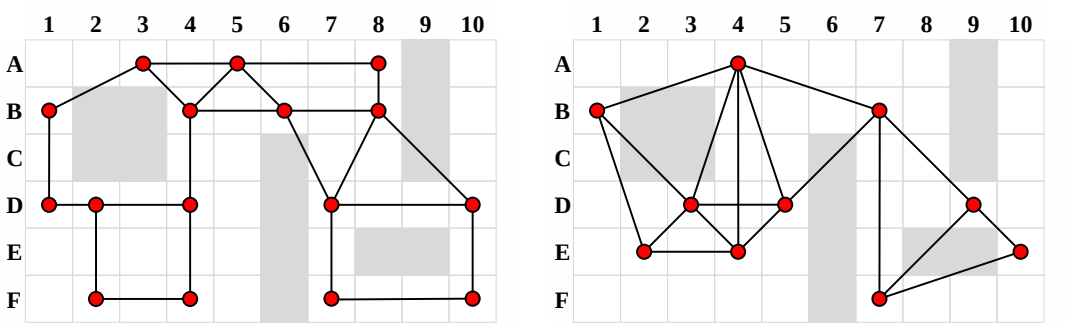
Direct-reachability with respect to a set of vertices $S \subseteq V$, that is, D_S, can also be considered to be a reachability relation: For each $n \in V$, $(n, n) \in D_S$ since no S can cover the unique shortest n-n path $\langle n \rangle$, and, for each $(u, v) \in E$, $(u, v) \in D_S$ since no S can cover the unique shortest u-v path $\langle u, v \rangle$ (Assumption 1.3).

The combination (intersection) of any two reachability relations R_1 and R_2 (that is, $R_1 \cap R_2$) is also a reachability relation. We refer to the combination of a reachability relation R with the direct-reachability relation with respect to S as the direct-R-reachability relation with respect to S, and denote it as R_S. That is, $R_S = R \cap D_S$.

Throughout the rest of this chapter, we use *bounded-distance reachability* as an example reachability relation. A vertex t is bounded-distance reachable from a vertex s if and only if $d(s, t) \leq b$ for some given reachability bound b. We use the notation BDb to denote bounded-distance reachability with bound b. For the 4-neighbor grid graphs that we use as examples in this chapter, for every $b \geq 1$, BDb is a reachability relation since, for every edge (u, v) of the 4-neighbor grid graph, $d(u, v) = 1$ and, therefore, $(u, v) \in$ BDb.

3.3.2 Subgoal Graphs

Suppose that we are given a reachability relation R and want to construct an overlay graph G_S such that, for every $s, t \in V$, the s-t query overlay graph $G_S^{s,t}$ contains R-reachable edges only. Recall that the set of edges of any s-t query overlay graph $G_S^{s,t}$ is a subset of the edges of the extended overlay graph $G_{S,V}$ (Definitions 3.4 and 3.6).



(a) An overlay graph that is not a BD4 subgoal graph: $S \not\sqsubset$ (A5,F5) and (A5,F5) $\notin$ BD4.

(b) A BD4 subgoal graph. For every $s, t \in V$, $S \sqsubset (s,t)$ or $(s,t) \in$ BD4.

Figure 3.7: A BD4 subgoal graph and an overlay graph that is not a BD4 subgoal graph.

Therefore, if we choose an S such that the extended overlay graph $G_{S,V}$ has R-reachable edges only, then we guarantee that every s-t query overlay graph $G_S^{s,t}$ has R-reachable edges only. Definition 3.8 captures this "constraint" on S with the concept of an R *shortest path cover*: Recall that, for every $u, v \in V$, $G_{S,V}$ contains an edge (u,v) if and only if $S \not\sqsubset (u,v)$. That is, if $(u,v) \notin R$, in order to not have the non-R-reachable edge (u,v) in $G_{S,V}$, it should hold that $S \sqsubset (u,v)$.

Definition 3.8 (Shortest path cover)**.** *S is an R shortest-path cover (R-SPC) if and only if, $\forall s,t \in V$, $(s,t) \in R$ or $S \sqsubset (s,t)$.*

We call an overlay graph G_S where S is an R-SPC an R *subgoal graph* or, simply, a subgoal graph if the specific R is not important. Definition 3.9 defines subgoal graphs.

Definition 3.9 (Subgoal graph)**.** *An overlay graph G_S is called an R subgoal graph if and only if S is an R-SPC.*

Figure 3.6b shows an example of a BD4 subgoal graph. The overlay graph in Figure 3.6a is not a BD4 subgoal graph although its edges are all BD4-reachable: Its set of vertices $S \not\sqsubset$ (A5,F5), which means that its extension into an F5-t query overlay graph (for any t) would require the non-BD4-reachable edge (A5,F5).

Definition 3.10 defines *partial R-SPCs*, which can be considered as R-SPCs for only a subset $T \subseteq V$ of the vertices of G. We use partial R-SPCs for our definition of N-level subgoal graphs in Section 3.4.2.

Definition 3.10 (Partial shortest path cover)**.** *S is a partial R shortest-path cover of T ((R,T)-SPC) if and only if, $\forall s,t \in T$, $(s,t) \in R$ or $S \sqsubset (s,t)$.*

Lemma 3.11 follows from the definition of (R,T)-SPCs, and also holds for R-SPCs since R-SPCs are equivalent to (R, V)-SPCs.

Lemma 3.11. *If S is an (R,T)-SPC, then, for every $S' \supseteq S$, $R' \supseteq R$ and $T' \subseteq T$, S' is an (R',T')-SPC.*

We conclude this section by proving that an extended overlay graph $G_{S,T}$ has only R-reachable edges if and only if S is an (R,T)-SPC.

Theorem 3.12. *An extended overlay graph $G_{S,T}$ has only R-reachable edges if and only if S is an (R,T)-SPC.*

Proof.

1. Let $G_{S,T} = (T, E', c')$.
2. If S is an (R,T)-SPC, then, $\forall (u,v) \in E'$, $(u,v) \in R$:
   - 2.1. Assume that S is an (R,T)-SPC.
   - 2.2. Assume (for contradiction) that there exists $(u,v) \in E'$ such that $(u,v) \notin R$.
   - 2.3. $u, v \in T$ (Definition 3.4, since $(u,v) \in E'$).
   - 2.4. $(u,v) \in R$ or $S \sqsubset (u,v)$ (Definition 3.10, since S is an (R,T)-SPC and $u, v \in T$).
   - 2.5. $S \sqsubset (u,v)$ (since $(u,v) \notin R$).
   - 2.6. $(u,v) \notin E'$ (Definition 3.4, since $S \sqsubset (u,v)$).
   - 2.7. $\perp$ (since $(u,v) \in E'$ and $(u,v) \notin E'$).
3. If, $\forall (u,v) \in E'$, $(u,v) \in R$, then S is an (R,T)-SPC:
   - 3.1. Assume that, $\forall (u,v) \in E'$, $(u,v) \in R$.
   - 3.2. Assume (for contradiction) that S is not an (R,T)-SPC.
   - 3.3. Let $s, t \in T$, such that $S \not\sqsubset (s,t)$ and $(s,t) \notin R$. Such (s,t) exists (Definition 3.10, since S is not an (R,T)-SPC).
   - 3.4. $s \neq t$ (Definition 3.7, since $(s,t) \notin R$).
   - 3.5. $(s,t) \in E'$ (Definition 3.4, since $s \neq t$, $s, t \in T$, and $S \not\sqsubset (s,t)$).
   - 3.6. $(s,t) \in R$ (since $(s,t) \in E'$ and, $\forall (u,v) \in E'$, $(u,v) \in R$).
   - 3.7. $\perp$ (since $(s,t) \in R$ and $(s,t) \notin R$).

□

3.3.3 Shortest-Path Covers and Highway Dimension

In this section, we relate our definition of R shortest-path covers to the definitions of shortest-path covers that appear in the literature, and prove that it is possible to construct *locally sparse* BDb shortest-path covers on graphs that have a small *highway dimension*.

Generally speaking, a "path cover" on G can be considered as a set of vertices S that "cover" certain paths on G, with respect to some notion of "covering". For instance, our definition of an R shortest-path cover is an instantiation of this general concept of path covers, where we aim to cover at least one shortest s-t path π for every $(s,t) \notin R$, and define "covering" as at least one vertex $n \in S \setminus \{s,t\}$ appearing as a vertex on π. An alternative notion of "covering" allows a vertex n to "cover" a path π if it appears as

any vertex on π (that is, including the first and last vertex of π). We refer to this notion of covering as "loose covering", since it is easier to "loosely cover" paths than it is to "cover" paths, as per our Definition 3.1.

The notion of loose covering is used in the literature to define k-hop shortest-path covers (Tao, Sheng, & Pei, 2011; Funke, Nusser, & Storandt, 2014), which loosely cover all shortest paths with at least k vertices; k-hop all-path covers (Funke et al., 2014), which loosely cover all paths with at least k vertices; and (r, k) shortest-path covers (Abraham et al., 2010), which loosely cover all shortest paths π with $r < l(\pi) \leq 2r$ and are (r, k)-*sparse*, as we explain later in this section. k-hop shortest-path covers have been used to generate overlay graphs on road networks (Tao et al., 2011), and k-hop all-path covers have been used to generate overlay graphs G' with additional edges, such that, when an edge length on G changes, G' can be updated quickly by changing only the lengths of some of its edges. (r, k)-shortest path covers have been used in conjunction with the notion of *highway dimension*, to provide runtime bounds for reach, transit-node routing, hub-labeling, and answering queries using contraction hierarchies (Abraham et al., 2010). We now briefly describe (r, k) shortest-path covers and the notion of highway dimension in our own terminology. [2]

Let $S \subseteq V$, $r \in \mathbb{R}_{>0}$, and $k \in \mathbb{Z}_{>0}$. S is an r *loose shortest-path cover* (r-LSPC) if and only if all shortest paths π on G with $r < l(\pi) \leq 2r$ are loosely covered by S. S is (r, k)-*sparse* if and only if, for every $s \in V$, the number of vertices $n \in S$ with $\min(d(s, n), d(n, s)) \leq 2r$ is at most k (that is, informally, the "ball of radius r" around every vertex contains at most k vertices from S). S is an (r, k) *loose shortest path cover* ((r, k)-LSPC) if and only if S is an (r, k)-sparse r-LSPC. The highway dimension h of a graph G is the smallest $h \in \mathbb{Z}_{>0}$, such that, for every $r \in \mathbb{R}_{>0}$, an (r, h)-LSPC exists on G. Abraham et al. show that, on graphs with highway dimension h and diameter D, after running a polynomial-time preprocessing routine, queries can be answered using contraction hierarchies or reach in $O((h \log h \log D)^2)$ time, hub labeling in $O(h \log h \log D)$ time, and long-range transit-node routing in $O(h^2)$ time (Abraham et al., 2010)

Observe that our definition of BDr-SPCs (that is, BDb-SPCs with $b = r$) is very similar to the definition of r-LSPCs by Abraham et al.. Namely, whereas BDr-SPCs *cover at least one* shortest s-t path for every $s, t \in V$ with $d(s, t) > r$, r-LSPCs *loosely cover all* shortest s-t paths π with $r < L(\pi) \leq 2r$. The reasons for the differences are as follows: 1) Shortest paths on road networks are typically unique, and symmetries do not usually need to be taken into account. That is, in the case of r-LSPCs on road networks, (loosely) covering *at least one* shortest path between two vertices is equivalent to covering *all* shortest paths between them. Since, in this dissertation, we consider grid graphs and state lattices that can have multiple shortest paths between the same two vertices, our definition of BDr-SPCs specifies that covering *at least one* shortest path between two vertices is sufficient. 2) We want to guarantee that query BDr subgoal graphs have BDr-reachable edges only (or, more generally, query R subgoal graphs have R-reachable edges only) and, therefore, require that BDr-SPCs *cover* paths rather than *loosely cover* them. For instance, consider $u, v \in V$ such that $(u, v) \notin$ BDr. If the vertices of an overlay graph

[2] Although the definitions and theoretical results by Abraham et al. assume that G is undirected, Abraham et al. outline how their definitions and theoretical results can be modified for directed graphs. Our overview presents their definitions and results after applying these modifications.

were selected as an r-LSPC rather than a BDr-SPC, then it could be the case that (u, v) is not covered but only loosely covered (by having u or v in the r-LSPC). However, then, the u-v query overlay graph would have the non-BDr-reachable edge (u, v) (Definition 3.6). 3) Road networks can have long edges (for instance, corresponding to ferry connections). As a result, the definition for r-LSPCs specifies that paths with such edges do not need to be (loosely) covered if their lengths are longer than $2r$. Our definition of BDr-SPCs does not have a similar specification since it assumes that the maximum edge length is less than r (otherwise, BDr would not be a reachability relation, as per Definition 3.7).

The highway dimension of a graph can be considered as a measure of the "local sparsity" of r-LSPCs that can be constructed on G, for every r. Theorem 3.13 proves that, if G has highway dimension h, then it is possible to construct a $(r - 2m, h)$-sparse BDr-SPC on G for every $r > 3m$, where m is the maximum edge length in G. The proof of Theorem 3.13 relies on the fact that an $(r - 2m, h)$-sparse $(r - 2m)$-LSPC S exists on G, which loosely covers all shortest paths π' of length $r - 2m < l(\pi') \leq 2r - 4m$. Theorem 3.13 proves that every shortest path π with length $l(\pi) > r$ can be considered as an extension of one such π' with at least two edges, one added as a prefix and the other one as a suffix to π'. Since S loosely covers π', it therefore covers π.

Theorem 3.13. *If G has highway dimension h and maximum edge length m, then, for every $r > 3m$, there exists a $(r - 2m, h)$-sparse BDr-SPC on G.*

Proof.

1. Let $r > 3m$.
2. Let G have highway dimension h.
3. Let S be an $(r-2m, h)$-sparse $(r-2m)$-LSPC S of G. Such S exists (since $r-2m > 0$ and G has highway dimension h).
4. Claim: S is a BDr-SPC of G.
5. Assume (for contradiction) that, for some $s, t \in V$, $(s, t) \notin$ BDr and $S \not\sqsubset (s, t)$.
6. Let $\pi = \langle p_0, \ldots, p_k \rangle$ be a shortest s-t path.
7. $l(\pi) = d(s, t) > r$ (since $(s, t) \notin$ BDr).
8. $d(s, p_1) \leq m$ and $d(p_{k-1}, t) \leq m$ (since m is the maximum edge length).
9. $d(p_1, p_{k-1}) = d(s, t) - d(s, p_1) - d(p_{k-1}, t) > r - 2m$ (since $l(\pi) > r$, $d(s, p_1) \leq m$ and $d(p_{k-1}, t) \leq m$).
10. Let $i < k$ be the minimum index for which $d(p_1, p_i) > r - 2m$. Such i exists (since $d(p_1, p_{k-1}) > r - 2m$).
11. $d(p_1, p_i) \leq 2r - 4m$:
    - 11.1. Assume (for contradiction) that $d(p_1, p_i) > 2r - 4m$.
    - 11.2. $d(p_1, p_{i-1}) = d(p_1, p_i) - d(p_{i-1}, p_i)$.

11.3. $d(p_1, p_{i-1}) > 2r - 4m - d(p_{i-1}, p_i)$ (since $d(p_1, p_i) > 2r - 4m$).

11.4. $d(p_1, p_{i-1}) > 2r - 4m - m$ (since $d(p_{i-1}, p_i) \leq m$).

11.5. $d(p_1, p_{i-1}) > r + 3m - 4m - m$ (since $r > 3m$).

11.6. $d(p_1, p_{i-1}) > r - 2m$.

11.7. $i - 1$ is an index for which $d(p_1, p_{i-1}) > r - 2m$.

11.8. $\perp$ (since $i > i - 1$ is the smallest index for which $d(p_1, p_i) > r - 2m$).

12. $\pi' = \langle p_1, \ldots, p_i \rangle$ is a shortest p_1-p_i path with length $r - 2m < l(\pi') \leq 2r - 4m$.

13. S loosely covers π' (since S is a $(r - 2m)$-LSPC of G and π' is a shortest path with length $r - 2m < l(\pi) \leq 2r - 4m$).

14. For some $j \in 1, \ldots, i$, $p_j \in S$.

15. $S \sqsubset \pi$ (Definition 3.1, since, for some $j \in 1, \ldots, i$, $p_j \in S$, and $0 < j < k$).

16. $S \sqsubset (s, t)$.

17. $\perp$ ($S \sqsubset (s, t)$ and $S \not\sqsubset (s, t)$).

□

We refer to Theorem 3.13 in Chapter 4 to conjecture that state lattices might have large highway dimensions, based on our experimental results for generating BDr subgoal graphs.

3.3.4 Answering Queries Using Subgoal Graphs

R subgoal graphs can be used to answer path queries optimally in the same way that overlay graphs can be used to answer path queries optimally, by using the Connect-Search-Refine algorithm that we have discussed in Section 3.2.4. In this section, we formally introduce the Connect-Search-Refine algorithm, introduce a class of algorithms, called R-connect and R-refine algorithms, and characterize the criteria for connection and refinement algorithms to be "correct".

Algorithm 4 outlines the Connect-Search-Refine algorithm, which can be used to answer path queries using R subgoal graphs. As we have discussed in Section 3.2.4, s-t path queries can be answered using overlay (subgoal) graphs in three phases: During the connection phase (lines 1–2), the overlay graph G_S is extended to an s-t query overlay graph $G_S^{s,t}$. During the search phase (line 3), a shortest s-t path Π on $G_S^{s,t}$ is found, which is guaranteed to have $l(\Pi) = d(s, t)$ (as we prove in Theorem 3.14). Finally, during the refinement phase (lines 4–7), each edge (u, v) on Π is replaced with a shortest s-t path on G. The resulting path π is guaranteed to be a shortest s-t path on G (as we prove in Theorem 3.14).

Since all edges of R query subgoal graphs are guaranteed to be R-reachable, as discussed in Section 3.3.2, the connection phases of queries need to identify only R-reachable edges, and the refinement phases need to find shortest paths only between R-reachable

Algorithm 4 Connect-Search-Refine

Input: G, reachability relation R, subgoal graph $G_S = (S, E', c')$ with respect to R, start vertex s, goal vertex t

Output: A shortest s-t path π on G

1: $E^+, c^+ \leftarrow R\text{-Connect}(G, S, s, t)$ ▷ Connection phase
2: $G' = (S \cup \{s, t\}, E' \cup E^+, \text{combine}(c', c^+))$
3: $\Pi \leftarrow$ a shortest s-t path on G' ▷ Search phase
4: $\pi \leftarrow \langle\rangle$ ▷ Refinement phase
5: **for all** $(u, v) \in \Pi$, in order **do**
6: $\quad \pi' \leftarrow R\text{-Refine}(G, u, v)$
7: $\quad \pi \leftarrow \pi \cdot \pi'$
8: return π

vertices. We call an algorithm that extends an R subgoal graph into an s-t query R subgoal graph an R-connect algorithm, and an algorithm that finds a shortest path between R-reachable vertices an R-refine algorithm. The premise of Theorem 3.14 summarizes the criteria for an R-connect or R-refine algorithm to be "correct", by using "R-Connect" (line 1) as a placeholder for any R-connect algorithm and "R-Refine" (line 6) as a placeholder for any R-refine algorithm.

Theorem 3.14. *Let G_S be an R subgoal graph, $s, t \in V$ be arbitrary vertices of G, and E^* be the set of edges that appear in $G_S^{s,t}$ but not in G_S. Algorithm 4 finds a shortest s-t path on G if:*

1. *R-Connect(G, S, s, t) returns E^+, c^+ such that:*
   - 1.1. *$E^* \subseteq E^+ \subseteq R$;*
   - 1.2. *$\forall (u, v) \in E^*$, $c^+(u, v) = d(u, v)$; and*
   - 1.3. *$\forall (u, v) \in E^+ \setminus E^*$, $c^+(u, v) \geq d(u, v)$.*
2. *R-Refine(u, v) returns a shortest (u, v) path on G if $(u, v) \in R$.*

Proof. We first prove the theorem by assuming that R-Connect identifies exactly the set of edges $E^+ = E^*$ (rather than a superset) that extend G_S to $G_S^{s,t}$, and correctly determines their lengths as distances on G. Under this assumption, the graph G' constructed by Algorithm 4 (line 2) is precisely $G_S^{s,t}$. By Theorem 3.3, the s-t distance on the s-t query overlay graph $G_S^{s,t}$ is $d(s, t)$. Therefore, the search phase (line 3) is guaranteed to find an s-t path Π on G' with length $d(s, t)$. By Theorem 3.12, all edges of $G_S^{s,t}$ are R-reachable and, by Definition 3.6, their lengths correspond to distances on G. Therefore, R-Refine can replace every edge (u, v) on Π with a corresponding shortest path on G to generate a shortest s-t path π on G.

We now extend this proof to the case where R-Connect may identify additional edges, as stated in the premise of the theorem, namely the set of edges $E^+ \setminus E^* \subseteq R$ such that, for every $(u, v) \in E^+ \setminus E^*$, $c^+(u, v) \geq d(u, v)$. That is, G' is formed by adding additional edges to $G_S^{s,t}$. As discussed earlier, by Theorem 3.3, the s-t distance $G_S^{s,t}$ is

$d(u, v)$. The additional edges cannot increase the s-t distance on $G_S^{s,t}$. The additional edges cannot decrease the s-t distance on $G_S^{s,t}$ since, for every $(u, v) \in E^+ \setminus E^*$, $c^+(u, v) \geq d(u, v)$. Therefore, the search phase is still guaranteed to find an s-t path Π on G' with length $d(s, t)$. Finally, if any of the edges $E^+ \setminus E^*$ appear on Π, R-refine can still find a corresponding shortest path on G since $E^+ \setminus E^* \subseteq R$. □

3.3.5 Heavy R Contractions

As discussed in Section 3.2.5, contracting a vertex $n \in S$ from an overlay graph G_S results in the overlay graph $G_{S\setminus\{n\}}$, and heavy contracting a vertex $n \in S$ from an extended overlay graph $G_{S,T}$ results in the extended overlay graph $G_{S\setminus\{n\},T}$. Furthermore, contractions can be used to construct overlay graphs G_S by repeatedly contracting the vertices $n \in V \setminus S$ from $G = G_V$, and heavy contractions can be used to construct extended overlay graphs by repeatedly heavy contracting the vertices $n \in T \setminus S$ from overlay graphs $G_T = G_{T,T}$. In this section, we introduce *heavy R contractions*, which heavy contract vertices from extended overlay graphs if and only if doing so introduces R-reachable shortcut edges only, and show that they can be used to construct extended overlay graphs $G_{S,T}$ with R-reachable edges only, or to construct *minimal* (R, T)-SPCs S, that is, no $S' \subset S$ is an (R, T)-SPC.

Algorithm 5 Heavy$R$Contract

Blue text: Modifications to heavy contractions.

Input: Extended overlay graph $G_{S,T} = (T, E', c')$ such that S is an (R, T)-SPC, reachability relation R, vertex n to remove from S

Output: Extended overlay graph $G_{S\setminus\{n\},T}$ if $S \setminus \{n\}$ is an (R, T)-SPC, $G_{S,T}$ otherwise

```
1:  E+ ← ∅
2:  for all (p, n) ∈ E' do
3:      for all (n, s) ∈ E' do
4:          d ← length of a shortest p-s subgoal path that is not covered by n
5:          if d > c'(p, n) + c'(n, s) then
6:              if (p, s) ∉ R then                          ▷ Cannot heavy contract n
7:                  return G_{S,T}
8:              else
9:                  E+ := E+ ∪ {(p, s)}
10:                 c'(p, s) = c'(p, n) + c'(n, s)
11: return G_{S\{n},T} = (S \ {n}, T, E' ∪ E+, c')
```

Heavy R contracting a vertex $n \in S$ from an extended overlay graph $G_{S,T}$ heavy contracts n from $G_{S,T}$ if and only if doing so introduces R-reachable shortcut edges only. Algorithm 5 outlines the Heavy$R$Contract operation, where its differences from the HeavyContract operation are highlighted in blue. Heavy$R$Contract first simulates heavy contracting n from $G_{S,T}$ to identify the set of shortcut edges E^+ that appear in $G_{S\setminus\{n\},T}$ but not in $G_{S,T}$ (lines 2–5, 9–10). If a non-R-reachable shortcut edge is identified (line 6), n is not heavy contracted and $G_{S,T}$ remains unchanged (line 7). Otherwise, if all shortcut edges in E^+ are R-reachable, n is heavy contracted from $G_{S,T}$ as usual (lines 8, 11).

Algorithm 6 Prune (R, T)-SPC.

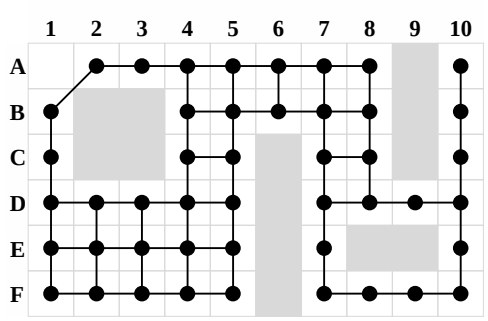
Input: Overlay graph $G_T = (T, E', c')$ where T is a (R, T)-SPC, reachability relation R
Output: A minimal (R, T)-SPC $S \subseteq T$, or the extended overlay graph $G_{S,T}$

```
1: S ← T, G_{S,T} ← G_T
2: for all n ∈ S, in some order do
3:     G_{S,T} ← HeavyRContract(G_{S,T}, R, n)
4: return S or G_{S,T}
```

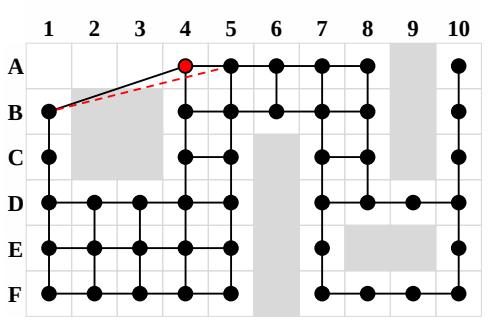
Since heavy R contractions introduce R-reachable shortcut edges only, they can be used to construct extended overlay graphs with R-reachable edges only. Specifically, given an overlay graph $G_T = G_{T,T}$ with R-reachable edges only, heavy R contracting all vertices $n \in T$ from $G_{T,T}$ generates an extended overlay graph $G_{S,T}$, where the vertices in S are exactly the vertices that did not get heavy contracted by heavy R contractions. $G_{S,T}$ is guaranteed to have R-reachable edges only, and, by Lemma 3.12, S is guaranteed to be an (R, T)-SPC. Algorithm 6 outlines this algorithm for constructing (R, T)-SPCs (or extended overlay graphs with R-reachable edges only). We prove in Theorem 3.16 that (R, T)-SPCs identified by Algorithm 6 are *minimal* (R, T)-SPCs, that is, no $S' \subset S$ exists such that S' is an (R, T)-SPC. We informally refer to Algorithm 6 as the *pruning* variant of constructing (R, T)-SPCs, since it can start with an R-SPC $S = T$ and repeatedly remove (heavy contract) vertices from S while maintaining that S is an R-SPC. In the next section, we introduce a *growing* variant of constructing (R, T)-SPCs, that starts with $S = \emptyset$ and adds vertices to S until it becomes an (R, T)-SPC. Note that Algorithm 6 can also be used to identify the vertices of R subgoal graphs as R-SPCs, by running it on the overlay graph $G_V = G$. In this dissertation, we use Algorithm 6 to construct subgoal graphs on state lattices (Section 4.6.5) and to construct N-level subgoal graphs on grid graphs (Section 5.5).

Figure 3.8 shows an example trace of Algorithm 6, where heavy R contractions are used to generate a (minimal) BD4-SPC (that is, a (BD4, V)-SPC). Each subfigure shows only the edges between vertices in S (that is, G_S) in the extended overlay graph maintained by Algorithm 6. Initially, the extended overlay graph $G_{V,V} = G_V = G$ only contains BD4-reachable edges. Heavy BD4 contracting A1 heavy contracts A1 and adds the R-reachable edge (B1, A2) (Figure 3.8a). Algorithm 6 proceeds to heavy BD4 contract vertices in lexical order. After A1, A2, and A3 are heavy R contracted, A4 is selected for heavy BD4 contraction (Figure 3.8b). Since heavy contracting A4 from the extended overlay graph maintained by Algorithm 6 would introduce the non-BD4-reachable edge (B1, A5), it is not heavy contracted (and marked red). Figures 3.8c-h show the state of the extended overlay graph maintained by Algorithm 6 after heavy BD4 contracting all vertices in each row.

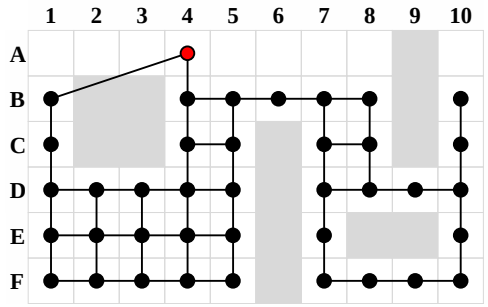
Unlike contractions and heavy contractions, the order of heavy R contractions affects the resulting graph, since heavy R contractions determine whether to heavy contract a vertex based on the current state of the extended overlay graph. Figure 3.9 shows an example of the resulting graph if the vertices are heavy R contracted in the reverse lexical order, rather than the lexical order. We discuss heavy R contraction orders for



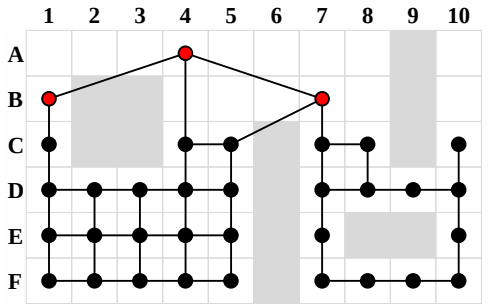
(a) Heavy BD4 contracting A1 heavy contracts A1 and adds the BD4-reachable edge (B1,A2).



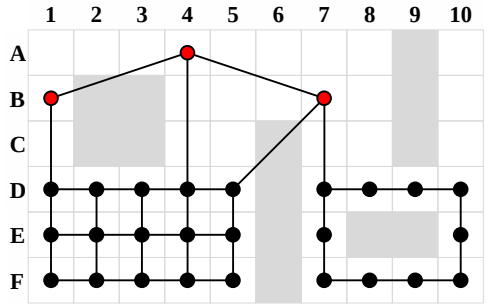
(b) Heavy-BD4 contracting A4 does not heavy contract A4 since the edge (B1,A5) is not BD4-reachable.



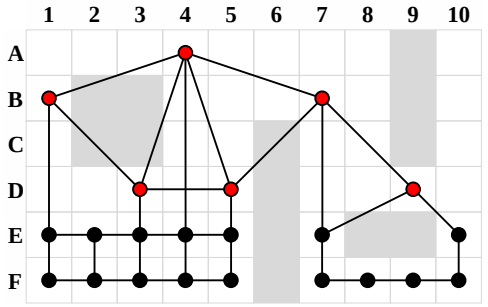
(c) After heavy BD4 contracting row A.



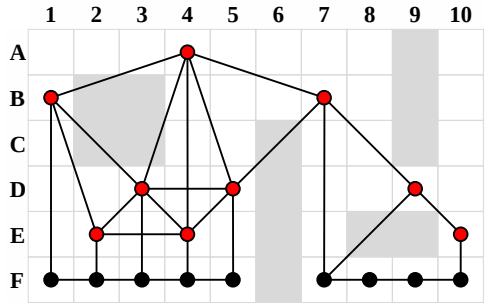
(d) After heavy BD4 contracting rows A-B.



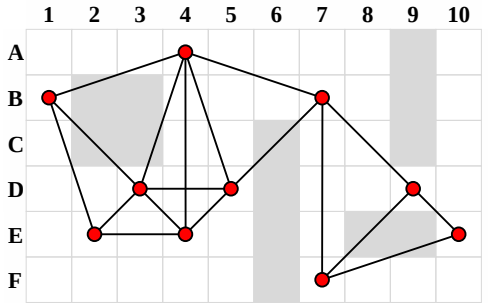
(e) After heavy BD4 contracting rows A-C.



(f) After heavy BD4 contracting rows A-D.

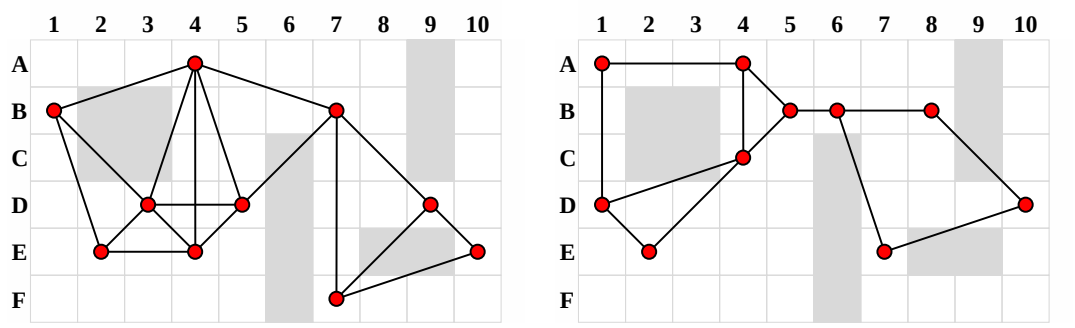


(g) After heavy BD4 contracting rows A-E.



(h) After heavy BD4 contracting rows A-F.

Figure 3.8: Constructing a BD4-SPC by heavy BD4 contractions. Heavy BD4 contraction does not heavy contract the vertices shown in red, since doing so would introduce non-BD4-reachable shortcut edges.



(a) After heavy BD4 contracting all vertices in lexical order.

(b) After heavy BD4 contracting all vertices in reverse-lexical order.

Figure 3.9: R-SPCs constructed with different orderings of heavy R contractions.

constructing subgoal graphs on state lattices in Section 4.6.5 and N-level subgoal graphs in Section 3.4.3.

We conclude this section by proving that the (R,T)-SPCs S constructed by Algorithm 6 are minimal. The proof relies on the observation that, if heavy R contracting a vertex at some point during the execution of Algorithm 6 does not heavy contract it, then heavy R contracting it afterwards cannot heavy contract it either. That is, no vertex can be removed from S while maintaining it being an (R,T)-SPC.

Lemma 3.15. *Algorithm 5 heavy contracts n if and only if $S \setminus \{n\}$ is an (R,T)-SPC.*

Proof.

1. Heavy R contract heavy contracts n if and only if all edges of $G_{S\setminus\{n\},T}$ are R-reachable (Algorithm 5).
2. $S \setminus \{n\}$ is an (R,T)-SPC if and only if all edges of $G_{S\setminus\{n\},T}$ are R-reachable (Theorem 3.12).
3. Therefore, heavy R contract heavy contracts n if and only if $S \setminus \{n\}$ is an (R,T)-SPC.

□

Theorem 3.16. *The set of vertices S returned by Algorithm 6 is a minimal (R,T)-SPC (that is, no $S' \subset S$ is an (R,T)-SPC).*

Proof.

1. Let S be the set of vertices returned by Algorithm 6.
2. Assume (for contradiction) that there exists $S' \subset S$ such that S' is an (R,T)-SPC.
3. Let $n \in S \setminus S'$. Such n exists (since $S' \subset S$).
4. Let S^n be the value of S directly before Algorithm 6 attempts to heavy R contract n (line 3).

5. n was not heavy contracted (since $n \in S$).
6. $S^n \setminus \{n\}$ is not an (R,T)-SPC (Lemma 3.15, since n was not heavy contracted).
7. $S \subseteq S^n$ (since further heavy R contractions can only remove vertices from S^n and not add them).
8. $S' \subset S^n$ (since $S' \subset S$ and $S \subseteq S^n$).
9. $S' \subseteq S^n \setminus \{n\}$ (since $S' \subset S^n$, $n \in S^n$, and $n \notin S'$).
10. $S^n \setminus \{n\}$ is an (R,T)-SPC (Lemma 3.11, since $S' \subseteq S^n \setminus \{n\}$ and S' is an (R,T)-SPC).
11. $\perp$ (since $S^n \setminus \{n\}$ is an (R,T)-SPC and $S^n \setminus \{n\}$ is not an (R,T)-SPC).

□

3.3.6 Incrementally Constructing an R-SPC

In this section, we discuss an alternative variant, called the *growing* variant, for constructing (R,T)-SPCs. Unlike the pruning variant, that starts with $S = T$ and removes vertices from S while maintaining that S is an (R,T)-SPC, the growing variant starts with $S = \emptyset$ and adds vertices to S until it becomes an (R,T)-SPC. While the growing variant does not guarantee the minimality of S, it typically constructs (R,T)-SPCs faster than the pruning variant. We use this variant for constructing subgoal graphs on state lattices (Section 4.6.5).

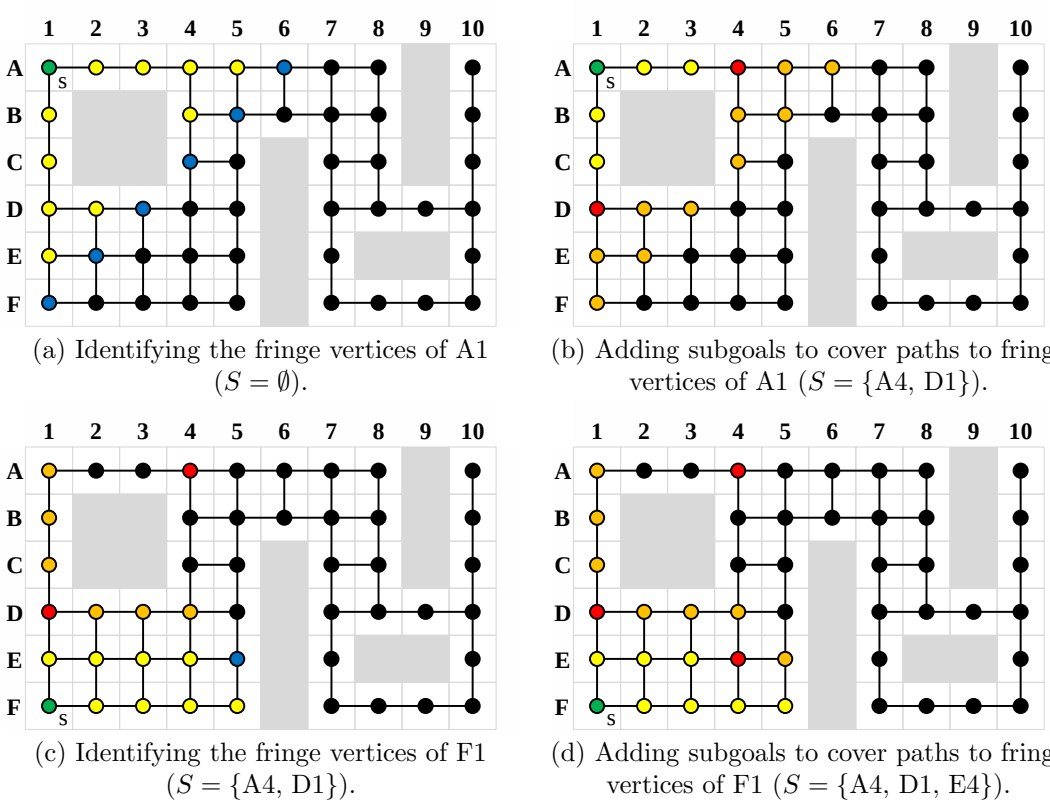
Algorithm 7 Grow (R,T)-SPC

Input: Overlay graph $G_T = (T, E', c')$ where T is a (R,T)-SPC, reachability relation R
Output: An (R,T)-SPC $S \subseteq T$

1: $S \leftarrow \emptyset$
2: **for all** $s \in T$, in some order **do**
3: $\quad F \leftarrow$ IdentifyFringeVertices(G_T, s, S)
4: $\quad S \leftarrow S \cup$ IdentifyCoveringSubgoals(G_T, s, F)
5: return S

Algorithm 7 outlines the growing variant of constructing an (R,T)-SPC S. Recall that an (R,T)-SPC S guarantees that, for every $s,t \in T$, if $(s,t) \notin R$, then $S \sqsubset (s,t)$ (Definition 3.10). To ensure that the set S of vertices it identifies is an (R,T)-SPC, Algorithm 7 starts with $S = \emptyset$ (line 1) and, for each $s \in T$ (line 2), adds vertices to S so that, for every $t \in T$, if $(s,t) \notin R$, then $S \sqsubset (s,t)$ (lines 3–4).

We first describe a hypothetical algorithm that performs this operation (lines 3–4): A modified Dijkstra search could be run from s on G to identify the set A of vertices $t \in T$ such that $(s,t) \notin R$ and $S \not\sqsubset (s,t)$, where the Dijkstra search maintains "covered" values, similar to the conservative variant of Direct-Connect (Algorithm 2), and terminates when its OPEN list is empty. Then, a set of vertices S' could be added to S to cover, for each $t \in A$, at least one shortest s-t path on G. S' could be identified heuristically, for instance



(a) Identifying the fringe vertices of A1 ($S = \emptyset$).

(b) Adding subgoals to cover paths to fringe vertices of A1 ($S = \{\text{A4, D1}\}$).

(c) Identifying the fringe vertices of F1 ($S = \{\text{A4, D1}\}$).

(d) Adding subgoals to cover paths to fringe vertices of F1 ($S = \{\text{A4, D1, E4}\}$).

Figure 3.10: Incrementally constructing a BD4-SPC. Red vertices: S. Yellow vertices: Direct-BD4-reachable from s. Orange vertices: A shortest path from s is covered by S. Blue vertices: Fringe vertex of s. Black vertices: Not generated by the search to identify fringe vertices of S.

by greedily selecting a vertex in the search tree rooted at s that covers shortest paths from s to most vertices in A.

Algorithm 7 performs this operation (lines 3–4) similarly to the hypothetical algorithm we have outlined above, but terminates its Dijkstra search early, after identifying the set F of *fringe vertices* of s (line 3), and identifies S' based on F instead of A (line 4). The set of fringe vertices of s (with respect to the current contents of S) is the set of vertices $F \subseteq A$ such that, for every $t \in F$, $A \not\sqsubseteq (s, t)$. For every vertex $t \in A \setminus F$, at least one shortest s-t path is covered by a vertex $n \in F$ (otherwise, $t \in F$). Therefore, adding subgoals to S to cover shortest paths to fringe vertices F is sufficient for covering shortest paths to all vertices in A, and, consequently, it is sufficient to identify F instead of A, and sufficient for identifying S' to cover shortest paths from s to vertices in F rather than to vertices in A. The Dijkstra search of the hypothetical algorithm that we have outlined above can be terminated early, after generating F, when its OPEN list contains only vertices that are "covered" (by the current contents of S) or not R-reachable from s. At this point, any vertex t that is not generated by the Dijkstra search cannot be a fringe vertex of s, since all shortest s-t paths on G are covered by at least one vertex in the OPEN list: either by a vertex that is already "covered", in which case t would also be

"covered" and thus $t \notin A \supseteq F$; or by a vertex that is in A, in which case $t \notin F$. F can then be determined as those vertices t in the search tree of the Dijkstra search such that: 1) t is not marked as "covered", 2) t is not R-reachable from s, and 3) the parent of t cannot be set, without changing the g-value of t, to a vertex that is not marked as "covered" and not R-reachable from s. We discuss how S' is selected from F in Section 4.6.5, when we use Algorithm 7 to construct subgoal graphs on state lattices.

Figure 3.10 shows an example of the operation of Algorithm 7 for constructing a BD4-SPC, that is, a (BD4, V)-SPC. In Figure 3.10a, $S = \emptyset$ and the Dijkstra search from $s =$ A1 terminates when its OPEN list contains only vertices that are not BD4-reachable from s. Coincidentally, these vertices all become the fringe vertices of s (with respect to $S = \emptyset$). In Figure 3.10b, A4 and D1 are (identified heuristically and) added as subgoals to S, so that at least one shortest path from s to each of its fringe vertices is covered. In Figure 3.10c, $S = \{$A4, D1$\}$, and the Dijkstra search from $s =$ F1 terminates when its OPEN list contains only vertices that are not BD4-reachable from s (E5) or already covered by S (A1 and D4); and E5 is identified as the only fringe vertex of s. In Figure 3.10d, E4 is added to S to cover a shortest s-E5 path. Algorithm 7 would continue this process by performing a Dijkstra search from all vertices $s \in V$, eventually constructing an S that is a BD4-SPC.

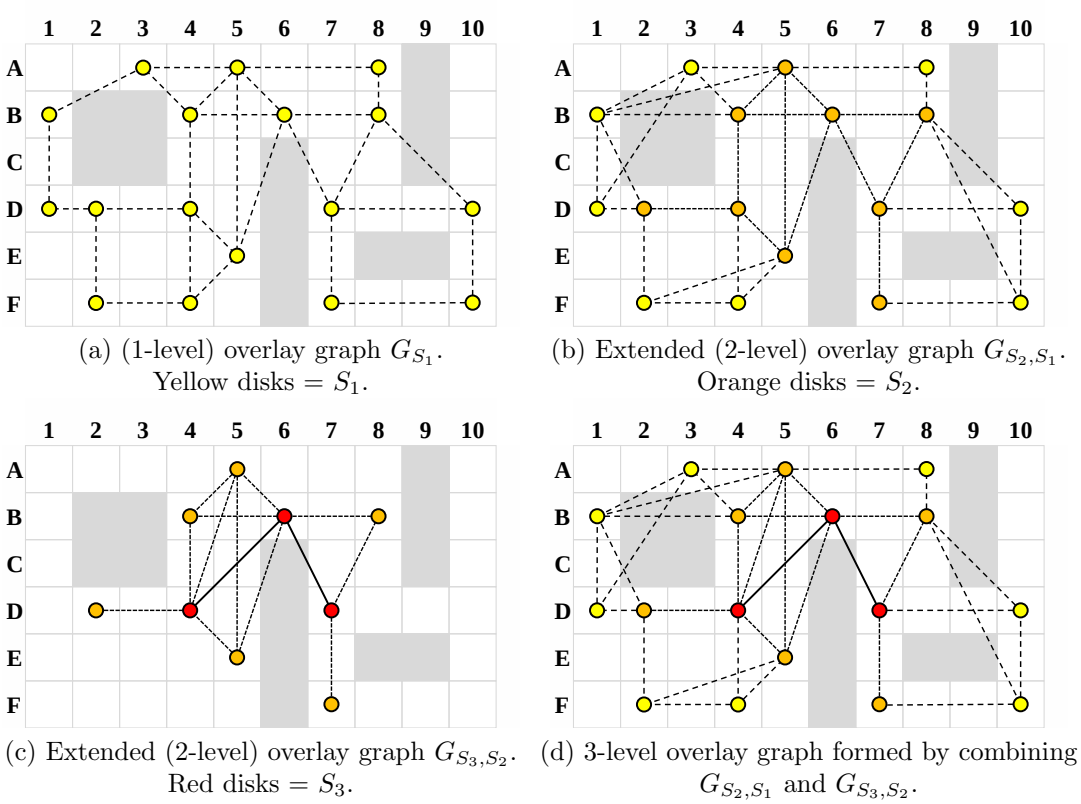
3.4 N-level Overlay and Subgoal Graphs

In this section, we introduce N-level overlay graphs that are formed by combining multiple extended overlay graphs. As proven in Section 3.2.3, extended overlay graphs contain, between every pair s and t of their vertices, an overlay s-t path with length $d(s,t)$. In this section, we prove that N-level overlay graphs contain, between every pair s and t of their vertices, an *arching* s-t path with length $d(s,t)$, which allows them to be searched for arching paths efficiently by using modified bidirectional Dijkstra searches. We introduce R N-level subgoal graphs as N-level overlay graphs with R-reachable edges only that are constructed on R subgoal graphs, describe contraction hierarchies as N-level overlay graphs without level edges, and introduce R contraction hierarchies as N-level overlay graphs that have R-reachable edges only and level edges only between vertices at the highest level of the hierarchy. We compare these hierarchies in more detail, both analytically and experimentally, on grid graphs in Chapter 5.

This section is organized as follows: In Section 3.4.1, we introduce N-level overlay graphs and arching paths, and prove that N-level overlay graphs contain, between every pair s and t of their vertices, an arching s-t path with length $d(s,t)$ In Section 3.4.2, we describe contraction hierarchies as a subclass of N-level overlay graphs, and introduce R N-level subgoal graphs and R contraction hierarchies as two other subclasses. In Sections 3.4.3 and 3.4.4, we discuss how these hierarchies can be constructed and be used to answer path queries optimally, respectively.

3.4.1 N-Level Overlay Graphs

N-level overlay graphs are constructed by assigning levels to the vertices of an overlay graph G_S and adding extra shortcut edges to guarantee that, for every $s,t \in S$, an



(a) (1-level) overlay graph G_{S_1}. Yellow disks = S_1.

(b) Extended (2-level) overlay graph G_{S_2,S_1}. Orange disks = S_2.

(c) Extended (2-level) overlay graph G_{S_3,S_2}. Red disks = S_3.

(d) 3-level overlay graph formed by combining G_{S_2,S_1} and G_{S_3,S_2}.

Figure 3.11: N-level overlay graphs.

arching s-t path with length $d(s,t)$ exists on the resulting hierarchy. An overlay graph G_{S_1} can be considered to be a 1-level overlay graph, where every vertex $n \in S_1$ has the same level (Figure 3.11a). An extended overlay graph G_{S_2,S_1} can be considered to be a 2-level overlay graph, where the vertices in S_2 can be considered to be level 2 vertices, whereas the vertices in $S_1 \setminus S_2$ can be considered as level 1 vertices (Figure 3.11b). Recall that, for every $s,t \in S_1$, G_{S_2,S_1} is guaranteed to contain an overlay s-t path with length $d(s,t)$ (Lemma 3.2) that uses only vertices in S_2, with the possible exception of s and t (Definition 3.5). Therefore, to find a shortest s-t path on G_{S_2,S_1}, one only needs to search over overlay paths (since at least one overlay s-t path is guaranteed to be a shortest s-t path), and can ignore any vertices in $S_1 \setminus S_2$ except for s and t.

We can take this idea one step further: For some $S_3 \subseteq S_2$, consider the extended overlay graph G_{S_3,S_2} (Figure 3.11c) which, for every $u,v \in S_2$, is guaranteed to contain a u-v path with length $d(u,v)$ that uses only vertices in S_3 on G_{S_3,S_2}, with the possible exception of u and v. The combination of G_{S_3,S_2} with G_{S_2,S_1} can be considered as a 3-level overlay graph (Figure 3.11d), which is guaranteed to contain, for every $s,t \in S_1$, an s-t path $\pi = \langle p_0, \ldots, p_k \rangle$ with $l(\pi) = d(s,t)$, $p_0, p_k \in S_1$, $p_1, p_{k-1} \in S_2$, and $p_2, \ldots, p_{k-2} \in S_3$. To find a shortest s-t path on the 3-level overlay graph, one needs to look only for such a path, and 1) can ignore any vertices in $S_1 \setminus S_2$ as long as they are not s or t, and 2) can ignore any vertices in $S_2 \setminus S_3$ as long as they are not s or t, or share an edge with s or t.

More generally, an N-level overlay graph can be considered as a combination of $N-1$ extended overlay graphs $G_{S_1,S_2}, \ldots, G_{S_{N-1},S_N}$, for some $V \supseteq S_1 \supseteq \cdots \supseteq S_N \neq \emptyset$, and is guaranteed to contain, for every $s,t \in S_1$, an *arching s-t* path with length $d(s,t)$. We succinctly represent the sequence of vertex sets $V \supseteq S_1, \ldots, \supseteq S_N \neq \emptyset$ by using a *level function* $\mathbb{L}$, and use $G_{\mathbb{L}}$ to denote an N-level overlay graph constructed with respect to $\mathbb{L}$. We formally define level functions, N-level overlay graphs, and arching paths in Definitions 3.11, 3.12, and 3.13, respectively.

Definition 3.11 (Level function)**.** *A level function is a function $\mathbb{L} : V \to \mathbb{Z}_{\geq 0}$ that assigns a non-negative level to each vertex, where $\max(\mathbb{L}) = \max_{n \in V} \mathbb{L}(n)$ is the maximum level of $\mathbb{L}$. The level of a path π with respect to $\mathbb{L}$ is $\mathbb{L}(\pi) = \max_{n \in \pi} \mathbb{L}(n)$.*

Definition 3.12 (N-Level overlay graph)**.** *Given a level function $\mathbb{L}$ with $\max(\mathbb{L}) = N$:*

- *$\forall i = 1, \ldots, N$, let $S_i = \{u \in V : \mathbb{L}(u) \geq i\}$ be the set of level $\geq i$ vertices.*
- *Let E_1 be the edges of G_{S_1}.*
- *$\forall i = 2, \ldots, N$, let E_i be the edges of $G_{S_i,S_{i-1}}$.*

The N-level overlay graph induced by $\mathbb{L}$ on G is the graph $G_{\mathbb{L}} = (S, E', c')$, where:

1. *$S = S_1$;*
2. *$E' = E_1 \cup \cdots \cup E_N$; and*
3. *$\forall (s,t) \in E'$, $c'(s,t) = d(s,t)$.*

G_{S_1} and G_{S_N} are referred to as the base *and* core *(graphs) of $G_{\mathbb{L}}$, respectively.*

In this dissertation, we construct hierarchies (N-level overlay graphs and their subclasses that we introduce in the next section) $G_{\mathbb{L}}$ on either the input graph G or a subgoal graph G_S. Definitions 3.11 and 3.12 allow us to distinguish between these two cases: If $G_{\mathbb{L}}$ is constructed on G, then all vertices $n \in V$ are assigned a level $\mathbb{L}(n) \geq 1$. If $G_{\mathbb{L}}$ is constructed on G_S, then a vertex $n \in V$ is assigned a level $\mathbb{L}(n) \geq 1$ if $n \in S$, or level $\mathbb{L}(n) = 0$ otherwise. We refer to G or G_S as the *base (graph)* of the hierarchy. We refer to its highest level vertices and edges as *core* vertices and *core* edges, respectively. The core vertices and edges of a hierarchy form the *core (graph)* of the hierarchy. By definition, all edges in the base graph are contained in the hierarchy.

Definition 3.13 (Arching path)**.** *Given an N-level overlay graph $G_{\mathbb{L}} = (S, E', c')$, an edge $(u,v) \in E'$ is level if and only if $\mathbb{L}(u) = \mathbb{L}(v)$, upward if and only if $\mathbb{L}(u) < \mathbb{L}(v)$, and downward if and only if $\mathbb{L}(u) > \mathbb{L}(v)$.*

A path $\pi = \langle v_0, \ldots, v_k \rangle$ on $G_{\mathbb{L}}$ is level if and only if all its edges are level, upward if and only if all its edges are upward, downward if and only if all its edges are downward. π is an arching path if and only if there exists $0 \leq i \leq j \leq k$, such that:

1. *$\pi_{\uparrow} = \langle v_0, \ldots, v_i \rangle$ is upward;*
2. *$\pi_{\leftrightarrow} = \langle v_i, \ldots, v_j \rangle$ is level;*

3. $\pi_\downarrow = \langle v_j, \ldots, v_k \rangle$ *is downward; and*

4. *if* $\mathbb{L}(v_i) < \max(\mathbb{L})$*, then* $j = i$ *or* $j = i + 1$*.*

$\pi_\uparrow$*,* $\pi_\leftrightarrow$*,* $\pi_\downarrow$ *are called the upward, level, and downward parts of* π*, respectively.*

Informally, an arching path is a path that can be split into three parts: The *upward* part visits vertices in ascending order of levels, the *level* part visits vertices only on the same level, and the *downward* part visits vertices in descending order of levels. The ascending part of an arching path must be its prefix, the descending part must be its suffix, and the level part may contain at most one non-core edge. For instance, the path ⟨D1, D2, D4, B6, D7, D10⟩ on the 3-level overlay graph shown in Figure 3.11d is an arching path, since it can be split into the upward part ⟨D1, D2, D4⟩, the level part ⟨D4, B6, D7⟩, and the downward part ⟨D7,D10⟩, where its level part uses core edges and, therefore, can have any number of them. The path ⟨D1, B1, A3⟩ can also be split into the upward path ⟨D1⟩, the level part ⟨D1, B1, A3⟩ and the downward part ⟨A3⟩. However, it is not an arching path since its level part contains two edges between level 1 (non-core) vertices. The path ⟨F2, F4, E5⟩ is not an arching path since the upward part ⟨F4, E5⟩ is not its prefix.

We conclude this section by proving that, for every pair of vertices s and t in an N-level overlay graph $G_\mathbb{L}$, an arching s-t path exists on $G_\mathbb{L}$ with length $d(s,t)$. The proof considers the series of 1-level, ..., N-level overlay graphs $G_{\mathbb{L}_1}, \ldots, G_{\mathbb{L}_N}$, where the level function $\mathbb{L}_i$ considers vertices n with $\mathbb{L}(n) \geq i$ as level i vertices; and shows, by induction, that an arching s-t path with length $d(s,t)$ exists on each of these hierarchies. Namely, an arching s-t path π_{i+1} with length $d(s,t)$ on $G_{\mathbb{L}_{i+1}}$ can be constructed from an arching s-t path with length $d(s,t)$ on $G_{\mathbb{L}_i}$, by replacing its level part with a corresponding overlay path on the extended overlay graph that extends $G_{\mathbb{L}_i}$ to $G_{\mathbb{L}_{i+1}}$. We omit the proof that the s-t distance on $G_\mathbb{L}$ is no shorter than the s-t distance on G, which trivially follows from the fact that edge lengths on $G_\mathbb{L}$ are equal to distances on G (Definition 3.12).

Theorem 3.17. *Let* $G_\mathbb{L} = (S, E', c')$*. For every* $s, t \in S$*, there exists an arching* s-t *path* π *on* $G_\mathbb{L}$ *with* $l(\pi) = d(s,t)$*.*

Proof.

1. Let $G_\mathbb{L} = (S, E', c')$.
2. Let $s, t \in S$.
3. For $i = 1, \ldots, N$, let $S_i = \{n \in V : \mathbb{L}(n) \geq i\}$.
4. For $i = 1, \ldots, N$, let $\mathbb{L}_i$ be a level function such that, $\forall n \in V$, $\mathbb{L}_i(n) = \max(\mathbb{L}(n), i)$.
5. For $i = 1, \ldots, N$, let $\mathcal{S}_i$ be the statement "there exists an s-t arching path π on $G_{\mathbb{L}_i}$ with $l(\pi) = d(s,t)$."
6. For $i = 1, \ldots, N$, $\mathcal{S}_i$ is true (by induction on $i = 1, \ldots, N-1$):
   6.1. (Base case): $\mathcal{S}_1$ is true:

6.1.1. $G_{\mathbb{L}_1} = G_{S_1}$ (Definition 3.12).
6.1.2. There exists an s-t path π on G_{S_1} with $l(\pi) = d(s,t)$ (Lemma 3.2).
6.1.3. $\forall p \in \pi$, $\mathbb{L}(p) \geq 1$ (since $p \in S_1$).
6.1.4. $\forall p \in \pi$, $\mathbb{L}_1(p) = 1$ (since $\mathbb{L}(p) \geq 1$).
6.1.5. Therefore, π is an arching path on $G_{\mathbb{L}_1}$ (Definition 3.13).
6.2. (Induction step): If $\mathcal{S}_i$ is true, then $\mathcal{S}_{i+1}$ is true:
6.2.1. Assume $\mathcal{S}_i$ is true.
6.2.2. Let π be an s-t arching path on $G_{\mathbb{L}_i}$ with $l(\pi) = d(s,t)$. Such π exists (since $\mathcal{S}_i$ is true).
6.2.3. $G_{\mathbb{L}_{i+1}}$ has all the edges of $G_{\mathbb{L}_i}$ (Definition 3.12).
6.2.4. Let $\pi_\uparrow$, $\pi_\leftrightarrow = \langle p_0, \ldots, p_j \rangle$, $\pi_\downarrow$ be the upward, level, and downward parts of π, respectively (Definition 3.13, since π is an arching path).
6.2.5. If $j < 2$ then $\mathcal{S}_{i+1}$ is true (since π is an arching path on $G_{\mathbb{L}_{i+1}}$):
6.2.5.1. Assume $j < 2$.
6.2.5.2. π is a path on $G_{\mathbb{L}_{i+1}}$ (since π is a path on $G_{\mathbb{L}_i}$, and $G_{\mathbb{L}_{i+1}}$ has all the edges of $G_{\mathbb{L}_i}$).
6.2.5.3. π is an arching path on $G_{\mathbb{L}_{i+1}}$ (Definition 3.13, since π is a path on $G_{\mathbb{L}_{i+1}}$ with the upward part $\pi_\uparrow$, level part $\pi_\leftrightarrow = \langle p_0, \ldots, p_j \rangle$ with $j < 2$, and the downward part $\pi_\downarrow$).
6.2.6. If $j \geq 2$, then $\mathcal{S}_{i+1}$ is true (since we can replace the level part $\pi_\leftrightarrow$ of π with a shortest overlay path π' on G_{S_{i+1},S_i} to get an arching path π'' on $G_{\mathbb{L}_{i+1}}$ with $l(\pi'') = d(s,t)$):
6.2.6.1. Assume $j \geq 2$.
6.2.6.2. $\mathbb{L}_i(p_0) = \mathbb{L}_i(p_j) = i$ (Definition 3.13, since $\langle p_0, \ldots, p_j \rangle$ is the level-part of an arching path on $G_{\mathbb{L}_i}$ with $j \geq 2$).
6.2.6.3. $\mathbb{L}(p_0) \geq i$, $\mathbb{L}(p_j) \geq i$ (since $\mathbb{L}_i(p_0) = \mathbb{L}_i(p_j) = i$).
6.2.6.4. $p_0, p_j \in S_i$ (since $\mathbb{L}(p_0) \geq i$, $\mathbb{L}(p_j) \geq i$).
6.2.6.5. Let $\pi' = \langle p'_0, \ldots, p'_k \rangle$ be a overlay p_0-p_j path on G_{S_{i+1},S_i} with $l(\pi) = d(p_0, p_j)$. Such π' exists (Lemma 3.2, since $p_0, p_j \in S_i$).
6.2.6.6. Let $\pi'' = \pi_\uparrow \cdot \pi' \cdot \pi_\downarrow$.
6.2.6.7. G_{L_i+1} contains all the edges of G_{S_{i+1},S_i} (Definition 3.12).
6.2.6.8. Therefore, π' is a path on $G_{\mathbb{L}_{i+1}}$.
6.2.6.9. Therefore, π'' is a path on $G_{\mathbb{L}_{i+1}}$.
6.2.6.10. $l(\pi'') = l(\pi) = d(s,t)$ (since π' replaces $\pi_\leftrightarrow$ in π as a shortest p_0-p_j path).
6.2.6.11. $p'_1, \ldots, p'_{k-1} \in S_{i+1}$ (Definition 3.5, since π' is an overlay path on G_{S_{i+1},S_i}).
6.2.6.12. $\mathbb{L}_{i+1}(p'_1) = \cdots = \mathbb{L}_{i+1}(p'_{k-1}) = i + 1$ (since $p'_1, \ldots, p'_{k-1} \in S_{i+1}$).
6.2.6.13. $i \leq \mathbb{L}(p'_0)$ and $i \leq \mathbb{L}(p'_k)$ (since $p'_0 = p_0$, $p'_k = p_j \in S_i$).
6.2.6.14. $i \leq \mathbb{L}_{i+1}(p'_0) \leq i + 1$ and $i \leq \mathbb{L}_{i+1}(p'_k) \leq i + 1$.
6.2.6.15. π'' is an arching path (Definition 3.13, further details omitted).

□

3.4.2 Subclasses of N-Level Overlay Graphs

In this section, we describe contraction hierarchies as a subclass of N-level overlay graphs, and introduce two new subclasses, R N-level subgoal graphs, and R contraction hierarchies. Definition 3.14 formally defines these three subclasses, and Figure 3.12 shows an example.

Definition 3.14 (Contraction hierarchy, R contraction hierarchy, R N-level subgoal graph)**.** *Let $G_{\mathbb{L}} = (S, E', c')$ be an N-level overlay graph. Let R be a reachability relation.*

- *$G_{\mathbb{L}}$ is a* contraction hierarchy *if and only if, $\forall (u, v) \in E'$, $\mathbb{L}(u) \neq \mathbb{L}(v)$.*
- *$G_{\mathbb{L}}$ is an R* contraction hierarchy *if and only if, $\forall (u, v) \in E'$, ($(u, v) \in R$ and, if $\mathbb{L}(u) = \mathbb{L}(v)$, then $\mathbb{L}(u) = \max(\mathbb{L})$).*
- *$G_{\mathbb{L}}$ is an R N-level subgoal graph if and only if, $\forall (u, v) \in E'$, $(u, v) \in R$.*

Contraction hierarchies, as described in the literature (Geisberger et al., 2008), are constructed by contracting the vertices of a graph one by one using the vertex contraction operation discussed in Section 3.2.6. Shortcut edges identified by contractions are added to the graph, and each vertex is assigned a level based on the order of contractions. We discuss this construction scheme in more detail in Section 3.4.3. The resulting contraction hierarchy has the property that, between every pair of vertices s and t, there exists an *up-down s-t* path with length $d(s, t)$, that is, an arching path with only the upward and downward parts. We discuss how this property can be exploited during searches in Section 3.4.4.2.

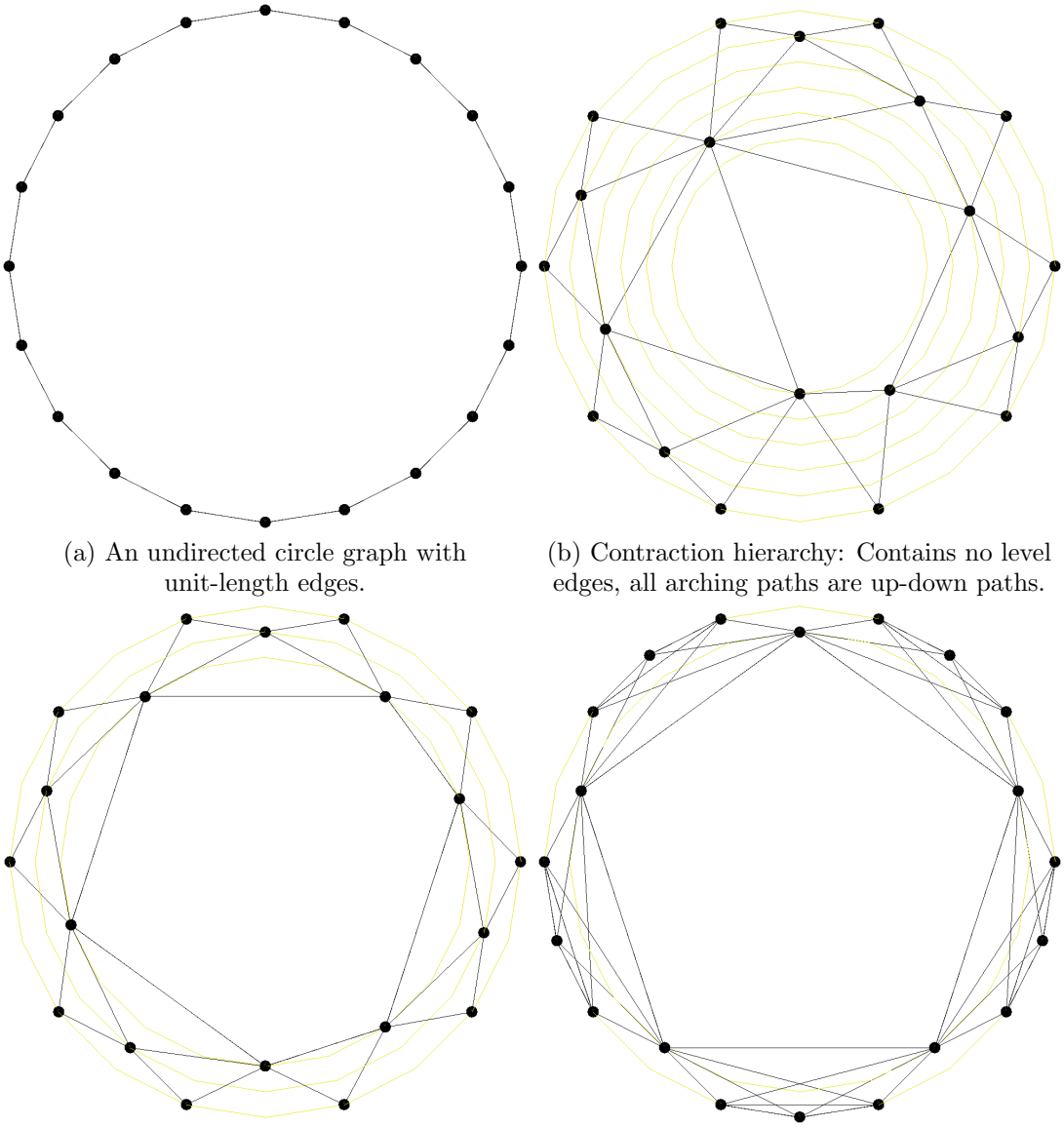
Observe that, if an N-level overlay graph $G_{\mathbb{L}} = (S, E', c')$ does not have any level edges, then it can be considered as a contraction hierarchy (Definition 3.14): For every $s, t \in S$, there exists an arching s-t path π on $G_{\mathbb{L}}$ with length $d(s, t)$ (Theorem 3.17). Since $G_{\mathbb{L}}$ does not have level edges, π cannot have level edges and, therefore, π is an up-down path. We therefore define contraction hierarchies as a subclass of N-level overlay graphs, which do not contain level edges.

R N-level subgoal graphs are N-level overlay graphs constructed with respect to a reachability relation R, and have R-reachable edges only. Since N-level overlay graphs include all edges of their base graphs (Section 3.4.1), it is not possible to construct R N-level overlay graphs on graphs with non-R-reachable edges. We therefore construct R N-level subgoal graphs on R subgoal graphs in this dissertation, which are guaranteed to have R-reachable edges only.

R contraction hierarchies can be considered to be an intermediate hierarchy between contraction hierarchies and R N-level subgoal graphs. Similar to contraction hierarchies, they do not have level edges between non-core vertices. Similar to R N-level subgoal graphs, they have R-reachable edges only. We allow R contraction hierarchies to have level edges between core vertices because it might not be possible to construct contraction hierarchies with R-reachable edges only. We discuss this in more detail in Section 3.4.3.

3.4.3 Constructing N-Level Overlay Graphs

In this section, we discuss how N-level overlay graphs and their different subclasses can be constructed by using different types of contractions. Namely, the (regular) contractions,



(a) An undirected circle graph with unit-length edges.

(b) Contraction hierarchy: Contains no level edges, all arching paths are up-down paths.

(c) BD4 contraction hierarchy: BD4-reachable edges only, level edges only in the core.

(d) BD4 2-level subgoal graph: BD4-reachable edges only. (May contain level edges anywhere in the hierarchy).

Figure 3.12: Subclasses of N-level overlay graphs. The yellow concentric circles denote the levels of vertices, where vertices in the outermost circle have level 1.

Shortcut edges \ Operates on (graph)	Overlay	Extended overlay
Can introduce non-R-reachable shortcut edges	Contract	Heavy contract
Introduces R-reachable shortcut edges only	R contract	Heavy R contract

Table 3.2: Variants of contractions.

Hierarchy	Paths and edges	Construction
N-level overlay graph	Preserves distances in arching paths	Heavy contractions
R N-level subgoal graph	Preserves distances in arching paths All edges are R-reachable	Heavy R contractions
Contraction hierarchy	Preserves distances in up-down paths No level edges	Contractions
R contraction hierarchy	Preserves distances in arching paths Level edges only in the core All edges are R-reachable	R contractions

Table 3.3: Classes of hierarchies and their associated contraction variants for constructing them.

heavy contractions, and heavy R contractions discussed in Sections 3.2.6 and 3.3.5, and a new type of contraction, called R contractions, that we introduce in this section.

Recall that contracting a vertex $n \in S$ from an overlay graph G_S produces the overlay graph $G_{S\setminus\{n\}}$, and heavy contracting a vertex $n \in S$ from an extended overlay graph $G_{S,T}$ produces the extended overlay graph $G_{S\setminus\{n\},T}$ (Section 3.2.6). Both contractions and heavy contractions can introduce new shortcut edges, and contractions (but not heavy contractions) remove incident edges of the contracted vertex from the graph. Also recall that heavy R contracting a vertex $n \in S$ from $G_{S,T}$ heavy contracts n from $G_{S,T}$ if and only if doing so introduces R-reachable shortcut edges only (Section 3.3.5). Our new contraction variant, called R contractions, contracts n from an overlay graph G_S if and only if doing so introduces R-reachable shortcut edges only. Table 3.2 summarizes these four variants of contractions, which differ in whether they operate on overlay or extended overlay graphs, and whether they introduce R-reachable shortcut edges only.

Table 3.3 summarizes the four classes of hierarchies and the type of contractions used for constructing them. We discuss each case below. For constructing N-level overlay graphs $G_{\mathbb{L}}$, we assume that a level function $\mathbb{L}$ is given, since any $\mathbb{L}$ uniquely defines an N-level overlay graph. For constructing subclasses of N-level overlay graphs, we outline construction methods for identifying level functions $\mathbb{L}$ that satisfy their respective constraints.

3.4.3.1 N-Level Overlay Graphs and Heavy Contractions

Suppose that we are given a level function $\mathbb{L}$ and want to construct the N-level overlay graph $G_{\mathbb{L}}$. Let S_i denote the level $\geq i$ vertices of $G_{\mathbb{L}}$. As discussed in Section 3.4.1, $G_{\mathbb{L}}$ is a combination of the extended overlay graphs $G_{S_N,S_{N-1}}, \ldots, G_{S_2,S_1}$, and, as discussed in Section 3.2.6, we can construct each extended overlay graph G_{S_{i+1},S_i} by heavy contracting

all vertices $n \in S_i \setminus S_{i+1}$ from the overlay graph $G_{S_i} = G_{S_i,S_i}$. Therefore, N-level overlay graphs can be constructed using heavy contractions.

The overlay graph $G_{S_{i+1}}$ is the core of the extended overlay graph G_{S_{i+1},S_i}. This allows us to construct the constituent extended overlay graphs of $G_{\mathbb{L}}$ in the following order: First, we construct G_{S_2,S_1} from the overlay graph G_{S_1}. Then, we construct the extended overlay graph G_{S_3,S_2} from the core G_{S_2} of G_{S_2,S_1}, and repeat until $G_{S_N,S_{N-1}}$ is constructed. We discuss the construction of the subclasses of N-level overlay graphs within this framework.

3.4.3.2 R N-level Subgoal Graphs and Heavy R Contractions

Suppose that we are given a reachability relation R, and an overlay graph G_S with R reachable edges only, and we want to construct an R N-level overlay graph $G_{\mathbb{L}}$ on G_S. We can use the same framework outlined above for construction, but also identify $\mathbb{L}$ during the construction. As discussed in Section 3.3.5, heavy R contracting all vertices from the overlay graph $G_S = G_{S,S}$ results in an extended overlay graph $G_{S',S}$ with only R-reachable edges. We can use this method to identify S_2 from S_1, S_3 from S_2, and so on, until we identify an S_{k+1} with $S_{k+1} = S_k$ or $S_{k+1} = \emptyset$. The sequence $S_1, \ldots, S_k$ identifies a level function $\mathbb{L}$ (with $\max(\mathbb{L}) = k$) and, since each G_{S_{i+1},S_i} contains only R-reachable edges, $G_{\mathbb{L}}$ is an R N-level subgoal graph.

3.4.3.3 Contraction Hierarchies and Contractions

Suppose that we are given an overlay graph G_S and want to construct a contraction hierarchy (an N-level overlay graph with no level edges) $G_{\mathbb{L}}$. It is easy to identify a level function $\mathbb{L}$ such that $G_{\mathbb{L}}$ is a contraction hierarchy: If no two vertices have the same level with respect to $\mathbb{L}$, then $G_{\mathbb{L}}$ cannot have level edges.[3] We can therefore use the level function in the construction schema discussed for N-level overlay graphs, and construct a contraction hierarchy using *heavy* contractions.

Consider the construction of the extended overlay graph G_{S_{i+1},S_i} using *heavy* contractions as discussed above. Since no two vertices have the same level with respect to $\mathbb{L}$, there is a unique level i vertex n, where $S_i \setminus S_{i+1} = \{n\}$. Therefore, G_{S_{i+1},S_i} can be constructed with a single heavy contraction from G_{S_i}. As we have discussed in Section 3.2.6, *heavy* contracting a single vertex n from an overlay graph performs the same witness searches and adds the same set of shortcut edges as contracting n does. Therefore, each *heavy* contraction performed during the construction of a contraction hierarchy can be considered to be a (regular) contraction.

The level function that we have described above can be considered to be the *contraction order*, where vertices with higher levels are contracted after vertices with lower levels. Geisberger et al. note that searches over contraction hierarchies constructed using different contraction orders can have significantly different execution times, and suggest various heuristics for determining good contraction orders. Although an overview of these

[3] As we discuss later in this section, it is not necessarily the case that each vertex in a contraction hierarchy has a distinct level.

heuristics is beyond the scope of this dissertation, we outline the heuristic used in the contraction hierarchy entry in the Grid-Based Path-Planning Competition, which determines the next vertex n to contract from an overlay graph G_{S_i} as the vertex with the minimum *importance* $I(n)$ (Sturtevant et al., 2015). This method of determining contraction orders is referred to as an "online" contraction order, since the contraction order is determined during the construction of a contraction hierarchy, by updating the importance $I(n)$ of vertices after each contraction. The following expression summarizes the various factors that $I(n)$ considers when estimating the importance of vertices:

$$I(n) = L(n) + \frac{|A(n)|}{|D(n)|} + \frac{\sum_{(u,v)\in A(n)} h(u,v)}{\sum_{(u,v)\in D(n)} h(u,v)}.$$

We now explain the various terms that appear in this expression:

- $L(n)$ can be considered to be the "level" of n. Our suggestion of using a level function $\mathbb{L}$ that assigns a unique level to each vertex can be considered to be a simplification of how levels are assigned to vertices in contraction hierarchies. As Definition 3.14 suggests, two vertices in a contraction hierarchy can have the same level as long as no two vertices of the same level share an edge. A different way to assign levels L (to distinguish it from $\mathbb{L}$, that we have discussed above) to vertices is as follows: Initially, for every vertex n, $L(n) = 1$. When a vertex u is contracted from the overlay graph G_{S_j}, for every neighbor v of u in G_{S_j}, $L(v)$ is updated to $\max(L(v), L(u) + 1)$. This ensures that, when a vertex is contracted, its neighbors that have already been contracted have levels strictly lower than $L(n)$, and its neighbors that have not yet been contracted have a level strictly higher than $L(n)$, thereby ensuring that the resulting contraction hierarchy has no level edges. $I(n)$ considers vertices with lower levels to be less "important" and suggests contracting them first. This helps with contracting vertices more "uniformly", which has been shown to improve query times (Geisberger et al., 2008).

- $A(n)$ is the set of shortcut edges added when contracting n from G_{S_i}, and $D(n)$ is the set of edges (or shortcut edges) removed from G_{S_i}. $I(n)$ considers vertices whose contraction removes more shortcut edges than it adds to be less "important" and suggests contracting them first. This helps to add fewer shortcut edges to contraction hierarchies, which has been shown to improve query times (Geisberger et al., 2008). As a further benefit, the remaining "core" $G_{S_{i+1}}$ after contracting n from G_{S_i} is likely to contain fewer edges than it would have if another vertex were contracted, which can result in subsequent contractions requiring fewer witness searches and, therefore, less time.

- $h(u, v)$ is the number of edges that corresponds to a shortest u-v path on G, and is used to denote the "hop-length" of a shortcut edge (u, v). Recall that each shortcut edge (u, v) introduced during the contraction of a vertex m is the combination of the edges (or shortcut edges) (u, m) and (m, v). Therefore, during the construction of a contraction hierarchy, $h(u, v)$ can be maintained for all edges (or shortcut edges) as follows: For every edge (u, v) of G_S, assign $h(u, v) = 1$. When a shortcut edge (u, v) is introduced as a combination of the edges (or shortcut edges) (u, m) and

(m, v), assign $h(u, v) = h(u, m) + h(m, v)$. $I(n)$ considers vertices whose contraction removes shortcut edges with a higher sum of hops than it adds to be less "important" and suggests contracting them first. This helps with contracting vertices more uniformly, as well as helps add fewer shortcut edges to contraction hierarchies.

The contraction hierarchy entry in the Grid-Based Path-Planning Competition computes $I(n)$ for each vertex in G_S (by simulating its contraction from G_S) before determining the first vertex to contract and, after contracting a vertex u, recomputes $I(v)$ for every neighbor v of u that has not yet been contracted. This is considered to be a "comparatively slow but high-quality bottom-up ordering approach" (Sturtevant et al., 2015). In our experiments on grid graphs in Section 5.5, we use this ordering for determining the contraction orders during the construction of the other classes of hierarchies as well.

3.4.3.4 R Contraction Hierarchies and R Contractions

Suppose that we are given a reachability relation R and an overlay graph G_S with R-reachable edges only and want to construct an R contraction hierarchy $G_{\mathbb{L}}$ on G_S. We can use the same construction method for constructing contraction hierarchies, but avoid performing any contractions that introduce non-R-reachable edges (that is, performing R contractions only), which guarantees that the resulting hierarchy has R-reachable edges only. This hierarchy can have an "uncontracted core", where contracting any of the vertices from the core would introduce a non-R-reachable edge (Figure 3.12c shows an example). When constructing R contraction hierarchies, we set the "importance" $I(n)$ of a vertex n to infinity if contracting it would introduce a non-R-reachable edge.

3.4.4 Answering Queries Using N-Level Overlay Graphs

N-level overlay graphs can be used to answer path queries optimally in the same way that overlay graphs can be used to answer path queries optimally, by using the Connect-Search-Refine algorithm discussed in Section 3.2.4. In this section, we discuss how the start and goal vertices can be connected to hierarchies as level 0 vertices, describe the bidirectional search algorithm used for searching contraction hierarchies for shortest up-down paths (Geisberger et al., 2008) and how it can be extended for searching the other hierarchies for shortest arching paths, and describe the *unpacking* algorithm for refining up-down paths into paths on the base graph (Geisberger et al., 2008). Table 3.4 provides a summary of how the search and refinement phases of answering queries differ when using different classes of hierarchies.

3.4.4.1 Connection

As mentioned in Section 3.4.1, we construct hierarchies on subgoal graphs or G in this dissertation. The connection phases of queries can be skipped for hierarchies constructed on G. For hierarchies $G_{\mathbb{L}}$ constructed on a subgoal graph G_S, the start and goal vertices can be connected to $G_{\mathbb{L}}$ in the same way how they can be connected to G_S, by using an R-connect algorithm. If the start and goal vertices do not already appear as vertices in $G_{\mathbb{L}}$, we assign them a level of 0, which ensures that the resulting *query* hierarchy has

Hierarchy	Forward search	Refine
N-level overlay graph	Constructs the upward and level part Generates successors using upward and core level edges	Unpack
R N-level subgoal graph	Generates "stalled" successors using non-core-level edges	R-refine
Contraction hierarchy	Constructs the upward part Generates successors using upward edges	Unpack
R contraction hierarchy	Constructs the upward and (core) level part Generates successors using upward and level edges	R-refine

Table 3.4: Answering queries using different subclasses of N-level overlay graphs. All hierarchies are searched with bidirectional Dijkstra (or A*) searches, where the backward search constructs the downward part of an arching path.

an arching s-t path with length $d(s,t)$. To see why this is the case, consider combining $G_{\mathbb{L}}$ with the extended overlay graph $G_{S,V}$.[4] By Theorem 3.17, the resulting hierarchy is guaranteed to contain an arching s-t path π with length $d(s,t)$. Figure 3.13c shows an example, where t is connected to $G_{\mathbb{L}}$ as a level 0 vertex.

It could be the case that $\pi = \langle s,t \rangle$, where the edge (s,t) is level and non-core. We assume that the search algorithm we describe below checks for this case and returns the solution $\pi = \langle s,t \rangle$ immediately, to ensure that searches over contraction hierarchies can look for up-down paths only.

3.4.4.2 Search

The bidirectional search algorithm used for searching contraction hierarchies differs from (regular) bidirectional Dijkstra searches in three ways: 1) The forward search uses only upward edges and the backward search uses only downward edges, 2) the termination criterion is different, and 3) it uses stall-on-demand. In this section, we describe these three differences, and then discuss how the resulting search algorithm can be adapted for searching R contraction hierarchies and N-level overlay (subgoal) graphs for shortest *arching* s-t paths instead. Figure 3.13 shows an example of the operation of this algorithm on an N-level overlay graph.

Bidirectional search with upward and downward edges: Contraction hierarchies can be searched for shortest up-down paths by a bidirectional Dijkstra search, where the forward search searches over upward paths that originate at the start vertex s, and the backward search searches over downward paths that terminate at the goal vertex t. Therefore, the forward search only generates successors v of expanded vertices u that are reachable from u by upward edges (u,v), and the backward search only generates predecessors (that is, successors on the inverse graph) v of expanded vertices u that reach u with downward edges (v,u). Any vertex that is not reachable from the start vertex with an upward path and that cannot reach the goal vertex with a downward path is thus effectively pruned

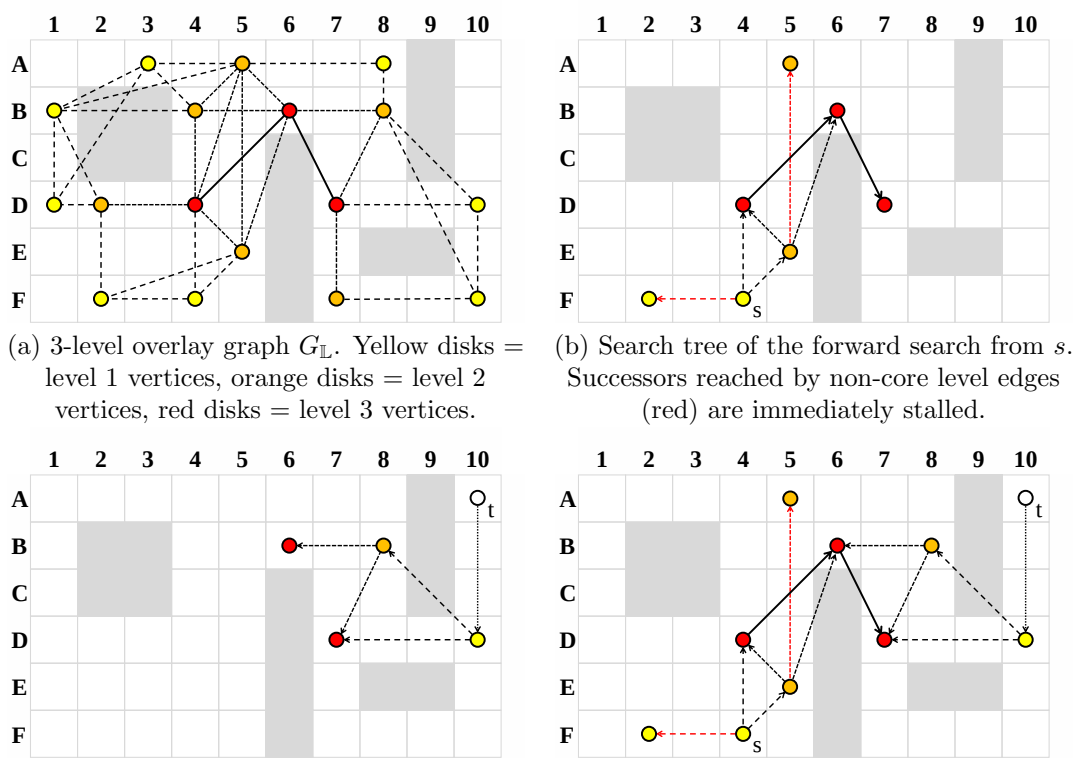
[4] More specifically, consider the N-level overlay graph $N_{\mathbb{L}'}$ where, for all $n \in V$, $\mathbb{L}'(n) = \mathbb{L}(n) + 1$.

from both searches, which allows searches over contraction hierarchies to be significantly faster than searches over their base graphs.

Termination criterion: Recall that a bidirectional Dijkstra search for an s-t path terminates when the two searches "meet" at a vertex n, that is, when n is expanded by both searches. However, bidirectional Dijkstra searches over contraction hierarchies that use this termination criterion can find s-t paths with lengths greater than $d(s,t)$, for the following reason: Although the contraction hierarchy is guaranteed to have an up-down s-t path with length $d(s,t)$, it is not necessarily the case that all shortest upward or downward paths are shortest paths themselves. It just means that there exists some *apex* vertex a for which the combination of the shortest upward s-a path with the shortest downward a-t path is a shortest s-t path. Therefore, the first time when the two searches meet at a vertex n, it is not necessarily the case that n is the apex vertex of a shortest up-down s-t path. To guarantee that shortest arching s-t paths are found, bidirectional Dijkstra searches over contraction hierarchies therefore terminate when the radii (the lowest g-value of a vertex in OPEN) of both searches are greater than or equal to the length of the shortest s-t path found so far by the search (Geisberger et al., 2008), since any vertices that were expanded after this point would be reached with paths that are longer than $d(s,t)$.

Stall-on-demand: Bidirectional Dijkstra searches over contraction hierarchies can use the stall-on-demand technique (which we have discussed in Section 3.2.5 in the context of the OverlayConnect algorithm) to "stall" (avoid expanding) vertices reached with suboptimal upward or downward paths. Stall-on-demand operates as follows in the context of contraction hierarchies: Whenever a vertex v is selected for expansion in the forward search, for each downward edge (u,v), the forward search checks if $g(u) + d(u,v) < g(v)$. If so, v is stalled and not expanded since it has a provably suboptimal g-value. Otherwise, it is expanded as usual. That is, the forward search uses upward edges to generate successors, but uses downward edges to perform stall-on-demand checks. Similarly, the backward search uses upward edges to perform stall-on-demand checks.

Finding arching paths: We can adapt bidirectional Dijkstra searches for finding shortest up-down paths over contraction hierarchies to find shortest arching paths over N-level overlay graphs, as follows: The forward search searches over combinations of upward paths and level paths (that is, arching paths without downward parts), whereas the backward search searches over downwards paths (similar to how it operates on contraction hierarchies). Since arching paths cannot have more than a single non-core level edge (Definition 3.13), successors generated using non-core level edges are not added to OPEN, and are used simply to check if the two searches meet. That is, they are generated as "stalled" vertices. Searches over R contraction hierarchies can ignore this case, since they do not have non-core level edges. Figure 3.13b shows an example, where non-core level edges are shown in red. When F4 (level 1) is expanded, F2 (level 1) is generated as a stalled vertex, since it is reached with a non-core level edge.



(a) 3-level overlay graph $G_{\mathbb{L}}$. Yellow disks = level 1 vertices, orange disks = level 2 vertices, red disks = level 3 vertices.

(b) Search tree of the forward search from s. Successors reached by non-core level edges (red) are immediately stalled.

(c) Search tree of the backward search from t connected to $G_{\mathbb{L}}$ as a level 0 vertex. Level edges are ignored during the backward search.

(d) Search trees of both searches meet at B6 (and D7). Vertices that are not reachable from s with a combination of upward and level paths and cannot reach t with a downward path are pruned from the search.

Figure 3.13: Searching N-level overlay graphs for shortest arching paths.

3.4.4.3 Refinement

Since R N-level subgoal graphs and R contraction hierarchies have R-reachable shortcut edges only, each shortcut edge on a shortest arching path can be replaced with corresponding shortest paths on G directly by using an R-refine algorithm. Contraction hierarchies (and N-level overlay graphs), on the other hand, can have non-R-reachable shortcut edges. If the base graph is G, these shortcut edges can be replaced with corresponding shortest paths on G by *unpacking* them, as we describe below. If the base graph is an R subgoal graph G_S, shortcut edges can first be unpacked into shortest paths on G_S, which can then be R-refined into corresponding shortest paths on G.

Recall that each shortcut edge (u, v) introduced during the contraction of a vertex n corresponds to a combination of two (shortcut) edges (u, n) and (n, v). Therefore, when replacing (u, v) with a corresponding shortest path on the base graph, it can first be replaced with the sequence of edges (u, n) (a downward edge since n is contracted before u) and (n, v) (an upward edges since n is contracted before v), which can then be recursively replaced with their two corresponding (shortcut) edges in the sequence, until all edges in the sequence are edges of the base graph. This operation is called

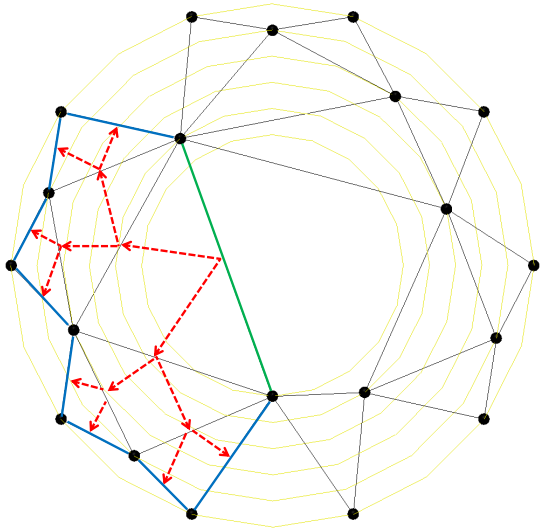


Figure 3.14: Unpacking a shortcut edge (green) recursively replaces it with its two associated edges (red arrows) until the shortcut edge is replaced with a path on G (blue).

unpacking (Geisberger et al., 2008), and is illustrated in Figure 3.14. Observe that each time a shortcut edge is replaced with two corresponding (shortcut) edges, the number of edges in the sequence increases exactly by one. Therefore, unpacking a shortcut edge (u, v) replaces a shortcut edge with two corresponding (shortcut) edges exactly $k - 1$ times, where k is the number of edges of a corresponding shortest u-v path on the base graph.

Unpacking requires additional information to be stored for each shortcut edge. We explain two different ways of doing this. *2-pointer unpacking* requires, for each shortcut edge (u, v), the storage of its two corresponding edges (u, n) and (n, v). *Midpointer unpacking*, on the other hand, requires the storage of n instead, and can reconstruct (u, n) and (n, v) by scanning the downward in-edges of n for the edge (u, n), and the upward out-edges of n for the edge (n, v). Therefore, 2-pointer unpacking requires more information to be stored for each shortcut edge, and midpointer unpacking can take longer to execute.

3.5 Strongly-Connected Subgoal Graphs

In this section, we introduce a suboptimal variant of subgoal graphs, called strongly connected subgoal graphs. Similar to subgoal graphs, strongly connected subgoal graphs can be used to answer path queries using the Connect-Search-Refine algorithm discussed in Section 3.3.4. However, the resulting path is not guaranteed to be a shortest s-t path. In this dissertation, we use strongly connected subgoal graphs as an alternative to subgoal graphs for answering queries on state lattices, since answering queries on state lattices using subgoal graphs achieves only a small speed-up over A* searches on G, as we discuss in Section 4.6.5.

This section is organized as follows. In Section 3.5.1, we formally define strongly connected subgoal graphs. In Section 3.5.2, we show that strongly connected subgoal graphs can be used to find s-t paths on G for every pair of vertices $s, t \in V$, by using the Connect-Search-Refine algorithm. In Section 3.5.3, we introduce an algorithm for constructing strongly connected subgoal graphs.

3.5.1 Strongly-Connected Subgoal Graphs

Recall that an R subgoal graph is an overlay graph G_S where S is an R-SPC on G (Section 3.3), which can be extended, for every $s, t \in V$, into an s-t query R subgoal graph that has only R-reachable edges (Theorem 3.12) and contains an s-t path π with $l(\pi) = d(s, t)$ (Theorem 3.3). However, the requirement that S is an R-SPC limits the set of R subgoal graphs that can be constructed on G, which could result in many vertices of G becoming subgoals.

Our definition of R strongly connected subgoal graphs aims to capture the minimum requirements from a graph $G' = (S, E', c')$, such that, for every $s, t \in V$, it can be extended into an s-t query graph that has only R-reachable edges (using an R-connect algorithm that satisfies the criteria outlined in Section 3.3.4) and contains an s-t path π with $l(\pi) < \infty$ (rather than $l(\pi) = d(s, t)$). Namely, we require that (1) G' is strongly connected (otherwise, for some $s, t \in S$, $d_{G'}(s, t) = \infty$), (2) for every vertex $n \in V$, there is at least one vertex in S that is R-reachable from n and one vertex in S from which n is R-reachable (otherwise, R-connect cannot connect n to the strongly connected subgoal graph as a start or a goal vertex), and (3) all edges of G' are R-reachable (otherwise, R-refine cannot operate on the edges of G'). Definition 3.15 formally defines R strongly connected subgoal graphs as graphs that satisfy these requirements.

Definition 3.15. *A graph $G' = (S, E', c')$ is an R strongly connected subgoal graph on G if and only if:*

1. *G' is strongly connected;*
2. *for every $s \in V$, there exists an $n \in S$ such that $(s, n) \in R$;*
3. *for every $t \in V$, there exists an $n \in S$ such that $(n, t) \in R$; and*
4. *$\forall (u, v) \in E'$, $(u, v) \in R$ and $c'(u, v) = d(u, v)$.*

3.5.2 Answering Queries Using Strongly-Connected Subgoal Graphs

R strongly connected subgoal graphs can be used to answer path queries in the same way that R subgoal graphs can be used to answer path queries (except, not necessarily optimally), by using the Connect-Search-Refine algorithm algorithm discussed in Section 3.3.4. Theorem 3.18 shows that using an R strongly connected subgoal graph instead of an R subgoal graph in the SG-Query algorithm (Algorithm 4) is guaranteed to return, for every $s, t \in V$, an s-t path on G. In this dissertation, we use the same R-connect and R-refine algorithms when answering queries using R subgoal graphs or R strongly connected subgoal graphs. However, in general, the criteria for the "correctness" of R-connect and R-refine algorithms are "less strict" when using them for answering queries

using R strongly connected subgoal graphs rather than R subgoal graphs. The premise of Theorem 3.18 outlines these "less strict" criteria.

Theorem 3.18. *Let $G' = (S, E', c')$ be an R strongly connected subgoal graph and $s, t \in V$ be arbitrary vertices of G. SG-Query(G, R, G', s, t) finds an s-t path on G if:*

1. *R-connect(G, S, s, t) returns E^+ and c^+, such that:*
   - 1.1. *For some n such that $(s, n) \in R$, $(s, n) \in E^+$;*
   - 1.2. *For some n such that $(n, t) \in R$, $(n, t) \in E^+$;*
   - 1.3. *$E^+ \subseteq R$; and*
   - 1.4. *$\forall (u, v) \in E^*$, $c^+(u, v) < \infty$.*
2. *R-refine(u, v) returns a (u, v) path on G if $(u, v) \in R$.*

Proof. Let $u \in S$ such that $(s, u) \in R$ and $(s, u) \in E^+$ with $c^+(s, u) < \infty$. By Definition 3.15, such $u \in S$ exists with $(s, u) \in R$ and, in the premise of the theorem, we assume that R-connect identifies at least one such u (if $s \in S$, we use $u = s$ with $c^+(s, u) = 0$). Similarly, let $v \in S$ such that $(v, t) \in R$ and $(v, t) \in E^+$ with $c^+(v, t) < \infty$. Let $\pi_{u,v}$ be a u-v path on G' with $l(\pi_{u,v}) < \infty$. By Definition 3.15, such $\pi_{u,v}$ exists since $u, v \in S$ and G' is strongly connected. Then, $\pi_{s,t} = \langle s, u \rangle \cdot \pi_{u,v} \cdot \langle v, t \rangle$ is an s-t path on the s-t query R strongly connected subgoal graph with $l(\pi_{s,t}) = c^+(s, u) + l(\pi_{u,v}) + c^+(v, t) < \infty$. Therefore, the search phase is able to find some s-t path π on the s-t query R strongly connected subgoal graph. All edges on π are guaranteed to be R-reachable since, by Definition 3.15, G' has only R-reachable edges and, in the premise of the theorem, we assume that $E^+ \subseteq R$. Therefore, the refinement phase can replace each edge (u', v') on π with some u'-v' path on G. □

3.5.3 Constructing Strongly-Connected Subgoal Graphs

In this section, we introduce a two-phase algorithm for constructing R strongly connected subgoal graphs. In the first phase, the algorithm identifies a set of *access subgoals* A to ensure that every vertex $n \in V$ can be connected to an R-reachable subgoal in both the forward and backward directions. That is, for every $n \in V$, there exists $u, v \in A$ with $(u, n), (n, v) \in R$. In the second phase, the algorithm constructs a graph that strongly-connects the access subgoals, adding extra subgoals as necessary to guarantee that the graph has R-reachable edges only. In this dissertation, we use this algorithm to construct strongly connected subgoal graphs on state lattices in Section 4.6.6, with respect to various reachability relations.

3.5.3.1 Identifying Access Subgoals

Algorithm 8 outlines our algorithm for identifying access subgoals, which starts with an empty set of access subgoals A (line 1). For every vertex $n \in V$, Algorithm 8 maintains two flags: forward[n], which indicates whether there exists a vertex $u \in A$ with $(n, u) \in R$, and backward[n], which indicates whether there exists a vertex $v \in A$ with $(v, n) \in R$. Initially, for all $n \in V$, forward[n] = backward[n] = false (lines 2–3) since $A = \emptyset$.

Algorithm 8 IdentifyAccessSubgoals

Input: G, reachability relation R

Output: Set of subgoals A such that, for every $n \in V$, $\exists u, v \in A$ with $(n, u), (v, n) \in R$

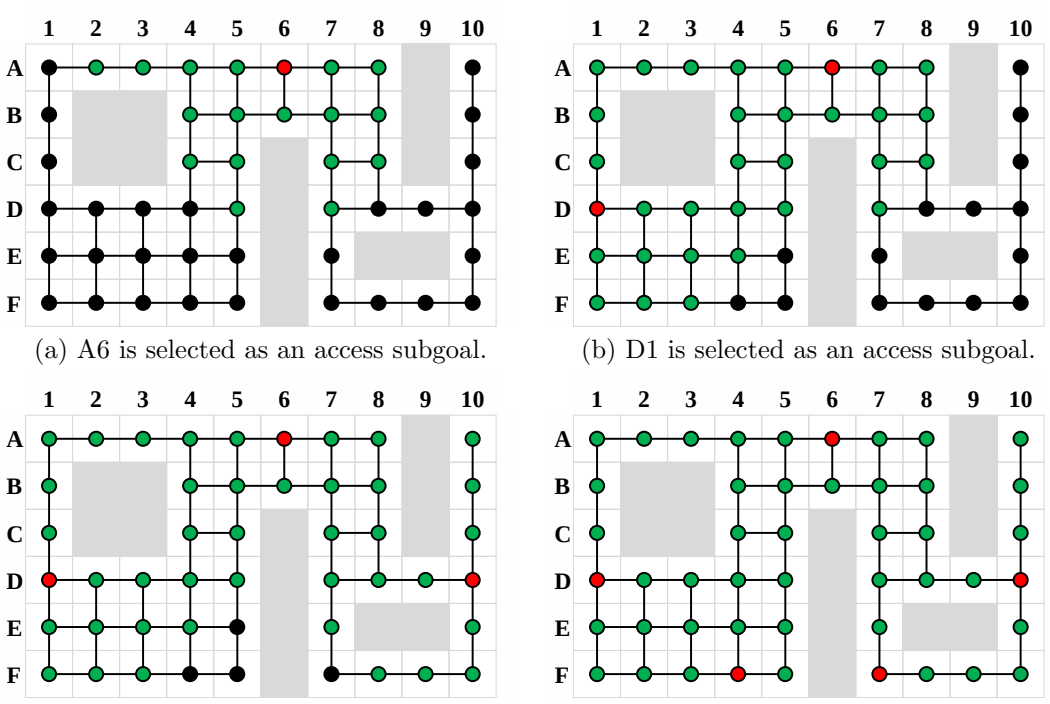
```
1:  A ← ∅
2:  for all n ∈ V do
3:      forward[n] ← false, backward[n] ← false
4:  for all n ∈ V in some order do
5:      if ¬forward[n] ∨ ¬backward[n] then
6:          A ← A ∪ {n}
7:          for all t ∈ V : (n, t) ∈ R do
8:              backward[t] ← true
9:          for all s ∈ V : (s, n) ∈ R do
10:             forward[t] ← true
11: return A
```

Algorithm 8 then iterates over all vertices $n \in V$ (line 4) and adds them to A if and only if forward$[n]$ = false or backward$[n]$ = false (line 5,6). That is, if n cannot be connected to a vertex in A with an R-reachable edge in the forward or the backward direction, it is added to A. When a vertex n is added to A, the backward flag of every vertex t with $(n, t) \in R$ and the forward flag of every vertex s with $(s, n) \in R$ is set to true (lines 7–10). In our implementation, we use a modified R-connect algorithm to quickly iterate over all such vertices and update their flags.

Figure 3.15 shows an example for identifying access subgoals for BD4-reachability on a 4-neighbor grid graph. Since BD4-reachability is symmetric on undirected graphs, the example uses a single flag for each vertex n that indicates whether an access subgoal u exists with $d(n, u) = d(u, n) \leq 4$. Such vertices are shown in green, and access subgoals are shown in red. Algorithm 8 first arbitrarily selects A6, makes it an access subgoal since there is no access subgoal within distance 4 from it, and marks every vertex within distance 4 from it. These vertices will not be added to the set of access subgoals. After randomly selecting an unmarked vertex D1, Algorithm 8 similarly makes it an access subgoal and marks all vertices within distance 4 from it. After all vertices are processed, a total of five access subgoals have been identified, and every vertex has an access subgoal within distance 4 from it.

3.5.3.2 Strongly Connecting Access Subgoals Using R-Reachable Edges

A straightforward method for strongly-connecting access subgoals using R-reachable edges only would be to first construct an overlay graph G_A of access subgoals (which is guaranteed to be strongly connected, since G is strongly connected) and then, for each non-R-reachable edge (u, v) that appears on G_A, break it down into a sequence of R-reachable edges $(u, n_1), (n_1, n_2), \ldots, (n_k, v)$, by adding $n_1, \ldots, n_k$ to the set of subgoals. Algorithm 9 outlines our algorithm for strongly connecting access subgoals using R-reachable edges, which follows a similar idea, but aims to add as few extra subgoals as possible.



(a) A6 is selected as an access subgoal.

(b) D1 is selected as an access subgoal.

(c) D10 is selected as an access subgoal.

(d) F4 and F7 are selected as access subgoals.

Figure 3.15: Identifying access subgoals for BD4-reachability. Red vertices are access subgoals. Green vertices can connect to a subgoal in both the forward and backward directions.

Rather than constructing G_A, Algorithm 9 instead constructs two spanning trees of access subgoals rooted at an arbitrarily selected vertex $n \in A$ (line 3), one an out-tree (edges point away from the root) and one an in-tree (edges point toward the root), in order to reduce the number of additional subgoals required to break down non-R-reachable edges. The edges of these two trees are sufficient to strongly connect all access subgoals, since the out-tree contains an n-t path for every $t \in A$ and the in-tree contains an s-n path for every $s \in A$. Therefore, for every $s, t \in A$, the s-n path in the in-tree and the n-t path in the out-tree can be combined to form an s-t path. The out-tree and the in-tree rooted at n can be constructed by using Prim's algorithm (Prim, 1957), by initializing their roots to n and repeatedly adding a vertex $v \in A$ to the tree that has the minimum u-v or v-u distance, respectively, for some u that is already in the tree. Algorithm 9 performs this construction by using two Dijkstra searches, one in the forward direction and one in the backward direction (lines 4–9 and 18), with the following modification: When a vertex $v \in A$ is selected for expansion, it is connected to the spanning tree (lines 10–14) and its g-value is updated to 0 (line 15) before it is expanded. This ensures that every vertex that is already in the spanning tree has g-value 0 (that is, it can be considered as one of multiple start vertices of a Dijkstra search), and the g-value of the next vertex $v \in A$ selected for expansion correctly represents its distance from/to the closest vertex u already in the spanning tree. Since G is strongly connected, these modified Dijkstra

Algorithm 9 StronglyConnectSubgoals

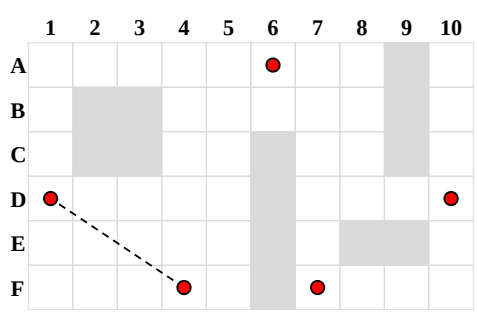
Input: G, reachability relation R, access subgoals A
Output: Strongly-connected subgoal graph $G' = (S, E', c')$

```
 1: S ← A
 2: E' ← ∅
 3: randomly select s ∈ S
 4: for both the forward and backward directions do
 5:     for all n ∈ V do
 6:         g[n] ← ∞, parent[n] ← undefined
 7:     g[s] ← 0, OPEN ← {s}
 8:     while OPEN ≠ ∅ do
 9:         v ← pop the vertex with minimum g-value in OPEN
10:         if v ∈ S and g(v) > 0 then
11:             π ← follow parents from v to a u ∈ S
12:             Reverse π if searching backward
13:             S+, E+, c+ ← SplitIntoRReachable(π)
14:             S ← S ∪ S+, E' ← E' ∪ E+, update c' with c+
15:             g[v] ← 0
16:             for all n ∈ S+ do
17:                 g[n] ← 0, insert/update n in OPEN
18:         Expand(v)
19: return G' = (S, E', c')
```

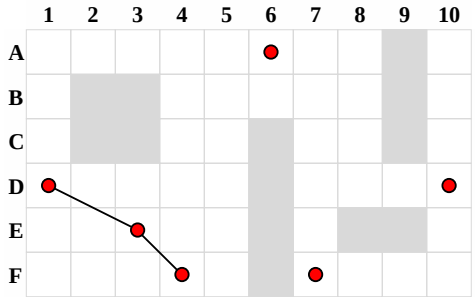
searches are guaranteed to eventually visit every vertex $v \in A$ and construct the desired spanning trees.

Algorithm 9 interleaves the construction of the spanning trees with the breaking down of non-R-reachable edges. That is, any time an edge (u, v) is identified as an edge of the out- or in-tree, it is broken down into a sequence of R-reachable edges by adding more subgoals to S (lines 13–14). The g-values of the new subgoals are also set to 0 (lines 16–17), which allows adding new vertices to the spanning trees through shorter edges if a new subgoal is closer to them than to an access subgoal that is already in the spanning tree. The aim of this interleaving is to allow breaking down of multiple non-R-reachable edges by adding a single subgoal, which we discuss further in the context of an example. In our implementation, we select the subgoals to break down a non-R-reachable edge (u, v) by first extracting a representative u-v path π on G from the Dijkstra search (lines 11–12) and then placing the fewest number of subgoals as evenly as possible along π.

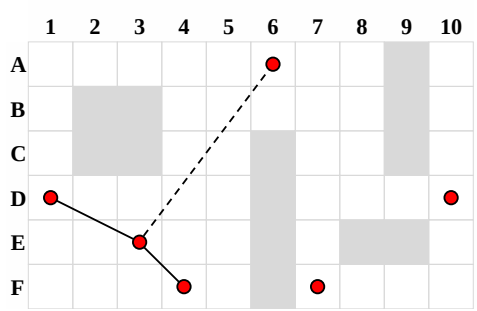
Figure 3.16 shows an example of the construction of a strongly connected BD4 subgoal graph using Algorithm 9, using the access subgoals identified in Figure 3.15. Since G is undirected in this example, it is sufficient to construct a single spanning tree that can be interpreted as both the spanning out- and in-trees. Algorithm 9 randomly selects F4 $\in A$ as the root of the spanning tree and starts a Dijkstra search from F4, which selects D1 $\in A$ with g(D1) = 5 for expansion before any other vertex in $A \setminus \{\text{F4}\}$, indicating that F4 is the access subgoal that can be connected to the spanning tree using the shortest edge (Figure 3.16a). Since the edge (D1,F4) is not BD4-reachable, it is broken down



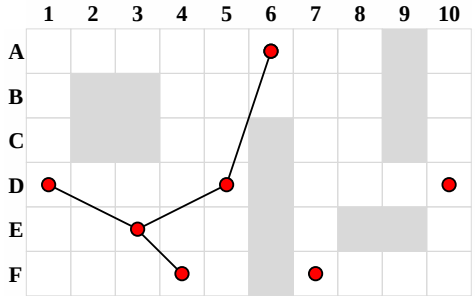
(a) The spanning tree rooted at F4 is extended with the edge (D1, F4).



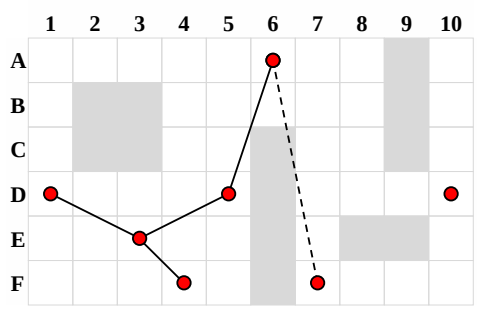
(b) E3 is added as a subgoal to break down the non-R-reachable edge (D1, F4).



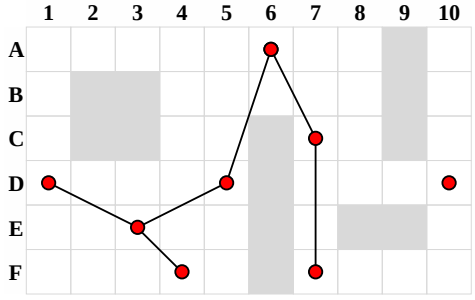
(c) The spanning tree rooted at F4 is extended with the edge (E3, A6).



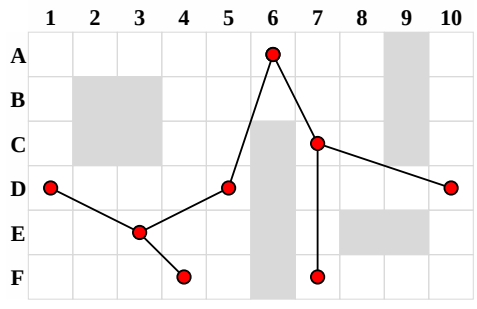
(d) D5 is added as a subgoal to break down the non-R-reachable edge (E3, A6).



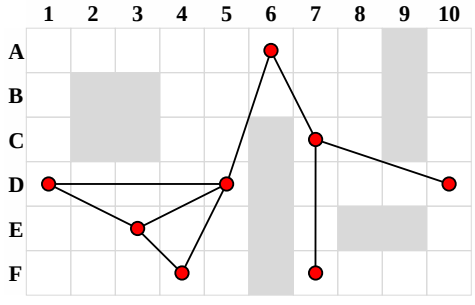
(e) The spanning tree rooted at F4 is extended with the edge (A6, F7).



(f) C6 is added as a subgoal to break down the non-R-reachable edge (A6, F7).



(g) The spanning tree rooted at F4 is extended with the edge (C7,D10).



(h) Direct-R-reachable edges are added to reduce distances.

Figure 3.16: Strongly connecting access subgoals with BD4-reachable edges. Solid line: R-reachable edge. Dashed line: Non-R-reachable edge.

into the sequence of BD4-reachable edges (F4,E3) and (E3,D1), by adding a subgoal E3 (Figure 3.16b). The g-values of the access subgoal D1 and the new subgoal E3 are both set to 0, and the Dijkstra search is resumed, eventually selecting A6 $\in A$ for expansion. Since the edge (E3, A6) required to add A6 to the spanning tree is not BD4-reachable, a new subgoal D5 is added to break it down into the sequence of BD4-reachable edges (E3, D5) and (D5, A6) (Figures 3.16c and 3.16d). Similarly, the non-BD4-reachable edge (A6, F7) required for connecting F7 to the spanning tree is broken down into a sequence of BD4-reachable edges with a new subgoal C7 (Figures 3.16e and 3.16f). Finally, D10 is connected to the spanning tree with the BD4-reachable edge (C7, D10) (Figure 3.16g). If Algorithm 9 did not interleave the construction of the spanning tree with the breaking-down of non-R-reachable edges, the initial spanning tree it constructs would contain the non-R-reachable edges (A6, F7) and (F7, D10). Breaking down these two edges into sequences of R-reachable edges individually might require two subgoals. Instead, Algorithm 9 identifies the single subgoal C7 in this example, which it first uses to break down (A6, F7), and then to connect D10 to the spanning tree directly via an R-reachable edge.

Algorithms 8 and 9 can be modified in several ways to reduce the number of subgoals that are identified, possibly at the cost of additional preprocessing time. For instance, in the example shown in Figure 3.16, the additional subgoal D5 alone is sufficient to strongly connect the access subgoals D1, F4, and A6 with BD4-reachable edges, and E3 is unnecessary. Due to its greedy nature (for constructing a spanning tree), Algorithm 9 fails to address this issue. Furthermore, no pair of access subgoals identified by Algorithm 8 in Figure 3.15 is BD4-reachable, which eventually leads Algorithm 9 to introduce multiple new subgoals to ensure that they can be strongly connected with BD4-reachable edges only.

Although minimizing the number of subgoals in R strongly connected subgoal graphs can help to reduce the search times in queries, it can also result in queries finding longer paths and increase the connection times in queries. A detailed analysis of different construction strategies for R strongly connected subgoal graphs and the resulting query-time/path-length trade-offs is beyond the scope of this dissertation. We use R strongly connected subgoal graphs mainly as a means of comparing different reachability relations on state lattices (Section 4.6.6) and assume that any improvements made to the construction of R strongly connected subgoal graphs affect the query-time/path-length trade-offs similarly for different reachability relations. Our aim with Algorithms 8 and 9 is to find a decent query-time/path-length trade-off to show that this is a feasible approach that can be applied in real-world scenarios. To that end, we also add edges between all direct-R-reachable pairs of subgoals to reduce the distances on R strongly connected subgoal graphs, as shown in Figure 3.16h.

3.6 Conclusions

In this chapter, we have introduced the subgoal graph framework, which can be specialized to exploit structure in different classes of graphs, by choosing a reachability relation that captures structure in that class of graph and developing specialized connection and refinement algorithms that exploit this structure. We have proved that subgoal graphs

can be used to answer path queries optimally with the Connect-Search-Refine algorithm, and that it is possible to construct locally sparse subgoal graphs with respect to bounded-distance reachability on graphs with small highway dimensions. We have introduced a hierarchical variant of subgoal graphs, called N-level subgoal graphs, and introduced variants of contraction hierarchies within this framework. We have introduced a suboptimal variant of subgoal graphs, called strongly connected subgoal graphs, that can be used to answer path queries, but without the guarantee of optimality. We have introduced algorithms for constructing subgoal graphs, N-level subgoal graphs, and strongly connected subgoal graphs. In Chapters 4 and 5, we apply the subgoal graph framework to state lattices and grid graphs to exploit their freespace structure.

Chapter 4

Exploiting the Freespace Structure of State Lattices

In this chapter, we apply the subgoal graph framework to state lattices, by introducing freespace-reachability and canonical-freespace-reachability as reachability relations to capture the freespace structure of state lattices, and developing efficient connection and refinement algorithms that exploit this structure. Specifically, we characterize the freespace structure of state lattices as the *translation invariance of freespace distances and freespace-canonical paths*, and show that it can be exploited to efficiently compute and compactly store freespace information, such as pairwise distances or shortest path trees on freespace state lattices. We introduce freespace-reachability and canonical-freespace-reachability as reachability relations to distinguish those pairs of vertices on state lattices between which the freespace information is accurate, and develop connection and refinement algorithms for these reachability relations that use freespace information to efficiently explore freespace-shortest and freespace-canonical paths, respectively. We experimentally demonstrate that, answering queries using freespace-reachability or canonical-freespace-reachability strongly connected subgoal graphs achieves a *dominating query-time/path-suboptimality trade-off* compared to answering queries using bounded-distance-reachability strongly connected subgoal graphs, and a *non-dominated query-time/path-suboptimality* trade-off compared to answering queries using weighted A* searches on G. These results validate the hypothesis of this dissertation that one can develop preprocessing-based path-planning algorithms for state lattices that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.

This chapter is organized as follows. In Section 4.1, we provide a detailed summary of the main ideas that we use in this chapter. In Section 4.2, we formally define state lattices and introduce notation that we use throughout this chapter. In Section 4.3, we state our assumptions for state lattices that are necessary to satisfy our assumptions for G. In Section 4.4, we define freespace state lattices, prove that distances on freespace state lattices are translation invariant, introduce freespace-reachability as a reachability relation on state lattices, and introduce connection and refinement algorithms for freespace-reachability that use precomputed freespace distances. In Section 4.5, we introduce a lexical canonical ordering on freespace paths, prove that freespace-canonical paths are translation invariant, introduce canonical-freespace-reachability as a reachability relation on

state lattices, and introduce connection and refinement algorithms for canonical-freespace-reachability that use precomputed freespace-canonical successors, predecessors, and parents. In Section 4.6, we experimentally compare different connection algorithms with respect to their connection times, and compare answering queries using subgoal graphs and strongly connected subgoal graphs constructed with respect to bounded-distance-reachability, freespace-reachability, and canonical-freespace-reachability, with respect to their query-time/path-suboptimality trade-offs. Finally, in Section 4.7, we summarize our results.

4.1 Introduction

In Section 2.1.2, we introduced state lattices as graphs that are constructed by systematically discretizing the configuration space of an agent. In this chapter, we show that the systematicity in the construction of state lattices gives rise to their *freespace structure*, which can be captured with reachability relations and exploited with efficient connection and refinement algorithms. In particular, we show that, on state lattices constructed on obstacle-free and infinitely extending environments, called *freespace state lattices*, *translating* shortest paths by changing the x- or y-coordinates of all their vertices by the same integral amount produces other shortest paths, and translating *canonical* paths produces other canonical paths. We refer to these properties as the *translation invariance of freespace distances* and *freespace-canonical paths*, respectively.

The translation invariance of freespace distances allows for the efficient computation and compact storage of pairwise freespace distances. Freespace distances accurately represent distances on freespace state lattices, and may accurately represent distances between *some* pairs of vertices on the (actual) state lattices. We define *freespace-reachability* as the set of all pairs of vertices on a state lattice between which the freespace distance is accurate. When answering path queries using freespace-reachability subgoal or strongly connected subgoal graphs, connection phases of queries need to identify only freespace-reachable edges, and the refinement phases need to find shortest paths only between freespace-reachable vertices. We introduce a connection algorithm for freespace-reachability which uses a depth-first search (rather than a Dijkstra search) and uses freespace distances to avoid expanding vertices that are not freespace-reachable from the start vertex (or from which the goal vertex is not freespace-reachable). We introduce a refinement algorithm for freespace-reachability which also uses a depth-first search, and can be considered to be similar to an A* search that uses freespace distances as a perfect heuristic.

We impose a canonical ordering on the paths on freespace state lattices, that uniquely designates, among multiple symmetric shortest paths between two vertices, one as the *freespace-canonical* path. Our canonical ordering ensures that the set of all freespace-canonical paths that originate (or terminate) at a vertex form a tree. The translation invariance of freespace-canonical paths allows for the efficient computation and compact storage of these trees. We define *canonical-freespace-reachability* as the list of all pairs of vertices on a state lattice between which the freespace-canonical path is unblocked. We introduce a connection algorithm for canonical-freespace-reachability, which is similar to our

connection algorithm for freespace-reachability, but only considers a small set of *freespace-canonical* successors for each expanded vertex and does not perform duplicate detection. We introduce a refinement algorithm for canonical-freespace-reachability, which simply returns the freespace-canonical path between two canonical-freespace-reachable vertices.

We experimentally compare various connection algorithms for various reachability relations with respect to their connection times, and experimentally compare answering queries using subgoal graphs and strongly connected subgoal graphs constructed with respect to various reachability relations with respect to their query times, memory requirements, and path suboptimalities. Our results show that our connection algorithm for canonical-freespace-reachability has the highest "DR-rate" (direct-R-reachable vertices expanded per second), answering queries using subgoal graphs constructed with respect to various reachability relations is only slightly faster than A* searches on state lattices, and answering queries using freespace-reachability and canonical-freespace-reachability strongly connected subgoal graphs achieves dominating query-time/path-suboptimality trade-offs compared to answering queries using bounded-distance reachability strongly connected subgoal graphs, and non-dominated query-time/path-suboptimality trade-offs compared to answering queries using weighted A* searches on state lattices.

4.2 Preliminaries and Notation

As described in Section 2.1.2, state lattices are constructed by discretizing the configuration spaces of agents into graphs. In this section, we formally define state lattices and introduce notation that we use throughout this chapter.

Cell, grid, orientation, state: A cell is a pair of integers $(x, y) \in \mathbb{Z}^2$. A grid is a set of cells $\mathcal{G} \subseteq \mathbb{Z}^2$. A cell $(x, y) \in \mathcal{G}$ is unblocked (with respect to $\mathcal{G}$) if $(x, y) \in \mathcal{G}$ and blocked otherwise. A state is a tuple (x, y, θ) where (x, y) is a cell and $\theta \in \mathbb{Z}_{\geq 0}$ is a non-negative integer that indicates the orientation of the agent (or, more generally, any combination of its discretized features).

Primitive: A (motion) primitive $\vec{m}$ is a tuple $(\theta^s_m, \theta^e_m, x_m, y_m, l_m, C_m)$ such that: $\theta^s_m, \theta^e_m \in \mathbb{Z}_{\geq 0}$ are the start and end orientations of $\vec{m}$; x_m, $y_m \in \mathbb{Z}$ are the difference in the x- and y-coordinates, respectively, of the start and end cells of $\vec{m}$; $l_m \in \mathbb{R}_{>0}$ is the length of $\vec{m}$; and $C_m \supseteq \{(0, 0), (x_m, y_m)\}$ is the list of cells that the footprint of the agent intersects with when executing m from state $(0, 0, \theta^s_m)$. $\vec{m}$ is *kinematically feasible* to execute from state (x, y, θ) if and only if $\theta^s_m = \theta$. $\vec{m}$ is *collision free* to execute from (x, y, θ) (with respect to $\mathcal{G}$) if and only if, for all $(x, y) \in C_m$, cell $(x_s + x, y_s + y)$ is unblocked. $\vec{m}$ is *executable* from (x, y, θ) if and only if it is kinematically feasible and collision free to execute from (x, y, θ). Executing $\vec{m}$ from state $s = (x, y, \theta^s_m)$ results in state $e = (x + x_m, y + y_m, \theta^e_m)$, denoted as $s + \vec{m} = e$.

State lattice, G: Let $\mathcal{G}$ be a grid and $\mathcal{M}$ be a set of primitives. Let $O_\mathcal{M}$ be the set of *induced orientations* of M, such that $\theta \in O_\mathcal{M}$ if and only if there exists $\vec{m} \in \mathcal{M}$ with $\theta = \theta^s_m$ or $\theta = \theta^e_m$. A state lattice is a graph $\mathcal{L}_{\mathcal{G},\mathcal{M}} = (V', E', c')$, defined as follows: For each unblocked cell $(x, y) \in \mathcal{G}$ and each induced orientation $\theta \in O_\mathcal{M}$, V' contains a vertex that corresponds to state (x, y, θ). We use the terms state and vertex interchangeably unless it results in ambiguity. For each vertex $s \in V'$ and all primitives $\vec{m} \in \mathcal{M}$ executable from s, $\vec{m}$ *induces* an edge $(s, s + \vec{m}) \in E'$ with length $c'(s, s + \vec{m}) = l_m$. As we discuss

in the next section, we assume that each edge in a state lattice is induced by a unique primitive, and that the graph G that we use for path planning is the largest strongly connected component of a state lattice.

Primitive path: A primitive path is a sequence of primitives $\vec{\pi} = \langle \vec{m}_1, \ldots, \vec{m}_k \rangle$ where, for all $i = 1, \ldots, k-1$, $\theta^e_{m_i} = \theta^s_{m_{i+1}}$.

Path notation with primitives: Given our assumption that each edge in a state lattice is induced by a unique primitive, it holds that, for every path $\pi = \langle p_0, \ldots, p_k \rangle$ with $p_0, \ldots, p_k \in V$ on a state lattice, there exists a unique corresponding primitive path $\vec{\pi} = \langle \vec{m}_1, \ldots, \vec{m}_k \rangle$ with $\vec{m}_1, \ldots, \vec{m}_k \in \mathcal{M}$, such that, for all $i = 1, \ldots, k$, $p_{i-1} + \vec{m}_i = p_i$. Throughout this chapter and the next chapter, we sometimes describe paths by interleaving them with their primitive paths, as follows: $\pi = \langle p_0, \vec{m}_1, p_1, \ldots, p_{k-1}, \vec{m}_k, p_k \rangle$. Furthermore, we sometimes equivalently describe them by leaving out up to $k-1$ of their vertices, as follows: $\pi = \langle p_0, \vec{m}_1, \ldots, \vec{m}_k \rangle = \langle \vec{m}_1, \ldots, \vec{m}_j, p_j, \vec{m}_{j+1}, \ldots \vec{m}_k \rangle$.

Unicycle and Urban primitives: We use state lattices constructed with respect to two different sets of primitives in this chapter, namely the Unicycle and Urban primitives. Although we describe them in greater detail in Section 4.6.1 in the context of our experimental results, we provide a brief description of them here as well, since we use them in various figures and examples throughout this chapter. The *Unicycle* primitives have 16 induced orientations and 4 (kinematically feasible) primitives per orientation. All Unicycle primitives for the orientation "Up" are shown in Figure 4.1a. The *Urban* primitives have 32 induced orientations and between 26 to 32 primitives per orientation. All Urban primitives for the orientation "Right" are shown in Figure 4.1b. For both the Unicycle and Urban primitives, the length of each primitive is equal to the distance traveled by the agent multiplied by a factor: For Unicycle primitives, backward moves are multiplied by 5 and turns are multiplied by 2. For Urban primitives, backward moves are multiplied by 3. All other primitives are multiplied by 1.

4.3 Assumptions

In order to ensure that G satisfies Assumptions 1.1 (G is finite), 1.2 (G is strongly connected), and 1.3 (every edge $(u, v) \in E$ is the unique shortest u-v path on G), we define G as the largest strongly connected component of a state lattice $\mathcal{L}_{\mathcal{G},\mathcal{M}}$, where $\mathcal{G}$ and $\mathcal{M}$ are finite and $\mathcal{M}$ is *well-behaved*. Assumptions 4.1, 4.2, and 4.3 summarize our assumptions about $\mathcal{G}$, $\mathcal{M}$, and how G relates to $\mathcal{L}_{\mathcal{G},\mathcal{M}}$.

Assumption 4.1. *$\mathcal{G}$ and $\mathcal{M}$ are finite.*

Assumption 4.2. *G is the largest strongly connected component of $\mathcal{L}_{\mathcal{G},\mathcal{M}}$.*

Assumption 4.3. *$\mathcal{M}$ is well-behaved: For every $\vec{m} \in \mathcal{M}$, there is no primitive path $\vec{\pi} = \langle \vec{m}_1, \ldots, \vec{m}_k \rangle$ such that:*

1. *$\vec{m}_1, \ldots, \vec{m}_k \in M \setminus \{\vec{m}\}$;*
2. *$\theta^s_{m_1} = \theta^s_m$, $\theta^e_{m_k} = \theta^e_m$;*
3. *$x_{m_1} + \cdots + x_{m_k} = x_m$;*

4. $y_{m_1} + \cdots + y_{m_k} = y_m$; *and*

5. $l_{m_1} + \cdots + l_{m_n} \leq l_m$.

Lemma 4.1. *G satisfies Assumptions 1.1, 1.2, and 1.3.*

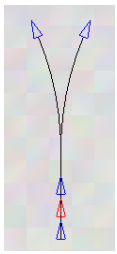
Proof.

1. Assumption 1.1: G is finite:
   1.1. $\mathcal{L}_{\mathcal{G},\mathcal{M}}$ is finite since $\mathcal{G}$ and $\mathcal{M}$ are finite (Assumption 4.1).
   1.2. G is finite since it is a component of the finite graph $\mathcal{L}_{\mathcal{G},\mathcal{M}}$ (Assumption 4.2).
2. Assumption 1.2: G is strongly connected (Assumption 4.2).
3. Assumption 1.3: Every edge $(u, v) \in E$ is the unique shortest u-v path on G:
   3.1. Let $\vec{m}$ be the primitive that induces the edge (u, v) in G.
   3.2. Assume (for contradiction) that there exists a u-v path $\pi = \langle p_0, \ldots, p_k \rangle \neq \langle u, v \rangle$ with $l(\pi) \leq c(u, v)$.
   3.3. For $i = 0, \ldots, k-1$, let $\vec{m}_{i+1}$ be a primitive that induces the edge (p_i, p_{i+1}) in G.
   3.4. $\mathcal{M}$ is not well-behaved (Assumption 4.3, since $\vec{m}$ and the sequence $\vec{m}_1, \ldots, \vec{m}_k$ violate it).
   3.5. $\mathcal{M}$ is well-behaved (Assumption 4.3).
   3.6. $\perp$ (since $\mathcal{M}$ is well-behaved and $\mathcal{M}$ is not well-behaved).

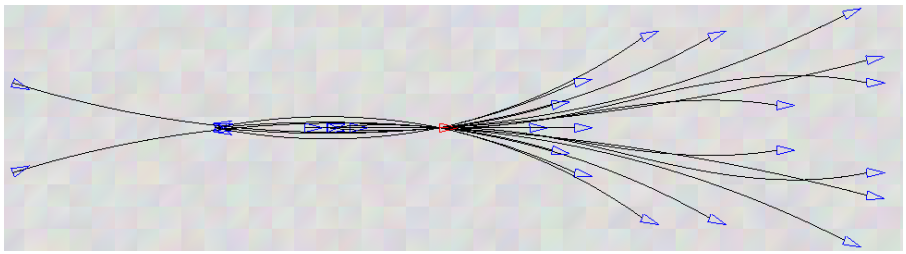
□

4.4 Freespace Reachability

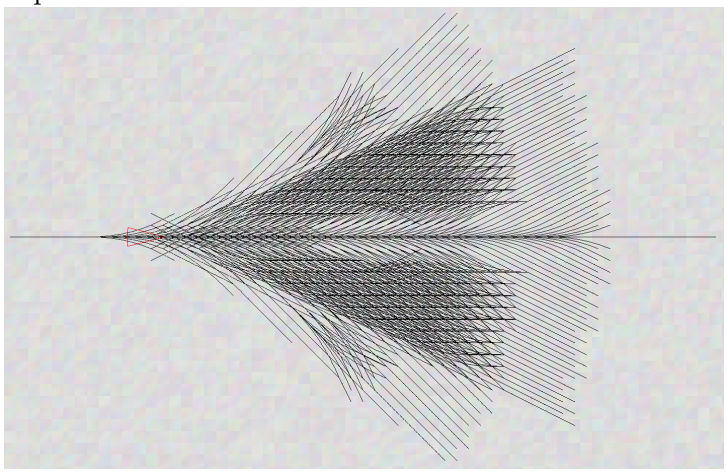
In this section, we introduce *freespace state lattices* as state lattices constructed on obstacle-free and infinitely-extending environments, and *freespace-shortest paths* as shortest paths on freespace state lattices. We prove the *translation invariance of freespace distances*, that is, that *translating* freespace-shortest paths by changing the x- or y-coordinates of all their vertices by the same integral amount produces other freespace-shortest paths, and show that this property can be exploited to efficiently compute and compactly store pairwise distances on freespace state lattices. We introduce *freespace-reachability* as a reachability relation on state lattices as a set of pairs of vertices between whose distance on the state lattice is equal to the distance on the freespace state lattice. We introduce a connection algorithm for freespace-reachability, which uses a depth-first search that uses precomputed freespace distances to explore only freespace-shortest paths that originate (or terminate) at a given vertex. We introduce a refinement algorithm for freespace-reachability, which uses a depth-first search that uses precomputed freespace distances to explore only freespace-shortest paths between two vertices.



(a) Unicycle primitives.



(b) Urban primitives.



(c) Freespace-shortest paths up to length 50 (Unicycle). Figure scaled up by a factor of 2.15 compared to (d).

(d) Freespace-shortest paths up to length 50 (Urban).

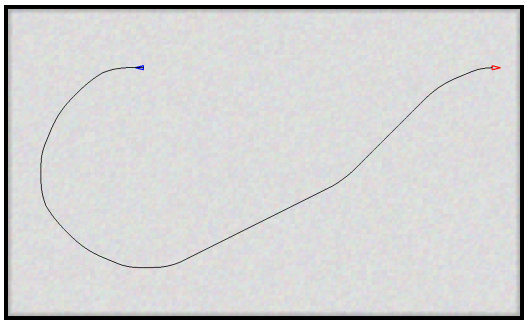
Figure 4.1: Unicycle and Urban primitives and their corresponding freespace-shortest paths up to length 50. The red triangle shows the current state. In (c) and (d), blue triangles show states reachable from the current state with a single primitive.

This section is organized as follows. In Section 4.4.1, we introduce freespace state lattices and discuss some of their properties. In Section 4.4.2, we prove the translation invariance of freespace distances. In Section 4.4.3, we introduce freespace-reachability as a reachability relation on state lattices. In Sections 4.4.4 and 4.4.5, we introduce connection and refinement algorithms for freespace-reachability, respectively.

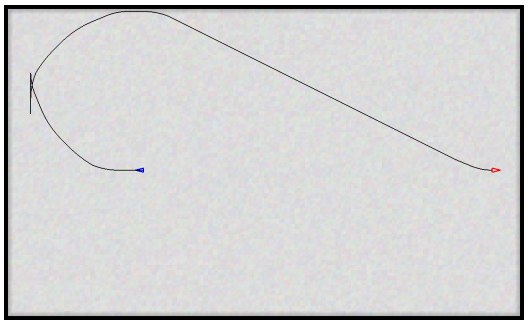
4.4.1 Freespace State Lattice

The freespace state lattice $\mathcal{F}$ is the state lattice $\mathcal{L}_{\mathbb{Z}^2,\mathcal{M}}$, that is, the state lattice constructed with respect to an infinitely-extending grid with no blocked cells. Definition 4.1 formally defines freespace grids, state lattices, and paths. Figures 4.1c and 4.1d show shortest paths of lengths up to 50 that originate at a single state on a freespace state lattice constructed from the Unicycle and Urban primitives, respectively.

Definition 4.1 (Freespace grid, state lattices, and paths)**.** *The freespace grid is the grid* $\mathbb{Z}^2$ *(which extends infinitely in all directions and does not contain any blocked cells). The*



(a) A shortest path on G that is freespace-shortest.



(b) A shortest path on G that is not freespace-shortest. The boundaries of the grid act as obstacles, blocking a path that is equivalent to the one shown in (a).

Figure 4.2: Shortest paths on state lattices. The relative x- and y- coordinates of the start and goal vertices are the same in both figures, but only one of them is freespace-shortest.

freespace state lattice is the state lattice $\mathcal{F} = \mathcal{L}_{\mathbb{Z}^2,\mathcal{M}} = (V_\mathcal{F}, E_\mathcal{F}, c_\mathcal{F})$. *A freespace* s*-*t *path is an* s*-*t *path on* $\mathcal{F}$. *An* s*-*t *path (on* G *or* $\mathcal{F}$*) is freespace-shortest if and only if it is a shortest* s*-*t *path on* $\mathcal{F}$. *The freespace* s*-*t *distance is the length of a shortest* s*-*t *path on* $\mathcal{F}$*, and is denoted as* $d_\mathcal{F}(s,t)$. *We say that a freespace path is unblocked (on* G*) if and only if it is also a path on* G.

Our definition of the freespace grid is influenced by the *freespace assumption*, the assumption that the underlying environment has no obstacles (or the corresponding grid has no blocked cells). The freespace assumption is typically used in path planning in partially-known environments (Zelinsky, 1992; Foux, Heymann, & Bruckstein, 1993; Stentz, 1994; Nourbakhsh & Genesereth, 1996; Koenig & Smirnov, 1997): any region of the environment about which the agent has no information is assumed to be obstacle-free, in order to always move the agent along a shortest, potentially unblocked, path in the environment. However, even under the freespace assumption, the environment is assumed to be bounded (usually within a rectangle). Our definition of the freespace grid also assumes that the environment (or the corresponding grid) extends infinitely in all directions. Without this assumption, the translation invariance of freespace distances that we discuss in Section 4.4.2 does not hold.

Figure 4.2 shows an example that motivates our definition of the freespace grid as an infinite grid, rather than the smallest rectangle of unblocked cells that contains $\mathcal{G}$. The underlying grid has a rectangle of unblocked cells, surrounded by (a boundary of) blocked cells. The path shown in Figure 4.2(a) is the shortest possible path between the start (blue arrow) and goal (red arrow) states: even if the rectangle of blocked cells were infinitely extended, there would not be a shorter path. Therefore, it is a freespace-shortest path. The orientations and relative x- and y-coordinates of the start and goal states in Figure 4.2(b) are the same as in Figure 4.2(a). However, the shortest path between them is longer. If the rectangle of unblocked cells is sufficiently extended in all directions, a path equivalent to the one in Figure 4.2(a) (that is, with the same sequence

of primitives) would be a shorter path. Therefore, the path shown in Figure 4.2(b) is not freespace-shortest.

As we have discussed in Section 4.2, a primitive $\vec{m}$ may induce multiple edges in a state lattice, namely, an edge for every state from which $\vec{m}$ is kinematically feasible and collision-free to execute. The set of states from which $\vec{m}$ is kinematically feasible to execute can be considered to be "regular", in the sense that they can be characterized easily as those states with orientation θ_m^s. The set of states from which $\vec{m}$ is collision-free to execute can be considered to be "irregular", depending on the underlying grid $\mathcal{G}$. On freespace state lattices, the set of states from which $\vec{m}$ is collision-free to execute is also "regular": Since there are no blocked cells, every state from which $\vec{m}$ is kinematically feasible to execute is also a state from which $\vec{m}$ is collision-free to execute. We use this "regularity" of freespace state lattices in Section 4.4.2 to prove the translation invariance of freespace-distances, and in Section 4.5.2 to prove the *translation invariance of freespace-canonical paths*.

The freespace state lattice $\mathcal{F} = \mathcal{L}_{\mathbb{Z}^2,\mathcal{M}}$ is a supergraph of the state lattice $\mathcal{L}_{\mathcal{G},\mathcal{M}}$, that is, $\mathcal{L}_{\mathbb{Z}^2,\mathcal{M}}$ contains a superset of the edges of $\mathcal{L}_{\mathcal{G},\mathcal{M}}$, since the underlying grid $\mathbb{Z}^2$ of the freespace state lattice $\mathcal{F} = \mathcal{L}_{\mathbb{Z}^2,\mathcal{M}}$ contains a superset of the unblocked cells of the underlying grid $\mathcal{G}$ of $\mathcal{L}_{\mathcal{G},\mathcal{M}}$. Furthermore, since G is defined as the largest strongly-connected component of $\mathcal{L}_{\mathcal{G},\mathcal{M}}$ (Assumption 4.2), $\mathcal{F} = \mathcal{L}_{\mathbb{Z}^2,\mathcal{M}}$ is also a supergraph of G. Lemmata 4.2, 4.3, 4.4, and 4.5 follow from this observation, and allow us to use precomputed freespace distances (or trees of freespace-shortest paths) in the connection and refinement algorithms that we develop in this dissertation.

Lemma 4.2. *Every path π on G is a path on $\mathcal{F}$ with the same length. For every $s, t \in V$, $d_{\mathcal{F}}(s,t) \leq d(s,t)$.*

Proof. By Definition 4.1, since $\mathcal{G} \subseteq \mathbb{Z}^2$, $\mathcal{F} = \mathcal{L}_{\mathbb{Z}^2,\mathcal{M}}$ is a supergraph of $\mathcal{L}_{\mathcal{G},\mathcal{M}}$. By Assumption 4.2, $\mathcal{L}_{\mathcal{G},\mathcal{M}}$ is a supergraph of G. Therefore, $\mathcal{F}$ is a supergraph of G and contains all edges of G with the same lengths, and all paths on G with the same lengths. Consequently, distances on $\mathcal{F}$ cannot be larger than distances on G.

□

Lemma 4.3. *A freespace-shortest s-t path π on G is a shortest s-t path on G.*

Proof. Since π is freespace-shortest, by Definition 4.1, $l(\pi) = d_{\mathcal{F}}(s,t)$. From Lemma 4.2, $d_{\mathcal{F}}(s,t) \leq d(s,t)$. Therefore, $l(\pi) \leq d(s,t)$ and, consequently, π is a shortest s-t path on G. □

Lemma 4.4. *A freespace-shortest s-t path exists on G if and only if $d_{\mathcal{F}}(s,t) = d(s,t)$.*

Proof. Assume that a freespace-shortest s-t path π exists on G. By Definition 4.1, $l(\pi) = d_{\mathcal{F}}(s,t)$, and, by Lemma 4.3, $l(\pi) = d(s,t)$. Therefore, $d_{\mathcal{F}}(s,t) = d(s,t)$.

Assume that $d_{\mathcal{F}}(s,t) = d(s,t)$. Let π be a shortest s-t path on G with $l(\pi) = d(s,t) = d_{\mathcal{F}}(s,t)$. By Definition 4.1, π is freespace-shortest. □

Lemma 4.5. *Every subpath of a freespace-shortest path is freespace-shortest.*

Proof. A freespace-shortest path is a shortest path on $\mathcal{F}$ (Definition 4.1). Therefore, its subpaths are also shortest paths on $\mathcal{F}$ and thus are freespace-shortest. □

4.4.2 Translation Invariance of Freespace Distances

In Euclidean geometry, a translation is a transformation that moves every point of an object by the same distance in a given direction. On state lattices, we define a translation as changing the x- or y-coordinates of every point (state) of an object (a state, a set of states, an edge, or a path) by the same *integral* amount. In this section, we prove the *translation invariance of freespace distances*, that is, that translating a pair of states does not change the distance between them on the freespace state lattice, which allows for the efficient computation and compact storage of freespace distances.

Lemmata 4.6, 4.7, and 4.8 prove the translation invariance of edges, paths, and shortest paths on freespace state lattices, respectively. Theorem 4.19 proves the translation invariance of freespace distances. We use the following notation to denote translations on state lattices: Translating a state $s = (x_s, y_s, \theta_s)$ by (x, y) (by the integers x and y) results in the state $s' = (x_s + x, y_s + y, \theta_s)$, which we denote as $s + (x, y) = s'$. Translating a path $\pi = \langle p_0, \dots, p_k \rangle$ by (x, y) results in the path $\pi' = \langle p_0 + (x, y), \dots, p_k + (x, y) \rangle$, which we denote as $\pi + (x, y) = \pi'$.

Lemma 4.6 (Translation invariance of edges on $\mathcal{F}$)**.** *Let (u, v) be an edge of $\mathcal{F}$. For any two integers x and y, let $u' = u + (x, y)$ and $v' = v + (x, y)$. (u', v') is an edge of $\mathcal{F}$ with $c_{\mathcal{F}}(u, v) = c_{\mathcal{F}}(u', v')$.*

Proof. Since $\vec{m}$ induces the edge (u, v) in $\mathcal{F}$, it holds that $u + \vec{m} = v$ and $l_{\vec{m}} = c_{\mathcal{F}}(u, v)$. Since $u' = u + (x, y)$ has the same orientation as u, $\vec{m}$ is kinematically feasible to execute from u'. Since $\mathcal{G}$ has no blocked cells, $\vec{m}$ is collision-free to execute from u'. Therefore $\vec{m}$ is executable from u'. When $\vec{m}$ is executed from u', the resulting state is $u' + \vec{m} = u + (x, y) + \vec{m} = u + \vec{m} + (x, y) = v + (x, y) = v'$. Therefore, $\vec{m}$ induces the edge (u', v') in $\mathcal{F}$ with $c_{\mathcal{F}}(u', v') = l_{\vec{m}} = c_{\mathcal{F}}(u, v)$. □

Lemma 4.7 (Translation invariance of freespace paths)**.** *Let $\pi = \langle p_0, \dots, p_k \rangle$ be a freespace path. For any two integers x and y, let $\pi' = \langle p'_0, \dots, p'_k \rangle$ such that, for all $i = 0, \dots, k$, $p'_i = p_i + (x, y)$. π' is a freespace path with $l(\pi') = l(\pi)$.*

Proof. For all $i = 0, \dots, k - 1$, since (p_i, p_{i+1}) appears on π, it is an edge of $\mathcal{F}$ and, consequently, (p'_i, p'_{i+1}) is an edge of $\mathcal{F}$ with $c_{\mathcal{F}}(p'_i, p'_{i+1}) = c_{\mathcal{F}}(p_i, p_{i+1})$ (Lemma 4.6). Therefore π' is a path on $\mathcal{F}$ with length $l(\pi) = l(\pi')$. □

Lemma 4.8 (Translation invariance of freespace-shortest paths)**.** *Let π be a freespace-shortest path. For any two integers x and y, $\pi' = \pi + (x, y)$ is a freespace-shortest path.*

Proof. Let s and t denote the first and last vertex on π, and let s' and t' denote the first and last vertex on π', respectively. Assume (for contradiction) that an s'-t' path π'' exists on $\mathcal{F}$ with $l(\pi'') < l(\pi')$. Then $\pi'' + (-x, -y)$ is a freespace s-t path with length $l(\pi'') < l(\pi') = l(\pi)$ (Lemma 4.7), contradicting that π is a freespace-shortest s-t path (Definition 4.1). □

Theorem 4.9 (Translation invariance of freespace distances)**.** *Let s and t be two states. For any two integers x and y, let $s' = s + (x, y)$ and $t' = t + (x, y)$. $d_{\mathcal{F}}(s, t) = d_{\mathcal{F}}(s', t')$.*

Proof. Our assumption that G is strongly connected (Assumption 4.2) does not apply to $\mathcal{F}$. If $d_{\mathcal{F}}(s,t) = \infty$, then $d_{\mathcal{F}}(s',t') = \infty$ because, otherwise, a freespace s'-t' path π' exists with $l(\pi') < \infty$ and, therefore, the freespace s-t path $\pi' + (-x,-y)$ exists with length $l(\pi) = l(\pi') < \infty$ (Lemma 4.7), contradicting that $\mathcal{F}(s,t) = \infty$. If $d_{\mathcal{F}}(s,t) < \infty$, then $d_{\mathcal{F}}(s,t) = d_{\mathcal{F}}(s',t')$ follows from Lemmata 4.7 and 4.8. □

The translation invariance of freespace distances allows for the efficient computation and compact storage of pairwise freespace distances. Namely, by Theorem 4.9, for every $s = (x_s, y_s, \theta_s)$ and $t = (x_t, y_t, \theta_t)$, the freespace s-t distance is equal to the freespace s'-t' distance, where $s' = s + (-x_s, -y_s) = (0, 0, \theta_s)$ and $t' = t + (-x_s, -y_s)$. That is, computing and storing freespace distances from states $(0, 0, \theta)$, for all $\theta \in O_{\mathcal{M}}$, is sufficient for computing and storing pairwise freespace distances between all states that appear on state lattices.[1]

4.4.3 Freespace Reachability (F-Reachability)

In order to be able to use efficiently computed and compactly stored freespace distances during the connection and refinement phases of queries answered using subgoal graphs, we can construct subgoal graphs with respect to *freespace-reachability*: A state t is freespace-reachable from a state s if and only if a freespace-shortest s-t path is unblocked on G, or, equivalently, if and only if $d_{\mathcal{F}}(s,t) = d(s,t)$ (Lemma 4.4). Definition 4.2 formally defines freespace-reachability as well as a variant, called bounded-freespace-reachability.

Definition 4.2. *A vertex t is freespace-reachable (F-reachable) from a vertex s, denoted by $(s,t) \in F$, if and only if a freespace-shortest s-t path exists on G (equivalently, by Lemma 4.4, if and only if $d_{\mathcal{F}}(s,t) = d(s,t)$). t is bounded-freespace-reachable with reachability bound $b > 0$ (Fb-reachable) from s if and only if t is F-reachable from s and $d(s,t) \leq b$.*

Lemma 4.10. *Every edge (u,v) of $\mathcal{F}$ is the unique freespace-shortest u-v path.*

Proof. Proof is similar to Step 3 of the proof of Lemma 4.1, and follows from $\mathcal{M}$ being well-behaved (Assumption 4.3). □

Theorem 4.11. *F is a reachability relation. Fb is a reachability relation if, for all $\vec{m} \in \mathcal{M}$, $l_{\vec{m}} \leq b$.*

Proof. For every $n \in V$, $(n,n) \in \mathrm{F}$ and $(n,n) \in \mathrm{F}b$ since $\langle n \rangle$ is a freespace-shortest n-n path on G with $d(n,n) = 0 \leq b$. For every $(u,v) \in E$, $(u,v) \in \mathrm{F}$ since $\langle u,v \rangle$ is a freespace-shortest u-v path (Lemma 4.10) on G. If, for every $\vec{m} \in \mathcal{M}$, $l_{\vec{m}} \leq b$, then $c(u,v) = l_{\vec{m}} \leq b$, where $\vec{m}$ is the primitive that induces the edge (u,v). Consequently, $(u,v) \in \mathrm{F}b$. □

[1] Since we define freespace state lattices to be infinite graphs (Definition 4.1), it is not possible to store pairwise distances between all pairs of states on a freespace state lattice. However, we are only interested in freespace distances between vertices that appear on (actual) state lattices, which are finite (Assumption 4.1).

Using a reachability bound b for F-reachability allows us to limit the calculation and storage of freespace distances to only those pairs of vertices between which the freespace distance is no more than b. It also allows us to influence the trade-off between the connection and search times of queries answered using (strongly-connected) subgoal graphs, as we discuss further and experimentally evaluate in Section 4.6. Fb-reachability can be considered as the combination of F-reachability and BDb-reachability, which we have introduced in Section 3.3.1. That is $\mathrm{F}b = \mathrm{F} \cap \mathrm{BD}b$.

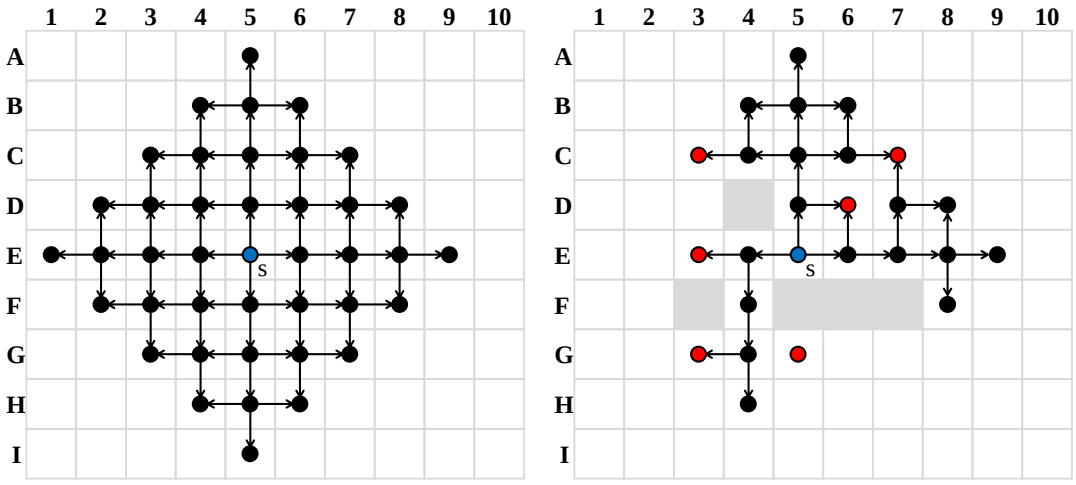
4.4.4 F-Connect

In this section, we introduce a connection algorithm for F-reachability, called F-Connect, and prove its correctness with respect to the criteria that we have outlined in Section 3.3.4 for answering queries using F (strongly-connected) subgoal graphs. That is, we show that F-Connect connects a given start vertex s (or a given goal vertex t) to all direct-F-reachable subgoals from s, never connects s to a subgoal that is not F-reachable from s, and determines the correct lengths for the connecting edges as distances on G. For brevity, we only describe a version of F-Connect, called $\mathrm{F}^{\rightarrow}$-Connect, that connects the start vertex, but not the goal vertex, to an F (strongly-connected) subgoal graph, and describe briefly how it can be modified to also connect the goal vertex. Furthermore, we omit providing a connection algorithm for Fb-reachability, since F-Connect can be modified to become a connection algorithm for Fb-reachability by simply disallowing it from generating vertices whose distances from the start vertex are larger than b.

A connection algorithm for F-reachability needs to connect a given start vertex to subgoals (or the goal vertex) that are direct-F-reachable from it, and only to subgoals that are F-reachable from it, as discussed in Section 3.3.4. Algorithm 10 outlines $\mathrm{F}^{\rightarrow}$-Connect, which explores exactly the freespace-shortest paths that originate at a given start vertex s that are unblocked on G and not covered by subgoals: Any vertex that is reached with a shortest path that is not freespace-shortest cannot be F-reachable from s (Definition 4.2), and any vertex that is reached with a freespace-shortest path that is covered by a subgoal cannot be direct-F-reachable from s (Definition 3.7). Figure 4.3 shows an example of the operation of $\mathrm{F}^{\rightarrow}$-Connect on a 4-neighbor grid graph, and Figure 4.4 shows an example on a state lattice constructed with respect to Unicycle primitives.

$\mathrm{F}^{\rightarrow}$-Connect operates similarly to the aggressive variant of Overlay-Connect (that is, a Dijkstra search that does not expand subgoals), that we have discussed in Section 3.2.5, with two differences that are highlighted in blue in Algorithm 10:

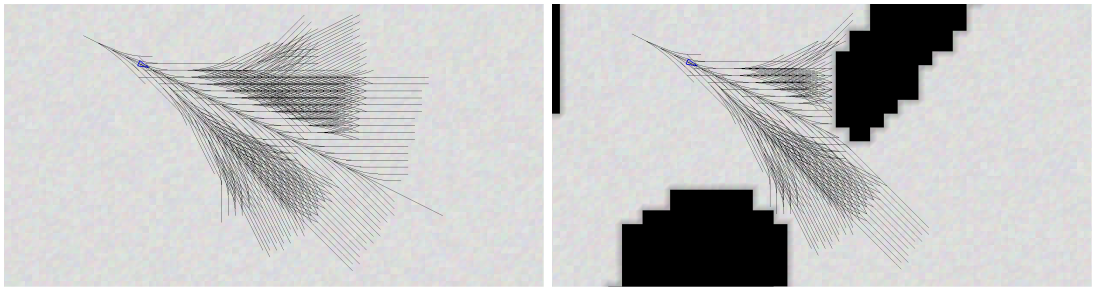
- $\mathrm{F}^{\rightarrow}$-Connect avoids generating vertices that are not F-reachable from s by checking, for each successor v of an expanded vertex u, whether $d_{\mathcal{F}}(s,u) + c(u,v) = d_{\mathcal{F}}(s,v)$ (line 11). As we prove later in Lemma 4.12, this check is sufficient to maintain the invariant that all vertices placed in the OPEN list are F-reachable from s. This proof can be summarized as follows: Intuitively, assuming that the invariant holds, for every vertex u selected for expansion, the search tree contains a freespace-shortest s-u path. If the edge (u,v) extends this path to a freespace-shortest s-v path (that is, if $d_{\mathcal{F}}(s,u) + c(u,v) = d_{\mathcal{F}}(s,v)$), then we can verify that $(s,v) \in \mathrm{F}$ and v is added to OPEN, which does not violate the invariant. Otherwise, v is not added to OPEN



(a) All freespace-shortest paths on $\mathcal{F}$ that originate at s and have length 4 or less. $\mathcal{F}$ is shown only partially, since it is infinite.

(b) F4-Connect explores all freespace-shortest paths on G that originate at s, have length 4 or less, and are not covered by subgoals (red).

Figure 4.3: F4-Connect on a 4-neighbor grid graph.



(a) All freespace-shortest paths on $\mathcal{F}$ that originate at a state and have length 50 or less. $\mathcal{F}$ is only shown partially, since it is infinite.

(b) F50-Connect explores all freespace-shortest paths on G that originate at s, have length 4 or less, and are not covered by subgoals (none are present in this example).

Figure 4.4: F50-Connect on a state lattice constructed using Unicycle primitives.

(but it may still be added to OPEN eventually if a freespace-shortest path to it is found through another vertex).

- F$^{\rightarrow}$-Connect can expand vertices in an arbitrary order (line 5) and therefore does not require its OPEN list to be implemented as a priority queue (line 2). Recall that, in order to guarantee that each vertex is expanded with the correct g-value, a Dijkstra search always chooses the next vertex to expand as the vertex with the minimum g-value in its OPEN list. However, since Algorithm 10 maintains the invariant that all vertices in OPEN are F-reachable from s, the distance from s to any vertex $u \in$ OPEN is $d(s,u) = d_{\mathcal{F}}(s,u)$ (Definition 4.2). Therefore, Algorithm 10 can determine the correct distance for the edges it identifies by simply performing a lookup for $d_{\mathcal{F}}(s,u)$ (line 8).

Algorithm 10 $F^{\rightarrow}$-Connect

Blue text: Differences from the aggressive variant of Overlay-Connect (Section 3.2.5).

Input: A state lattice $G = (V, E, c)$, start vertex s, covering vertices $S \subseteq V$, target vertices $T \subseteq V$

Output: A set of edges E^+ such that $F_S^{s \rightarrow T} \subseteq E^+ \subseteq F$, edge lengths c^+ such that, $\forall (u, v) \in E^+$, $c^+(u, v) = d_{\mathcal{F}}(u, v) = d(u, v)$

1: $E^+ \leftarrow \emptyset$
2: OPEN $\leftarrow \{s\}$ ▷ Implemented as a stack for depth-first search
3: CLOSED $\leftarrow \{\}$ ▷ For duplicate detection
4: **while** OPEN $\neq \emptyset$ **do**
5: $u \leftarrow$ any vertex in OPEN
6: Move u from OPEN to CLOSED
7: **if** $u \in T \setminus \{s\}$ **then**
8: Add (s, u) to E^+ with $c^+(u, v) = d_{\mathcal{F}}(s, u)$
9: **if** $u \notin S \setminus \{s\}$ **then**
10: **for all** successors v of u such that $v \notin$ OPEN∪CLOSED **do**
11: **if** $d_{\mathcal{F}}(s, u) + c(u, v) = d_{\mathcal{F}}(s, v)$ **then**
12: Add v to OPEN
13: return E^+, c^+

Given that $F^{\rightarrow}$-Connect can expand vertices in an arbitrary order, its OPEN list can be implemented as a stack to expand vertices in a depth-first order. Each insertion and removal from the OPEN list thus requires $O(1)$ time. Since only vertices that are F-reachable from s are placed in OPEN (Lemma 4.12), the total time to perform insertions and removals from OPEN is $O(|F^{s \rightarrow V}|)$, where $F^{s \rightarrow V}$ is the set of vertices that are F-reachable from s. Since each expansion iterates over all successors of the expanded vertex (line 10) and performs two $O(1)$ time freespace-distance lookups per successor (line 11), the overall runtime of Algorithm 10 is $O(|F^{s \rightarrow V}|B)$, where B is the maximum out-degree of a vertex in G. (For an expanded vertex u, $d_{\mathcal{F}}(s, u)$ needs to be looked-up only once, bringing down the number of table lookups per expansion to no more than $|B| + 1$.)

$F^{\rightarrow}$-Connect can be modified to connect a goal vertex t to all vertices from which it is direct-F-reachable by running it on the reverse graph of G, and reversing the order of states in freespace-distance table look-ups. That is, when expanding a vertex u, for each predecessor v of u, it can be determined that t is F-reachable from v if $d_{\mathcal{F}}(u, t) + c(u, v) = d_{\mathcal{F}}(v, t)$. We refer to this algorithm as $F^{\leftarrow}$-Connect, and assume that F-Connect operates by first running $F^{\rightarrow}$-Connect from the start vertex and then running $F^{\leftarrow}$-Connect from the goal vertex.

We now prove the correctness of $F^{\rightarrow}$-Connect with respect to the criteria that we have outlined in Section 3.3.4 for answering queries using F (strongly-connected) subgoal graphs. Namely, we show that $F^{\rightarrow}$-Connect connects a given source vertex s to all direct-F-reachable subgoals from s, never connects s to a subgoal that is not F-reachable from s, and determines the correct lengths for the connecting edges as distances on G. We omit the proof of correctness for $F^{\leftarrow}$-Connect, which is similar to the proof of correctness for $F^{\rightarrow}$-Connect. We also omit the proof that $F^{\rightarrow}$-Connect terminates, which follows from

the fact that it uses OPEN and CLOSED lists to expand each vertex at most once, and our assumption that G is finite (Assumption 1.1), similar to the search framework that we have discussed in Section 2.2. We prove in Lemma 4.12 that F$^{\rightarrow}$-Connect expands only vertices that are F-reachable from s, and we prove in Lemma 4.13 that it is guaranteed to expand all vertices that are direct-F-reachable from s.

Lemma 4.12. *At any point during the execution of Algorithm 10, every vertex $u \in$ OPEN is F-reachable from s.*

Proof.

1. Let $\mathcal{S}_i$ be the statement "The ith time that line 4 is executed, for every vertex $n \in$ OPEN, $(s, n) \in$ F".
2. For all $i = 1, \ldots, k$, where k is the number of times that line 4 is executed, S_i is true (proof by induction on i):
   - 2.1. Base case: $\mathcal{S}_1$ is true (since OPEN $= \{s\}$, $(s, s) \in F$).
   - 2.2. Induction step: If $\mathcal{S}_i$ is true, then $\mathcal{S}_{i+1}$ is true:
     - 2.2.1. Assume $\mathcal{S}_i$ is true.
     - 2.2.2. Let u be the vertex selected for expansion on line 5 immediately after line 4 is executed for the ith time.
     - 2.2.3. Let v be any vertex added to OPEN (line 12) after line 4 is executed for the ith time but before the $(i+1)$st time.
     - 2.2.4. $(s, u) \in F$ (since $\mathcal{S}_i$ is true).
     - 2.2.5. $(u, v) \in E$ (line 10, since v is a successor of u).
     - 2.2.6. $d_{\mathcal{F}}(s, u) + c(u, v) = d_{\mathcal{F}}(s, v)$ (line 11).
     - 2.2.7. Let π_u be a freespace-shortest s-u path on G with $l(\pi_u) = d_{\mathcal{F}}(s, u)$. Such π_u exists (Definition 4.2, since $(s, u) \in F$).
     - 2.2.8. Let $\pi_v = \pi_u.(u, v)$ with $l(\pi_v) = l(\pi_u) + c(u, v)$.
     - 2.2.9. $l(\pi_v) = l(\pi_u) + c(u, v) = d_{\mathcal{F}}(\pi_u) + c(u, v) = d_{\mathcal{F}}(s, v)$.
     - 2.2.10. π_v is freespace-shortest (Definition 4.1, since $l(\pi_v) = d_{\mathcal{F}}(s, v)$).
     - 2.2.11. π_v is a path on G (since $\pi_v = \pi_u.(u, v)$, π_u is a path on G, and $(u, v) \in E$).
     - 2.2.12. $(s, v) \in F$ (Definition 4.2, since π_v is a freespace-shortest s-v path on G).

□

Lemma 4.13. *For every $n \in V$, if n is direct-F-reachable from s, then Algorithm 10 selects n for expansion (line 5) at some point during its execution.*

Proof.

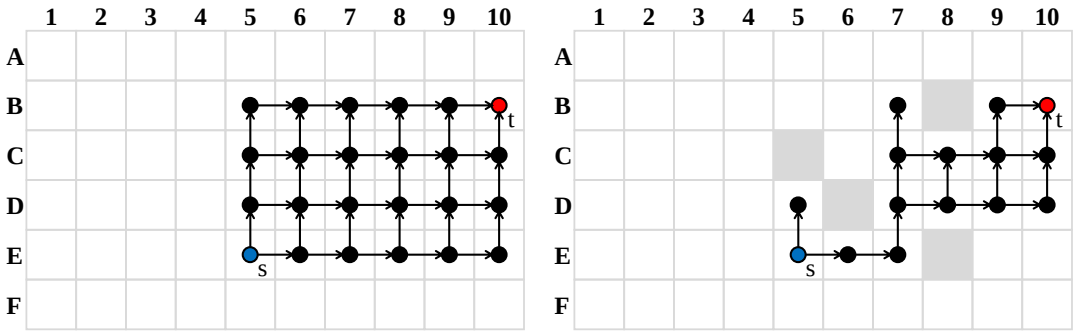
1. Let $n \in V$ with $(s, n) \in \mathrm{F}_S^{s \rightarrow V}$.
2. Let $\pi = \langle p_0, \ldots, p_k \rangle$ be a freespace-shortest s-n path on G. Such π exists (Definition 4.2, since $(s, n) \in$ F).

3. π is a shortest s-n path on G (Lemma 4.3, since π is freespace-shortest).
4. $S \not\sqsubset (s, n)$ (Definition 3.2, since $(s, n) \in \mathrm{F}_S^{s \to V}$).
5. $S \not\sqsubset \pi$ (Definition 3.1, since $S \not\sqsubset (s, n)$ and π is a shortest s-n path).
6. For $i = 0, \ldots, k-1$, $p_i \notin S \setminus \{s\}$:
   6.1. If $i = 0$, then $p_i = p_0 = s \notin S \setminus \{s\}$.
   6.2. If $0 < i < k$, then $p_i \notin S$ (Definition 3.1, since $S \not\sqsubset \pi$).
7. Let $\mathcal{S}_i$ be the statement "Algorithm 10 selects p_i for expansion on line 6 at some point during its execution".
8. For all $i = 0, \ldots, k$, S_i is true (proof by induction on $i = 0, \ldots, k-1$):
   8.1. Base case: $\mathcal{S}_0$ is true (since OPEN $= \{s\}$ the first time line 4 is executed).
   8.2. Induction step: If $\mathcal{S}_i$ is true, then $\mathcal{S}_{i+1}$ is true:
      8.2.1. Assume $\mathcal{S}_i$ is true.
      8.2.2. p_i is selected for expansion (since $\mathcal{S}_i$ is true).
      8.2.3. Consider the point in execution when p_i is selected for expansion.
      8.2.4. If $p_{i+1} \in$ OPEN, p_{i+1} is eventually selected for expansion (line 4).
      8.2.5. If $p_{i+1} \in$ CLOSED, p_{i+1} has already been selected for expansion (line 6).
      8.2.6. Otherwise, p_{i+1} is added to OPEN:
         8.2.6.1. Lines 10–12 are executed for p_i (line 9, since $p_i \notin S \setminus \{s\}$).
         8.2.6.2. $\pi_i = \langle p_0, \ldots, p_i \rangle$ and $\pi_{i+1} = \langle p_0, \ldots, p_{i+1} \rangle$ are freespace-shortest paths on G (Lemma 4.5, since π_i and π_{i+1} are subpaths of the freespace-shortest path π on G).
         8.2.6.3. $l(\pi_i) = d_{\mathcal{F}}(s, p_i)$ and $l(\pi_{i+1}) = d_{\mathcal{F}}(s, p_i)$ (Definition 4.1, since π_i and π_{i+1} are freespace-shortest).
         8.2.6.4. $(p_i, p_{i+1}) \in E$ (since π is a path on G).
         8.2.6.5. $d_{\mathcal{F}}(s, p_{i+1}) = l(\pi_{i+1}) = l(\pi_i) + c(u, v) = d_{\mathcal{F}}(s, p_i) + c(p_i, p_{i+1})$.
         8.2.6.6. p_{i+1} is added to OPEN (since lines 10–12 are executed for p_i, $(p_i, p_{i+1}) \in E$, and $d_{\mathcal{F}}(s, p_i) + c(p_i, p_{i+1}) = d_{\mathcal{F}}(s, p_{i+1})$).

□

Theorem 4.14. *Algorithm 10 returns a set of edges E^+ with lengths c^+ such that $\mathrm{F}_S^{s \to T} \subseteq E^+ \subseteq F$ and, for every $(u, v) \in E^+$, $c^+(u, v) = d(u, v)$.*

Proof. By Lemma 4.12, Algorithm 10 only expands vertices that are F-reachable from s. By Lemma 4.13, Algorithm 10 is guaranteed to expand all vertices that are direct-F-reachable from s. Therefore, Algorithm 10 identifies a set of edges E^+ with $\mathrm{F}_S^{s \to T} \subseteq E^+ \subseteq \mathrm{F}$. Furthermore, since $E^+ \subseteq \mathrm{F}$, Algorithm 10 correctly determines the lengths of edges in E^+ as their freespace-distances (Definition 4.2). □



(a) All freespace-shortest s-t paths on $\mathcal{F}$. $\mathcal{F}$ is only shown partially, since it is infinite.

(b) F-Refine only explores freespace-shortest paths on G to states that appear on freespace-shortest s-t paths on $\mathcal{F}$.

Figure 4.5: F-Refine.

4.4.5 F-Refine

In this section, we introduce a refinement algorithm for F-reachability, called F-Refine, and prove its correctness with respect to the criteria that we have outlined in Section 3.3.4 for answering queries using F (strongly-connected) subgoal graphs. That is, we show that F-Refine is guaranteed to find a shortest s-t path on G, given any $(s,t) \in \mathrm{F}$. F-Refine can be used as a refinement algorithm for Fb-reachability as well, without any changes.

Algorithm 11 F-Refine

Input: $(s,t) \in \mathrm{F}$, $G = (V, E, c)$
Output: A shortest (freespace) s-t path π on G

```
1:  OPEN := {s}                                  ▷ Implemented as a stack for DFS
2:  CLOSED := {}
3:  while OPEN ≠ ∅ do
4:      u := any vertex in OPEN
5:      if u = t then
6:          break
7:      Move u from OPEN to CLOSED
8:      for all successors v of u such that v ∉ OPEN ∪ CLOSED do
9:          if d_F(s,u) + c(u,v) + d_F(v,t) = d_F(s,t) then
10:             parent(v) = u
11:             add v to OPEN
12: π ← ⟨t⟩
13: u ← t
14: while u ≠ s do
15:     π ← ⟨parent(u), u⟩ · π
16:     u ← parent(u)
17: return π
```

Algorithm 11 outlines F-Refine, which only explores freespace-shortest paths on G to states that appear on freespace-shortest s-t paths on $\mathcal{F}$: If a vertex does not appear on a

freespace-shortest s-t path on $\mathcal{F}$, then it cannot appear on a freespace-shortest s-t path on G, since, as discussed in Section 4.4.1, the set of paths on $\mathcal{F}$ is a superset of the set of paths on G. F-Refine maintains the invariant that each vertex in OPEN appears on at least one shortest s-t path on $\mathcal{F}$. It does so by checking, for each successor v of an expanded vertex u, whether $d_{\mathcal{F}}(s,u)+c(u,v)+d_{\mathcal{F}}(v,t) = d_{\mathcal{F}}(s,t)$ (lines 8–9). Intuitively, if the edge (u,v) appears on a freespace-shortest path $\pi = \pi' \cdot \langle u,v\rangle \cdot \pi''$, it must hold that $d_{\mathcal{F}}(s,t) = l(\pi) = l(\pi') + c(u,v) + l(\pi'') = d_{\mathcal{F}}(s,u) + c(u,v) + d_{\mathcal{F}}(v,t)$. If so, then v is added to OPEN since it could be verified that v can appear on a freespace-shortest s-t path (lines 10–11). Otherwise, v is not added to OPEN.

F-Refine can be considered as a "goal-directed" version of F$^{\rightarrow}$-Connect. Whereas F$^{\rightarrow}$-Connect avoids generating vertices that are not freespace-reachable from s, F-Refine avoids generating vertices that are not freespace-reachable from s *and* vertices that cannot appear on freespace-shortest s-t paths on $\mathcal{F}$. We omit an extensive proof of the correctness of F-Refine, which is similar to the proof of correctness of F-Connect, but only prove that its "goal-directed" pruning is correct.

Lemma 4.15. *For every $(s,t) \in F$ and edge $(u,v) \in E$, if $d_{\mathcal{F}}(s,u) + c(u,v) + d_{\mathcal{F}}(v,t) \neq d_{\mathcal{F}}(s,t)$, then (u,v) cannot appear on a shortest s-t path on G.*

Proof.

1. Let $(s,t) \in$ F.
2. Let $(u,v) \in E$.
3. Let $d_{\mathcal{F}}(s,u) + c(u,v) + d_{\mathcal{F}}(v,t) \neq d_{\mathcal{F}}(s,t)$.
4. Assume (for contradiction) that a shortest s-t path $\pi = \pi' \cdot \langle u,v\rangle \cdot \pi''$ exists on G.
5. $l(\pi) = d(s,t)$ (since π is a shortest s-t path on G).
6. $\quad = d_{\mathcal{F}}(s,t)$ (Definition 4.2, since $(s,t) \in$ F).
7. π is a freespace-shortest s-t path (Definition 4.1, since $l(\pi) = d_{\mathcal{F}}(s,t)$).
8. π', $\langle u,v\rangle$, and π'' are freespace-shortest (Lemma 4.5, since they are subpaths of the freespace-shortest path π).
9. $l(\pi) = l(\pi') + l(\langle u,v\rangle) + l(\pi'')$ (since $\pi = \pi' \cdot \langle u,v\rangle \cdot \pi''$).
10. $d_{\mathcal{F}}(s,t) = d_{\mathcal{F}}(s,u) + d_{\mathcal{F}}(u,v) + d_{\mathcal{F}}(v,t)$ (Definition 4.1, since π, π', $\langle u,v\rangle$, and π'' are s-t, s-u, u-v, and v-t freespace-shortest paths, respectively).
11. $d_{\mathcal{F}}(u,v) = c(u,v)$ (Lemma 4.10, since $(u,v) \in E$).
12. $d_{\mathcal{F}}(s,t) = d_{\mathcal{F}}(s,u) + c(u,v) + d_{\mathcal{F}}(v,t)$.
13. $\perp$ (since $d_{\mathcal{F}}(s,t) = d_{\mathcal{F}}(s,u) + c(u,v) + d_{\mathcal{F}}(v,t) \neq d_{\mathcal{F}}(s,t)$).

□

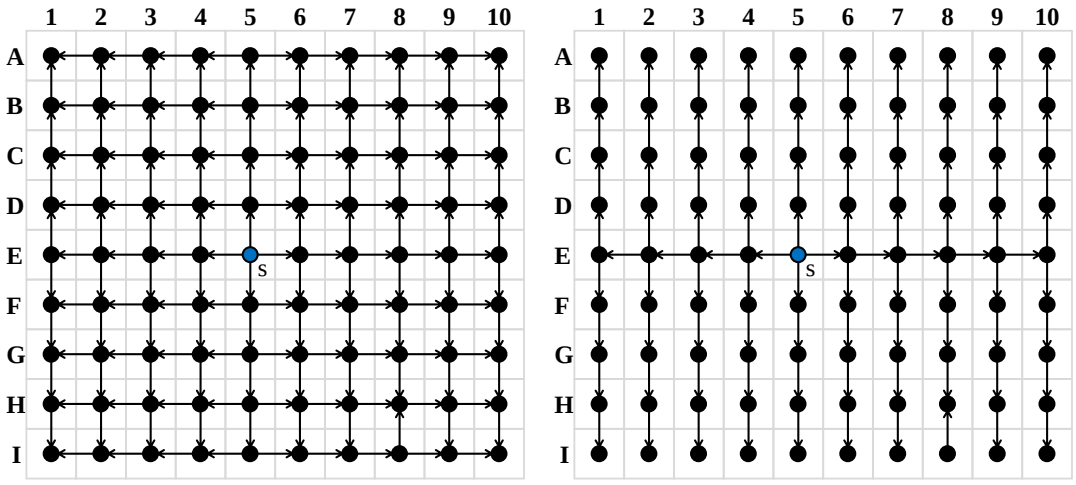
4.5 Canonical Freespace Reachability

In the previous section, we have introduced F-reachability and developed connection and refinement algorithms for it that use precomputed freespace distances, which can be efficiently computed and compactly stored by exploiting the translation invariance of freespace distances. In this section, we introduce a variant of F-reachability, called *canonical-freespace-reachability* (CF-reachability) and develop connection and refinement algorithms for it. The main idea behind CF-reachability is to use a canonical ordering to distinguish, among multiple freespace-shortest paths between two vertices, exactly one as the *freespace-canonical* path. A vertex is CF-reachable from another vertex if and only if the freespace-canonical path between them is unblocked on G. We introduce a connection algorithm for CF-reachability that explores only the freespace-canonical paths rather than all freespace-shortest ones, considers only a subset of the successors of expanded vertices, and does not have to perform duplicate detection (that is, check whether each successor of an expanded vertex already appears in OPEN or CLOSED). We introduce a refinement algorithm for CF-reachability that simply generates the freespace-canonical path between two CF-reachable vertices. These connection and refinement algorithms use precomputed *freespace-canonical successors, predecessors, and parents* rather than freespace distances. We prove that, similar to freespace distances, these values can be efficiently computed and compactly stored by exploiting the *translation invariance of freespace-canonical paths*.

This section is organized as follows. In Section 4.5.1, we describe canonical orderings defined over the primitives of state lattices, and show how they can be used to uniquely designate, among multiple freespace-shortest paths between two vertices, exactly one as the *freespace-canonical* path. In Section 4.5.2, we prove the translation invariance of freespace-canonical paths, as well as prove that the set of freespace-canonical paths that originate (or terminate) at a vertex form a tree. In Section 4.5.3, we introduce canonical-freespace-reachability as a reachability relation on state lattices. In Sections 4.5.4 and 4.5.5, we introduce connection and refinement algorithms for canonical-freespace-reachability.

4.5.1 Symmetries and Canonical Orderings

On graphs with arbitrary edge lengths, such as road networks, there is typically a unique shortest path between any two vertices. However, on state lattices that are constructed with respect to a small set of primitives (or, more generally, on state spaces constructed with respect to a small set of actions), different permutations of a sequence of primitives (actions) may correspond to multiple *symmetric* shortest paths between two states. Canonical orderings are total or partial orderings on the primitives (actions) available to an agent, and can be used to distinguish one or multiple (shortest) paths between two states as the *canonical* one(s). Canonical orderings have been used in search algorithms that cannot perform proper duplicate detection (due to limited memory) to reduce the number of duplicate nodes in their search trees (Taylor & Korf, 1993; Holte & Burch, 2014), and by Jump Point Search for symmetry breaking on grid graphs (Harabor &



(a) All freespace-shortest paths on $\mathcal{F}$ that originate at s.

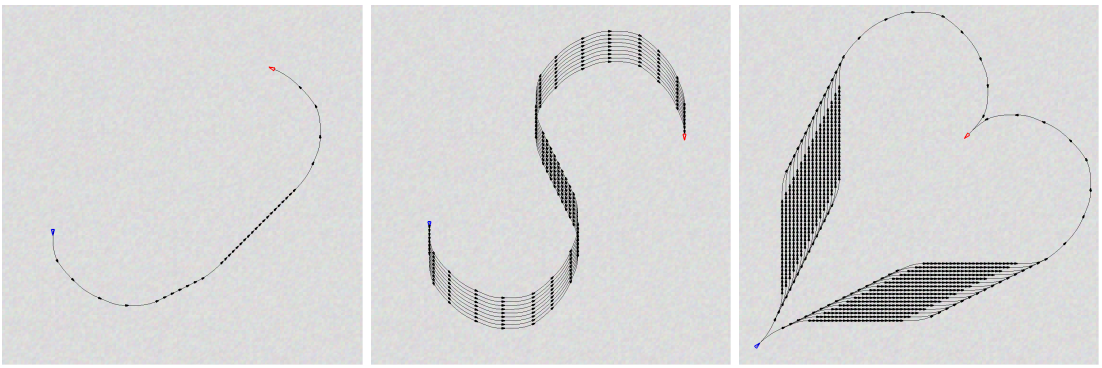
(b) All freespace-canonical paths on $\mathcal{F}$ that originate at s.

Figure 4.6: Canonical ordering based on the lexical ordering Left $<_{\mathscr{L}}$ Right $<_{\mathscr{L}}$ Up $<_{\mathscr{L}}$ Down of the four primitives that define a 4-neighbor grid graph. $\mathcal{F}$ is only shown partially, since it is infinite.

Grastien, 2011; Harabor et al., 2014; Sturtevant & Rabin, 2016). In this section, we define a canonical ordering on the primitives of a state lattice to distinguish, among multiple symmetric shortest paths on $\mathcal{F}$, exactly one as the *freespace-canonical* path.

Figure 4.6 shows the symmetries on a 4-neighbor grid graph, which can be considered as a state lattice constructed with respect to the four primitives Up, Down, Left, and Right. There are two different freespace-shortest E5-D6 paths, namely $\langle \text{E5}, \vec{U}, \vec{R} \rangle$ and $\langle \text{E5}, \vec{R}, \vec{U} \rangle$, which correspond to executing the primitives $\vec{U}$ and $\vec{R}$ in different orders. There are 10!/5!5! freespace-shortest E5-A10 paths, each corresponding to a different ordering of five $\vec{U}$ and five $\vec{R}$ primitives. Searches without proper duplicate detection might explore all these shortest paths, resulting in many redundant expansions.

State lattices, in general, have fewer symmetries than grid graphs, since some permutations of a sequence of primitives might not be kinematically feasible if a primitive with start orientation θ follows a primitive with end orientation $\theta' \neq \theta$. However, symmetries are still present on state lattices, which is why F-Connect needs to perform duplicate detection and F-Refine needs to search for a freespace-shortest path that is unblocked on G. Figure 4.7 shows an example of symmetries on state lattices constructed using Unicycle primitives. The path in Figure 4.7a is the unique freespace-shortest path between the states that it connects. Every other permutation of its primitives produces a path that is kinematically infeasible to execute. All paths in Figure 4.7b correspond to different permutations of a single sequence of primitives, and differ in how much the agent moves in a straight line in the very beginning or at the very end of its planned path. The paths in Figure 4.7c correspond to different permutations of one of two sequences of primitives: in one sequence the agent initially steers toward the right and in the other one it steers toward the left. This example also shows that not all symmetric shortest paths are due to different permutations of a single sequence of primitives.



(a) There is a single (unique) freespace-shortest path. Every other permutation of the corresponding sequence of primitives are kinematically infeasible.

(b) The freespace-shortest paths correspond to different permutations of a single sequence of primitives.

(c) The freespace-shortest paths correspond to different permutations of one of two distinct sequences of primitives.

Figure 4.7: All freespace-shortest paths between two states on the freespace state lattice constructed using the Unicycle primitives.

We use a total canonical ordering $<_{\mathscr{L}}$ on the primitives of a state lattice to designate, among multiple freespace-shortest paths between two vertices, exactly one as the freespace-canonical path. Intuitively, $<_{\mathscr{L}}$ associates each primitive with a unique "letter", such that, for every pair of different primitives, one is *lexically smaller* than the other one. As a result, primitive paths, that is, sequences of primitives, form "words", such that, for every pair of different primitive paths, one is lexically smaller than the other one. Furthermore, since there is a unique primitive path that correspond to any path on $\mathcal{F}$, we can also lexically compare two different symmetric shortest paths, by lexically comparing their corresponding primitive paths. We define the freespace-canonical path between two vertices on $\mathcal{F}$ as the lexically smallest freespace-shortest path between them. Figure 4.6b shows all lexically smallest freespace-shortest paths that originate at a vertex on the freespace 4-neighbor grid graph, assuming that we use the canonical ordering $<_{\mathscr{L}}$ with $\vec{W} <_{\mathscr{L}} \vec{E} <_{\mathscr{L}} \vec{N} <_{\mathscr{L}} \vec{S}$. Among all 10!/5!5! symmetric paths that correspond to different permutations of five $\vec{N}$ and five $\vec{E}$ primitives, the path $\langle s, \vec{E}, \vec{E}, \vec{E}, \vec{E}, \vec{E}, \vec{N}, \vec{N}, \vec{N}, \vec{N}, \vec{N} \rangle$ is selected as the freespace-canonical path, since it is the lexically smallest.

Definition 4.3 formally defines our lexical ordering of paths, and Definition 4.4 formally defines freespace-canonical paths. Lemma 4.16 proves that freespace-canonical paths are unique.

Definition 4.3 (Lexical ordering of paths.)**.** *Let $<_{\mathscr{L}}$ be a total ordering on all primitives in $\mathcal{M}$. Let $\vec{\pi} = \langle \vec{m}_0, \ldots, \vec{m}_k \rangle \neq \vec{\pi}' \langle \vec{m}'_0, \ldots, \vec{m}'_{k'} \rangle$ be two primitive paths.*

$\vec{\pi}$ is lexically smaller than $\vec{\pi}'$, denoted as $\vec{\pi} <_{\mathscr{L}} \vec{\pi}'$ if and only if: there exists $j \leq \min(k, k')$ such that ($\vec{m}_j <_{\mathscr{L}} \vec{m}'_j$ and, for all $i < j$, $\vec{m}_i = \vec{m}'_i$) or, if no such j exists, if and only if $k < k'$.

For every s and s', the path $\pi = \langle s, \vec{m}_0, \ldots, \vec{m}_k \rangle <_{\mathscr{L}} \pi' = \langle s', m'_0, \ldots, \vec{m}'_{k'} \rangle$ if and only if $\vec{\pi} <_{\mathscr{L}} \vec{\pi}'$.

Definition 4.4 (Freespace-canonical)**.** *An s-t path (on G or $\mathcal{F}$) is freespace-canonical if and only if it is the lexically smallest shortest s-t path on $\mathcal{F}$.*

Lemma 4.16. *For every pair of states s and t with $d_{\mathcal{F}}(s,t) < \infty$, there is a unique freespace-canonical s-t path.*

Proof. For every pair of freespace-shortest s-t paths $\pi \neq \pi'$, their corresponding primitive paths $\vec{\pi} \neq \vec{\pi}'$. Therefore, it holds that either $\pi <_{\mathscr{L}} \pi'$ or $\pi' <_{\mathscr{L}} \pi$ (Definition 4.3). Consequently, there cannot be more than one lexically smallest freespace-shortest s-t path. □

4.5.2 Properties of Freespace-Canonical Paths

In this section, we prove that all freespace-canonical paths that originate at a vertex s form an *out-tree* rooted at s, and all freespace-canonical paths that terminate at a vertex t form an *in-tree* rooted at t, and prove the *translation invariance of freespace-canonical paths*, that is, that translating freespace-canonical paths on $\mathcal{F}$ produces other freespace-canonical paths on $\mathcal{F}$, which allows for the efficient computation and compact storage of freespace-canonical successors, predecessors, and parents of states on $\mathcal{F}$.

Consider all freespace-canonical paths that originate at a single vertex s on a 4-neighbor grid graph, using the lexical ordering $\vec{W} <_{\mathscr{L}} \vec{E} <_{\mathscr{L}} \vec{N} <_{\mathscr{L}} \vec{S}$, as shown in Figure 4.6. The collection of all edges that appear on these paths form an out-tree rooted at s, that is, a tree where all edges point away from the root. Lemma 4.17 proves that subpaths of freespace-canonical paths are freespace-canonical, and Theorem 4.18 proves that all freespace-canonical paths that originate at a vertex s form an out-tree rooted at s and all freespace-canonical paths that terminate at a vertex t form an in-tree (all edges point towards the root) rooted at t.

Lemma 4.17. *Every subpath of a freespace-canonical path is freespace-canonical.*

Proof. Sketch: If a subpath $\pi_{u,v}$ of a freespace-canonical path $\pi_{s,t}$ is not freespace-canonical, then we can replace $\pi_{u,v}$ in $\pi_{s,t}$ with a lexically-smaller freespace-shortest path $\pi'_{u,v}$ to get a path $\pi'_{s,t}$ such that $\pi'_{s,t} <_{\mathscr{L}} \pi_{s,t}$, contradicting that $\pi_{s,t}$ is freespace-canonical.

1. Let $\pi_{s,t} = \langle s, \vec{m}_1, \ldots, u, \vec{m}_i, \ldots, \vec{m}_j, v, \ldots, \vec{m}_k, t \rangle$ be the freespace-canonical s-t path.
2. Assume (for contradiction) that $\pi_{u,v} = \langle u, \vec{m}_i, \ldots, \vec{m}_j, v \rangle$ is not freespace-canonical.
3. $\pi_{s,t}$ is freespace-shortest (Definition 4.4, since $\pi_{s,t}$ is freespace-canonical).
4. $\pi_{u,v}$ is freespace-shortest (Lemma 4.5, since $\pi_{u,v}$ is a subpath of $\pi_{s,t}$).
5. Let $\pi'_{u,v} = \langle u, \vec{m}'_i, \ldots, \vec{m}'_{j'}, v \rangle$ be a freespace-shortest u-v path with $\pi'_{u,v} <_{\mathscr{L}} \pi_{u,v}$. Such $\pi'_{u,v}$ exists (Definition 4.4, since $\pi_{u,v}$ is not freespace-canonical).
6. $l(\pi_{u,v}) = l(\pi'_{u,v}) = d_{\mathcal{F}}(u,v)$ (Definition 4.1, since $\pi_{u,v}$ and $\pi'_{u,v}$ are freespace-shortest u-v paths).
7. $\pi_{u,v}$ is not a prefix of $\pi'_{u,v}$ and $\pi'_{u,v}$ is not a prefix of $\pi_{u,v}$ (since $l(\pi_{u,v}) = l(\pi'_{u,v})$ and, for every $\vec{m} \in \mathcal{M}$, $l_m > 0$).

8. Let $a \geq i$ be the smallest number for which $\vec{m}_a \neq \vec{m}'_a$. Such $a \leq \min(j, j')$ exists (Definition 4.3, since $\pi'_{u,v} <_{\mathscr{L}} \pi_{u,v}$, $\pi_{u,v}$ is not a prefix of $\pi'_{u,v}$, and $\pi'_{u,v}$ is not a prefix of $\pi_{u,v}$).

9. $\vec{m}'_a <_{\mathscr{L}} \vec{m}_a$ (Definition 4.3, since $\pi'_{u,v} <_{\mathscr{L}} \pi_{u,v}$).

10. Let $\pi'_{s,t} = \langle s, \vec{m}_1, \ldots, u, \vec{m}'_i, \ldots, \vec{m}'_{j'}, v, \ldots, \vec{m}_k, t\rangle$ (replace $\pi_{u,v}$ in $\pi_{s,t}$ with $\pi'_{u,v}$).

11. $\pi'_{s,t} = \langle s, \vec{m}_1, \ldots, \vec{m}_{a-1}, \vec{m}'_a, \ldots, \vec{m}'_{j'}, \vec{m}_{j+1} \ldots \vec{m}_k\rangle$ (since a is the smallest number for which $\vec{m}_a \neq \vec{m}'_a$).

12. $\pi'_{s,t} <_{\mathscr{L}} \pi_{s,t}$ (Definition 4.3, since $m'_a <_{\mathscr{L}} m_a$).

13. $\pi'_{s,t}$ is a freespace-shortest s-t path (since $\pi'_{s,t}$ replaces the freespace-shortest u-v path $\pi_{u,v}$ in the freespace-shortest s-t path $\pi_{s,t}$ with the freespace-shortest u-v path $\pi'_{u,v}$).

14. $\pi_{s,t}$ is not freespace-canonical (Definition 4.4, since $\pi'_{s,t}$ is a freespace shortest s-t path with $\pi'_{s,t} <_{\mathscr{L}} \pi_{s,t}$).

15. $\perp$ ($\pi_{s,t}$ is freespace-canonical and not freespace-canonical).

□

Theorem 4.18. *All freespace-canonical paths that originate at a vertex s form an out-tree rooted at s. All freespace-canonical paths that terminate at a vertex t form an in-tree rooted at t.*

Proof. Let π and π' be two freespace-canonical paths with the start vertex s. Assume (for contradiction) that an edge (u, n) appears on π and an edge (v, n) appears on π', with $u \neq v$. That is, $\pi = \pi_{s,u} \cdot \langle u, n\rangle \cdot \pi_{n,t}$ and $\pi' = \pi'_{s,v} \cdot \langle v, n\rangle \cdot \pi'_{n,t'}$. By Lemma 4.17, both $\pi_{s,u} \cdot \langle u, n\rangle$ and $\pi'_{s,v} \cdot \langle v, n\rangle$ are freespace-canonical, since they are subpaths of freespace-canonical paths. By Lemma 4.16, the freespace-canonical s-n path is unique. Therefore, $\pi_{s,u} \cdot \langle u, n\rangle = \pi'_{s,v} \cdot \langle v, n\rangle$, contradicting that $u \neq v$.

We can similarly show that two freespace-canonical paths that terminate at a vertex t cannot contain both the edge (n, u) and (n, v) with $u \neq v$. □

As we discuss in the following sections, our connection algorithm for CF-reachability uses *freespace-canonical successors and predecessors*, and our refinement algorithm uses *freespace-canonical parents*, which correspond to the edges of the out-trees or in-trees formed by freespace-canonical paths that originate or terminate at a vertex. Namely, the freespace-canonical successors of a vertex u with respect to a source vertex s are the children of u in the out-tree formed by the freespace-canonical paths that originate at s, or, equivalently, those successors v of u on $\mathcal{F}$ that extend the freespace-canonical s-u path to freespace-canonical s-v paths. Similarly, the freespace-canonical predecessors of a vertex u with respect to a destination vertex t are the children of u in the in-tree formed by the freespace-canonical paths that terminate at t, or, equivalently, those predecessors v of u on $\mathcal{F}$ that extend the freespace-canonical u-t path to the freespace-canonical v-t path. Finally, the freespace-canonical parent of a vertex u with respect to a source vertex

s is the parent of u in the out-tree formed by the freespace-canonical paths that originate at s.

We now prove the *translation invariance of freespace-canonical paths*, that is, that translating freespace-canonical paths on $\mathcal{F}$ produces other freespace-canonical paths on $\mathcal{F}$, which can be used to efficiently compute and compactly store freespace-canonical successors, predecessors, and parents. Recall that we use the terminology $s+(x,y)=s'$ to denote that translating a state s by (x,y) results in the state s', and use the terminology $\pi+(x,y)=\pi'$ to denote that translating a path π by (x,y) results in the path π'. Intuitively, if π is freespace-canonical, $\pi'=\pi+(x,y)$ must also be freespace canonical since, otherwise, a lexically-smaller freespace-shortest path π'' would exist with the same length as π'. However, then $\pi''-(x,y)$ would show that π is not freespace-canonical, resulting in a contradiction.

Theorem 4.19 (Translation invariance of freespace-canonical paths)**.** *Let π be a freespace-canonical path. For any two integers x and y, $\pi'=\pi+(x,y)$ is a freespace-canonical path as well.*

Proof. Since π is freespace-canonical, π is freespace-shortest (Definition 4.4). Since π' is generated by translating the freespace-shortest path π, π' is also freespace-shortest (Lemma 4.8). Let s and t denote the first and last vertices on π, and let s' and t' denote the first and last vertices on π', respectively. Assume (for contradiction) that π' is not freespace-canonical. That is, an s'-t' path π'' exists on $\mathcal{F}$ with $l(\pi'')=l(\pi')$ and $\pi''<_{\mathscr{L}}\pi'$ (Definition 4.4). Then, $\pi''+(-x,-y)$ is a freespace s-t path with length $l(\pi''+(-x,-y))=l(\pi)$ (Lemma 4.7). Since π and π' have the same primitive sequence and $\pi''<_{\mathscr{L}}\pi'$, it holds that $\pi''<_{\mathscr{L}}\pi$ (Definition 4.3). Since π'' and $\pi''+(-x,-y)$ have the same primitive sequence and $\pi''<_{\mathscr{L}}\pi$, it holds that $\pi''+(-x,-y)<_{\mathscr{L}}\pi$ (Definition 4.3), contradicting that π is freespace-canonical (Definition 4.4). □

The translation invariance of freespace-canonical paths ensures that we can store, for each $\theta\in O_{\mathcal{M}}$, a single out-tree of freespace-canonical paths rooted at $(0,0,\theta)$ (that is, a single set of freespace-canonical successors or parents), and a single in-tree rooted at $(0,0,\theta)$ (that is, a single set of freespace-canonical predecessors), since these trees are translationally equivalent to the trees rooted at every vertex (x,y,θ). Furthermore, these trees can be computed efficiently by first performing a Dijkstra search from $s=(0,0,\theta)$ (using out-edges or in-edges for constructing the out-trees or in-trees, respectively) to generate a directed acyclic graph of freespace-shortest path that originate or terminate at a vertex, and then performing a depth-first search on this directed acyclic graph, which evaluates successors of expanded vertices in increasing lexical order, guaranteeing that, when a vertex is visited for the first time, it is visited through the freespace-canonical path. The search tree of this depth-first search then corresponds to the out-tree or the in-tree of all freespace-canonical paths rooted at $(0,0,\theta)$.

4.5.3 Canonical Freespace Reachability (CF-Reachability)

In order to be able to use efficiently computed and compactly stored freespace-canonical successors, predecessors, and parents during the connection and refinement phases of queries answered using subgoal graphs, we can construct subgoal graphs with respect to

canonical-freespace-reachability: A state t is canonical-freespace-reachable from a state s if and only if the freespace-canonical s-t path is unblocked on G. Definition 4.5 formally defines canonical-freespace-reachability, as well as a variant, called bounded-canonical-freespace-reachability.

Definition 4.5 (Canonical-freespace-reachability)**.** *A vertex t is canonical-freespace-reachable (CF-reachable) from a vertex s, denoted as $(s,t) \in$ CF, if and only if the freespace-canonical s-t-path is unblocked on G. t is bounded-canonical-freespace-reachable with reachability bound $b > 0$ (CFb-reachable) from s if and only if t is CF-reachable from s and $d(s,t) \leq b$.*

Theorem 4.20. *CF is a reachability relation. CFb is a reachability relation if, for all $\vec{m} \in \mathcal{M}$, $l_{\vec{m}} \leq b$.*

Proof. For every $n \in V$, $(n,n) \in$ CF and $(n,n) \in$ CFb since $\langle n \rangle$ is the unique freespace-shortest n-n path on G (that is, no other freespace-shortest n-n path exists that is lexically smaller). For every $(u,v) \in E$, $(u,v) \in$ CF since $\langle u,v \rangle$ is the unique freespace-shortest u-v path (Lemma 4.10) on G. If, for every $\vec{m} \in \mathcal{M}$, $l_{\vec{m}} \leq b$, then $c(u,v) = l_{\vec{m}} \leq b$, where $\vec{m}$ is the primitive that induces the edge (u,v). Consequently, $(u,v) \in$ CFb. □

Similar to how Fb reachability can be considered to be the combination of F-reachability and BDb-reachability, CFb reachability can be considered to be the combination of CF-reachability and BDb-reachability. That is, CFb = CF $\cap$ BDb and, therefore, CF$b \subseteq$ BDb. Furthermore, if $(s,t) \in$ CF, a freespace-shortest s-t path exists on G (namely, the canonical one). Therefore, $(s,t) \in$ F and, consequently, CF $\subseteq$ F. We discuss and experimentally evaluate in Section 4.6 how the relationship CF$b \subseteq$ F$b \subseteq$ BDb affects the trade-off between the connection, search, and refinement times of queries answered using (strongly-connected) subgoal graphs constructed with respect to these reachability relations.

4.5.4 CF-Connect

In this section, we introduce a connection algorithm for CF-reachability, called CF-Connect, and prove its correctness with respect to the criteria that we have outlined in Section 3.3.4 for answering queries using CF (strongly-connected) subgoal graphs. That is, we show that CF-Connect connects a given start vertex s (or a given goal vertex t) to all direct-CF-reachable subgoals from s, never connects s to a subgoal that is not CF-reachable from s, and determines the correct lengths for the connecting edges as distances on G. For brevity, we only describe a version of CF-Connect, called CF$^{\rightarrow}$-Connect, that connects the start vertex, but not the goal vertex, to a CF (strongly-connected) subgoal graph, and describe briefly how it can be modified to also connect the goal vertex. Furthermore, we omit providing a connection algorithm for CFb-reachability, since CF-Connect can be modified to become a connection algorithm for CFb-reachability by simply disallowing it from generating vertices whose distances from the start vertex are larger than b.

Algorithm 12 outlines CF$^{\rightarrow}$-Connect, which operates similarly to F$^{\rightarrow}$-Connect, except that it explores freespace-canonical rather than freespace-shortest paths on G that originate at a vertex s. That is, it explores exactly the freespace-canonical paths that originate

Algorithm 12 CF$^{\rightarrow}$-Connect

Blue text: Differences from F$^{\rightarrow}$-Connect. CF$^{\rightarrow}$-Connect does not maintain a CLOSED list, since it does not have to perform duplicate detection.

Input: A state lattice $G = (V, E, c)$, start vertex s, covering vertices $S \subseteq V$, target vertices $T \subseteq V$

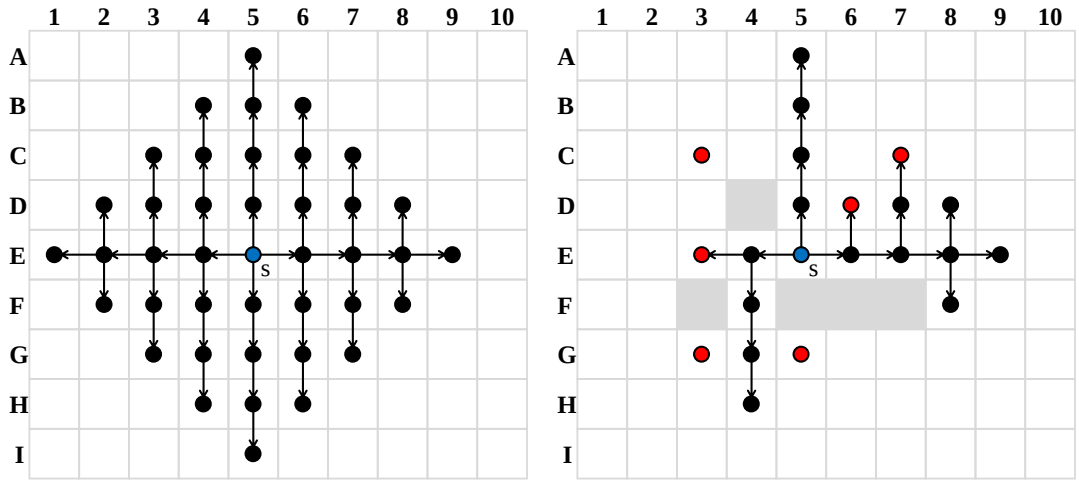
Output: A set of edges E^+ such that $\mathrm{CF}_S^{s \rightarrow T} \subseteq E^+ \subseteq \mathrm{CF}$, edge lengths c^+ such that, $\forall (u, v) \in E^+$, $c^+ = d_{\mathcal{F}}(u, v) = d(u, v)$

1: $E^+ \leftarrow \emptyset$
2: OPEN $\leftarrow \{s\}$ ▷ Implemented as a stack for depth-first search
3: $g(s) \leftarrow 0$ ▷ g-values maintained since $d_{\mathcal{F}}$ is not available
4: **while** OPEN $\neq \emptyset$ **do**
5: $u \leftarrow$ any vertex in OPEN
6: Remove u from OPEN
7: **if** $u \in T \setminus \{s\}$ **then**
8: Add (s, u) to E^+ with $c^+(u, v) = g(u) = d_{\mathcal{F}}(s, u)$
9: **if** $u \notin S \setminus \{s\}$ **then**
10: **for all** freespace-canonical successors v of u with respect to s **do**
11: **if** $(u, v) \in E$ **then**
12: $g(v) \leftarrow g(u) + c(u, v)$
13: Add v to OPEN
14: return E^+, c^+

at a given start vertex s that are unblocked on G and not covered by subgoals: Any vertex that cannot be reached with a freespace-canonical path cannot be CF-reachable from s (Definition 4.5), and any vertex that is reached with a freespace-canonical path that is covered by a subgoal cannot be direct-CF-reachable from s (Definition 3.7). Figure 4.8 shows an example of the operation of CF$^{\rightarrow}$-Connect on a 4-neighbor grid graph.

The operation of CF$^{\rightarrow}$-Connect differs from the operation of F$^{\rightarrow}$-Connect in two ways, which are highlighted in blue in Algorithm 12:

- CF$^{\rightarrow}$-Connect generates only the (unblocked) freespace-canonical successors (with respect to s) of expanded vertices u (lines 10–11), that is, successors that extend the freespace-canonical s-u path to the freespace-canonical s-v path, whereas F$^{\rightarrow}$-Connect iterates over the (unblocked) successors v of u to determine whether they are *freespace-shortest successors*, that is, whether they extend a freespace-shortest s-u path to a freespace-shortest s-v path. Therefore, CF$^{\rightarrow}$-Connect iterates over fewer successors per expansion (only the freespace-canonical ones rather than all successors) and does not perform a check to see if the successor is freespace-canonical (or freespace-shortest). However, since CF$^{\rightarrow}$-Connect does not use precomputed freespace distances, it maintain g-values for generated vertices to correctly determine the lengths of the edges it finds (lines 3, 12). We introduce a variant of F$^{\rightarrow}$-Connect in Section 4.6.2 that uses precomputed freespace-shortest successors rather than freespace distances, so that it also performs expansions more efficiently.



(a) All freespace-canonical paths on $\mathcal{F}$ based on the lexical ordering Left $<_{\mathscr{L}}$ Right $<_{\mathscr{L}}$ Up $<_{\mathscr{L}}$ Down that originate at s and have length 4 or less. $\mathcal{F}$ is only shown partially, since it is infinite.

(b) CF4-Connect explores all freespace-canonical paths on G that originate at s, have length 4 or less, and are not covered by subgoals (red).

Figure 4.8: CF4-Connect on a 4-neighbor grid graph.

- CF$^{\rightarrow}$-Connect does not perform duplicate detection since the freespace-canonical paths that originate at s form a tree (Theorem 4.18). That is, it does not maintain a CLOSED list, unlike F$^{\rightarrow}$-Connect (Algorithm 10, lines 3, 6, and 10).

CF$^{\rightarrow}$-Connect can be modified to connect a goal vertex t to vertices from which it is direct-CF-reachable by running it on the reverse graph of G, and using freespace-canonical predecessors rather than successors. That is, when expanding a vertex u, CF$^{\leftarrow}$-Connect only generates the freespace-canonical predecessors of u with respect to t.

We now prove the correctness of CF$^{\rightarrow}$-Connect with respect to the criteria that we have outlined in Section 3.3.4 for answering queries using CF (strongly-connected) subgoal graphs. Namely, we show that CF$^{\rightarrow}$-Connect connects a given source vertex s to all direct-CF-reachable subgoals from s, never connects s to a subgoal that is not CF-reachable from s, and determines the correct lengths for the connecting edges as distances on G. We omit the proof of correctness for CF$^{\leftarrow}$-Connect, which is similar to the proof of correctness for CF$^{\rightarrow}$-Connect. We also omit the proof that CF$^{\rightarrow}$-Connect terminates, which follows from the fact that it never generates duplicate vertices, and our assumption that G is finite (Assumption 1.1). Lemmata 4.21 and 4.22 are similar to Lemmata 4.12 and 4.13, respectively, that we used to prove the correctness of F$^{\rightarrow}$-Connect.

Lemma 4.21. *At any point during the execution of Algorithm 12, every vertex $u \in$ OPEN is CF-reachable from s with $g(u) = d_{\mathcal{F}}(s, u) = d(s, u)$.*

Proof.

1. Let $\mathcal{S}_i$ be the statement "The ith time that line 4 is executed, for every vertex $n \in$ OPEN, $(s, n) \in$ CF and $g(n) = d_{\mathcal{F}}(s, n) = d(s, n)$".

2. For all $i = 1, \ldots, k$, where k is the number of times that line 4 is executed, S_i is true (proof by induction on i):

   2.1. Base case: $\mathcal{S}_1$ is true (since OPEN $= \{s\}$, $(s, s) \in$ CF, $g(s) = 0 = d_{\mathcal{F}}(s, s) = d(s, s)$).

   2.2. Induction step: If $\mathcal{S}_i$ is true, then $\mathcal{S}_{i+1}$ is true:

      2.2.1. Assume $\mathcal{S}_i$ is true.

      2.2.2. Let u be the vertex selected for expansion on line 5 immediately after line 4 is executed for the ith time.

      2.2.3. Let v be any vertex added to OPEN (line 12) after line 4 is executed for the the ith time but before the $(i+1)$st time.

      2.2.4. $(s, u) \in$ CF (since $\mathcal{S}_i$ is true).

      2.2.5. Let π_u be the canonical freespace s-u path on G. Such π_u exists (Definition 4.5, since $(s, u) \in$ CF).

      2.2.6. v is a canonical successor of u with respect to s (line 10).

      2.2.7. $\pi_v = \pi_u \cdot \langle u, v \rangle$ is a canonical freespace s-v path (since π_u is the canonical freespace s-u path and v is a canonical successor of u with respect to s).

      2.2.8. $(s, v) \in$ CF:

         2.2.8.1. $(u, v) \in E$ (line 11).

         2.2.8.2. π_v is a path on G (since π_u is a path on G and $(u, v) \in E$).

         2.2.8.3. $(s, v) \in$ CF (Definition 4.5, since π_v is a canonical freespace s-v path on G).

      2.2.9. $g(v) = d(s, v) = d_{\mathcal{F}}(s, v)$:

         2.2.9.1. $g(v) = g(u) + c(u, v)$ (line 12).

         2.2.9.2. $= d_{\mathcal{F}}(s, u) + c(u, v)$ (since $\mathcal{S}_i$ is true).

         2.2.9.3. $= l(\pi_u) + c(u, v)$ (Definition 4.1, since π_u is a freespace s-v path).

         2.2.9.4. $= l(\pi_v)$ (since $\pi_v = \pi_u \cdot \langle u, v \rangle$).

         2.2.9.5. $= d_{\mathcal{F}}(s, v)$ (Definition 4.1, since π_v is a freespace s-v path on G).

         2.2.9.6. $= d(s, v)$ (Lemma 4.4, since π_v is a freespace s-v path on G).

□

Lemma 4.22. *For every $n \in V$, if n is direct-CF-reachable from s, then Algorithm 12 selects n for expansion (line 5) at some point during its execution.*

Proof.

1. Let $n \in V$ with $(s, n) \in \mathrm{CF}_S^{s \to V}$.

2. Let $\pi = \langle p_0, \ldots, p_k \rangle$ be a canonical freespace s-n path on G. Such π exists (Definition 4.5, since $(s, n) \in$ CF).

3. $S \not\sqsubset (s, n)$ (Definition 3.2, since $(s, n) \in \mathrm{CF}_S^{s \to V}$).

4. $S \not\sqsubset \pi$ (Definition 3.1, since $S \not\sqsubset (s, n)$ and π is a shortest s-n path).

5. For $i = 0, \ldots, k-1$, $p_i \notin S \setminus \{s\}$:

   5.1. If $i = 0$, then $p_i = p_0 = s \notin S \setminus \{s\}$.

   5.2. If $0 < i < k$, then $p_i \notin S$ (Definition 3.1, since $S \not\sqsubset \pi$).

6. Let $\mathcal{S}_i$ be the statement "Algorithm 12 selects p_i for expansion on line 6 at some point during its execution".

7. For all $i = 0, \ldots, k$, S_i is true (proof by induction on $i = 0, \ldots, k-1$):

   7.1. Base case: $\mathcal{S}_0$ is true (since OPEN $= \{s\}$ the first time line 4 is executed).

   7.2. Induction step: If $\mathcal{S}_i$ is true, then $\mathcal{S}_{i+1}$ is true:

      7.2.1. Assume $\mathcal{S}_i$ is true.

      7.2.2. p_i is selected for expansion (since $\mathcal{S}_i$ is true).

      7.2.3. Consider the point during execution when p_i is selected for expansion.

      7.2.4. If $p_{i+1} \in$ OPEN, then p_{i+1} is eventually selected for expansion (line 4).

      7.2.5. If $p_{i+1} \in$ CLOSED, then p_{i+1} has already been selected for expansion (line 6).

      7.2.6. Otherwise, p_{i+1} is added to OPEN:

         7.2.6.1. Lines 10–13 are executed for p_i (line 9, since $p_i \notin S \setminus \{s\}$).

         7.2.6.2. $\pi_i = \langle p_0, \ldots, p_i \rangle$ is a canonical freespace path on G (since π_i is a subpath of the canonical freespace s-n path π on G).

         7.2.6.3. $\pi_{i+1} = \langle p_0, \ldots, p_{i+1} \rangle$ is a canonical freespace path on G (since π_{i+1} is a subpath of the canonical freespace s-n path π on G).

         7.2.6.4. p_{i+1} is a canonical successor of π_i with respect to s (since (p_i, p_{i+1}) extends the canonical freespace s-p_i path π_i to the canonical freespace s-p_{i+1} path).

         7.2.6.5. $(p_i, p_{i+1}) \in E$ (since π is a path on G).

         7.2.6.6. p_{i+1} is added to OPEN (since lines 10–13 are executed for p_i, p_{i+1} is a canonical successor of π_i with respect to s, and $(p_i, p_{i+1}) \in E$).

□

Theorem 4.23. *Algorithm 12 returns a set of edges E^+ with lengths c^+ such that $CF_S^{s \to T} \subseteq E^+ \subseteq CF$ and, for every $(u, v) \in E^+$, $c^+(u, v) = d(u, v)$.*

Proof. By Lemma 4.21, Algorithm 12 only expands vertices that are CF-reachable from s. By Lemma 4.22, Algorithm 12 is guaranteed to expand all vertices that are direct-CF-reachable from s. Therefore, Algorithm 12 identifies a set of edges E^+ with $\mathrm{CF}_S^{s \to T} \subseteq E^+ \subseteq \mathrm{CF}$. Furthermore, by Lemma 4.21, Algorithm 12 correctly maintains the distances of expanded vertices from the start vertex on G as g-values. Therefore, Algorithm 12 correctly determines the lengths of edges in E^+ as distances on G. □

Algorithm 13 CF-Refine

Input: $(s, t) \in \text{CF}$
Output: The freespace-canonical s-t path π.

```
1: π ← ⟨t⟩
2: u ← t
3: while u ≠ s do
4:     π ← ⟨freespace-canonical-parent(s, u), u⟩ · π
5:     u ← freespace-canonical-parent(s, u)
6: return π
```

4.5.5 CF-Refine

Our refinement algorithm for CF-reachability, CF-Refine, uses freespace-canonical parents to generate the freespace-canonical path between two CF-reachable vertices. Algorithm 13 outlines CF-Refine.

4.6 Experimental Evaluation

In this section, we perform three sets of experiments: First, we compare various connection algorithms for BDb-, Fb-, and CFb-reachability with respect to their connection times. Second, we compare answering queries using BDb, Fb, and CFb subgoal graphs (SGs) with respect to their query times. Third, we compare answering queries using BDb, Fb, and CFb strongly-connected subgoal graphs (SCSGs) with respect to their path suboptimalities and query times.

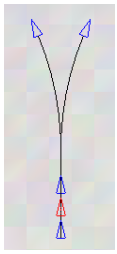
4.6.1 Benchmarks

We use four benchmarks in our experiments, a small grid paired with a small set of primitives, a small grid paired with a large set of primitives, a large grid paired with a small set of primitives, and a large grid paired with a large set of primitives. We now explain these benchmarks in more detail.

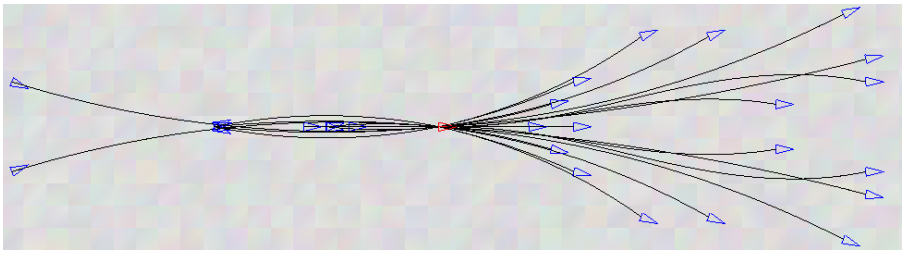
- **Unicycle and Urban primitives:**[2] *Unicycle* primitives have 16 induced orientations and 5 (kinematically feasible) primitives per orientation. *Urban* primitives have 32 induced orientations and 36 primitives per orientation. Urban primitives have been used in Carnegie Mellon University's "Boss" entry in the DARPA Urban Challenge for navigation in unstructured and cluttered environments such as parking lots (Urmson et al., 2008), and can be considered as the more "realistic" set of primitives.

  In both the Unicycle and Urban primitives, the length of each primitive is equal to the distance traveled by the agent, with the following exceptions: In Unicycle primitives, the lengths of primitives that correspond to backward moves or turns

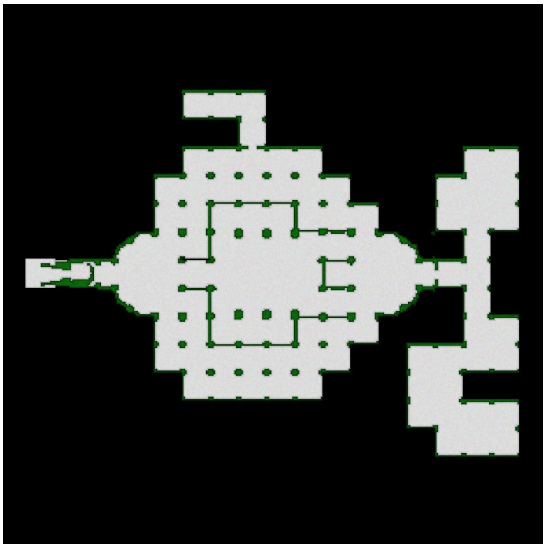
[2] We thank Maxim Likhachev for making these primitives available to us.



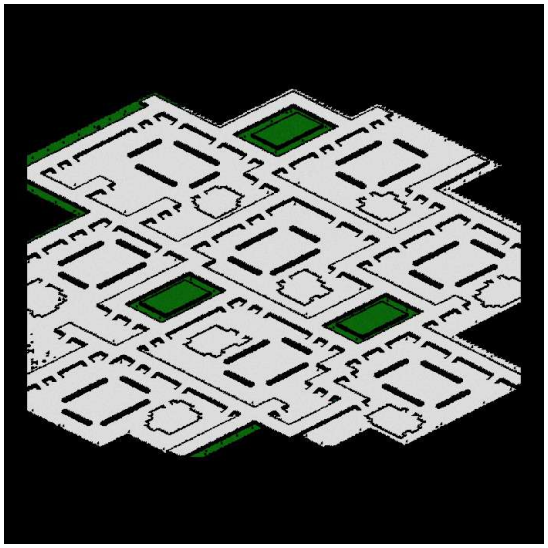
(a) Unicycle primitives.



(b) Urban primitives.



(c) Arena2.



(d) Aurora.

Primitives	Grid	$\lvert O_{\mathcal{M}} \rvert$	$\lvert \theta_{\mathcal{M}} \rvert$	Dimensions	Unblocked	Vertices	Edges
Unicycle	Arena2	16	4	209 × 281	24,311	375,391	1,262,812
	Aurora	16	4	768 × 1024	493,772	7,584,050	27,203,641
Urban	Arena2	32	32	209 × 281	24,311	755,949	14,089,647
	Aurora	32	32	768 × 1024	493,772	15,255,281	333,465,922

(e) Number of primitives, grid dimensions, and the size of G as the largest strongly-connected component of the resulting state lattice. $\lvert O_{\mathcal{M}} \rvert$: number of induced orientations, $\lvert \theta_M \rvert$: largest number of primitives per orientation.

Figure 4.9: Experimental setup: Primitives and grids. In both (c) and (d), unblocked cells are colored white and blocked cells are colored black or green.

are multiplied by 5 or 3, respectively. In Urban primitives, the lengths of primitives that correspond to backward moves are multiplied by 3.

Neither Unicycle nor Urban primitives are well-behaved. For instance, Unicycle primitives contain a primitive $\vec{m}$ for moving the agent forward by one cell, and another primitive for moving the agent forward by eight cells that is equivalent to executing $\vec{m}$ eight times. To satisfy our assumption that we construct state lattices with well-behaved sets of primitives (Assumption 4.3), we remove those primitives from the Unicycle and Urban primitives, respectively, that can be replaced with sequences of other primitives. After the removal of such primitives, Unicycle primitives have 4 primitives per orientation, and Urban primitives have between 27 to 32. Figures 4.9a and 4.9b show the "well-behaved" Unicycle and Urban primitives, respectively.

- **Arena2 and Aurora grids:** We use two different grids from Nathan Sturtevant's benchmark grid maps (Sturtevant, 2012a) in our experiments. Arena2 is a 209 × 281 grid with 24,311 unblocked cells from the video game Dragon Age: Origins. Aurora is a 768 × 1024 grid with 493,772 unblocked cells from the video game Starcraft. Arena2 and Aurora are shown in Figures 4.9c and 4.9d, respectively.

- **State lattices:** We use four state lattices in our experiments, which correspond to the four combinations of the two Unicycle and Urban primitives with the two Arena2 and Aurora grids. To satisfy Assumption 4.2, we designate our four benchmark graphs, Unicycle-Arena2, Unicycle-Aurora, Urban-Arena2, and Urban-Aurora, as the largest strongly connected components of these four state lattices. Figure 4.9e shows the number of vertices and edges of these four benchmark graphs (benchmarks, for short).

- **Instances:** For each benchmark, we randomly generate 1000 start and goal pairs, which we use to evaluate execution times of queries answered using subgoal graphs and strongly-connected subgoal graphs constructed with respect to different reachability relations. We use only the start vertices of these 1000 start and goal pairs when evaluating the execution times of different connection algorithms.

- **Server:** We run our experiments on a PC with 3.6GHz Intel Core i7-7700 CPU and 32GB of RAM.

4.6.2 Algorithms

We distinguish between six different ways of implementing connection and refinement algorithms for BDb, Fb, and CFb reachability, which are shown in Table 4.2 and explained below.

- **BDb-reachability:** We distinguish between three different ways of implementing BDb-Connect, which correspond to the three variants of Overlay-Connect discussed in Section 3.2.5, namely, the conservative (BDb-C), aggressive (BDb-A), and stall-on-demand (BDb-S) variants. We perform refinement for BDb-reachability using A* searches with the Euclidean distance heuristic.

R	R-Connect	R-Refine
BDb	Dijkstra search. Prune v if $g(v) > b$. **(C)onservative variant:** Propagate covered, terminate when OPEN is covered. **(A)ggressive variant:** Stall u if $u \in S$. **(S)tall-on-demand variant:** Stall u if $u \in S$ or if its parent can be changed to a stalled vertex without increasing $g(u)$.	A* search
Fb	Depth-first search. Prune v if $g(v) > b$. Stall u if $u \in S$. **(D)istance variant:** Prune v if $d_{\mathcal{F}}(s,u) + c(u,v) \neq d_{\mathcal{F}}(s,v)$. **(F)lag variant:** Skip v if not a freespace-shortest successor of u wrt s.	Depth-first search. **(D)istance variant:** Prune v if $d_{\mathcal{F}}(s,t) \neq d_{\mathcal{F}}(s,u) + c(u,v) + d_{\mathcal{F}}(v,t)$. **(F)lag variant:** Skip v if not a freespace-shortest parent of u.
CFb	Depth-first search. Prune v if $g(v) > b$. Stall u if $u \in S$. Skip v if not a freespace-canonical successor of u wrt s. No duplicate detection.	Follow freespace-canonical parents.

Table 4.1: Summary of three reachability relations CF$b \subseteq$ F$b \subseteq$ BDb on state lattices, and their six variants. S = set of subgoals. s = start vertex. t = goal vertex. u = vertex selected for expansion. v = successor (or predecessor, if connecting the goal) of u. "Prune" = v is evaluated but not generated. "Skip" = v is not evaluated. "Stall' = u is selected for expansion but not expanded.

- **Fb-reachability:** We distinguish between two different ways of implementing the connection and refinement algorithms for Fb-reachability, namely, the *distance variant* (Fb-D) that uses precomputed freespace distances as discussed in Section 4.4, and a *flag variant* (Fb-F) that uses precomputed *freespace-shortest successors, predecessors, and parents* that are similar to their freespace-canonical counterparts that we discussed in Section 4.5. For instance, the freespace-shortest successors of a state u with respect to a start vertex s are those successors v of u that can extend a freespace-shortest s-u path to a freespace-shortest s-v path. The flag variant of Fb-Connect performs a single look-up for the freespace-shortest successors (or predecessors) of expanded vertices, whereas the distance variant of Fb-Connect iterates over all successors and determines whether they are freespace-shortest successors by looking-up freespace distances for each successor.

- **CFb-reachability:** We implement the connection and refinement algorithms for CFb-reachability as discussed in Section 4.5 and do not distinguish between multiple variants.

4.6.3 Implementation Details

We now discuss implementation details for our algorithms.

- **OPEN:** We use a binary heap to implement priority queues, which are used for A* searches (on G, subgoal graphs, or strongly-connected subgoal graphs) and all three variants of BDb-Connect. We use a stack for CFb-Connect, both variants of Fb-Connect, and both variants of Fb-refine.

- **Preallocation:** We preallocate memory for all data structures used during queries to ensure that no new memory is allocated during queries. This includes the memory required for maintaining g-, h-, or f-values; parent pointers; covered, stalled, open, and closed flags; and OPEN lists implemented as either binary heaps or stacks.

- **Avoiding collision detection using executable primitive flags:** As described in Section 4.2, state lattices are implicitly defined by a grid and a set of primitives. Searches over state lattices using this "implicit" representation should perform collision detection to generate the successors of expanded vertices. That is, for every primitive $\vec{m}$ that is kinematically feasible to execute from an expanded vertex s, the search should check if $\vec{m}$ is also collision-free to execute from s by iterating over the list of cells that the footprint of the agent intersects with when executing $\vec{m}$ from s. We avoid this overhead during queries by performing these checks during preprocessing and caching the results: We store a bitfield for every state, called *executable primitive flags*, where the ith bit indicates whether the ith primitive that is kinematically feasible to execute from s is also collision-free to execute from s. The successors of a state s can then be determined by iterating over its executable primitive flags and generating a successor for every set bit, by looking up the resulting state and length of the corresponding primitive. We store these flags in a 3D array which can be accessed using the x- and y-coordinates and the orientation of a state. Furthermore, we unset bits that correspond to edges that are

not part of the largest strongly-connected component of the state lattice, allowing us to represent the benchmark graphs exactly. This approach uses 96MB for storing the executable primitive flags in the Urban-Aurora benchmark, where no state has more than 32 successors and executable primitive flags for each state can thus be stored using 32 bits. We similarly store "predecessors" for every state using another set of executable primitive flags.

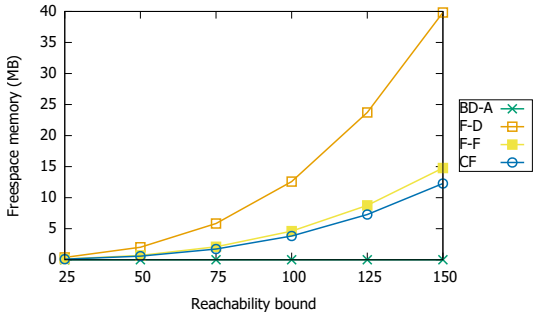
- **Storing freespace information:** As discussed in Section 4.4.2, freespace distances up to a bound b can be stored compactly by exploiting the translation invariance of freespace distances: For each orientation θ, we store a 3D table T_θ where the entry (x, y, θ') stores the freespace distance of state (x, y, θ') from $(0, 0, \theta)$. We adjust the x- and $y-$dimensions of these tables to correspond to the smallest rectangle that contain all the states with freespace-distance up to b from $(0, 0, \theta)$. We use 32 bits to store each freespace-distance entry in these tables.

  We store the freespace-shortest successors, predecessors, and parents (used for the flag variants of Fb-Connect and Fb-Refine) using bitfields that are similar to executable primitive flags: For instance, the ith bit in the freespace-shortest successor flags for a state $u = (x, y, \theta')$ in table T_θ is set if and only if the ith primitive that is kinematically feasible to execute from u is a freespace-canonical successor of u with respect to $(0, 0, \theta)$. During connection, we determine the unblocked freespace-shortest successors of expanded states by performing a "bitwise and" operation on the corresponding executable primitive flags and freespace-shortest successor flags, iterating over the resulting bits, and generating a successor for each set bit. We store the freespace-shortest successors, predecessors, and parents by using 4 and 32 bits each per entry on the Unicycle and Urban benchmarks, respectively.

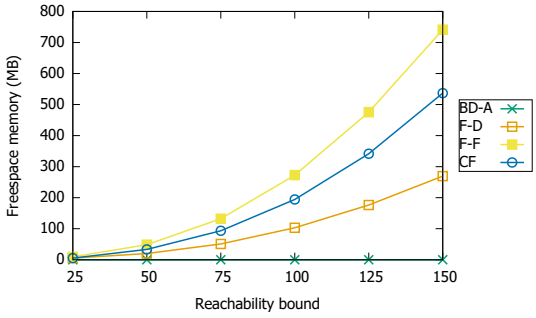
  We store the freespace-canonical successors and predecessors (used for CFb-Connect) in the same way that we store freespace-shortest successors and predecessors. However, since the freespace-canonical parents (used for CFb-refine) are unique for each state, we store them as the "ID" of the corresponding primitive rather than a bitfield with only one set bit, using 2 and 8 bits per entry on the Unicycle and Urban benchmarks, respectively.

  Figures 4.10a and 4.10b show the memory required for storing freespace information for Fb-D, Fb-F, and CFb on the Unicycle and Urban benchmarks, respectively. On Unicycle benchmarks, storing freespace distances (32 bits per entry) is more expensive than storing freespace-shortest successors, predecessors, and parents (12 bits per entry). On Urban benchmarks, storing freespace distances (32 bits per entry) is less expensive than storing freespace-shortest successors, predecessors, and parents (96 bits per entry). On all benchmarks, storing freespace-canonical successors, predecessors, and parents is slightly less expensive than storing freespace-shortest successors, predecessors, and parents, since storing freespace-canonical parents is less expensive that storing freespace-shortest parents.

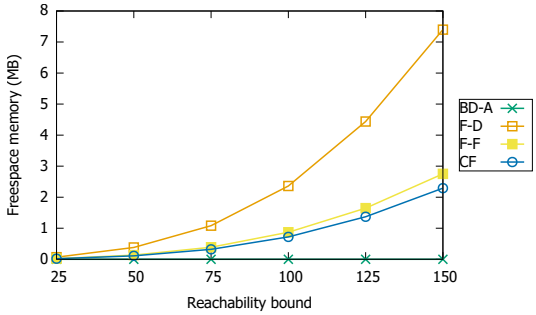
- **Further compression of freespace information (not used):** The translation invariance of freespace distances and freespace-canonical paths can be considered as translational symmetries, which hold for any set of primitives. However, many



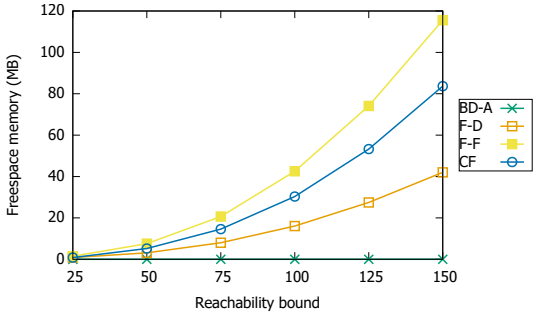
(a) Unicycle primitives, all 16 tables.



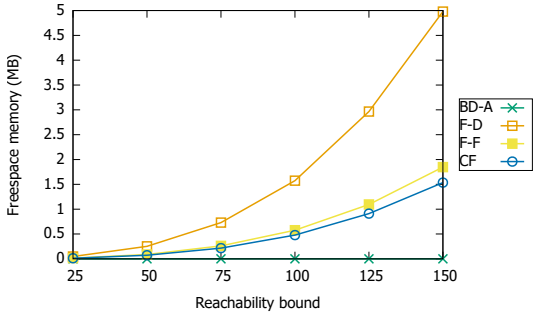
(b) Urban primitives, all 32 tables.



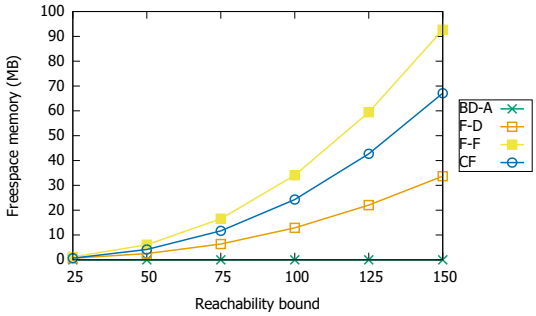
(c) Unicycle primitives, 3 tables.



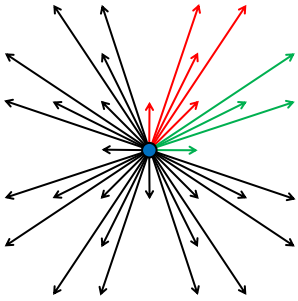
(d) Urban primitives, 5 tables.



(e) Unicycle primitives, 3 tables, shrunk for cardinal and diagonal orientations.



(f) Urban primitives, 5 tables, shrunk for cardinal and diagonal orientations.



(g) Rotational and reflexive symmetries for Urban primitives. Each arrow denotes an orientation for which a freespace information table T_θ can be stored.
Red arrows: The five stored freespace information tables.
Green arrows: Can be obtained from red arrows by reflexive symmetry around the diagonal.
Black arrows: Can be obtained from the red and green arrows by rotational symmetries (or reflexive symmetries around the x- and y-axes).

Figure 4.10: Storage of freespace information.

sets of primitives, including the Unicycle and Urban primitives, also have rotational and reflexive symmetries that can be exploited to further reduce the memory required for storing freespace information. For instance, for both sets of primitives, the freespace-shortest and freespace-canonical paths (using an appropriate canonical ordering) that originate at (0,0,N) are rotationally symmetric to those that originate at (0,0,E), (0,0,S), (0,0,W). Therefore, the tables T_N, T_W, T_S, T_E are all rotationally symmetric and storing only one of them is sufficient. By exploiting these rotational symmetries, as well as reflexive symmetries along the cardinal and diagonal axes, the number of stored tables can be reduced from 16 to 3 for Unicycle primitives and from 32 to 5 for Urban primitives. Figure 4.10g shows these symmetries, and Figures 4.10c and 4.10d show the memory required for storing only these tables. Furthermore, the number of entries within each table T_θ can also be reduced for cardinal and diagonal θ, since freespace-shortest paths and freespace-canonical paths (using an appropriate canonical ordering) that originate at $(0, 0, \theta)$ may have reflexive symmetries along cardinal or diagonal axes, as can be seen in Figure 4.1. Figures 4.10e and 4.10f show the memory required for storing only 3 and 5 tables for Unicycle and Urban primitives, respectively, by also exploiting symmetries within tables.

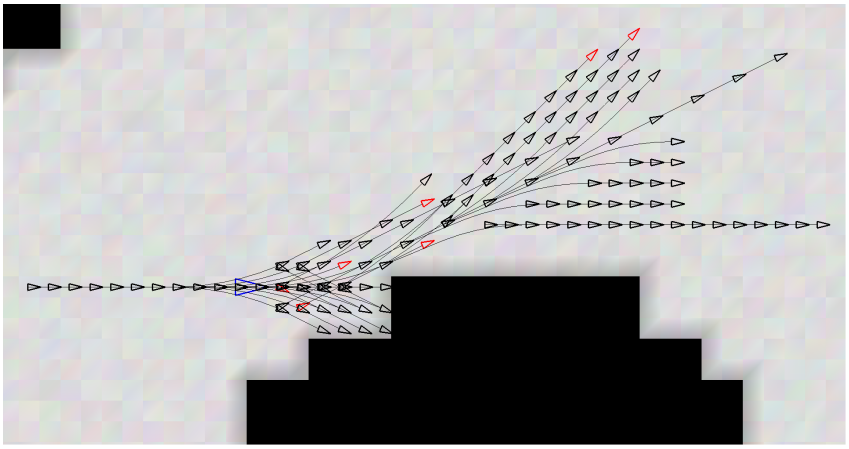
4.6.4 R-Connect

In this section, we compare the connection times of the six R-connect algorithms outlined in Table 4.1 and investigate how different factors contribute to their differences in connection times. As part of our analysis, we also investigate the average number of pairs of R-reachable vertices to help characterize the constraints on the construction of R SGs and R SCSGs, for various R.

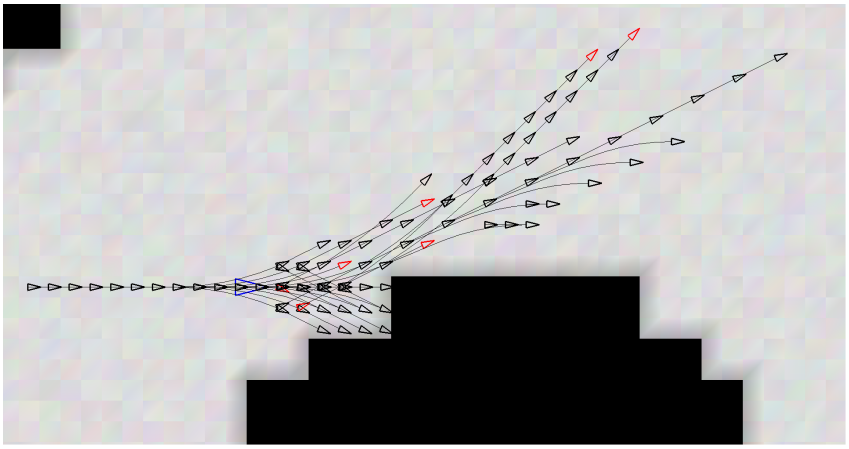
4.6.4.1 Setup

The six R-connect algorithms outlined in Table 4.1 differ in which vertices they expand, how they process the successors of expanded vertices, which data structures they use for OPEN, and whether they perform duplicate detection. These differences can be attributed broadly to the following three factors:

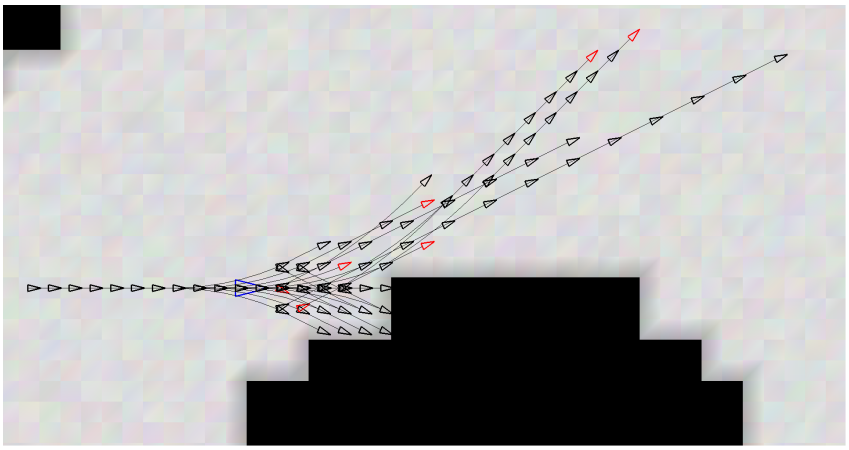
- **Reachability relation (number of R-reachable vertices from s):** As we have discussed in Sections 4.4.3 and 4.5.3, for a given reachability bound b, CF$b \subseteq$ F$b \subseteq$ BDb. Therefore, for any given start vertex s, reachability bound b, and set of subgoals S, there are at least as many vertices that are direct-BDb-reachable from s (with respect to S) as there are vertices that are direct-Fb-reachable or direct-CFb-reachable from s. Directly comparing the connection times of BDb-Connect variants to Fb-Connect variants or CFb-Connect for the same reachability bound is unfair since BDb-Connect variants inherently need to do more work to guarantee completeness. Namely, whereas BDb-Connect variants need to explore shortest paths to all direct-BDb-reachable vertices n from s with $d(s, n) \leq b$, Fb-Connect variants and CFb-Connect can avoid exploring paths that are not freespace-shortest or freespace-canonical, respectively. Figure 4.11 shows an example of the "extra



(a) BD50-Connect (aggressive). Expands all vertices within a distance of 50 from the start vertex, unless all paths to them with lengths greater than b are covered by subgoals.



(b) F50-Connect. Expands all vertices within a distance of 50 from the start vertex, unless all freespace paths to them are blocked by obstacles or covered by subgoals. Arbitrary expansion order. (Fb-F variant: only generate freespace-shortest successors.)



(c) CF50-Connect. Expands all vertices within a distance of 50 from the start vertex, unless the canonical freespace paths to them are blocked by obstacles or covered by subgoals. Arbitrary expansion order, only generate freespace-canonical successors, no duplicate detection.

Figure 4.11: Search trees of BD50-, F50-, and CF50-Connect. The start vertex is shown in blue, and the subgoals are shown in red.

work" performed by the aggressive variant of BDb-Connect, compared to Fb- and CFb-Connect. Although the relation CFb $\subseteq$ Fb $\subseteq$ BDb implies "extra work" that needs to be performed by BDb-Connect variants, it also means fewer constraints on the SGs or SCSGs constructed with respect to BDb-reachability, as we observe and discuss in Sections 4.6.5 and 4.6.6.

- **Optimizations for managing OPEN and processing successors of expanded vertices:** Since Fb-Connect variants and CFb-Connect perform depth-first searches to explore freespace-shortest and freespace-canonical paths, respectively, their OPEN lists can be implemented using a stack (Sections 4.4.4 and 4.5.4) instead of a priority queue, allowing for constant time insertions and removals from OPEN. Whereas the distance variant of Fb-Connect processes each successor of an expanded vertex u and performs freespace-distance look-ups to determine whether to generate that successor, the flag variant of Fb-Connect and CFb-Connect perform a single look-up to generate all freespace-shortest or freespace-canonical successors of u, respectively. That is, they avoid processing successors that are not freespace-shortest or freespace-canonical, respectively, reducing the time to perform each expansion. Additionally, CFb-Connect does not perform duplicate detection, since all freespace-canonical paths that originate at a vertex are guaranteed to form a tree (Section 4.5.4).

- **Strategies for avoiding to expand non-direct-reachable vertices:** The three variants of BDb-connect correspond to the three variants of the Overlay-Connect algorithm discussed in Section 3.2.5, which differ in how they try to void expanding vertices that are not direct reachable (from the start vertex or to the goal vertex, with respect to some set of subgoals), with various degrees of success and various degrees of overhead incurred per expansion: Whereas the conservative variant maintains covered values to terminate its search when all vertices in OPEN are covered, the aggressive and stall-on-demand variants do not expand (that is, stall) subgoals, and the stall-on-demand variant also checks to see if each vertex selected for expansion can be stalled instead of expanded by changing its parent to another stalled vertex without increasing its g-value. The two Fb-Connect variants and CFb-Connect operate as the aggressive variant of Overlay-Connect.

In order to evaluate how the three factors discussed above affect the connection times of the six R-connect algorithms, we report four sets of experimental results, using reachability bounds varying from 25 to 150 and percentage of subgoals varying from 0 to 64 (that is, 0, 1, 2, 4, 8, 16, 32, or 64 percent of the vertices of G are randomly selected as subgoals). First, we report the average number of vertices that are BDb-, Fb-, or CFb-reachable from 1000 randomly selected start vertices. Second, we report the average expansion times of the six R-connect algorithms when run from these 1000 vertices, assuming that there are no subgoals, which guarantees that the three variants of BDb-Connect expand the same set of vertices. Third, we report connection times and the number of non-direct-R-reachable vertices expanded by the six R-connect algorithms for different percentages of subgoals. Finally, we report the "direct-R-reachable expansion rates" of the six R-connect algorithms, which measure the number of direct-R-reachable

vertices expanded per millisecond. This measure normalizes the connection times of different R-connect algorithms with respect to the number of direct-R-reachable vertices that they need to expand to guarantee correctness, which, we think, provides a fair measure for comparison.

4.6.4.2 Number of R-Reachable Vertices from a Given Vertex

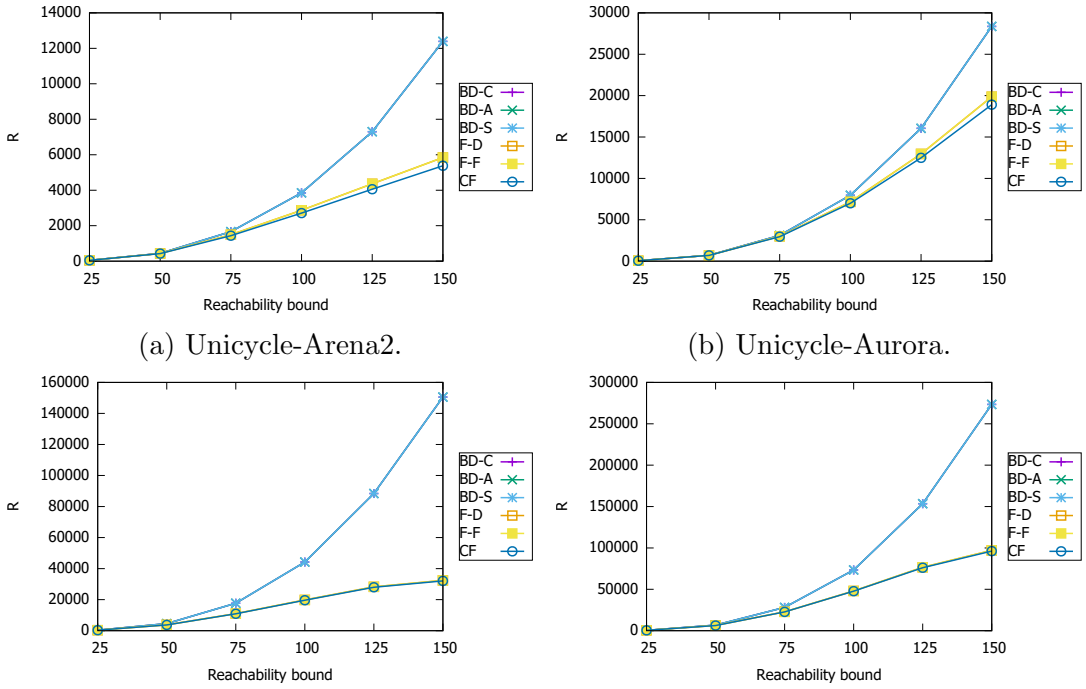
In this section, we compare the average number of vertices that are R-reachable from the start vertex, for varying reachability relations and reachability bounds. For brevity, we refer to this value as the "number of R-reachable vertices". Figure 4.12 reports the average number of BDb-, Fb-, and CFb-reachable vertices with b varying from 25 to 150, on each of the four benchmarks.

We first analyze how the number of BDb-reachable vertices varies, as the reachability bound, primitives, and grid vary.

- **BDb, varying b:** State lattices can be considered to be "3-dimensional" graphs, in the sense that their vertices are uniquely defined by their x-coordinates, y-coordinates, and orientations. However, the third dimension of state lattices, namely the range of possible orientations, is significantly smaller than their first two dimensions, namely the range of possible x- and y-coordinates. Consider a "ball of radius b" B_b centered at a vertex s in a state lattice, which contains exactly those vertices with distance less than or equal to b from s (that is, all BDb-reachable vertices from s). As b increases, we expect B_b to include new vertices that do not share their x-coordinates, y-coordinates, or orientations with the existing vertices, when b is sufficiently small. However, for sufficiently large b, B_b includes vertices for all possible orientations, in which case increasing b can only add new vertices that do not share their x- or y-coordinates with existing vertices, but not their orientations. Therefore, as b increases, we expect the number of vertices in B_b to increase cubically at first and then, eventually, quadratically. Our results show that B_b grows cubically for $b \leq 150$, as can be observed by the shape of the (blue) curves in Figure 4.13, that correspond to the cubic root of the number of vertices in B_b.

- **BDb, varying primitives:** State lattices constructed with respect to Urban primitives, compared to those constructed with respect to Unicycle primitives, can be considered as (1) "having a thicker third dimension", since Urban primitives have twice the number of induced orientations; and (2) "greater connectivity", since Urban primitives have more primitives that are kinematically feasible to execute from any given orientation, and smaller length multipliers. Both factors may contribute to increasing the number of BDb-reachable vertices. Our results show that the number of BDb-reachable vertices is ∼9–12 times larger on Urban benchmarks compared to Unicycle benchmarks.

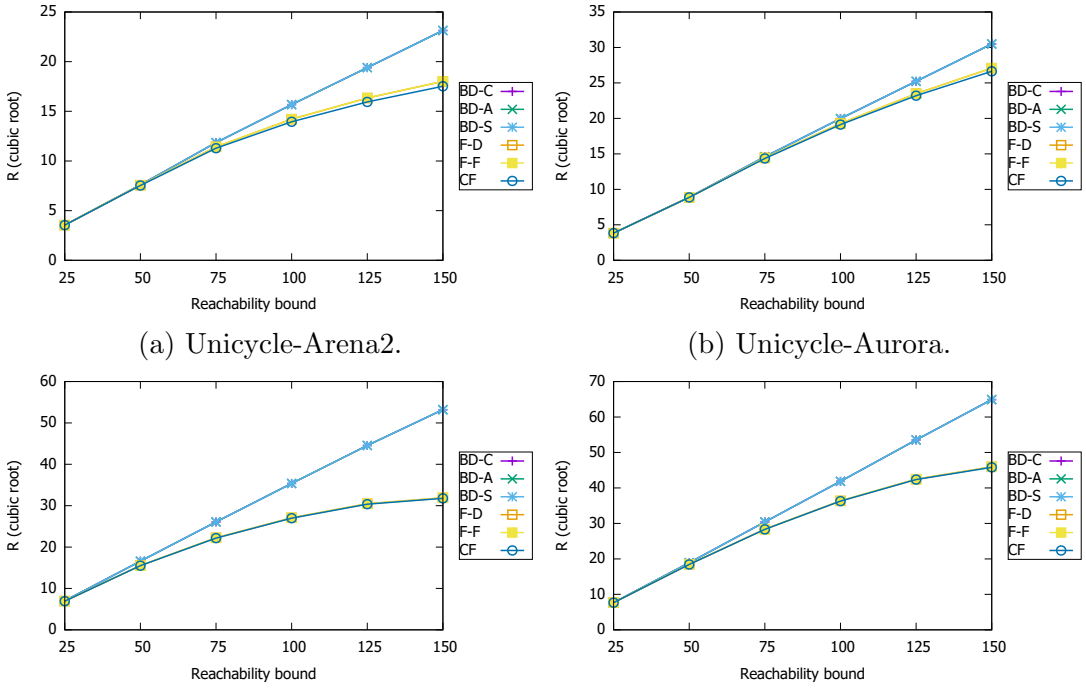
- **BDb, varying grids:** Aurora has larger contiguous regions of unblocked cells than Arena2, resulting in roughly twice as many BDb-reachable vertices in Aurora benchmarks compared to Arena2 benchmarks.



(a) Unicycle-Arena2. (b) Unicycle-Aurora.

(c) Urban-Arena2. (d) Urban-Aurora.

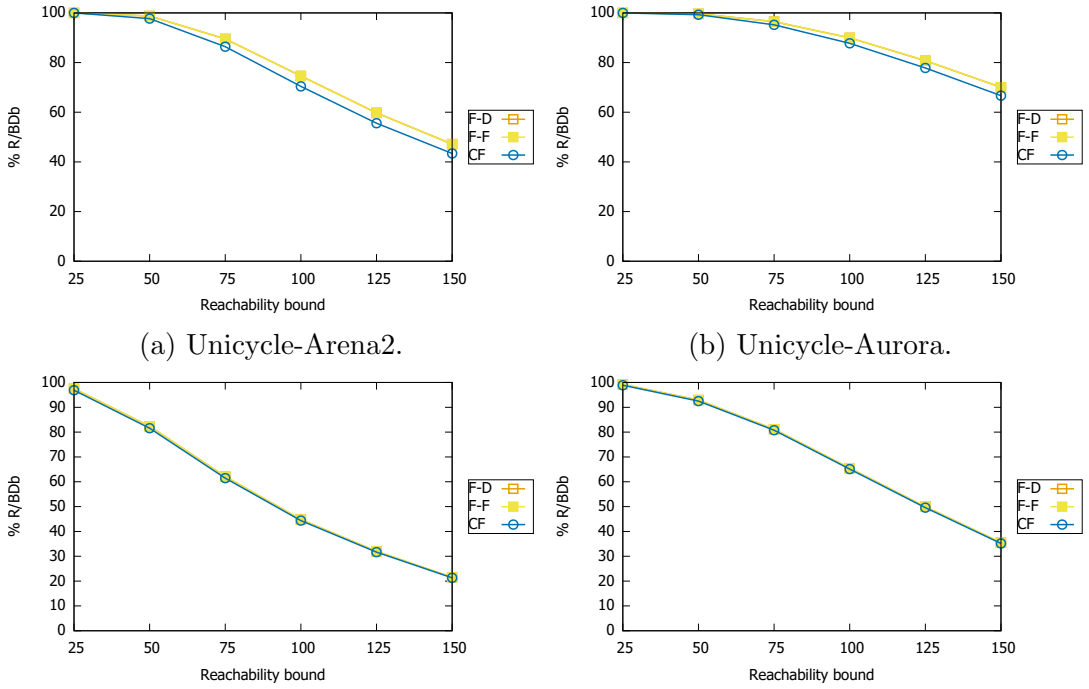
Figure 4.12: Average number of R-reachable vertices from a given source vertex.



(a) Unicycle-Arena2. (b) Unicycle-Aurora.

(c) Urban-Arena2. (d) Urban-Aurora.

Figure 4.13: Cubic root of the average number of R-reachable vertices from a given source vertex.



(a) Unicycle-Arena2. (b) Unicycle-Aurora.

(c) Urban-Arena2. (d) Urban-Aurora.

Figure 4.14: Percentage of BDb-reachable vertices that are also Fb- and CFb-reachable from a source vertex.

We now analyze how the number of Fb- and CFb-reachable vertices varies, as the reachability bound, primitives, and grid vary, and how these numbers compare to the number of BDb-reachable vertices. We perform our analysis in the context of the following observation: CFb = Fb = BDb on a freespace state lattice $\mathcal{F}$ since, for every s and t with $d_{\mathcal{F}}(s,t) \leq b$, there exists a freespace-canonical s-t path π with $l(\pi) \leq b$ on $\mathcal{F}$ (that is, $(s,t) \in$ CFb), which is also a freespace-shortest s-t path on $\mathcal{F}$ (that is, $(s,t) \in$ Fb), and an s-t path on $\mathcal{F}$ (that is, $(s,t) \in$ BDb). However, when the underlying grid contains blocked cells, CFb = Fb = BDb no longer necessarily holds. Instead, CFb $\subseteq$ Fb $\subseteq$ BDb: even if the freespace-canonical s-t path becomes blocked, a freespace-shortest s-t path might still be unblocked (that is, CFb $\subseteq$ Fb); and, if all freespace-shortest s-t paths become blocked, an s-t path π' with $l(\pi') \leq b$ might still be unblocked (that is, Fb $\subseteq$ BDb). In this sense, BDb reachability is more "robust" to cells becoming blocked on the underlying grid, since it is allowed to use a larger set of paths (any s-t path π' with $l(\pi') \leq b$) to go "around" them, whereas Fb-reachability is only allowed to use freespace-shortest paths, and CFb-reachability is only allowed to use the unique freespace-canonical s-t path. We perform our analysis from this perspective, by starting on $\mathcal{F}$ (where CFb = Fb = BDb), and then analyzing how each reachability relation "adapts" as cells become blocked when the freespace grid is replaced with Arena2 or Aurora. Figure 4.14 shows the percentage of BDb-reachable vertices that are also Fb- and CFb-reachable.

- **Fb and CFb vs. BDb, varying b:** As b increases, the percentage of BDb-reachable vertices that are also Fb- or CFb-reachable decreases. Consider the tree of all freespace-canonical paths with length less than or equal to b that originate at a vertex on $\mathcal{F}$. When blocked cells are introduced, this tree is more likely to remain intact for small b since it is less likely for a smaller tree to have one of its paths blocked by a blocked cell than it is for a larger tree. For instance, we observe that, for $b = 25$, almost all ($\sim$99–100%) BDb-reachable vertices are also Fb- or CFb-reachable, which is similar to the case CFb = Fb = BDb. As b grows large, it is more likely that the tree is no longer intact and is missing whole subtrees. Fb-reachability can "repair" this tree with alternate freespace-shortest paths, whereas BDb-reachability can "repair" it using any path with length less than or equal to b, which leads to the case CFb $\subseteq$ Fb $\subseteq$ BDb.

- **Fb and CFb vs. BDb, varying primitives:** Since Urban primitives have smaller length multipliers than Unicycle primitives, the trees of freespace-canonical paths are typically larger (that is, reach cells that are further away) on Urban freespace state lattices than on Unicycle freespace state lattices for a given reachability bound b. We therefore observe that, as b increases, the percentage of BDb-reachable vertices that are also Fb- or CFb-reachable decreases faster on Urban benchmarks than on Unicycle benchmarks.

- **Fb and CFb vs. BDb, varying grids:** Since Aurora has larger contiguous regions of unblocked cells than Arena2, the tree of freespace-canonical paths is more likely to remain intact on Aurora benchmarks than Arena2 benchmarks. We therefore observe that, as b increases, the percentage of BDb-reachable vertices that are also Fb- or CFb-reachable decreases slower on Urban benchmarks than Unicycle benchmarks.

- **Fb vs. CFb:** The difference between the number of Fb- and CFb-reachable vertices is small, up to 1% on Urban benchmarks and up to 8% on Unicycle benchmarks. If all freespace-shortest paths on a state lattice were unique shortest paths (that is, were freespace-canonical), it would hold that CFb = Fb (since the unique freespace-shortest path between them becomes blocked when the freespace-canonical path between them becomes blocked). We suggest that the degree of deviation from CFb = Fb could be used as a measure for the symmetries present in state lattices. Since our experiments show that the deviation from CFb = Fb is greater for Unicycle benchmarks than Urban benchmarks, we suspect that there are more symmetries present in Unicycle benchmarks than Urban benchmarks. We revisit this notion again when discussing our next set of results.

To summarize, as the reachability bound b increases from 25 to 150, the number of BDb-reachable vertices grows near-cubically, and the percentage of those vertices that are also Fb- or CFb-reachable decreases on all four benchmarks. Urban benchmarks have more BDb-reachable vertices than Unicycle benchmarks, and Aurora benchmarks have more BDb-reachable vertices than Arena2 benchmarks, which means that BDb-Connect variants might have to expand more vertices on Urban or Aurora benchmarks, and expand

the most vertices on the Urban-Aurora benchmark ($\sim$ 20 times more than the Unicycle-Arena2 benchmark). Similar trends hold for the number of Fb- and CFb-reachable vertices on the different benchmarks, but the percentage of BDb-reachable vertices that are also Fb- or CFb-reachable are the highest on the Unicycle-Aurora benchmark and the lowest on the Urban-Arena2 benchmark.

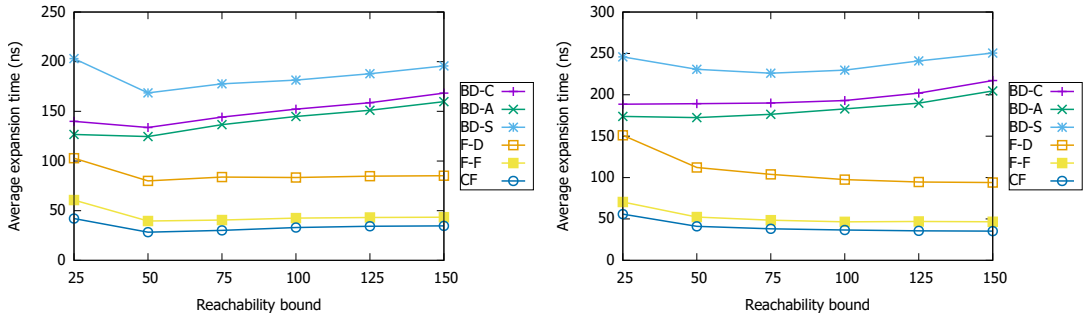
4.6.4.3 Average Expansion Time

In this section, we compare the connection times of the six R-connect algorithms, assuming that there are no subgoals present. That is, each R-connect algorithm expands all R-reachable vertices that we have measured in the previous section. Since there are typically more BDb-reachable vertices than there are Fb- or CFb-reachable ones, BDb-Connect variants expand more vertices than Fb-Connect variants and CFb-Connect. To normalize for this fact, we instead compare the average expansion times (that is, the connection times divided by number of expansions) of the six R-connect algorithms, reported in Figure 4.15. Figure 4.16 reports the average number of successors processed per expansion, that is, the successors explicitly iterated over by the various R-connect algorithms.

We make the following observations:

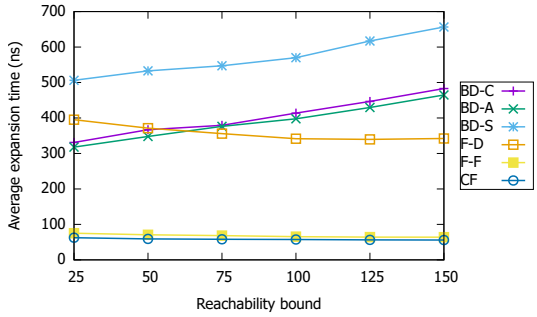
- **BDb, OPEN list:** As the reachability bound b increases, the average expansion time increases for the BDb-Connect variants, which maintain their OPEN lists as priority queues: as b increases, the number of BDb-reachable vertices increases as discussed in the previous section, and, therefore, the average number of vertices in OPEN, the time to perform insertions, removals, and g-value updates in OPEN, and thus the BDb-connection times increase. This increase is more significant in Urban and Aurora benchmarks compared to Unicycle and Arena2 benchmarks, respectively, since the number of BDb-reachable vertices is typically higher in these benchmarks. For instance, the average expansion time of the aggressive variant of BD150-Connect is 1.10 times longer than it is for BD25-Connect on the Unicycle-Arena2 benchmark, but 1.12, 1.17, and 1.36 times longer on the Unicycle-Aurora, Urban-Arena2, and Urban-Aurora benchmarks, respectively. Since the Fb-Connect variants and CFb-Connect use a stack implementation of OPEN, their average expansion times are not affected by the size of OPEN and the reachability bound b.

- **BDb, avoiding expansions of non-BDb-reachable vertices:** All three BDb-Connect variants perform the same sequence of expansions and process the same number of successors per expansion, 3.21–3.58 on Unicycle benchmarks and 19.43–23.50 on Urban benchmarks (these numbers are higher on Aurora benchmarks compared to Arena2 benchmarks since Aurora has larger contiguous regions of unblocked cells than Arena2, as noted in the previous section). However, their average expansion times are different due to their different strategies for avoiding the expansion of non-direct-BDb-reachable vertices (these strategies have no effect in this set of experiments since we assume that there are no subgoals). The aggressive variant of BDb-Connect performs a single check for each vertex selected for expansion to

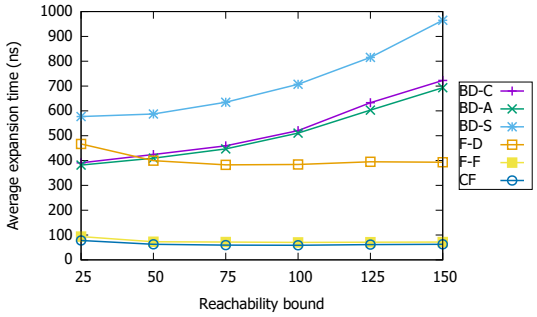


(a) Unicycle, arena2.

(b) Unicycle, Aurora.

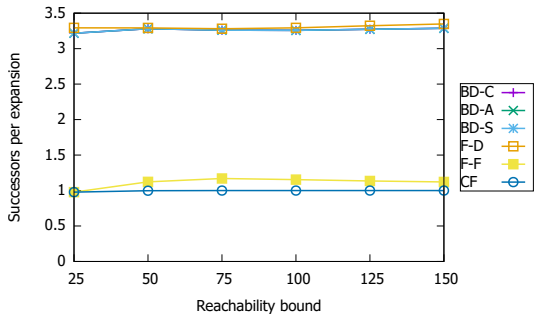


(c) Urban, arena2.

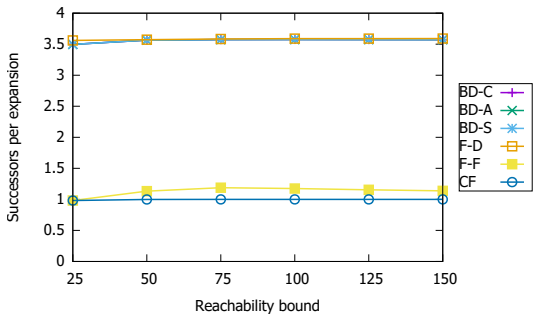


(d) Urban, Aurora.

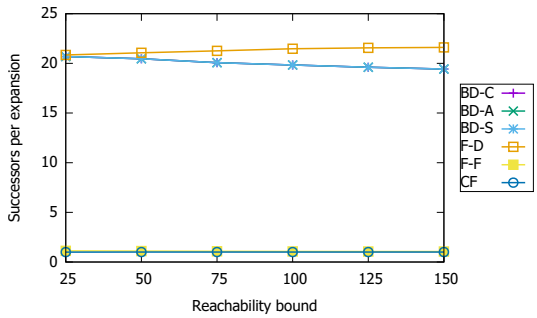
Figure 4.15: Average R-connect expansion times when no subgoals are present.



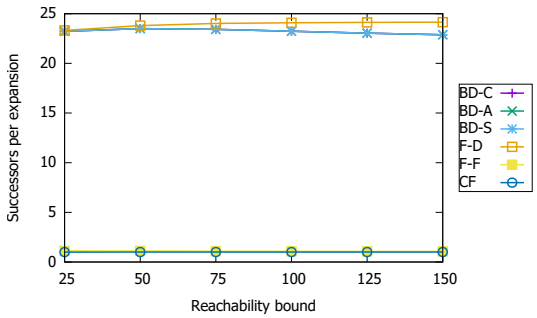
(a) Unicycle-Arena2.



(b) Unicycle-Aurora.



(c) Urban-Arena2.



(d) Urban-Aurora.

Figure 4.16: Average numbers of successors processed per expansion by R-connect when no subgoals are present.

see if it is a subgoal, and has an average expansion time of 153–191 nanoseconds on Unicycle benchmarks and 354–612 nanoseconds on Urban benchmarks. The conservative variant of BD*b*-Connect maintains covered values and therefore has slightly longer average expansion times, 7–16 nanoseconds longer than the aggressive variant on Unicycle benchmarks and 16–29 nanoseconds longer on Urban benchmarks. The stall-on-demand variant of BD*b*-Connect iterates over predecessors of vertices selected for expansion to see if the vertex can be stalled, and has the longest average expansion times, 35–50 nanoseconds longer than the aggressive variant on Unicycle benchmarks and 170–270 nanoseconds longer on Urban benchmarks.

- **F*b*-D vs. BD*b*-A:** The distance variant of F*b*-Connect iterates over all successors of expanded vertices to determine whether they are freespace-shortest successors, by looking up freespace distances from the start vertex to each successor. Therefore, for small b, the average expansion times are longer for the distance variant of F*b*-Connect than for the aggressive variant of BD*b*-Connect. However, as discussed earlier, as b increases, the average expansion time increases for the BD*b*-Connect variants, making them longer than for the distance variant of F*b*-Connect.

- **F*b*-F vs. F*b*-D:** The average expansion times for the flag variant of F*b*-Connect are significantly shorter than for the distance variant, since the flag variant performs a single look-up to determine all (unblocked) freespace successors of an expanded vertex, rather than performing a freespace distance look-up for each of its successors. On average, the distance variant of F*b*-Connect processes 3.29–3.59 and 20.84–24.14 successors per expansion on Unicycle and Urban benchmarks, respectively, whereas the flag variant processes only 1.12–1.19 and 1.06–1.13 freespace-shortest successors, respectively.

- **CF*b* vs. F*b*-F:** The average expansion times for CF*b*-Connect are shorter than for the flag variant of F*b*-Connect, since CF*b*-Connect processes fewer successors per expanded vertex and does not perform duplicate detection. CF*b*-Connect processes on average ~ 1 freespace-canonical successor per expanded vertex, since it essentially traverses a tree (namely, the tree of unblocked freespace-canonical paths originating at the start vertex), where the number of vertices in the tree is equal to one plus the number of edges in the tree. In other words, the numbers of successors processed per expanded vertex are 1.12–1.19 and 1.06–1.13 times smaller for CF*b*-Connect than the flag variant of F*b*-Connect on the Unicycle and Urban benchmarks, respectively. However, the average expansion times are 1.25–1.44 and 1.15–1.20 shorter for CF*b*-Connect than the flag variant of F*b*-Connect on the Unicycle and Urban benchmarks, respectively, suggesting that the avoidance of duplicate detection also contributes to shorter average expansion times.

- **Symmetries:** If all freespace-shortest paths on a state lattice were unique shortest paths (that is, were freespace-canonical), the average number of successors processed per expanded vertex would be one for the flag variant of F*b*-Connect as well, similar to CF*b*-Connect. We suggest that the degree of deviation from an average of 1 successor processed per expansion by the flag variant of F*b*-Connect could be used as another measure (in addition to the one mentioned in the previous section) for the

symmetries present in state lattices. Since our experiments show that this deviation is larger for Unicycle benchmarks than Urban benchmarks, we suspect that there are more symmetries present in Unicycle benchmarks than Urban benchmarks, which is consistent with our observation in the previous section.

To summarize, the average expansion times are the longest for the BDb-Connect variants, especially when b is large, since they use priority queues to maintain their OPEN lists. Among the BDb-Connect variants, the aggressive variant has the shortest average expansion times, and the stall-on-demand variant has the longest, due to their different strategies for avoiding the expansions of non-direct-BDb-reachable vertices. The average expansion times for the distance variant of Fb-Connect are comparable to those of the BDb-Connect variants, but typically a bit shorter since Fb-Connect maintains its OPEN list as a stack. The average expansion times for the flag variant of Fb-Connect are significantly shorter than for the distance variant, especially on Urban benchmarks, since the flag variant processes only a small number of successors per expansion; and even shorter for CFb-Connect since CFb-Connect processes on average only 1 successor per expansion and avoids performing duplicate detection.

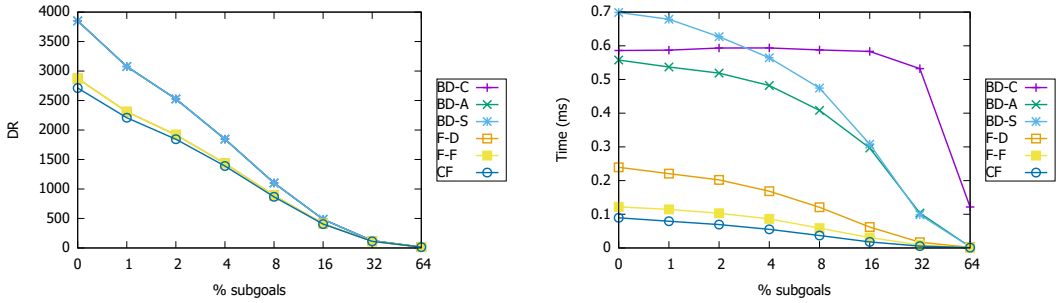
4.6.4.4 Avoiding Expansions of Non-Direct-Reachable Vertices

In this section, we compare how successful the six R-connect algorithms are in avoiding the expansions of non-direct-R-reachable vertices, by measuring the number of non-direct-R-reachable vertices they expand relative to all vertices they expand, for varying percentages p of vertices of G selected as subgoals. For brevity, we report results only for a fixed reachability bound of 100, and report results only on the Unicycle-Arena2 and Urban-Aurora benchmarks. The results for other reachability bounds and benchmarks have similar trends and are not reported. Figures 4.17 and 4.18 report the average number of direct-R-reachable vertices, the average connection times, the average number of vertices *popped* from OPEN (that is, expanded or stalled), and the percentage of popped vertices that are not direct-R-reachable, for the six R-Connect variants on the Unicycle-Arena2 and Urban-Aurora benchmarks, respectively. We count subgoals that are not expanded by the aggressive and stall-on-demand variants of BDb-Connect, both variants of Fb-Connect, and CFb-Connect as stalled (and, therefore, popped).

We make the following observations:

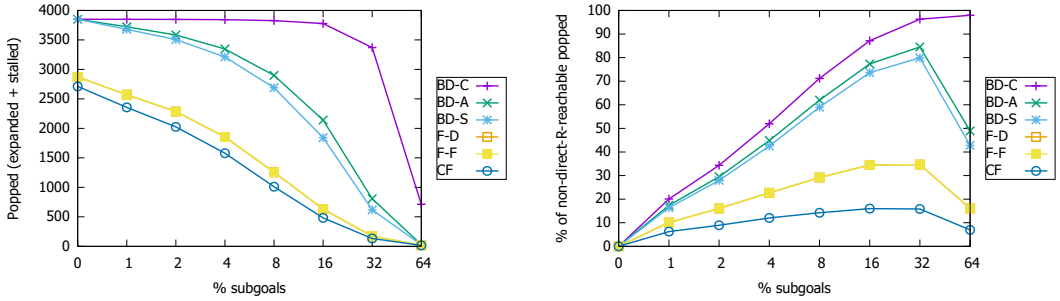
- **Number of direct-R-reachable vertices:** By definition of direct-reachability (Definition 3.2), as the number of subgoals increases (that is, as p increases), the number of direct-R-reachable vertices decreases. We observe that, in both benchmarks, this reduction is similar for all reachability relations. For instance, when 1% of the vertices of G become subgoals, for all R, 18.5%-20.0% and 9.3%-9.8% of the R-reachable vertices become non-direct-R-reachable in the Unicycle-Arena2 and Urban-Aurora benchmarks, respectively.

- **R-connect time, BDb-Connect variants:** As we have discussed in Section 3.2.5, the main downside of the aggressive variant of Overlay-Connect is that it can go "around" the subgoals, possibly via suboptimal paths. As we have suggested in



(a) Number of direct-R-reachable vertices.

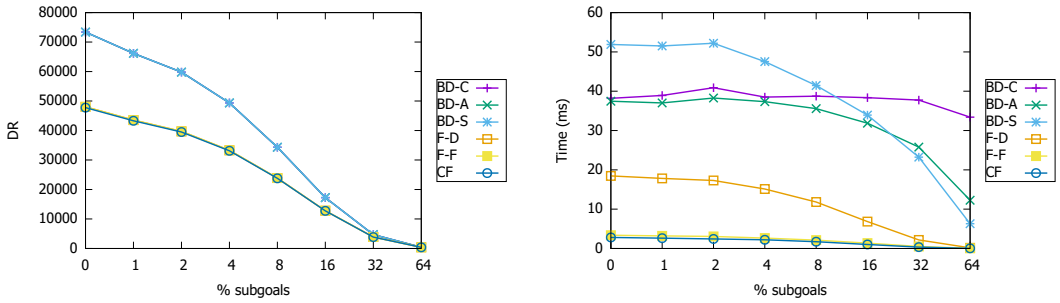
(b) R-connect time (ms).



(c) Popped.

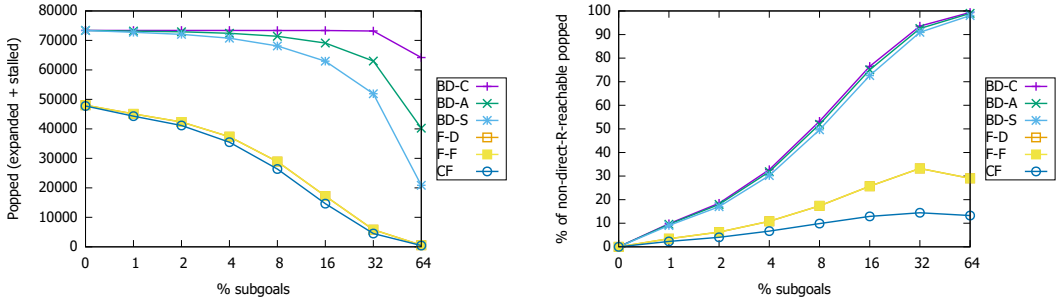
(d) % redundant popped.

Figure 4.17: R-connect statistics, Unicycle-Arena2, varying percentage of subgoals, reachability bound 100.



(a) Number of direct-R-reachable vertices.

(b) R-connect time (ms).



(c) Popped.

(d) % redundant popped.

Figure 4.18: R-connect statistics, Urban-Aurora, varying percentage of subgoals, reachability bound 100.

Section 3.3, bounding its search using a reachability relation can help to mitigate this problem, for instance, by using BDb-reachability as the reachability relation. Our results show that the aggressive variant is, overall, the fastest BDb-Connect variant in both benchmarks, for up to 16% subgoals. We analyze this result in further detail below.

- **Percentage of non-direct-R-reachable vertices popped, BDb-Connect variants:** The aggressive variant of BDb-Connect consistently pops fewer vertices for expansion than the conservative variant does and, since its average expansion times are shorter, its connection times are also consistently smaller. The stall-on-demand variant is guaranteed to pop no more vertices for expansion than the aggressive variant, since it stalls all the vertices that the aggressive variant stalls. However, the difference in the number of popped vertices for the stall-on-demand and aggressive variants is smaller than for the aggressive and conservative variants. Furthermore, the aggressive variant has shorter average expansion times than the stall-on-demand variant, as demonstrated in the previous section. The results show that the aggressive variant typically has shorter connection times than the stall-on-demand variant on both benchmarks when less than 16% of the vertices of G are subgoals, and similar or slightly longer connection times when more subgoals are introduced.

- **Fb-Connect variants and CFb-Connect:** Since Fb-Connect variants differ only in their implementations, they expand and stall the same set of vertices, so we will not distinguish between them in this analysis. Both Fb-Connect and CFb-Connect are based on the aggressive variant of Overlay-Connect. Similar to how using BDb-reachability as the reachability relation can help to prevent the aggressive variant of Overlay-Connect from going "around" subgoals by bounding its search, using Fb- or CFb-reachability as the reachability relation prevents this behavior by preventing it from exploring suboptimal paths: Fb-Connect and CFb-Connect explore only freespace-shortest and freespace-canonical paths on G, respectively, which are guaranteed to be shortest paths on G. As the results show, the Fb- and CFb-Connect variants have the smallest percentage of popped vertices that are non-direct-R-reachable from the start. This percentage is roughly half for CFb-Connect than what it is for Fb-Connect on both benchmarks, and less than half for Fb-Connect than what it is for all BDb-Connect variants.

To summarize, the percentage of popped vertices that are not direct-R-reachable is the lowest for the Fb-Connect variants and CFb-Connect, which only explore shortest paths and therefore have only a small tendency to go "around" subgoals; the highest for the conservative variant of BDb-Connect, which also explores shortest paths only but does not stall any vertices (besides terminating its search early); and lower for the stall-on-demand variant of BDb-Connect than the aggressive variant of BDb-Connect, since the stall-on-demand variant can stall more vertices than the aggressive variant. However, since the stall-on-demand variant of BDb-Connect performs an expensive check to determine whether it can stall vertices, its connection times are typically longer than the connection times of the aggressive variant, especially when only a small percentage of the vertices of G are subgoals. As we demonstrate in Section 4.6.5, answering queries

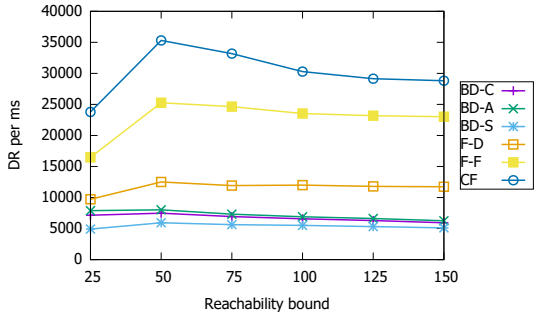
using SGs or SCSGs that have a high percentage of vertices of G as subgoals typically has execution times comparable to A* searches on G, and we therefore use the aggressive variant of BDb-Connect in our experiments only in Sections 4.6.5 and 4.6.6, since it has the smallest connection times among the BDb-Connect variants when only small percentages of the vertices of G are subgoals.

4.6.4.5 Direct-R-Reachable Expansion Rate

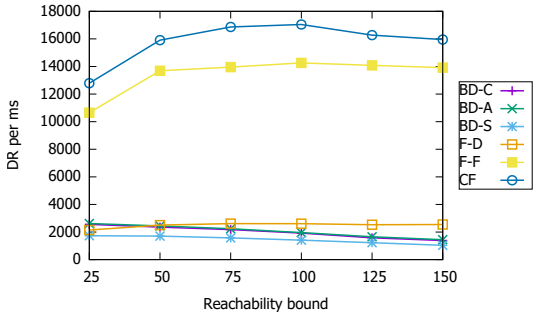
In this section, we compare the "efficiency" of the six R-connect algorithms by measuring their "direct-R-reachable expansion rates" (DR-rates, for short), that is, the number of direct-R-reachable vertices they expand divided by their connection times. The DR-rate can be considered to be the normalization of the connection times of different R-connect algorithms, with respect to the number of direct-R-reachable vertices that they have to expand to guarantee completeness; and, when no subgoals are present, it is equivalent to the inverse of the average expansion times (Section 4.6.4.3).

Figure 4.19 reports the DR-rates of the six R-connect algorithms, for varying reachability bounds and percentages of subgoals, on the Unicycle-Arena2 and Urban-Aurora benchmarks. The results on the Unicycle-Aurora and Urban-Arena2 benchmarks have similar trends and are thus not reported. Table 4.2 reports more detailed statistics for reachability bound 100 and percentages of subgoals 1% and 32%, by also providing the average number of R-reachable and direct-R-reachable vertices, subgoals encountered, expanded and stalled vertices, successors processed per expansion, and connection times. Since the DR-rates of different R-connect algorithms are determined by their average expansion times and how successfully they avoid the expansion of non-direct-R-reachable vertices, our discussion of DR-rates is simply a summary of our discussions from the previous sections.

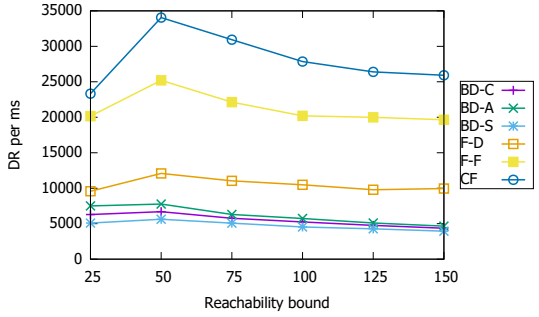
CFb-Connect and both variants of Fb-Connect have higher DR-rates than the three BDb-Connect variants since their OPEN lists are implemented as stacks rather than priority queues and a smaller percentage of the vertices expanded by them are non-direct-R-reachable. The flag variant of Fb-Connect has a higher DR-rate than the distance variant, especially on Urban benchmarks, since it only processes the freespace-successors of expanded vertices rather than iterating over all successors and determining whether they are freespace-shortest by using freespace distances; but it has a lower DR-rate than CFb-Connect since CFb-Connect processes fewer successors (namely, only the freespace-canonical successors) per expansion, does not perform duplicate detection, and has a smaller percentage of popped vertices that are not direct-R-reachable. The DR-rates of the BDb-Connect variants are similar to each other, where the aggressive variant has the higher DR-rate when a small percentage (less than 16%) of the vertices of G are subgoals, and the stall-on-demand variant has the higher DR-rate when a large percentage of the vertices of G are subgoals. For 1% subgoals and reachability bound 100 on the Urban-Aurora benchmark, the DR-rate of CFb-Connect is 1.21 and 6.75 times higher than that of the flag and distance variants of Fb-Connect, respectively, and 12.83, 9.22, and 9.70 times higher than that of the stall-on-demand, aggressive, and conservative variants of BDb-Connect, respectively. As we discuss in Section 4.6.6, the high DR-rates of CFb-Connect and the flag variant of Fb-Connect allow one to find good trade-offs between the



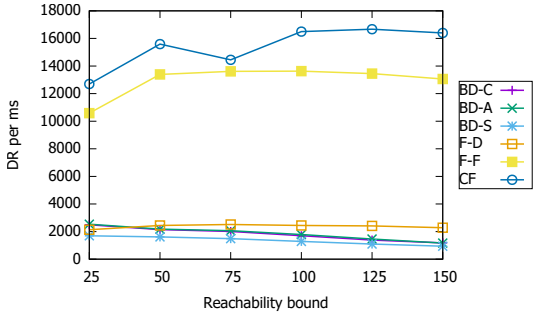
(a) Unicycle-Arena2, 0% subgoals.



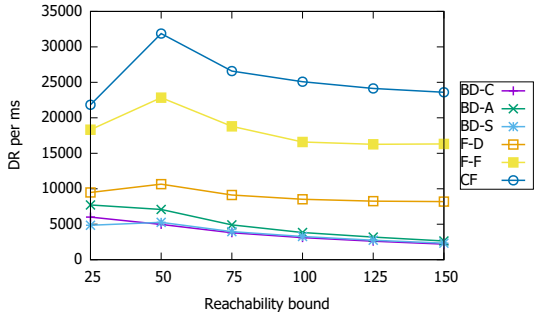
(b) Urban-Aurora, 0% subgoals.



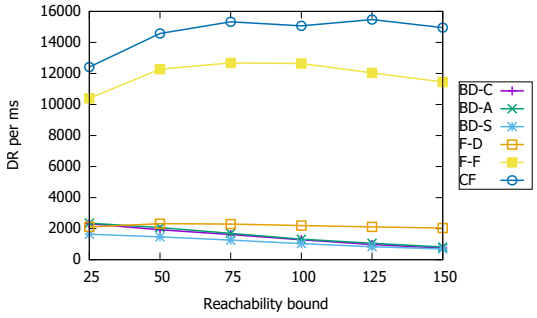
(c) Unicycle-arena2, 1% subgoals.



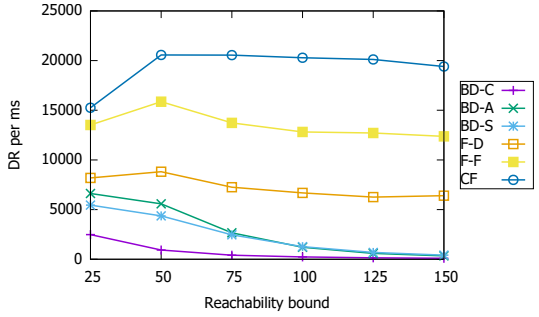
(d) Urban-Aurora, 1% subgoals.



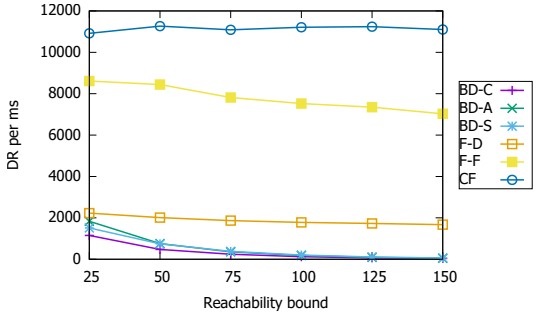
(e) Unicycle-Arena2, 4% subgoals.



(f) Urban-Aurora, 4% subgoals.



(g) Unicycle-Arena2, 32% subgoals.



(h) Urban-Aurora, 32% subgoals.

Figure 4.19: Direct-R-reachable expansion rates of R-connect algortihms on the small and large benchmarks, with different percentages of subgoals and varying reachability bounds.

connection and search times when answering queries using CFb or Fb SCSGs, resulting in short query times.

4.6.5 Subgoal Graphs

In this section, we compare answering queries using BDb, Fb, and CFb SGs with respect to their query-times.

4.6.5.1 Setup

We consider two different ways of constructing R SGs in our experiments, namely using the pruning and growing variants of constructing R-SPCs (Sections 3.3.5 and 3.3.6). We consider four different algorithms for answering queries using R SGs, which use four of the six different variants of performing connection and refinement outlined in Table 4.2, excluding the conservative and stall-on-demand variants of BDb-Connect (as discussed in Section 4.6.4.4). We vary the reachability bound from 25 to 150, similar to our experiments in Section 4.6.4. Due to long preprocessing times, we only present results on the Arena2-Unicycle benchmark, and do not present results for the pruning variant of constructing R-SPCs for reachability bounds above 75. We now provide a brief summary of the pruning and growing variants of constructing R-SPCs and discuss implementation details.

- **Identifying R-SPCs by pruning:** Recall that the pruning variant of identifying an R-SPC S starts by initializing $S = V$ and then removes vertices from S one by one while maintaining that S is an R-SPC (Section 3.3.5), that is, for every pair of vertices $(s, t) \notin R$, S covers at least one shortest s-t path on G. The set of subgoals S identified by the pruning variant is guaranteed to be a *minimal* R-SPC (that is, no $S' \subset S$ is an R-SPC) but not necessarily a *minimum* R-SPC (that is, there might exist an R-SPC S' with $|S'| < |S|$). One method of trying to find a small S is to vary the order with which the pruning variant iterates over the vertices in V. We use the "DFS-completion" ordering to iterate over the vertices in V for our experiments, inspired by Funke et al.'s method for constructing k-hop shortest path covers on road networks (Funke et al., 2014), which corresponds to a post-order traversal of the tree of a depth-first search run from a random vertex (that is, vertices whose children are processed first by the depth-first search appear earlier in the order). Our preliminary results suggest that this ordering is slightly better than a random ordering, producing on average ∼3.6% and up to ∼6.4% fewer subgoals.
- **Identifying R-SPCs by growing:** Recall that the growing variant of identifying an R-SPC starts by initializing $S = \emptyset$ and then adds subgoals to S until it becomes an R-SPC (Section 3.3.6). For each $n \in V$, it first identifies the "fringe vertices of n" F_n, that is, vertices u with $(n, u) \notin R$ and $S \not\sqsubset (n, u)$, and then extends S so that it covers at least one shortest path from n to each fringe vertex of n. We use a heuristic to identify the subgoals used to extend S to cover shortest paths from n to its fringe vertices, which we describe below.

Benchmark	% subgoals	R	R-reachable	Direct-R-reachable	Subgoals	Expanded	Stalled	Succ per expansion	Time (ms)	DR per ms
Unicycle	1	BD-C	3,851	3,077	31	3,849	0	3.257	0.587	5,242
arena2		BD-A	3,851	3,077	37	3,686	37	3.255	0.537	5,731
		BD-S	3,851	3,077	37	3,489	190	3.256	0.679	4,534
		F-D	2,872	2,310	26	2,545	26	3.292	0.221	10,478
		F-F	2,872	2,310	26	2,545	26	1.155	0.114	20,213
		CF	2,712	2,208	24	2,332	24	1.010	0.079	27,848
	32	BD-C	3,851	125	39	3,370	0	3.262	0.532	234
		BD-A	3,851	125	256	551	256	3.253	0.104	1,202
		BD-S	3,851	125	197	362	258	3.255	0.099	1,263
		F-D	2,872	114	55	119	55	3.342	0.017	6,695
		F-F	2,872	114	55	119	55	1.514	0.009	12,789
		CF	2,712	112	42	91	42	1.451	0.006	20,367
Urban	1	BD-C	73,379	66,176	662	73,379	0	23.236	38.923	1,700
Aurora		BD-A	73,379	66,176	732	72,438	732	23.238	37.012	1,788
		BD-S	73,379	66,176	728	66,856	5,896	23.233	51.513	1,285
		F-D	48,045	43,544	450	44,639	450	24.091	17.820	2,444
		F-F	48,045	43,544	450	44,639	450	1.092	3.195	13,629
		CF	47,770	43,305	443	43,877	443	1.010	2.625	16,495
	32	BD-C	73,379	4,701	1,503	73,208	0	23.236	37.751	125
		BD-A	73,379	4,701	20,168	42,848	20,168	23.358	25.781	182
		BD-S	73,379	4,701	16,614	14,103	37,806	23.313	23.236	202
		F-D	48,045	3,879	1,859	3,951	1,859	24.207	2.179	1,780
		F-F	48,045	3,879	1,859	3,951	1,859	1.570	0.516	7,523
		CF	47,770	3,864	1,445	3,072	1,445	1.470	0.345	11,211

Table 4.2: R-connect statistics with reachability bound 100.

We first construct a directed acyclic graph A_n of all shortest paths from n to its fringe vertices, and assign a value of 1 to each fringe vertex and 0 to each non-fringe vertex. We then back-propagate through A_n, starting from the fringe vertices and visiting each vertex before visiting any of its parents in A_n. For each visited vertex u with value $v(u)$ and set of parents P_u in A_n and each parent $p \in P_u$ of u, we increase $v(p)$ by $v(u)/|P_u|$. Intuitively, when the back-propagation terminates, the value of a non-fringe vertex u is high if making it a subgoal covers shortest paths from n to a large number of its fringe vertices, or if it covers shortest paths to fringe vertices to which only a small number of shortest paths from n exist. We then add the vertex u with the highest $v(u) \cdot d(n, u)$ to S (that is, make it a subgoal), update A_n by removing those fringe vertices to which a shortest path from n is covered by u, and repeat the process, until at least one shortest path from n to each of its fringe vertices is covered. When picking the next vertex u to add to S, we consider its distance $d(n, u)$ from n because, otherwise, u is typically selected as one of the successors of n since they typically cover shortest paths from n to the largest number of its fringe vertices.

- **Constructing the edges of R SGs:** We construct the edges of R SGs by performing an R-connect operation from each subgoal. Since Fb-Connect and CFb-Connect can identify a superset of the necessary edges, we perform an additional operation to remove the redundant ones, by performing a Dijkstra search on the SG (possibly with the redundant edges) from each subgoal u until all subgoals v that share an edge with n are expanded, and check for each v whether the Dijkstra search has found an alternative u-v path. We construct the edges of BDb SGs using the conservative variant of BDb-Connect, since it can do so exactly.

- **Memory requirements:** When reporting the memory requirements, we consider only the memory required to store freespace information and the edges of SGs. We store freespace information without exploiting symmetries between primitives. As discussed in Section 4.6.3, by exploiting such symmetries, we could reduce the memory requirements for freespace information by a factor of $\sim$8 for both the Unicycle and Urban primitives. When reporting the memory required to store the edges of SGs, we assume that each edge can be stored using 8 bytes, 4 bytes for its destination and 4 bytes for its length. For the distance variant of Fb-reachability, we assume that each edge of the SG can be stored using 4 bytes, only for its destination, since its length can be looked up from the stored freespace distances.

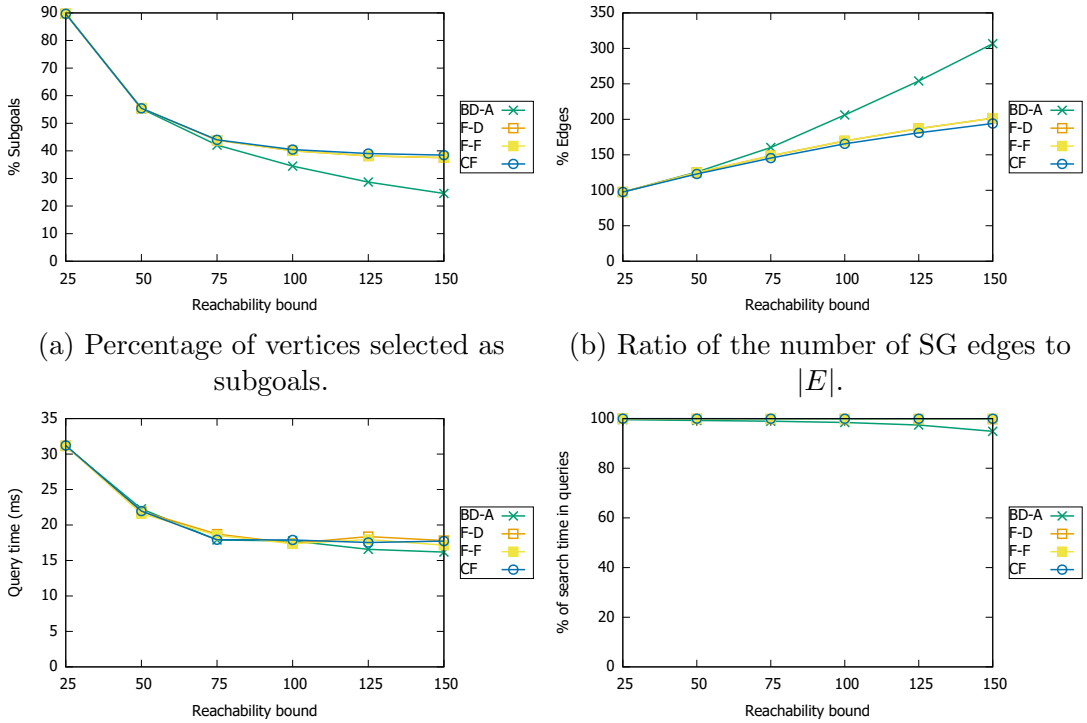
4.6.5.2 Results

Table 4.3 reports the time to construct BDb, Fb, and CFb SGs by pruning or growing the set of subgoals, the numbers of vertices and edges in these SGs relative to G, the memory required to store the edges of these SGs and freespace information, the execution times of queries answered using these SGs relative to A* search times on G (that is, "speed up"), and the percentage of query times spent in the search phase, on the Unicycle-Arena2 benchmark. Figure 4.20 shows the trends from Table 4.3 as the reachability bound varies.

We make the following observations:

R	C	b	Prep. time (s)	Memory (MB) FS	Memory (MB) Edges	Size vs G Subgoals	Size vs G Edges	Search / Query	Speed up
BD-A	P	25	13	0.00	9.33	87.29%	96.85%	99.61%	0.98
		50	622	0.00	13.85	46.36%	143.74%	99.08%	1.73
		75	16,232	0.00	19.04	33.48%	197.66%	98.67%	2.07
	G	25	6	0.00	9.40	89.68%	97.52%	99.58%	1.08
		50	23	0.00	12.11	55.27%	125.74%	99.24%	1.51
		75	69	0.00	15.45	42.09%	160.35%	98.95%	1.88
		100	162	0.00	19.86	34.48%	206.17%	98.43%	1.89
		125	309	0.00	24.48	28.70%	254.05%	97.40%	2.03
		150	532	0.00	29.54	24.57%	306.65%	94.88%	2.07
F-D	P	25	16	0.37	4.67	87.29%	96.85%	99.91%	0.96
		50	716	2.02	6.87	46.47%	142.52%	99.84%	1.65
		75	10,480	5.84	8.81	35.10%	182.95%	99.79%	2.06
	G	25	7	0.37	4.70	89.68%	97.52%	99.90%	1.08
		50	28	2.02	6.01	55.48%	124.82%	99.86%	1.53
		75	76	5.84	7.16	43.85%	148.63%	99.83%	1.79
		100	145	12.59	8.17	40.02%	169.55%	99.81%	1.93
		125	223	23.72	9.00	38.16%	186.90%	99.80%	1.83
		150	304	39.84	9.71	37.53%	201.59%	99.79%	1.89
F-F	P	25	16	0.12	9.34	87.29%	96.85%	99.95%	0.98
		50	716	0.71	13.74	46.47%	142.52%	99.91%	1.68
		75	10,480	2.09	17.62	35.10%	182.95%	99.88%	2.06
	G	25	7	0.12	9.40	89.68%	97.52%	99.95%	1.08
		50	28	0.71	12.02	55.48%	124.82%	99.92%	1.55
		75	76	2.09	14.32	43.85%	148.63%	99.90%	1.81
		100	145	4.61	16.34	40.02%	169.55%	99.89%	1.93
		125	223	8.76	18.00	38.16%	186.90%	99.89%	1.87
		150	304	14.77	19.42	37.53%	201.59%	99.88%	1.96
CF	P	25	17	0.10	9.33	87.29%	96.85%	99.98%	0.97
		50	706	0.58	13.69	46.80%	142.06%	99.95%	1.65
		75	9,979	1.72	17.27	36.50%	179.25%	99.93%	2.03
	G	25	7	0.10	9.40	89.68%	97.52%	99.98%	1.08
		50	27	0.58	11.84	55.43%	122.93%	99.96%	1.53
		75	76	1.72	14.00	43.99%	145.29%	99.94%	1.87
		100	143	3.82	15.94	40.48%	165.47%	99.94%	1.88
		125	214	7.28	17.44	39.03%	181.05%	99.93%	1.91
		150	292	12.28	18.70	38.46%	194.08%	99.93%	1.89

Table 4.3: Comparison of SGs constructed by pruning (P) or growing (G) an R-SPC.



(a) Percentage of vertices selected as subgoals.

(b) Ratio of the number of SG edges to $|E|$.

(c) Query time.

(d) Percentage of query time spent in the search phase.

Figure 4.20: Comparison of SGs on the Unicycle-Arena2 benchmark.

- **Number of subgoals, BDb vs. Fb vs. CFb:** Since CFb $\subseteq$ Fb $\subseteq$ BDb, BDb SGs have fewer subgoals than Fb SGs, which in turn have fewer subgoals than CFb SGs, when using the same reachability bound b and the same construction method. For instance, when using reachability bound 75 and the pruning variant of constructing SGs, the BD75 SG has ∼4.6% fewer subgoals than the F75 SG, which in turn has ∼3.8% fewer subgoals than the CF75 SG.

- **Number of subgoals and edges, varying b:** As the reachability bound increases, the number of subgoals in BDb, Fb, and CFb SGs decrease, since, for every $b_1 < b_2$, BDb_1 $\subseteq$ BDb_2, Fb_1 $\subseteq$ Fb_2, and CFb_1 $\subseteq$ CFb_2. However, as the reachability bound increases, the number of edges in BDb, Fb, and CFb SGs increases as well. We think that this is due to state lattices having large highway dimensions, as we discuss at the end of this section. Interestingly, the BD25, F25, and CF25 SGs constructed by pruning and growing are (almost) identical, which we think is due to the following reason: Edges that correspond to turns on the Unicycle-Arena2 benchmark have a length of 17.35, which means that any two successive turns correspond to a unique shortest path with length $34.7 > 25$, requiring its middle vertex to be a subgoal in any BD25-, F25-, and CF25-SPC.

- **Number of subgoals, pruning vs. growing:** The BD75 SG constructed by the pruning variant has 20.46% fewer subgoals than the one constructed by the growing

variant, since the pruning variant guarantees the minimality of the set of subgoals. The F75 and CF75 SGs constructed by the pruning variant have fewer subgoals than the F150 and CF150 SGs constructed by the growing variant. These results suggest that the heuristic that we use in the growing variant for identifying subgoals could be improved.

- **Preprocessing times:** The time to construct BDb, Fb, and CFb SGs increases as the reachability bound b increases, for both the pruning and growing variants, where the increase for the pruning variant is significantly larger. For instance, the pruning variant requires roughly four and a half hours of preprocessing time to construct a BD75 SG, whereas the growing variant requires only 69 seconds.

- **Query times:** All SGs in our experiments use a significant percentage of the vertices of G as subgoals ($\sim$24.6% at the lowest) and, for reachability bounds greater than 25, have more edges than G. Therefore, answering queries using them is slow, at best 2.07 times faster than A* searches on G.

- **Search times:** The search times make up at least $\sim$94.8% of the query times, significantly reducing the contribution of the more efficient connection and refinement algorithms for Fb- and CFb-reachability to the query times.

To summarize, our current construction methods fail to produce small SGs within a reasonable amount of time. We conjecture that this is in part due to state lattices having large highway dimensions. Recall that, on a graph with maximum edge length m and highway dimension h, it is possible to construct a $(b - 2m, h)$ sparse BDb-SPC for $b > 3m$ (Theorem 3.13). That is, there exists a BDb-SPC S such that, for every $u \in V$, the number of vertices $v \in S$ with $\max(d(u, v), d(v, u)) \leq 2b - 4m$ is at most h. For Unicycle benchmarks, $m \approx 17.35$, and this result applies for $b = 75 \geq 3 \cdot 17.35$. If G has a small highway dimension, we would expect a small number of subgoals within distance $2 \cdot 75 - 4 \cdot 17.35 \approx 80$ from any given vertex $n \in V$. However, this does not seem to be the case for the BD75 SGs that we generate via pruning, whose sets of subgoals S are guaranteed to be minimal BD75-SPC. We note that the minimality of S does not necessarily mean that it is the minimum set of subgoals, which is why our results (which construct minimal but not minimum R-SPCs) cannot be used to conclusively show that state lattices have large highway dimensions. Therefore, based on our results, we can only *conjecture* that state lattices, in general, may have large highway dimensions.

4.6.6 Strongly-Connected Subgoal Graphs

In this section, we compare answering queries using BDb, Fb, and CFb SCSGs with respect to their query times and path suboptimalities.

4.6.6.1 Setup

As we have shown in the previous section, our current construction methods fail to produce small SGs on state lattices, which we think is due to their large highway dimensions.

To get around this limitation, we now experiment with R SCSGs (Section 3.5), whose subgoals do not necessarily form R-SPCs.

We use a setup similar to the one that we have used for our experiments with SGs (Section 4.6.5). Namely, we omit the conservative and stall-on-demand variants of BDb-reachability in our comparison, report the amount of memory required to store SGs using 4 bytes per edge for the distance variant of Fb-reachability and 8 bytes per edge for the other reachability relations, and report the amount of memory required to store freespace information without exploiting symmetries among primitives. When constructing SCSGs, we first use Algorithm 8 to identify the access subgoals (such that any start and goal vertex can be connected to at least one access subgoal using R-Connect), then use Algorithm 9 to strongly-connect the access subgoals using only R-reachable edges, adding additional subgoals as necessary. In order to find short paths in practice, we add edges between all direct-R-reachable subgoals in SCSGs, as well as use all direct-R-reachable edges during the connection phases of queries. Similarly to constructing SGs, to construct the edges of SCSGs, we perform R-Connect from every subgoal and then eliminate any redundant edges, if necessary (for Fb- and CFb-reachability).

We are ultimately interested in the query-time/path-suboptimality trade-offs (trade-offs, for short) of SCSGs constructed with respect to different reachability relations. We treat the reachability bound b as a parameter that can be fine-tuned differently for different reachability relations, resulting in different trade-offs.[3] We therefore use the following scheme for comparing the trade-offs of SCSGs constructed with respect to different reachability relations: First, for each reachability relation R, we construct a set T_R of trade-offs, by varying the reachability bound b from 25 to 150 in increments of 25. We then eliminate those trade-offs t from T_R that are dominated by at least one other trade-off t' in T_R (that is t' has both a shorter query time and a smaller path suboptimality than t), essentially removing trade-offs that correspond to a "bad tuning" of b. The remaining set of trade-offs P_R forms the "query-time/path-suboptimality Pareto frontier" (Pareto frontier, for short) for SCSGs constructed with respect to R. We say that a Pareto frontier P_R dominates a trade-off t if and only if at least one trade-off $t' \in P_R$ dominates t. We say that a Pareto frontier P_R dominates a Pareto frontier $P_{R'}$ if and only if P_R dominates all trade-offs $t \in P_{R'}$. We say that a Pareto frontier P_R is non-dominated if and only if there is at least one $t \in P_R$ that is not dominated by any other Pareto frontier $P_{R'}$.

We also experiment with weighted A* searches on G, using the Euclidean distance (Euc) and *2D grid distance* (2D) heuristics, and similarly calculate Pareto frontiers for them by varying the suboptimality bound w from 1 to 3 in increments of 0.25. The 2D grid distance from a state $s = (x_s, y_s, \theta_s)$ to a state $t = (x_t, y_t, \theta_t)$ is the (x_s, y_s)-(x_t, y_t) distance on the 8-neighbor grid center graph constructed on the underlying grid of the state lattice, and is not necessarily admissible (Butzke, Sapkota, Prasad, MacAllister, & Likhachev, 2014). We compute 2D grid distances "on demand" during searches. That is, in the beginning of a (weighted) A* search on the state lattice, we also start a search on the corresponding 8-neighbor grid graph, backwards from the goal. When we want to

[3] To clarify, we do not claim that the reachability bound b can be used to trade off shorter query-times and smaller path-suboptimalities, but only remark that different values of b may result in different query-time/path-suboptimality trade-offs.

look-up the n-t 2D grid distance during the A* search on the state lattice, the A* search over the 8-neighbor grid graph expands vertices until the vertex that corresponds to n is expanded, and then return its g value. The time to compute 2D grid distances is included in our reported search and query times.

4.6.6.2 Results

We first compare answering queries using BDb, Fb, and CFb SCSGs on the Unicycle-Arena2 benchmark with respect to their query times and path suboptimalities, both to each other and to answering queries using BDb, Fb, and CFb SGs (Section 4.6.5). We then compare answering queries using BDb, Fb, and CFb SCSGs across the four benchmarks with respect to their query times and path suboptimalities, and focus on their differences across the different benchmarks.

Table 4.4 reports the time needed to construct BDb, Fb, and CFb SCSGs, the numbers of vertices and edges in these SCSGs relative to G, the memory required to store these edges and freespace information, the connection, search, and refinement times of queries answered using these SCSGs, and the average and maximum suboptimalities of the paths found by these queries, on the Unicycle-Arena2 benchmark. Figure 4.21 shows the trends from Table 4.4 as the reachability bound varies, as well as the query-time/path-suboptimality trade-off when answering queries using BDb, Fb, and CFb SCSGs, and weighted A* searches on G.

We make the following observations:

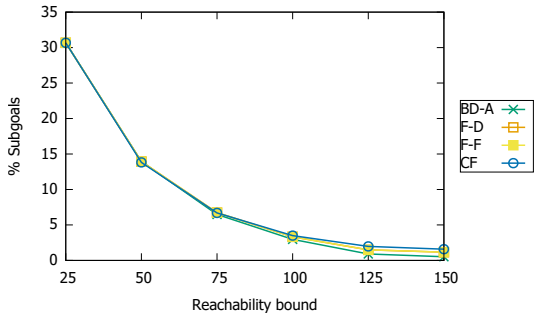
- **Preprocessing times:** R SCSGs can be constructed significantly faster than R SGs for the same R, since our construction algorithm (Section 3.5.3) only performs three (modified) R-connect operations for each identified subgoal n, namely once forward and once backward from n to mark the vertices of G that can R-connect to n in the forward and backward directions, and once more afterward to identify the edges of the SG. For instance, a BD150 SCSG can be constructed 58 times faster than a BD150 SG constructed by growing.

- **Number of subgoals:** R SCSGs have significantly fewer subgoals than R SGs for the same R, since there are fewer constraints on the placement of their subgoals (that is, their subgoals do not have to form R-SPCs). For instance, the BD25 SCSG has 64.9% fewer subgoals than the BD25 SG constructed by pruning, and the BD75 SCSG has 80.6% fewer subgoals than the BD75 SG constructed by pruning. We observe similar trends in the number of subgoals in R SCSGs as we have observed for R SGs as the reachability relation and reachability bound vary. Namely, BDb SCSGs have fewer subgoals than Fb and CFb SCSGs for the same b, and, as b increases, the number of subgoals decreases for all R SCSGs. Furthermore, similar to the results in Section 4.6.5, the BD25, F25, and CF25 SCSGs are identical, presumably due to all edges in the Unicycle-Arena2 benchmark having a length of 17.35 and, consequently, the length of any sequence of two turns exceeding the reachability bound.

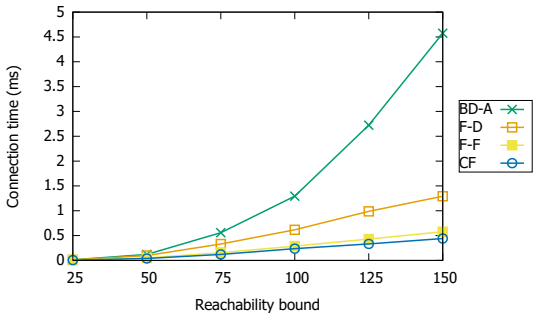
- **Memory requirements:** The memory required to store freespace information is the same for answering queries using R SCSGs and R SGs, for the same R. However,

		Prep.	Memory (MB)		Size vs G			Query time (ms)			Speed	Suboptimality	
R	b	time (s)	Freespace	Edges	Access	Subgoals	Edges	Connect	Search	Refine	up	Avg	Max
BD-A	25	1.45	0.00	3.48	15.37%	30.67%	36.07%	0.011	9.842	0.120	3.37	1.373	5.779
	50	5.98	0.00	6.27	8.36%	13.93%	65.08%	0.120	4.354	0.309	7.02	1.329	4.675
	75	11.89	0.00	5.93	3.90%	6.48%	61.53%	0.558	2.042	0.720	10.11	1.287	3.781
	100	15.31	0.00	3.48	1.91%	2.96%	36.13%	1.294	0.708	1.169	10.59	1.313	4.476
	125	9.72	0.00	0.61	0.60%	0.91%	6.29%	2.723	0.217	2.131	6.62	1.467	4.220
	150	9.12	0.00	0.33	0.35%	0.52%	3.37%	4.575	0.118	2.722	4.53	1.495	3.934
F-D	25	1.29	0.37	1.74	15.37%	30.67%	36.07%	0.011	9.735	0.028	3.43	1.373	5.779
	50	4.42	2.02	3.10	8.39%	13.93%	64.43%	0.099	4.110	0.023	7.93	1.309	4.675
	75	7.19	5.84	2.96	4.08%	6.74%	61.37%	0.330	2.031	0.020	14.10	1.213	4.931
	100	7.29	12.59	1.61	2.13%	3.32%	33.50%	0.616	0.839	0.017	22.80	1.181	3.292
	125	5.03	23.72	0.45	0.91%	1.48%	9.30%	0.988	0.366	0.016	24.49	1.169	2.788
	150	4.62	39.84	0.33	0.67%	1.16%	6.91%	1.292	0.296	0.016	20.94	1.154	2.566
F-F	25	1.29	0.12	3.48	15.37%	30.67%	36.07%	0.007	9.827	0.014	3.41	1.373	5.779
	50	4.42	0.71	6.20	8.39%	13.93%	64.43%	0.047	3.998	0.012	8.27	1.309	4.675
	75	7.19	2.09	5.92	4.08%	6.74%	61.37%	0.155	2.018	0.010	15.38	1.213	4.931
	100	7.29	4.61	3.22	2.13%	3.32%	33.50%	0.288	0.825	0.009	29.93	1.181	3.292
	125	5.03	8.76	0.90	0.91%	1.48%	9.30%	0.429	0.338	0.008	43.27	1.169	2.788
	150	4.62	14.77	0.66	0.67%	1.16%	6.91%	0.577	0.282	0.009	38.70	1.154	2.566
CF	25	0.94	0.10	3.48	15.37%	30.67%	36.07%	0.006	10.120	0.007	3.31	1.373	5.779
	50	2.21	0.58	6.11	8.40%	13.82%	63.44%	0.039	3.977	0.007	8.34	1.296	4.675
	75	3.17	1.72	5.77	4.17%	6.70%	59.93%	0.120	1.875	0.006	16.77	1.188	4.914
	100	3.37	3.82	3.49	2.31%	3.49%	36.19%	0.236	0.952	0.006	28.10	1.165	3.109
	125	3.15	7.28	1.50	1.29%	1.97%	15.57%	0.331	0.513	0.006	39.47	1.153	2.430
	150	4.31	12.28	1.16	0.99%	1.58%	12.02%	0.440	0.430	0.006	38.29	1.143	2.370

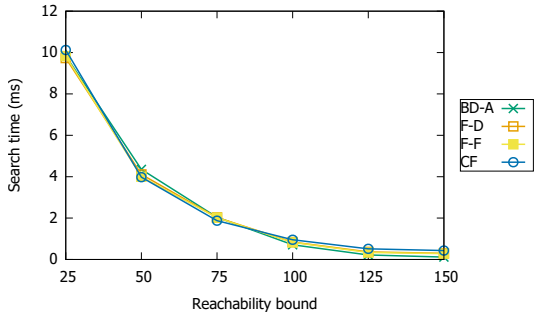
Table 4.4: Comparison of SCSGs on the Unicycle-Arena2 benchmark.



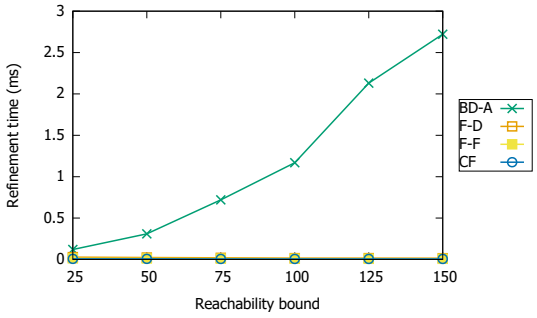
(a) Percentage of vertices selected as subgoals.



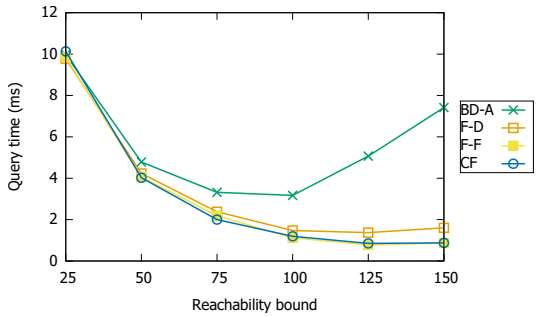
(b) Connection time.



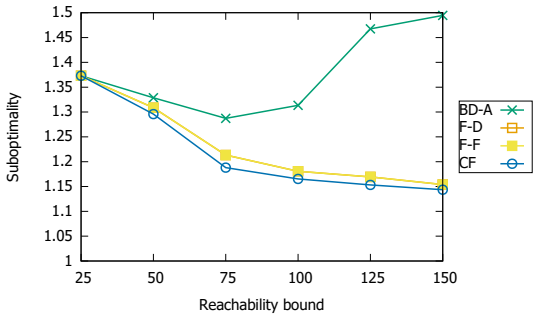
(c) Search time.



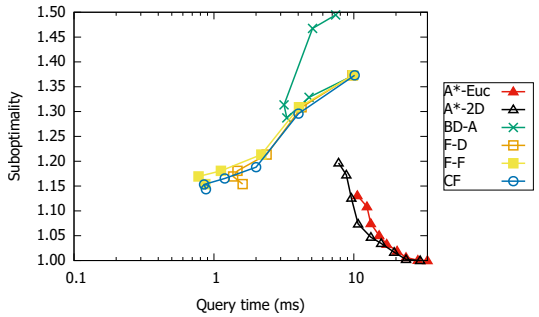
(d) Refine time.



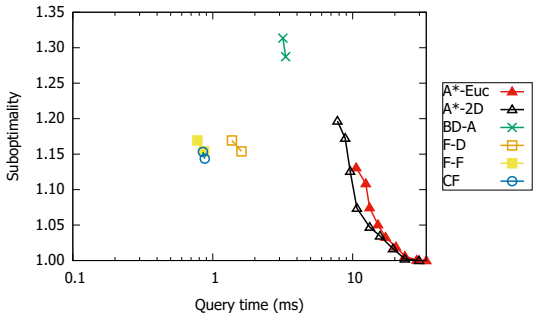
(e) Query time.



(f) Path suboptimality.



(g) Query-time/path-suboptimality trade-off.



(h) Query-time/path-suboptimality trade-off, Pareto frontiers.

Figure 4.21: Comparison of SCSGs on the Unicycle-Arena2 benchmark.

since R SCSGs are typically smaller than R SGs, the memory required to store their edges is smaller as well.

- **Connection times:** As the reachability bound increases, the connection times increase for all R-connect algorithms, and increase faster for BDb-Connect than the Fb-Connect variants and CFb-Connect. As discussed in Section 4.6.4, there are several factors that contribute to this result. As the reachability bound increases: 1) The number of BDb-reachable vertices increases faster than the number of Fb- and CFb-reachable vertices. 2) The average expansion time of BDb-Connect (variants) increases due to more expensive insertion and removal operations in OPEN, whereas the average expansion times of the Fb-Connect variants and CFb-Connect are unaffected. 3) Since BDb SCSGs have fewer subgoals than Fb and CFb SCSGs, a higher percentage of R-reachable vertices are direct-R-reachable (and, therefore, should be expanded by R-connect) for BDb-reachability, compared to Fb- and CFb-reachability. Similar to our results in Section 4.6.4, CFb-Connect has the shortest connection times, followed by the flag variant of Fb-Connect, since these variants process only a small number of successors per expansion and since CFb-Connect avoids duplicate detection.

- **Search times:** As the reachability bound increases, the search times decrease since the number of subgoals decrease for all R SCSGs. They are the smallest for BDb SCSGs, which contain fewer subgoals than Fb and CFb SCSGs for the same b.

- **Refinement times:** As the reachability bound increases, BDb-Refine times increase, whereas Fb- and CFb-Refine times remain mostly unchanged. This can be explained as follows. As the reachability bound increases, R SCSGs have fewer subgoals that need to be (and can be) connected through longer edges. As a result, as the reachability bound increases, paths on R SCSGs tend to have fewer edges that are longer. This negatively affects BDb-Refine times, since BDb-Refine is implemented as an A* search, and a single A* search between vertices that are far apart typically requires more time than multiple A* searches between vertices that are close together. CFb-Refine times remain mostly unaffected since they depend only on the number of edges on the resulting path. Similarly, Fb-Refine times are mostly unaffected since, although Fb-Refine performs searches, it typically operates similarly to CFb-Refine due to state lattices not having too many symmetric shortest paths. Interestingly, as b increases, CFb- and Fb-refine times can decrease, since the paths found on query CFb and Fb SCSGs become shorter, as we discuss later, when discussing path suboptimalities.

- **Query times:** As the reachability bound increases, the query times may increase or decrease, depending on how the fast connection and refinement times increase and the search times decrease. Our results show that query times typically first decrease and then increase, and the bound for which the query times are shortest varies with the reachability relations and connection and refinement algorithms. When using BDb-reachability (and the aggressive variant of BDb-Connect), the shortest query times are achieved at reachability bound 100. When using Fb-reachability (and both the flag and distance variants of Fb-Connect and Fb-Refine) or CFb-reachability, the

shortest query times are achieved at reachability bound 125. The shortest query times are achieved with a smaller reachability bound when using BDb-reachability because: 1) The connection and refinement times increase faster when using BDb-reachability rather than Fb- and CFb-reachability. 2) The search times decrease faster when using BDb-reachability rather than Fb- and CFb-reachability. Figures 4.21b, 4.21c, 4.21d, and 4.21e show how the reachability bound affects the connection, search, refinement, and query times.

- **Path suboptimality:** Paths found on BDb SCSGs are longer than those on Fb SCSGs, which in turn are longer than those on CFb SCSGs, for the same b. As the reachability bound increases, paths found on BDb SCSGs first get shorter and then longer, and paths found on Fb and CFb SCSGs get shorter. We suspect that there are several factors that contribute to these results: 1) As mentioned during our discussion of the refinement times, as the reachability bound increases, the paths found over R SCSGs typically consist of shorter sequences of longer edges. Since edges on these paths correspond to shortest paths on G, paths consisting of shorter sequences of longer edges typically have longer "stretches" that are "optimal" (that is, correspond to shortest paths on G), compared to paths consisting of longer sequences of shorter edges. Therefore, this factor contributes to paths found on R SCSGs getting shorter (since they have longer stretches that are optimal) as the reachability bound increases. 2) Our algorithm for constructing R SCSGs aims to find small R SCSGs with the highest priority, and reduce path lengths only with secondary priority, by adding more edges to R SCSGs *after* determining the set of subgoals. Fewer subgoals in R SCSGs mean that fewer possible "turning points" are considered, which can result more frequently in sequences of edges forming "non-taut" turns. Therefore, this factor contributes to paths found over R SCSGs getting longer as the reachability bound increases. We suspect that these two factors contribute to differences in path lengths found over R SCSGs for different reachability relations and reachability bounds. For instance, as the reachability bound increases, the number of subgoals of BDb SCSGs decreases faster than the number of subgoals in Fb and CFb SCSGs, which could be the reason why the lengths of paths found over BDb SCSGs increase as the reachability bound increases. As indicated in Section 3.5, a more detailed analysis of path suboptimalities on SCSGs is beyond the scope of this dissertation.

- **Query-time/path-suboptimality trade-off:** The (query-time/path-suboptimality) Pareto frontier for answering queries using Fb SCSGs and the flag variants of Fb-Connect and Fb-Refine dominate the Pareto frontier for answering queries using Fb SCSGs and the distance variants of Fb-Connect and Fb-Refine, since both algorithms have the same path-suboptimalities and search times, but the flag variants of Fb-Connect and Fb-Refine are faster than the distance variants. Throughout this discussion, we therefore assume that the flag variants of Fb-Connect and Fb-Refine are used when answering queries using Fb SCSGs (similar to how we assume that the aggressive variant of BDb-Connect is used when answering queries using BDb SCSGs).

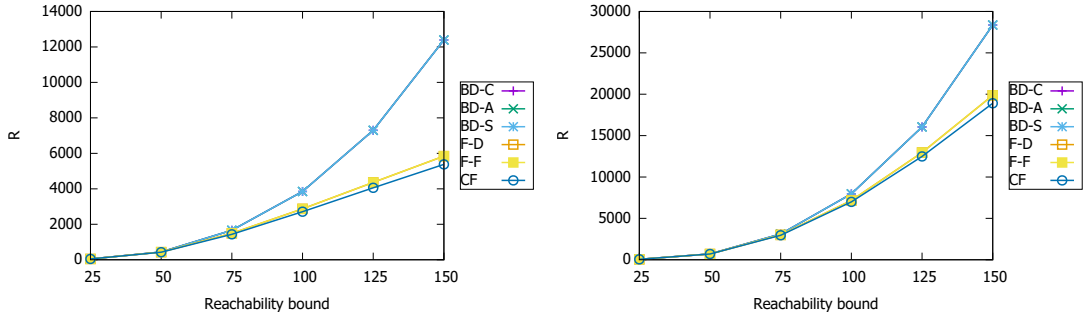
The Pareto frontiers for answering queries using Fb and CFb SCSGs dominate the Pareto frontier for answering queries using BDb SCSGs, since answering queries using Fb and CFb SCSGs is typically faster and finds shorter paths. The Pareto frontiers of these two algorithms are very similar and do not dominate each other, with answering queries using CFb SCSGs having slightly longer query times but smaller path suboptimalities. Furthermore, the Pareto frontiers of these two algorithms are non-dominated by the Pareto frontier when answering queries using weighted A* searches (with either the Euclidean or the 2D grid distance heuristics) on G, since they have shorter query times; but also do not dominate the Pareto frontier when answering queries using weighted A* searches on G, since they find longer paths.

We now analyze the results over all benchmarks, and focus on their differences across different benchmarks.

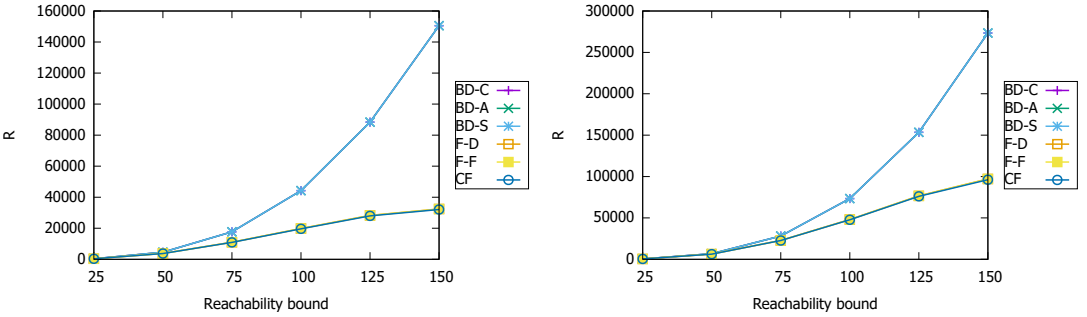
- **Number of R-reachable vertices, percentage of subgoals, and connection and search times:** Figures 4.22, 4.23, 4.24, and 4.25, respectively, report the number of R-reachable vertices (also reported in Figure 4.12), the number of vertices of R SCSGs relative to G, and the connection and search times of queries answered using R SCSGs across the four benchmarks. The refinement times are not reported since they are small compared to the connection and search times. We observe that the differences in the number of R-reachable vertices across the four benchmarks (as discussed in Section 4.6.4.2) result in differences in the number of vertices of R SCSGs relative to G and in the connection and search times of queries answered using R SCSGs across the four benchmarks.

- **Search times relative to query times:** Figure 4.26 reports the percentage of query times spent in the search phase when answering queries using R SCSGs, across the four benchmarks. We observe that, on all benchmarks and for all reachability relations, the percentage of query times spent in the search phase decreases as the reachability bound increases, since the search times decrease but the connection and refinement times (not reported) increase. The percentage of query times spent in the search phase decreases faster on Urban and Arena2 benchmarks compared to the Unicycle and Aurora benchmarks. The reason for this is two-fold: 1) Connection (and refinement) can be considered to be "local" operations since they are performed up to a bound, whereas search can be considered to be a "global" operation since it is performed on SCSGs that span the whole graph. As the grid size increases (that is, the Aurora benchmarks compared to the Arena2 benchmarks), the time to perform the "global" operation relative to the "local" operations increases. 2) The number of R-reachable vertices is larger on the Urban benchmarks than the Unicycle benchmarks, which results in longer connection times and smaller R SCSGs that can be searched faster.

- **Speed-up over weighted A* searches:** Figure 4.27 shows the speed-up achieved by answering queries using R SCSGs over weighted A* searches on G using the Euclidean distance heuristic, across the four benchmarks. We observe that how the

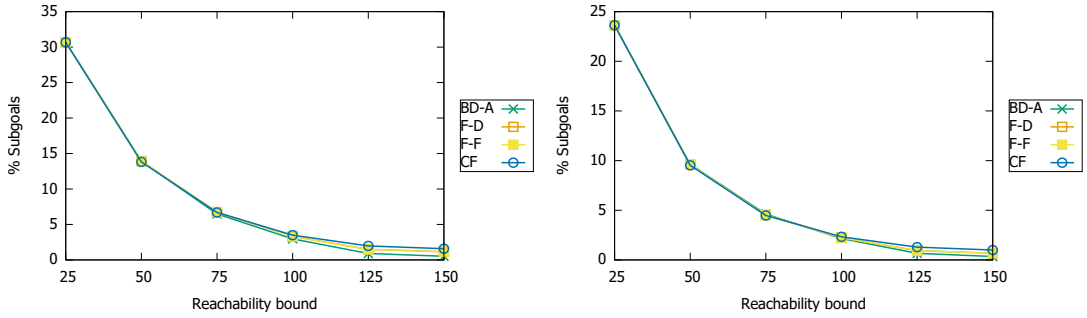


(a) Unicycle-Arena2. (b) Unicycle-Aurora.

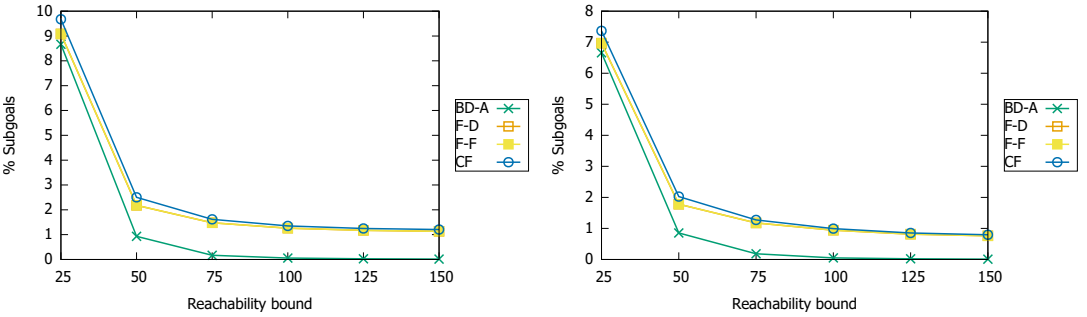


(c) Urban-Arena2. (d) Urban-Aurora.

Figure 4.22: Number of R-reachable vertices.

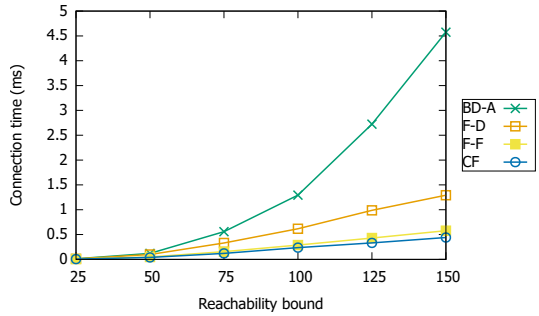


(a) Unicycle-Arena2. (b) Unicycle-Aurora.

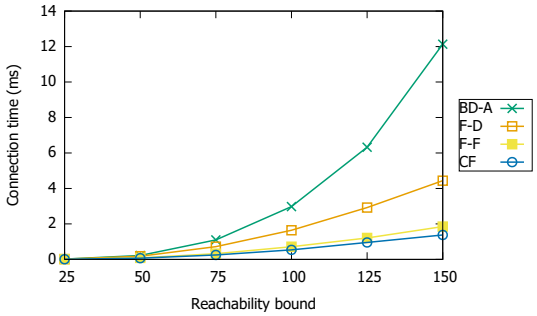


(c) Urban-Arena2. (d) Urban-Aurora.

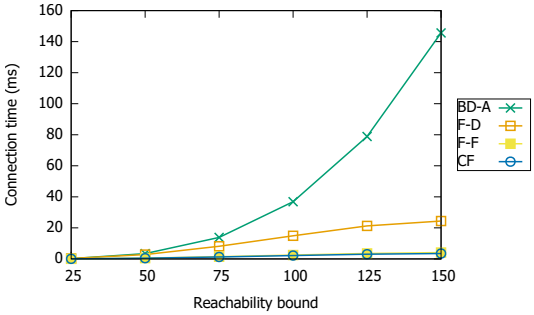
Figure 4.23: Percentage of subgoals in R SCSGs (relative to G).



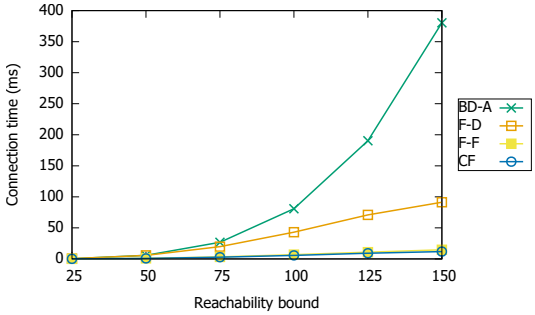
(a) Unicycle-Arena2.



(b) Unicycle-Aurora.

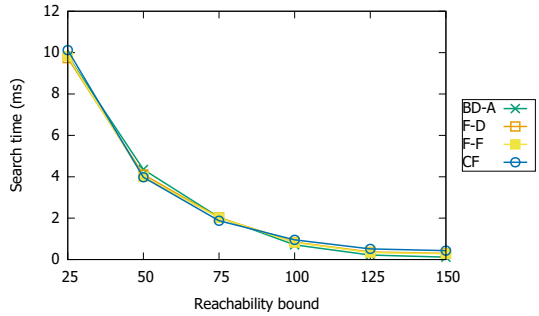


(c) Urban-Arena2.

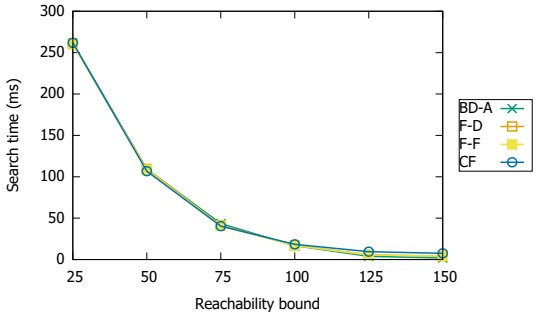


(d) Urban-Aurora.

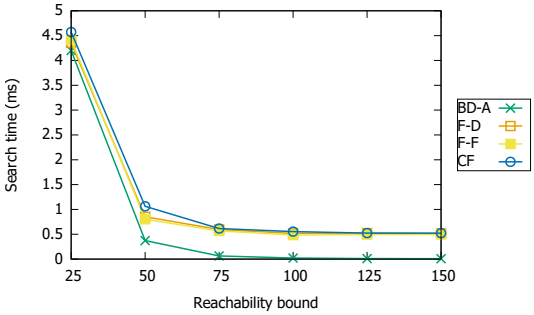
Figure 4.24: Connection times when answering queries using R SCSGs.



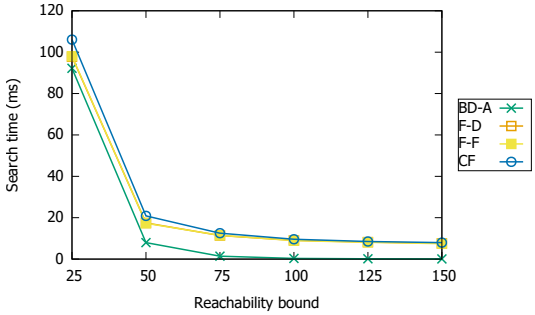
(a) Unicycle-Arena2.



(b) Unicycle-Aurora.

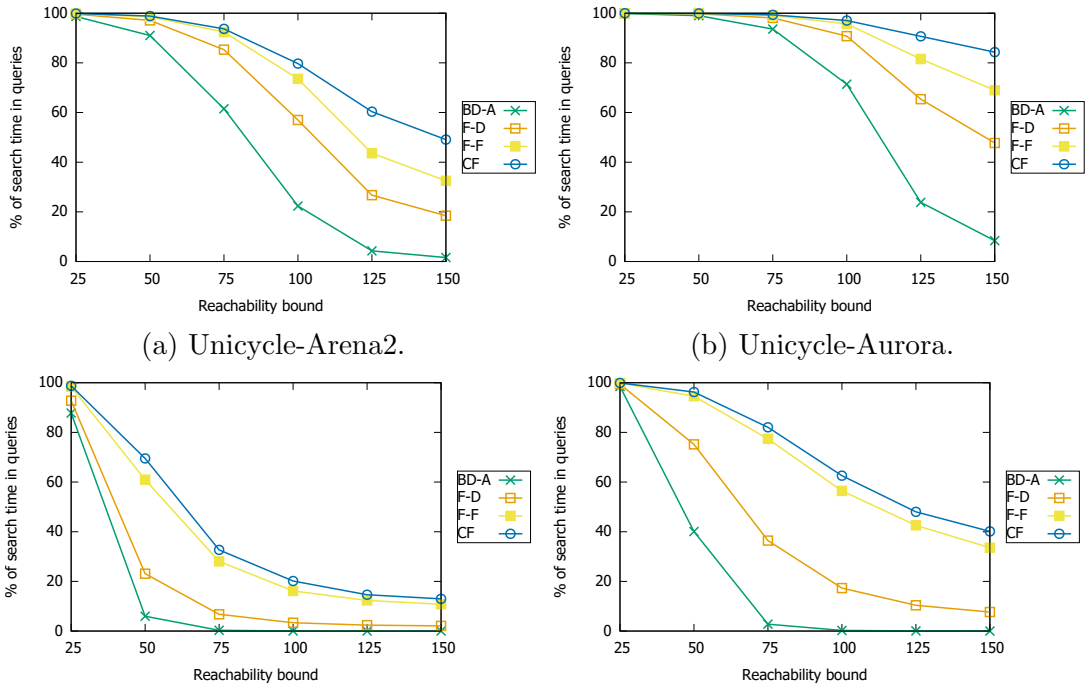


(c) Urban-Arena2.



(d) Urban-Aurora.

Figure 4.25: Search times when answering queries using R SCSGs.



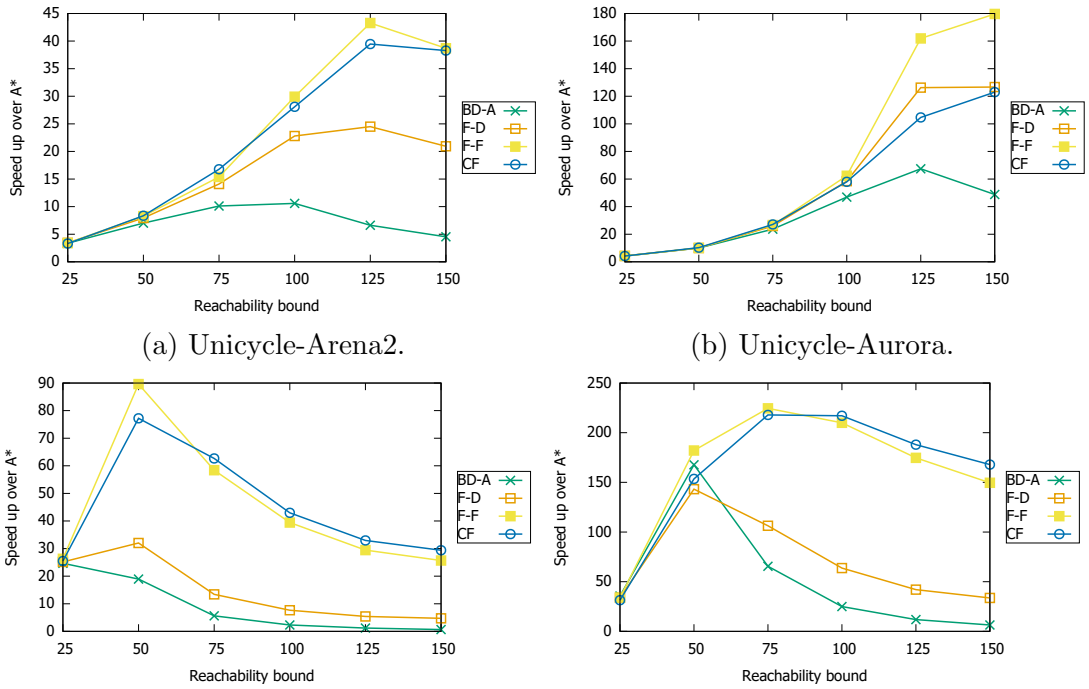
(a) Unicycle-Arena2.

(b) Unicycle-Aurora.

(c) Urban-Arena2.

(d) Urban-Aurora.

Figure 4.26: Percentage of query times spent in the search phase when answering queries using SCSGs.



(a) Unicycle-Arena2.

(b) Unicycle-Aurora.

(c) Urban-Arena2.

(d) Urban-Aurora.

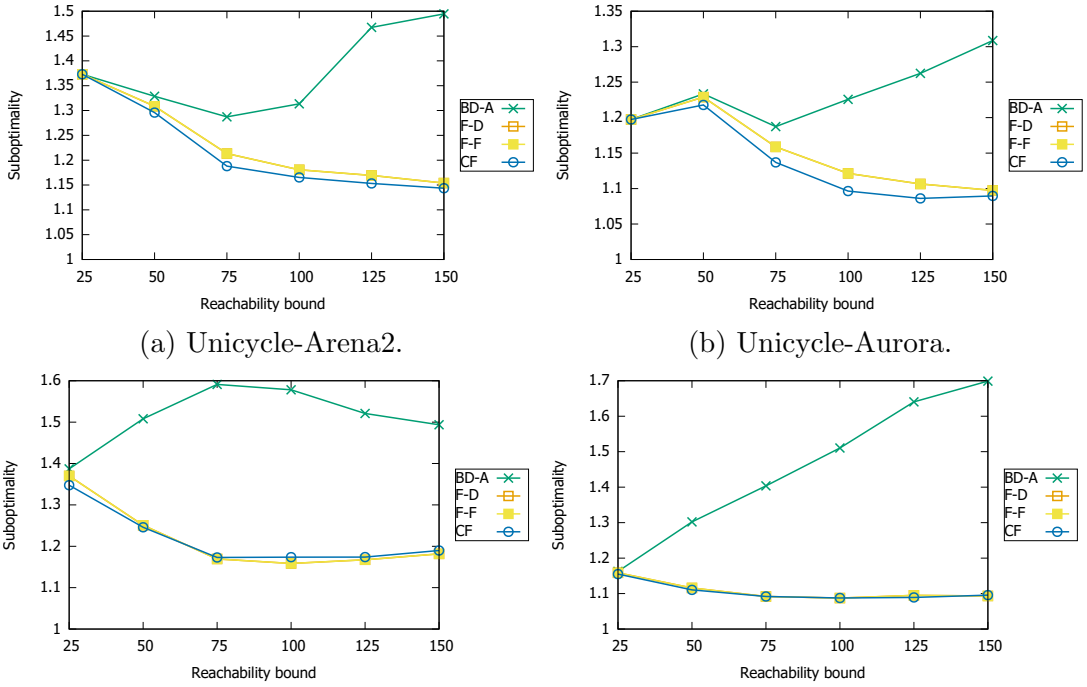
Figure 4.27: Speed-up over A* searches when answering queries using SCSGs.

percentage of query times spent in the search phase varies affects the reachability bound for which the highest speed-ups are achieved over weighted A* searches. On the Unicycle-Aurora benchmark, where the percentage of query times spent in the search phase decreases slowly as the reachability bound increases, the reachability bounds for which the highest speed-ups are achieved are high, namely 125 for BDb SCSGs and 150 for the other R SCSGs. On the Urban-Arena2 benchmark, where the percentage of query times spent in the search phase decreases quickly, the reachability bounds for which the highest speed-ups are achieved are low, namely 25 for BDb SCSGs and 50 for other R SCSGs.

- **Path suboptimalities:** Figure 4.27 shows the suboptimalities of the paths found when answering queries using R SCSGs, across the four benchmarks. We observe that the paths found when using BDb SCSGs have higher suboptimalities on the Urban benchmarks than the Unicycle benchmarks, which we think is due to BDb SCSGs having a smaller percentage of the vertices of G as subgoals on the Urban benchmarks. We observe that the paths found when using Fb and CFb SCSGs have lower suboptimalities on the Aurora benchmarks than the Arena2 benchmarks, which we think is due to the Aurora benchmarks containing larger contiguous regions of unblocked cells than the Arena2 benchmarks. As indicated in Section 3.5, a more detailed analysis of the suboptimalities of the paths found when using SCSGs is beyond the scope of this dissertation.

- **Query-time/path-suboptimality trade-off:** Figure 4.29 shows the (query-time/path-suboptimality) Pareto frontiers when answering queries using R SCSGs or weighted A* searches, across the four benchmarks. The dominance relations that we have observed on the Unicycle-Arena2 benchmark also hold on all other benchmarks. Namely, the Pareto frontiers when answering queries using CFb SCSGs or Fb SCSGs with the flag variants of performing connection and refinement dominate the Pareto frontiers when answering queries using BDb SCSGs or Fb SCSGs with the distance variants of performing connection and refinement, and are non-dominated by the Pareto frontiers when answering queries using weighted A* searches with the Euclidean or 2D grid-distance heuristics.

To summarize, R SCSGs can be constructed faster than R SGs, are smaller, and can be used to answer queries faster. Although R SCSGs cannot be used to find shortest paths, we observe that, in practice, short paths (10–15% longer than shortest paths) can be found by using Fb and CFb SCSGs. As the reachability bound increases, the connection and refinement times when answering queries using BDb SCSGs quickly increase, whereas the search times decrease. In comparison, as the reachability bound increases, the increase in connection times when answering queries using Fb and CFb SCSGs is much smaller, and the refinement times do not increase. As a result, the (query-time/path-suboptimality) Pareto frontiers (generated by varying the reachability bound) when answering queries using Fb and CFb SCSGs dominate the Pareto frontier when answering queries using BDb SCSGs.



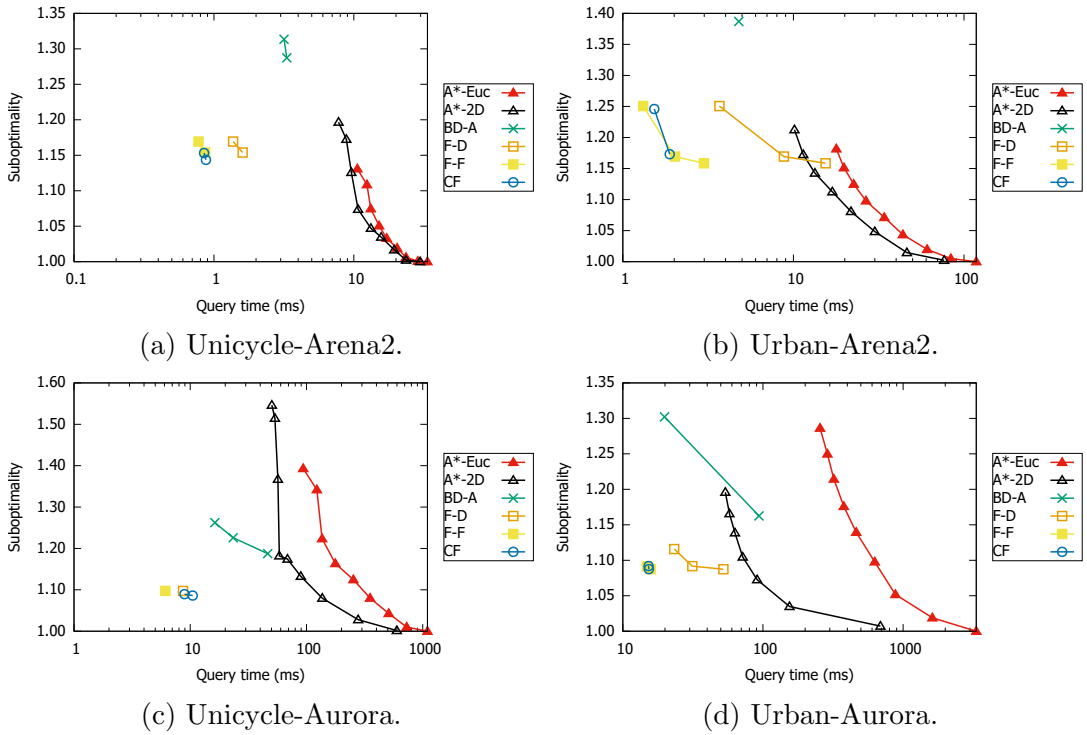
(a) Unicycle-Arena2.

(b) Unicycle-Aurora.

(c) Urban-Arena2.

(d) Urban-Aurora.

Figure 4.28: Path suboptimality when answering queries using SCSGs.



(a) Unicycle-Arena2.

(b) Urban-Arena2.

(c) Unicycle-Aurora.

(d) Urban-Aurora.

Figure 4.29: Query-time/path-length Pareto frontiers when answering queries using SCSGs or weighted A* searches.

4.7 Conclusions and Future Work

In this chapter, we have applied the subgoal graph framework to state lattices, by using freespace-reachability and canonical-freespace-reachability as reachability relations to capture the freespace structure of state lattices, and developing efficient connection and refinement algorithms that exploit this structure. Specifically, we have characterized the freespace structure of state lattices as the *translation invariance of freespace distances and freespace-canonical paths*, and showed that it can be exploited to efficiently compute and compactly store freespace information, such as pairwise distances or shortest path trees on freespace state lattices. We have introduced freespace-reachability and canonical-freespace-reachability as reachability relations to distinguish those pairs of vertices on state lattices between which the freespace information is accurate, and developed connection and refinement algorithms for these reachability relations, that use freespace information to efficiently explore freespace-shortest and freespace-canonical paths, respectively. We have experimentally demonstrated that answering queries using freespace-reachability or canonical-freespace-reachability strongly-connected subgoal graphs achieves a *dominating query-time/path-suboptimality trade-off* compared to answering queries using bounded-distance-reachability strongly-connected subgoal graphs, and a *non-dominated query-time/path-suboptimality* trade-off compared to answering queries using weighted A* searches. These results validate the hypothesis of this dissertation that one can develop preprocessing-based path-planning algorithms for state lattices that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.

Recall our discussion from Section 3.1 that the "benefit" of using overlay (subgoal) graphs for answering queries is that searches over them are faster, while the "cost" of using them is that one has to perform connection and refinement. Although we were not able to achieve short query times by using subgoal graphs, possibly due to the large highway dimensions of state lattices, we were able to do so using strongly-connected subgoal graphs (with the additional "cost" of sacrificing optimality). We have observed that, in order to accrue higher "benefits" by constructing smaller strongly-connected subgoal graphs that can be searched faster, we also pay higher "costs" of increased connection and refinement times, and achieve the shortest query times when we find a good balance between the two, by adjusting the reachability bound. Our results from Section 4.6.4 suggest that the "cost" of connection, as measured by the "DR-rate", can be significantly smaller for freespace-reachability and canonical-freespace-reachability, compared to bounded-distance-reachability. Our results from Section 4.6.6 suggest that the smaller "cost" of connection (and refinement) is the reason why queries can be answered faster by using freespace-reachability or canonical-freespace-reachability strongly-connected subgoal graphs than bounded-distance-reachability strongly-connected subgoal graphs.

As future work, we consider several directions for improving the answering of queries using freespace-reachability or canonical-freespace-reachability strongly-connected subgoal graphs. First, as we have discussed in Section 3.5.3, we think that our current algorithm for constructing strongly-connected subgoal graphs can be improved. Second, we think that our definition of strongly-connected subgoal graphs can be extended to use two different reachability relations R_c and R_r, where R_c is used for connection and

R_r is used for refinement. That is, access subgoals can be identified with respect to R_c, but they can be strongly-connected using R_r-reachable edges rather than R_c-reachable edges. Using a larger reachability bound for R_r could result in fewer additional subgoals being necessary to strongly-connect access subgoals, which in turn could result in smaller strongly-connected subgoal graphs that can be searched faster. Our experimental results from Section 4.6.6 show that the refinement times when answering queries using freespace-reachability or canonical-freespace-reachability strongly-connected subgoal graphs are very short, and do not increase as the reachability bound increases. Therefore, increasing the reachability bound *only* for refinement does not increase refinement times, and can decrease search (and thus query) times. Finally, we think that it would be interesting to develop a version of strongly-connected subgoal graphs that can be used to find bounded-suboptimal paths.

Chapter 5

Exploiting the Freespace Structure of Grid Graphs

In this chapter, we apply the subgoal graph framework to grid graphs by using safe-freespace-reachability as reachability relation, discuss the similarities and differences of subgoal graphs with jump-point search, and augment contraction hierarchies with reachability relations in various ways. Specifically, we show that the freespace structure (Octile property) of grid graphs allows for the construction of safe-freespace-reachability subgoal graphs by using only the convex corners of blocked cells, introduce a connection algorithm for safe-freespace-reachability that scans the grid efficiently by using clearance values, and prove that this algorithm can be used to construct subgoal graphs in time linear in the size of the underlying grid. We show that jump-point search can be understood as a search on a jump-point graph, which is a freespace-reachability subgoal graph on the direction-extended canonical grid graph. We experimentally demonstrate that answering queries using contraction hierarchies on subgoal graphs achieves a *dominating query-time/memory trade-off* compared to answering queries using contraction hierarchies on G or jump-point graphs. Our results further suggest (through interpolation) that answering queries using contraction hierarchies on subgoal graphs and performing freespace-based refinement is *2.34 times faster* than single-row compression, the fastest entry in the Grid-Based Path-Planning Competition, while requiring *139.06 times less memory*. These results validate the hypothesis of this dissertation that one can develop preprocessing-based path-planning algorithms for grid graphs that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.

This chapter is organized as follows. In Section 5.1, we provide a detailed summary of the main ideas that we use in this chapter. In Section 5.2, we formally define grid graphs and introduce notation that we use throughout this chapter. In Section 5.3, we characterize the Octile property of grid graphs, introduce safe-freespace-reachability as a reachability relation, prove that safe-freespace-reachability subgoal graphs can be constructed on grid graphs by using convex corners of blocked cells as subgoals, introduce a connection algorithm for safe-freespace-reachability, and prove that this algorithm can be used to construct safe-freespace-reachability subgoal graphs on grid graphs in time linear in the size of the underlying grid. In Section 5.4, we show that jump-point search can be understood as a search on a freespace-reachability subgoal graph constructed on a direction-extended canonical grid graph. In Section 5.5, we introduce various augmentations of contraction hierarchies with reachability relations, perform an extensive

experimental evaluation of these variants, and compare our results to results from the Grid-Based Path-Planning Competition. Finally, in Section 5.6, we summarize our results.

5.1 Introduction

In Chapter 4, we showed that the freespace structure of state lattices can be exploited to achieve non-dominated query-time/path-suboptimality trade-offs by: 1) constructing Fb or CFb strongly connected subgoal graphs, 2) storing freespace information compactly by exploiting the translation invariance of freespace distances or freespace-canonical paths, and 3) using freespace information to perform connection and refinement efficiently during queries. However, we have failed to construct small subgoal graphs on state lattices, and thus were not able to speed up the answering of path queries optimally beyond a factor of $\sim$2.

In this chapter, we consider a specific type of state lattice, namely the grid graph, where states describe only the locations of the agent, and primitives correspond to movements in one of the four cardinal or four diagonal directions. Grid graphs have the *Octile property*, the property that freespace-shortest paths on them consist of moves in at most two directions, a diagonal one and an associated cardinal one. We exploit the Octile property of grid graphs in three different ways: 1) The Octile property allows us to implement Fb and CFb connection and refinement algorithms without storing freespace information, since the required freespace information can be computed online, in constant time: Freespace distances are equal to *Octile distances*, and freespace-shortest and freespace-canonical successors, predecessors, and parents can be computed using diagonal and associated cardinal moves along freespace-shortest paths. As a consequence of this, we no longer need to use a reachability bound to limit the storage of freespace information, and can construct F and CF subgoal graphs instead of Fb and CFb subgoal graphs. 2) An arguably more important benefit of the Octile property is that it allows for the construction of small F and CF subgoal graphs: The convex corners of blocked cells form both an F-SPC and a CF-SPC, and can thus be used as the subgoals of F and CF subgoal graphs on grid graphs. That is, the F and CF subgoal graphs that we construct on grid graphs are the same. We therefore refer to F and CF subgoal graphs simply as "subgoal graphs" in this chapter. Furthermore, these subgoal graphs can also be considered to be *safe-freespace-reachability* (SF) subgoal graphs: Whereas F- and CF-reachability, respectively, require at least one freespace-shortest path or the freespace-canonical path between two vertices to be unblocked, SF-reachability can be considered to be a "stricter" reachability relation that requires *all* freespace-shortest paths between two vertices to be unblocked. Note that, since the F, CF, and SF subgoal graphs on grid graphs have the same set of subgoals, they have the same set of edges as well (Definition 3.3). 3) Finally, we exploit the Octile property of grid graphs by introducing a connection algorithm for SF-reachability that uses precomputed clearance values to efficiently scan the grid for subgoals, and prove that it can be used to construct subgoal graphs in time linear in the size of the underlying grid.

The subgoal graphs that we construct and use to answer queries on grid graphs have many similarities with jump-point search, an online path-planning algorithm on grid

graphs that we have discussed as related work in Section 2.3.3. Namely, similar to subgoal graphs on grid graphs, jump-point search also considers the convex corners of blocked cells as "important vertices", and, similar to our connection algorithm for SF-reachability, jump-point search also performs scans of the grid to efficiently perform "jumps". These similarities are not a coincidence: Sturtevant and Rabin break down jump-point search into three components, a best-first search, a canonical ordering, and a jumping policy (Sturtevant & Rabin, 2016). They also introduce canonical A* search, that can be considered both to be a jump-point search without the jumping policy, or an A* search that prunes successors of expanded vertices using a canonical ordering. We characterize the search space of canonical A* search as the *direction-extended canonical grid graph*, and show that the set of jump points used by jump-point search form an F-SPC on the direction-extended canonical grid graph. That is, we show that jump-point search can be understood as a search on a *jump-point graph*, which is a subgoal graph constructed on the direction-extended canonical grid graph.

Finally, we consider augmenting contraction hierarchies with reachability relations in three different ways. We have already described two of these augmentations in Section 3.4.2, namely constructing contraction hierarchies on subgoal graphs, or restricting the edges of contraction hierarchies to be R-reachable (R contraction hierarchies). The third augmentation is a simple modification that uses R-Refine rather than unpacking to refine the R-reachable edges of contraction hierarchies. Therefore, we do not explain these augmentations in further detail in this chapter, but simply experimentally evaluate the query-time/memory trade-offs associated with them on grid graphs.

5.2 Preliminaries and Notation

As described in Section 2.1.3, grid graphs can be considered to be instances of state lattices where states describe only the discretized location of the agent and edges correspond to straight-line motions in one of the four cardinal or four diagonal directions. We outline below the notation that we use for grid graphs, some of which is similar to the notation that we use for state lattices.

Cell, grid, freespace grid: As defined in Chapter 4, a grid $\mathcal{G} \subseteq \mathbb{Z}^2$ is a list of unblocked cells $(x, y) \in \mathcal{G}$ and the freespace grid is the grid $\mathbb{Z}^2$. In this chapter, we use the term "cell" instead of the term "state" (since states in grid graphs describe only the locations of the agent) and use the terms "cell" and "vertex" interchangeably (since there is a one-to-one correspondence between the vertices in grid graphs and the unblocked cells in the underlying grid).

Move, direction: We refer to the eight "primitives" on grid graphs as "moves". Each move is associated with one of the four cardinal directions (Up, Left, Down, Right) or one of the four diagonal directions that are formed by combining two perpendicular cardinal directions. We typically use $\vec{c}$ to indicate a move in a cardinal direction (cardinal move), $\vec{d}$ to indicate a move in a diagonal direction (diagonal move), and $\vec{v}$ to indicate a move in any direction. We sometimes treat moves $\vec{v}$ as their directions: We use the notation $\vec{v}_1 \perp \vec{v}_2$ to denote that the directions (of the moves) $\vec{v}_1$ and $\vec{v}_2$ are perpendicular (for instance, Up $\perp$ Left). We use the algebra $\vec{d} = \vec{c}_1 + \vec{c}_2$ to denote that combining two perpendicular cardinal directions $\vec{c}_1$ and $\vec{c}_2$ results in the diagonal direction $\vec{d}$ (for example, Up + Left

= Up-Left). We refer to $\vec{c}_1$ and $\vec{c}_2$ as the "associated cardinal directions" of $\vec{d}$. We use the algebra $t = s + k \cdot \vec{v}$ to denote that a cell t is reached from a cell s by moving k steps in direction $\vec{v}$.

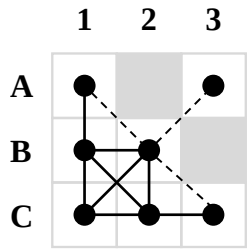


Figure 5.1: A grid graph. Corner-cutting diagonal moves (dashed lines) are not allowed.

Grid graph, freespace grid graph: Grid graphs are state lattices constructed with respect to a grid $\mathcal{G}$ and the set of primitives that correspond to the eight moves discussed above. We assume that cardinal moves have two swept cells, namely the source and destination cells, and length 1. We assume that diagonal moves have four swept cells, namely the 2×2 square of cells that contains the source and destination cells, and length $\sqrt{2}$. The swept cells of diagonal moves enforce the "no corner-cutting" constraint, that is, they disallow diagonal moves between two unblocked cells that share a neighboring blocked cell, as shown by the dashed lines in Figure 5.1. Similar to state lattices, we assume that G corresponds to the largest strongly connected component of the grid graph. For instance, in Figure 5.1, A3 is not part of G since it is disconnected from the largest strongly connected component of the grid graph. The freespace grid graph is a freespace state lattices as defined in Chapter 4, and we therefore use the same notation $\mathcal{F}$ for the freespace grid graph.

2-subpath, turn: We refer to any two-edge subpath $\langle n, \vec{v}_1, \vec{v}_2 \rangle$ of a path as a 2-subpath. We refer to a 2-subpath $\langle n, \vec{v}_1, \vec{v}_2 \rangle$ with $\vec{v}_1 \neq \vec{v}_2$ as a turn.

Taut, freespace-taut: We say that a path π is taut or freespace-taut if and only if all 2-subpaths of π are shortest paths or freespace-shortest paths, respectively. Note that, by definition, shortest or freespace-shortest paths are necessarily taut or freespace-taut, respectively. However, taut or freespace-taut paths are not necessarily shortest or freespace-shortest paths, respectively. For instance, in Figure 5.1 the path ⟨A1,B1,C2,C3⟩ is not freespace-shortest, but both of its 2-subpaths, namely ⟨A1,B1,C2⟩ and ⟨B1,C2,C3⟩ are freespace-taut.

Freespace-canonical, freespace-diagonal-first, freespace cardinal-first: As defined in Definition 4.4, an *s-t* path is freespace-canonical if and only if it is the lexically smallest freespace-shortest *s-t* path with respect to a given canonical ordering on the moves. We say that a canonical ordering is diagonal-first if and only if it orders all diagonal moves before cardinal ones. We say that a canonical ordering is cardinal-first if and only if it orders all cardinal moves before diagonal ones. We say that a path is freespace-diagonal-first if and only if it is a freespace-shortest path where all diagonal moves appear before cardinal ones. We say that a path is freespace-cardinal-first if and only if it is a freespace-shortest path where all cardinal moves appear before diagonal ones. That is,

freespace-diagonal-first and freespace-cardinal-first paths are freespace-canonical paths with respect to different canonical orderings (where diagonal moves are lexically smaller than cardinal ones or vice versa).

5.3 Subgoal Graphs on Grid Graphs

In Section 4.6.5, we have observed that it is difficult to construct small BDb, Fb, or CFb subgoal graphs on state lattices. In this section, we show that it is possible to construct small F or CF subgoal graphs (that is, without reachability bounds) on grid graphs by placing their subgoals at the convex corners of blocked cells. The resulting F and CF subgoal graphs are equivalent, and can also be considered to be a *safe-freespace-reachability* subgoal graph. We therefore refer to this F, CF, or safe-freespace-reachability subgoal graph as *the* subgoal graph throughout this chapter. We discuss how the F and CF refinement algorithms that we have developed for state lattices can be adapted to grid graphs to exploit the Octile property, introduce a connection algorithm for safe-freespace-reachability, and show that it can be used to construct subgoal graphs in time linear in the size of the underlying grid.

This section is organized as follows. In Section 5.3.1, we characterize the freespace structure of grid graphs as the *Octile property*. In Section 5.3.2, we adapt the F and CF refinement algorithms to grid graphs to exploit the Octile property. In Section 5.3.3, we introduce safe-freespace-reachability as a reachability relation, and, in Section 5.3.4, we prove that the convex corner cells form a *minimum* safe-freespace-reachability SPC. In Section 5.3.5, we introduce a connection algorithm for safe-freespace-reachability, and, in Section 5.3.6, we prove that it can be used to construct subgoal graphs in time linear in the size of the underlying grid.

5.3.1 Freespace Structure of Grid Graphs

Since grid graphs are instances of state lattices, they inherit the structural properties of state lattices, namely the translation invariance of freespace distances and freespace-canonical paths. However, since grid graphs are "very specific" instances of state lattices, they have other properties that are not necessarily present in arbitrary state lattices. Namely, on a grid with no blocked cells, shortest paths consist of moves in at most two directions, a diagonal one and an associated cardinal one. We refer to this property as the *Octile property*, which is used in the literature to efficiently calculate *Octile distances*, that is, freespace distances on grid graphs. In this section, we describe this property in greater detail.

Figure 5.2 shows all freespace-shortest paths between two cells on the freespace grid graph. Observe that all freespace-shortest E2-B9 paths consist of three Up-Right moves and four Right moves that are executed in different orders. In general, the Octile property can be stated as follows.

Theorem 5.1 (Octile property)**.** *For every pair of cells s and t, there exist a diagonal move $\vec{d}$, an associated cardinal move $\vec{c}$, and non-negative integers d and c, such that all freespace-shortest s-t paths consist of d moves in direction $\vec{d}$ and c moves in direction $\vec{c}$.*

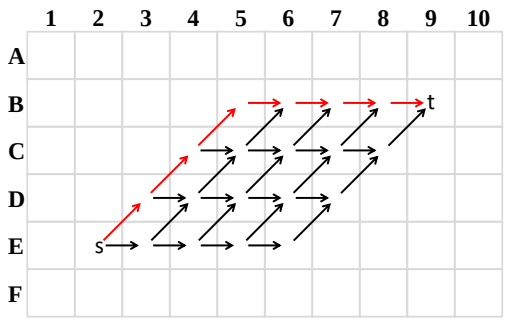


Figure 5.2: Freespace-shortest paths on grid graphs. Red path: Freespace-diagonal-first s-t path (equivalently, freespace-cardinal-first t-s path).

We omit the proof of Theorem 5.1 since it is a well-known property of grid graphs. The Octile property allows for the efficient calculation of the numbers and directions of moves along freespace-shortest paths between two cells, using only the coordinates of the cells. Given $s = (x_s, y_s)$ and $t = (x_t, y_t)$, with $t = s + d \cdot \vec{d} + c \cdot \vec{c}$, $\vec{d}$ and $\vec{c}$ can be determined in constant time by determining which ones of the inequalities $x_s < x_t$, $y_s < y_t$, and $|x_s - x_t| < |y_s - y_t|$ hold. Furthermore, d and c can be calculated in constant time as follows:

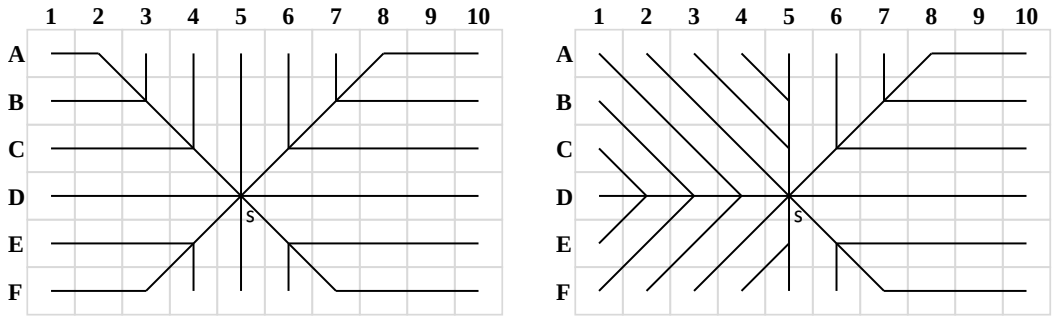
$$\Delta_x = |x_s - x_t|$$
$$\Delta_y = |y_s - y_t|$$
$$d = \min(\Delta_x, \Delta_y)$$
$$c = \max(\Delta_x, \Delta_y) - \min(\Delta_x, \Delta_y)$$

From these values, the freespace s-t distance, also known as the Octile distance, can be calculated as follows:

$$d_{\mathcal{F}}(s,t) = \sqrt{2} \cdot d + c = (\sqrt{2} - 1) \cdot \min(\Delta_x, \Delta_y) + \max(\Delta_x, \Delta_y)$$

5.3.2 F-Reachability and CF-Reachability on Grid Graphs

Since grid graphs are instances of state lattices, the definitions of F-reachability and CF-reachability on state lattices (Definitions 4.2 and 4.5) also apply to grid graphs. Namely, a cell t is F-reachable from a cell s if and only if a freespace-shortest s-t path is unblocked on G, and CF-reachable if and only if the freespace-canonical s-t path (with respect to a given canonical-ordering of the eight moves) is unblocked on G. In this section, we describe how the F-Refine and CF-Refine algorithms (Algorithms 11 and 13) that we have developed for state lattices can be adapted to grid graphs to exploit the Octile property, and discuss the canonical orderings we use on grid graphs. We do not use F-Connect and CF-Connect in this chapter, but introduce a connection algorithm for



(a) Diagonal-first canonical ordering. (b) Symmetric canonical ordering.

Figure 5.3: All freespace-canonical paths that originate at a cell, for the diagonal-first and symmetric canonical orderings.

safe-freespace-reachability in Section 5.3.5, which we use for constructing subgoal graphs on grid graphs.

F-Refine: We can implement F-Refine such that, for every $(s,t) \in \mathrm{F}$, it finds a freespace-shortest s-t path that is unblocked on a grid graph G as follows: As discussed in Section 5.3.1, there exist d, c, $\vec{d}$, and $\vec{c}$ such that every freespace-shortest s-t path consists of d diagonal moves in direction $\vec{d}$ and c cardinal moves in direction $\vec{c}$; and d, c, $\vec{d}$, $\vec{c}$ can be calculated in constant time from the coordinates of s and t. Since $(s,t) \in F$, at least one ordering of these moves corresponds to an unblocked s-t path on G. We can find this ordering by performing a depth-first search from s on G that only generates paths with at most d moves in direction $\vec{d}$ and c moves in direction $\vec{c}$ (and no moves in any other directions). Once the depth-first search finds an unblocked path with exactly d moves in direction $\vec{d}$ and c moves in direction $\vec{c}$, the resulting path is guaranteed to be a freespace-shortest s-t path on G.

Canonical orderings: Since freespace-shortest paths on grid graphs have moves in at most two directions, every canonical ordering $\mathscr{L}$ that specifies, for every diagonal direction $\vec{d}$ and associated cardinal direction $\vec{c}$, either $\vec{d} <_{\mathscr{L}} \vec{c}$ or $\vec{c} <_{\mathscr{L}} \vec{d}$, also uniquely designates exactly one freespace-shortest s-t path, either the diagonal-first or the cardinal-first one, as the freespace-canonical s-t path. We refer to the canonical ordering that specifies $\vec{d} <_{\mathscr{L}} \vec{c}$ for all such $\vec{d}$ and $\vec{c}$ as the *diagonal-first canonical ordering*. We refer to the canonical ordering that specifies $\vec{c} <_{\mathscr{L}} \vec{d}$ for all such $\vec{d}$ and $\vec{c}$ as the *cardinal-first canonical ordering*. Figure 5.3a shows all freespace-diagonal-first paths that originate at a cell s. Equivalently, it shows all freespace-cardinal-first paths that terminate at a cell s. Observe that, although grid graphs are undirected, the CF-reachability relation with respect to the diagonal-first or cardinal-first canonical ordering is not necessarily symmetric. For instance, if, for some $s,t \in V$, the freespace-diagonal-first s-t path is unblocked but the freespace-cardinal-first s-t path (equivalently, the freespace-diagonal-first t-s path) is blocked, then $(s,t) \in \mathrm{CF}$ but $(t,s) \notin \mathrm{CF}$.

We use the following canonical ordering on grid graphs, which we refer to as the *symmetric canonical ordering*, or, simply, as *the* canonical ordering, which is guaranteed to result in a symmetric CF-reachability relation (which we use in Section 5.5.9 to construct

CF contraction hierarchies with undirected edges only): For every diagonal direction $\vec{d} = \vec{c}_1 + \vec{c}_2$, if $\vec{c}_1 = \text{Right}$ or $\vec{c}_2 = \text{Right}$, then $\vec{d} <_{\mathscr{L}} \vec{c}_1$ and $\vec{d} <_{\mathscr{L}} \vec{c}_2$. Otherwise, $\vec{c}_1 <_{\mathscr{L}} \vec{d}$ and $\vec{c}_2 <_{\mathscr{L}} \vec{d}$. That is, for every $(s,t) \in \text{CF}$, if t is towards the Right of s, we require that the freespace-diagonal-first s-t path is unblocked and, otherwise, we require that the freespace-cardinal-first s-t path is unblocked. Figure 5.3b shows all freespace-canonical paths that originate at a cell s, with respect to the symmetric canonical ordering.

CF-Refine: For every canonical ordering, we can implement CF-Refine to simply generate, for any $(s,t) \in \text{CF}$, the freespace-diagonal-first or the freespace-cardinal-first s-t path, depending on the canonical ordering and the relative locations of s and t on the grid.

5.3.3 Safe-Freespace-Reachability (SF-Reachability)

In this section, we introduce a new reachability relation on grid graphs, called *safe-freespace-reachability* (SF-reachability), formally defined in Definition 5.1.

Definition 5.1 (Safe-freespace-reachability)**.** *A vertex t is safe-freespace-reachable (SF-reachable) from a vertex s, denoted as $(s,t) \in \text{SF}$, if and only if all freespace-shortest s-t paths are unblocked on G.*

F-, CF-, and SF-reachability relate to each other as follows: For every $(s,t) \in \text{SF}$, since *all* freespace-shortest s-t paths are unblocked on G, the freespace-canonical s-t path must be unblocked on G. Therefore, $(s,t) \in \text{CF}$ and, consequently, $\text{SF} \subseteq \text{CF}$. As discussed in Section 4.5.3, for every $(s,t) \in \text{CF}$, since the freespace-canonical s-t path is unblocked on G, at least one freespace-shortest s-t path must be unblocked on G. Therefore, $(s,t) \in \text{F}$ and, consequently, $\text{CF} \subseteq \text{F}$. Combining these two facts, we get $\text{SF} \subseteq \text{CF} \subseteq \text{F}$.

We introduce a connection algorithm for SF-reachability in Section 5.3.5. Since $\text{SF} \subseteq \text{CF}$, we use CF-Refine as a refinement algorithm for SF-reachability as well.

5.3.4 Using Convex Corner Cells as Subgoals

In this section, we prove that the set of *convex corner cells* is a minimum SF-SPC, CF-SPC, and F-SPC. We also discuss at the end of this section how subgoals could be placed on other types of grid graphs.

Figure 5.4 shows an example of a convex corner cell, namely A2, placed at the convex corner of blocked cell B1. Definition 5.2 defines convex corner cells.

Definition 5.2 (Convex Corner Cell)**.** *An unblocked cell s is a* convex corner cell *if and only if there are two cardinal directions $\vec{c}_1 \perp \vec{c}_2$ such that $s + \vec{c}_1 + \vec{c}_2$ is blocked and $s + \vec{c}_1$ and $s + \vec{c}_2$ are unblocked. $\mathcal{C}$ is the set of all convex corner cells.*

Observe that every convex corner cell has to be included in every SF-SPC, CF-SPC, and F-SPC: Recall that an R-SPC S has the property that, for every $s,t \in V$, $(s,t) \in R$ or $S \sqsubset (s,t)$ (S covers a shortest s-t path). Consider the example in Figure 5.4. The unique freespace-shortest A1-B2 path $\langle$A1,B2$\rangle$ is blocked by B1 since it corresponds to a corner-cutting diagonal move. Therefore, (A1, B2) $\notin \text{F} \supseteq \text{CF} \supseteq \text{SF}$, and every SF-SPC, CF-SPC, and F-SPC has to cover at least one shortest A1-B2 path on G. However,

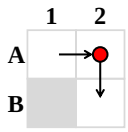


Figure 5.4: A convex corner cell (red disk). A2 must be included in every F-SPC, CF-SPC, or SF-SPC since B1 blocks the unique freespace-shortest A1-B2 path ⟨A1, B2⟩.

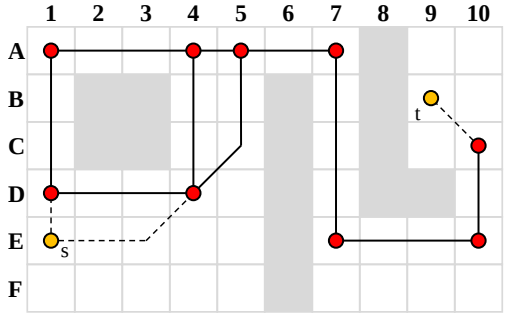


Figure 5.5: A query subgoal graph $G_{\mathcal{C}}^{s,t}$ on grid graphs. $\mathcal{C}$ is shown as red disks. All edges are $D_{\mathcal{C}}$-, $\mathrm{F}_{\mathcal{C}}$-, $\mathrm{CF}_{\mathcal{C}}$-, and $\mathrm{SF}_{\mathcal{C}}$-reachable.

there is only one shortest A1-B2 path on G, namely, ⟨A1,A2,B2⟩, which is covered by only a single cell, namely, the convex corner cell A2. Therefore, A2 must be included in every F-SPC, CF-SPC, or SF-SPC. Theorem 5.2 summarizes this result (proof omitted). This theorem is a direct consequence of the fact that the grid graphs that we consider in this dissertation exclude corner-cutting diagonal moves. As we discuss at the end of this section, this result does not necessarily hold for other types of grid graphs.

Theorem 5.2. *For every F-SPC, CF-SPC, or SF-SPC S, $\mathcal{C} \subseteq S$.*

Theorem 5.2 does not imply that $\mathcal{C}$ is an F-SPC, CF-SPC, or an SF-SPC, but only states that $\mathcal{C}$ has to be part of every such SPC. Figure 5.5 shows the overlay graph $G_{\mathcal{C}}$ of the convex corner cells, extended to a query overlay graph $G_{\mathcal{C}}^{s,t}$ using only $D_{\mathcal{C}}$-reachable edges. Notice that all edges shown in Figure 5.5 connect SF-reachable (and, therefore, CF- and F-reachable) cells. We now prove that $\mathcal{C}$ is an SF-SPC (and, by Lemma 3.11, therefore also a CF-SPC and an F-SPC).

Let π be a shortest s-t path on G that is not covered by $\mathcal{C}$. Lemma 5.3 proves that π is freespace-taut. Lemma 5.4 proves that "swapping" consecutive cardinal and diagonal moves on π produces unblocked paths. Lemma 5.5 uses the swapping operation introduced in Lemma 5.4 to prove that π is also freespace-shortest, which is sufficient to prove that $(s,t) \in F$. Theorem 5.6 uses the swapping operation introduced in Lemma 5.4 to rearrange the order of moves on π to produce any other freespace-shortest s-t path, to prove that $\mathcal{C}$ is an SF-SPC.

We note that the fact that the vertices of safe-freespace-reachability subgoal graphs on grid graphs are placed at convex corner cells is closely related to the fact that the

vertices of visibility graphs on Euclidean spaces with polygonal obstacles (Lozano-Pérez & Wesley, 1979) are placed at the convex corners of the polygons. Our proofs also reflect this similarity, by showing that shortest paths that are not freespace-shortest (in the case of visibility graphs, shortest paths that are not straight lines) must pass through a convex corner cell (convex corner of a polygon) because, otherwise, they cannot be shortest paths.

Lemma 5.3. *Let π be a shortest s-t path on G, such that $\mathcal{C} \not\sqsubset \pi$. Then, π is freespace-taut.*

Proof. We prove that, if π is a shortest path and $\mathcal{C} \not\sqsubset \pi$, then π is freespace-taut, that is, all its 2-subpaths are freespace-shortest paths. Figure 5.6 shows all possible 2-subpaths $\langle n, \vec{v}_1, \vec{v}_2 \rangle$ that can appear on a path with no cycles. The first move $\vec{v}_1$ is shown as a black arrow, and all possible second moves $\vec{v}_2$ are shown as red, blue, or green arrows.

(a) Cases where the first move is cardinal. (b) Cases where the first move is diagonal.

Figure 5.6: All possible 2-subpaths on grid graphs that do not form cycles. The black arrow denotes the first move. Red arrows denote moves that are guaranteed to form non-taut 2-subpaths with the first move. Blue arrows denote moves that may or may not form taut 2-subpaths with the first move. Green arrows denote moves that form freespace-taut 2-subpaths with the first move.

We consider three cases:

- *The second move is red:* Then, the two moves cannot form a taut 2-subpath and, therefore, π cannot be a shortest path, contradicting that it is. For instance, Right followed by Up-Left can be replaced with Up, which is strictly shorter. On any grid graph where Right followed by Up-Left is unblocked, Up must also be unblocked. This reasoning applies to all cases where the second move is red.

- *The second move is blue:* This case considers two perpendicular cardinal directions $\vec{c}_1 \perp \vec{c}_2$. Let $\vec{d} = \vec{c}_1 + \vec{c}_2$. We consider two subcases: 1) If $\langle n, \vec{d} \rangle$ is unblocked, then $\langle n, \vec{c}_1, \vec{c}_2 \rangle$ is not taut and, therefore, π cannot be a shortest path, contradicting that it is. 2) If $\langle n, \vec{d} \rangle$ is blocked (but $\langle n, \vec{c}_1, \vec{c}_2 \rangle$ is unblocked), then it must be the case that $n + \vec{c}_1 \in \mathcal{C}$, which covers $\langle n, \vec{c}_1, \vec{c}_2 \rangle$, contradicting that $\mathcal{C} \not\sqsubset \pi$.

- *The second move is green:* Then, the two moves form a freespace-taut 2-subpath.

Since, in any 2-subpath in π, the second move cannot be red or blue, the second move must be green, and, therefore, the 2-subpath must be freespace-taut. Therefore, π is freespace-taut.

□

Lemma 5.4. *Let π be a shortest s-t path on G, such that $\mathcal{C} \not\sqsubset \pi$. Then, swapping the order of any two consecutive moves on π results in an (unblocked) shortest s-t path on G.*

Proof. By Lemma 5.3, every pair of consecutive moves $\vec{v}_i$ and $\vec{v}_{i+1}$ on π form a freespace-taut 2-subpath. If $\vec{v}_1 = \vec{v}_2$, then swapping them in π results in π. Otherwise, it must be the case that exactly one of $\vec{v}_1$ or $\vec{v}_2$ is a diagonal move, and the other one an associated cardinal move. Figure 5.7 shows both cases (swapping the order of moves in one case results in the other case):

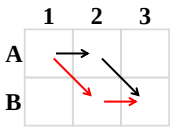


Figure 5.7: Replacing diagonal-to-cardinal turns with cardinal-to-diagonal turns.

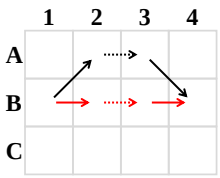
Without loss of generality, we prove the lemma for moving $\vec{c}$ = Right first and $\vec{d}$ = Down-Right second from cell A1. Since the (black) path $\langle \text{A1}, \vec{c}, \vec{d} \rangle = \langle \text{A1, A2, B3} \rangle$ is unblocked, the cells A1, A2, A3, B2, and B3 must be unblocked. Since $\mathcal{C} \not\sqsubset \pi$ and we assume that $\langle \text{A1, A2, B3} \rangle$ is a 2-subpath of π, $\mathcal{C} \not\sqsubset \langle \text{A1, A2, B3} \rangle$ (Definition 3.1). Therefore, A2 cannot be a convex corner cell. Consequently, B1 must be unblocked because, otherwise, A2 would be a convex corner cell. Therefore, all six cells must be unblocked, and the (red) path $\langle \text{A1}, \vec{d}, \vec{c} \rangle$ is also unblocked. □

Lemma 5.5. *Let π be a shortest s-t path on G, such that $\mathcal{C} \not\sqsubset \pi$. Then, π is freespace-shortest.*

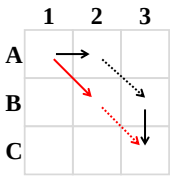
Proof. Let $\pi = \langle s, \vec{v}_1, \ldots, \vec{v}_k \rangle$ be a shortest path with $\mathcal{C} \not\sqsubset \pi$. We prove that *any* two moves on π (not necessarily consecutive ones) can be combined into a freespace-shortest path. That is, we prove that, for every $\vec{v}_i$ and $\vec{v}_j$ on π, it holds that $\vec{v}_i = \vec{v}_j$ or that exactly one of v_i or v_j is in a diagonal direction and the other one in an associated cardinal direction. Consequently, π must be freespace shortest, that is, there exist a diagonal direction $\vec{d}$ and an associated cardinal direction $\vec{c}$ such that, for all $\vec{v}_i$, either $\vec{v}_i = \vec{d}$ or $\vec{v}_i = \vec{c}$ (otherwise, if no such $\vec{d}$ and $\vec{c}$ exist, then at least two moves on π combine into a path that is not freespace-shortest).

Assume, for contradiction, that two non-consecutive moves $\vec{v}_i$ and $\vec{v}_j$ appear on π that combine into a 2-subpath that is not freespace-shortest. Let $\vec{v}_i$ and $\vec{v}_j$ be two such moves with $i < j$ and minimum $j - i$. By Lemma 5.3, all 2-subpaths of π are freespace-shortest. Intuitively, there can only be two cases that satisfy these criteria, which are shown in Figure 5.8. The path $\langle n, \vec{v}_i, \ldots, \vec{v}_j \rangle$ is shown in black (and the dotted line corresponds to the sequence of moves $\vec{v}_{i+1}, \ldots, \vec{v}_{j-1}$). In both cases, we can prove that either $\mathcal{C} \sqsubset \langle \vec{v}_i, \ldots, \vec{v}_j \rangle$, or that we can replace $\langle \vec{v}_i, \ldots, \vec{v}_j \rangle$ in π with a strictly shorter path, shown in red, contradicting that π is a shortest path with $\mathcal{C} \not\sqsubset \pi$.

Without loss of generality, assume that $\vec{v}_i = \vec{d}$ is a diagonal move. We prove that Case 1 holds. (A similar proof can be used to show that Case 2 holds if $\vec{v}_i = \vec{c}$ is a cardinal move.) First, we prove that there exists a cardinal move $\vec{c}$ associated with $\vec{d}$ such that, for every $m \in \{i+1, \ldots, j-1\}$, $\vec{v}_m = \vec{c}$, by induction on m. Base case: Since



(b) Case 1.



(a) Case 2.

Figure 5.8: Black paths are freespace-taut but not freespace-shortest. Red paths are freespace-shortest. Dashed arrows indicate any positive number of repeated moves in a given direction.

$\langle \vec{v}_i = \vec{d}, \vec{v}_{i+1} \rangle$ is freespace-shortest (Lemma 5.3), it holds that either $\vec{v}_{i+1} = \vec{v}_i$ or $\vec{v}_{i+1} = \vec{c}$ for some cardinal move $\vec{c}$ associated with $\vec{d}$. However, $\vec{v}_{i+1} \neq \vec{v}_i$ because, otherwise, $\vec{v}_{i+1}$ and $\vec{v}_j$ combine into a 2-subpath that is not freespace shortest, contradicting that $\vec{v}_i$ and $\vec{v}_j$ are two such moves with $i < j$ and the minimum $j - i$. Therefore, $\vec{v}_{i+1} = \vec{c}$ for some cardinal direction $\vec{c}$ associated with $\vec{d}$. Induction step: Assume that $\vec{v}_m = \vec{c}$ for some cardinal direction $\vec{c}$ associated with $\vec{d}$. Since $\langle \vec{v}_m = \vec{c}, \vec{v}_{m+1} \rangle$ is freespace-shortest (Lemma 5.3), it holds that either $\vec{v}_{m+1} = \vec{v}_m = \vec{c}$, $\vec{v}_{i+1} = \vec{d}$, or $\vec{v}_{i+1} = \vec{d'}$ for the other diagonal move $\vec{d'} \neq \vec{d}$ associated with $\vec{c}$. However, $\vec{v}_{m+1} \neq \vec{d}$ (and $\vec{v}_{m+1} \neq \vec{d'}$) because, otherwise, $\vec{v}_{m+1}$ and $\vec{v}_j$ (or $\vec{v}_i$ and $\vec{v}_{m+1}$) combine into a 2-subpath that is not freespace shortest, contradicting that $\vec{v}_i$ and $\vec{v}_j$ are two such moves with $i < j$ and the minimum $j - i$. Therefore, $\vec{v}_{m+1} = \vec{v}_m = \vec{c}$ and, consequently, $\vec{v}_{i+1} = \cdots = \vec{v}_{j-1} = \vec{c}$ for some cardinal move $\vec{c}$ associated with $\vec{d}$. Second, we prove that $\vec{v}_j = \vec{d'}$ for the other diagonal move $\vec{d'} \neq \vec{d}$ associated with $\vec{c}$. Since $\langle \vec{v}_{j-1} = \vec{c}, \vec{v}_j \rangle$ is freespace-shortest (Lemma 5.3), it holds that either $\vec{v}_j = \vec{v}_{j-1} = \vec{c}$, $\vec{v}_{i+1} = \vec{d}$, or $\vec{v}_{i+1} = \vec{d'}$ for the other diagonal move $\vec{d'} \neq \vec{d}$ associated with $\vec{c}$. However, since we assume that $v_i = \vec{c}$ and v_j do not combine into a 2-subpath that is freespace shortest, $\vec{v}_j \neq \vec{d}$ and $\vec{v}_j \neq \vec{c}$. Therefore, $\vec{v}_j = \vec{d'}$ and, consequently, Case 1 holds.

Assume that Case 1 (or, similarly, Case 2) holds. Let $n = s + \vec{v}_0 + \cdots + \vec{v}_{i-1}$ (that is, the vertex from which $\vec{v}_i$ is executed on π). We prove that $\langle n, \vec{v}_i, \ldots, \vec{v}_j \rangle$ is not a shortest path, by transforming it to a shorter path (that is, we transform the black path in Figure 5.8 to the red path). Let $\pi_0, \ldots, \pi_{j-i-1}$ be the sequence of paths such that $\pi_0 = \langle n, \vec{v}_i = \vec{d}, v_{i+1} = \vec{c}, \ldots, \vec{v}_{j-1} = \vec{c}, \vec{v}_j = \vec{d'} \rangle$ (no cardinal moves appear before the first diagonal move) and, for every $m \in \{1, \ldots, \pi_{j-i-1}\}$, $\pi_m = \langle n, \vec{v}_{i+1} = \vec{c}, \ldots, \vec{v}_{i+m} = \vec{c}, \vec{v}_i = \vec{d}, \vec{v}_{i+m+1} = \vec{c}, \ldots, \vec{v}_{j-1} = \vec{c}, \vec{v}_j = \vec{d'} \rangle$ (m cardinal moves appear before the first diagonal move). That is, π_{m+1} is obtained from π_m by swapping the order of the diagonal move $\vec{v}_i = \vec{d}$ with the cardinal move $\vec{v}_{i+m+1} = \vec{c}$ that follows it on π_m. We prove that, for $m = 0, \ldots, j - 1 - 1$, π_m is unblocked on G by induction on m. Base case: π_0 is a subpath of π, and is therefore unblocked on G. Induction step: Assume that π_m is unblocked on G. π_{m+1} is obtained by replacing the path $\langle n', \vec{v}_i = \vec{d}, n'', \vec{v}_{i+m+1} = \vec{c} \rangle$ in π_m with the path $\langle n', \vec{v}_{i+m+1}, \vec{v}_i \rangle$. Observe that, similar to π, n'' is reached from n on π_m with one diagonal move $\vec{d}$ and m cardinal moves $\vec{c}$. Therefore, n'' is an internal vertex π. Since $\mathcal{C} \not\sqsubset \pi$, $n' \notin \mathcal{C}$ (Definition 3.1). Since $\langle n', \vec{v}_i, n'', \vec{v}_{i+m+1} \rangle$ is a subpath of π_m, it is unblocked. Therefore, Lemma 5.4 applies to $\langle n', \vec{v}_i, n'', \vec{v}_{i+m+1} \rangle$, and the path

$\langle n', \vec{v}_{i+m+1}, \vec{v}_i \rangle$ is unblocked on G. Consequently, π_{m+1} is unblocked on G. Consequently, $\pi_{j-i-1} = \langle n, \vec{v}_{i+1} = \vec{c}, \ldots, \vec{v}_{j-1} = \vec{c}, \vec{v}_i = \vec{d}, \vec{v}_j = \vec{d'} \rangle$ is unblocked on G.

π_{j-i-1} has the same length as π_0, since it is obtained by reordering moves on π_0. However, replacing the subpath $\langle n''', \vec{v}_i = \vec{d}, \vec{v}_j = \vec{d'} \rangle$ in π_{j-i-1} with the path $\langle n''', \vec{c}, \vec{c} \rangle$ produces a path that is shorter than π_{j-i-1} and, therefore, shorter than π_0. Consequently, π_0 is not a shortest path and, since π_0 is a subpath of π, π cannot be a shortest path, contradicting our assumption that it is. Therefore, no $\vec{v}_i$ and $\vec{v}_j$ on π combine into a non-freespace-shortest path. Consequently, π must be freespace shortest, that is, there exist a diagonal direction $\vec{d}$ and an associated cardinal direction $\vec{c}$ such that, for all $\vec{v}_i$, either $\vec{v}_i = \vec{d}$ or $\vec{v}_i = \vec{c}$ (otherwise, if no such $\vec{d}$ and $\vec{c}$ exist, then at least two moves on π combine into a path that is not freespace-shortest).

□

Theorem 5.6. *$\mathcal{C}$ is an SF-SPC.*

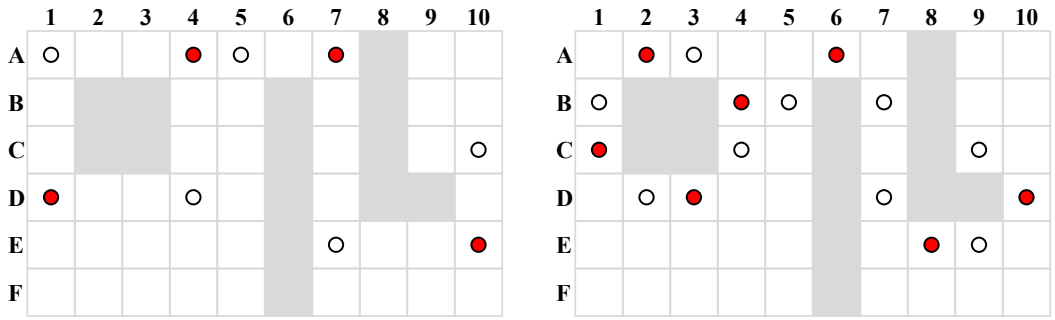
Proof. We show that, for every $s, t \in V$, $(s, t) \in \text{SF}$ or $\mathcal{C} \sqsubset (s, t)$. Assume, for contradiction, that there exists $s, t \in V$ with $(s, t) \notin \text{SF}$ and $\mathcal{C} \not\sqsubset (s, t)$. Let π be a shortest s-t path. Since $\mathcal{C} \not\sqsubset (s, t)$, $\mathcal{C} \not\sqsubset \pi$ (Definition 3.1). Then, by Lemma 5.5, π is freespace-shortest. (Note that, this would conclude the proof if we were proving that $\mathcal{C}$ is an F-SPC). We now prove that *every* freespace-shortest s-t path π' is unblocked on G, that is $(s, t) \in \text{SF}$, to arrive at a contradiction.

Let π' be any freespace-shortest s-t path. Since π and π' are both freespace-shortest s-t paths, they consists of the same set of moves, but possibly correspond to different orderings of these moves. Therefore, we can generate π' from π by swapping consecutive moves on π a finite number of times (that is, "sort" the moves on π' to match π). Let $\pi_0 = \pi, \ldots, \pi_k = \pi'$ be the sequence of paths, where, for each $i = 1, \ldots, k$, we can obtain π_i from π_{i-1} with a single "swap". We show that, by induction on $i = 0, \ldots, k$, π_i is unblocked on G. Base case: For $i = 0$, $\pi_0 = \pi$, which is known to be unblocked. Induction step: Assume that π_i is unblocked. Since π_i is freespace-shortest and unblocked, it is a shortest path on G (Lemma 4.3). Since $\mathcal{C} \not\sqsubset (s, t)$, $\mathcal{C} \not\sqsubset \pi_i$ (Definition 3.1). Therefore, Lemma 5.4 applies to π_i, and, since π_{i+1} is derived from π_i by "swapping" two consecutive moves, π_{i+1} is unblocked on G.

Since all freespace-shortest s-t paths are unblocked on G, $(s, t) \in \text{SF}$, contradicting that $(s, t) \notin \text{SF}$. Therefore, for every $s, t \in V$, $(s, t) \in \text{SF}$ or $\mathcal{C} \sqsubset (s, t)$, and, consequently, $\mathcal{C}$ is an SF-SPC.

□

By Theorems 5.2 and 5.6, the set of convex corner cells $\mathcal{C}$ is the minimum SF-SPC, CF-SPC, and F-SPC. The fact that $\mathcal{C}$ is the *minimum* SF-SPC is a direct consequence of the fact that the grid graphs that we consider in this dissertation exclude corner-cutting diagonal moves. However, the intuition that cells associated with convex corners of blocked cells are "important" applies to other variations of grid graphs as well. Figure 5.9 shows an example. On a 4-neighbor grid graph (Figure 5.9a), the set of convex corner cells also forms an SF-SPC. On an 8-neighbor grid graph with corner-cutting diagonal-moves (Figure 5.9b), the cells adjacent to convex corner cells form a SF-SPC. However, these sets of cells are not necessarily *minimal* SF-SPCs. For instance, in Figure 5.9b, C9 can be



(a) An F-SPC on a 4-neighbor grid graph.

(b) An F-SPC on an (8-neighbor) grid graph with corner-cutting diagonal-moves.

Figure 5.9: Placement of subgoals on other types of grid graphs. The set of all (red or white) disks shows the "important" cells associated with convex corners and forms an SF-SPC. The set of red disks show one possible placement of subgoals as F-SPCs.

excluded from an SF-SPC. More generally, on 8-neighbor grid graphs with corner-cutting diagonal moves, all *concave corner cells* can be excluded from SF-SPCs (since shortest paths do not "bend" around concave corners as they do around convex corners). As we will see in our experimental evaluation in Section 5.5.4, the exclusion of corner-cutting diagonal moves in the grid graphs that we consider in this dissertation results in the *staircase problem*, which negatively affects the execution times of answering queries using subgoal graphs. F-SPC can be significantly smaller than SF-SPCs on different types of grid graphs. The red disks in Figures 5.9a and 5.9b show F-SPCs that are smaller than SF-SPCs. Finally, minimal F-SPCs are not necessarily unique on different types of grid graphs (like they are on grid graphs without corner-cutting diagonal moves). For instance, if A6 were removed from the F-SPC shown in Figure 5.9b, it can be replaced with B5 and B7, resulting in a different minimal F-SPC. A more formal analysis of the placement of subgoals on other types of grid graphs is beyond the scope of this dissertation.

5.3.5 SF-Connect

In this section, we introduce a connection algorithm for SF-reachability. Algorithm 14 outlines the SF$^{\rightarrow}$-Connect algorithm, which uses precomputed *clearance values* to scan the grid efficiently, and, as we show in Section 5.3.6, can be used to construct $G_{\mathcal{C}}$ in time linear in the size of the underlying grid. SF$^{\rightarrow}$-Connect is exact, that is, identifies the set of SF$_{\mathcal{C}}$-reachable subgoals from a given vertex exactly. However, since it relies on *precomputed* clearance values, it might not connect the start vertex to the goal vertex even if they are SF-reachable. Algorithm 15 outlines the SF-Connect algorithm, which executes SF$^{\rightarrow}$-Connect twice, once from the start vertex and once from the goal vertex, and addresses the shortcoming of SF$^{\rightarrow}$-Connect by performing a separate check to see if the freespace-diagonal-first path from the start vertex to the goal vertex is unblocked.

We break down the operation of SF$^{\rightarrow}$-Connect (Algorithm 14) from a given vertex s into three components: 1) SF$^{\rightarrow}$-Connect explores only freespace-diagonal-first paths that originate at s, and only those paths that are not covered by $\mathcal{C}$. That is, its operation is

Algorithm 14 SF$^{\rightarrow}$-Connect.

Blue text: Only used for exactly identifying the SF$_{\mathcal{C}}$-reachable subgoals from s.

Input: Start vertex s

Output: The set D of SF$_{\mathcal{C}}$-reachable subgoals from s

```
1: function CardinalScan(n, c⃗, &D, &clearance_limit)
2:     if 0 < C[n, c⃗] ≤ clearance_limit then
3:         clearance_limit ← C[n, c⃗]
4:         if n + c⃗ · C[n, c⃗] is a subgoal then
5:             add n + c⃗ · C[n, c⃗] to D
6:             clearance_limit ← clearance_limit − 1
7: function DiagonalFirstScan(n, d⃗, &D)
8:     assign c⃗1, c⃗2 such that c⃗1 + c⃗2 = d⃗
9:     clearance_limit_1 ← C[s, c⃗1] − 1 if s + c⃗1 · C[s, c⃗1] is a subgoal, C[s, c⃗1] otherwise
10:    clearance_limit_2 ← C[s, c⃗2] − 1 if s + c⃗2 · C[s, c⃗2] is a subgoal, C[s, c⃗2] otherwise
11:    while can move from n to n + d⃗ do
12:        n ← n + d⃗
13:        if n is a subgoal then
14:            add n to D
15:            return
16:        CardinalScan(n, c⃗1, D, clearance_limit_1)
17:        CardinalScan(n, c⃗2, D, clearance_limit_2)
18: function SF→-Connect(s)
19:    D ← ∅
20:    for all cardinal directions c⃗ do
21:        CardinalScan(s, c⃗, D, ∞)
22:    for all diagonal directions d⃗ do
23:        DiagonalFirstScan(s, d⃗, D)
24:    return D
```

similar to that of CF$^{\rightarrow}$-Connect (Algorithm 12), assuming that the canonical ordering is diagonal-first. We informally refer to the exploration of freespace-diagonal-first paths as "scanning the grid". 2) SF$^{\rightarrow}$-Connect uses precomputed cardinal clearance values to scan the grid more efficiently. 3) SF$^{\rightarrow}$-Connect terminates some of its scans early to identify SF$_{\mathcal{C}}$-reachable subgoals from a given vertex exactly. We use Figure 5.10 as a running example throughout this section.

- *Exploring freespace-diagonal-first paths (scanning the grid):* SF$^{\rightarrow}$-Connect explores freespace-diagonal-first paths that originate at s, and only those paths that are not covered by $\mathcal{C}$. These paths are shown as a combination of black, green, and red lines in Figure 5.10. For every SF$_{\mathcal{C}}$-reachable vertex t from s, the freespace-diagonal-first s-t path must be unblocked and not covered by $\mathcal{C}$: If it is blocked, then t is not SF-reachable from s, and, if it is covered by $\mathcal{C}$, then t is not direct-reachable from s. Therefore, SF$^{\rightarrow}$-Connect can identify all SF$_{\mathcal{C}}$-reachable subgoals from s.

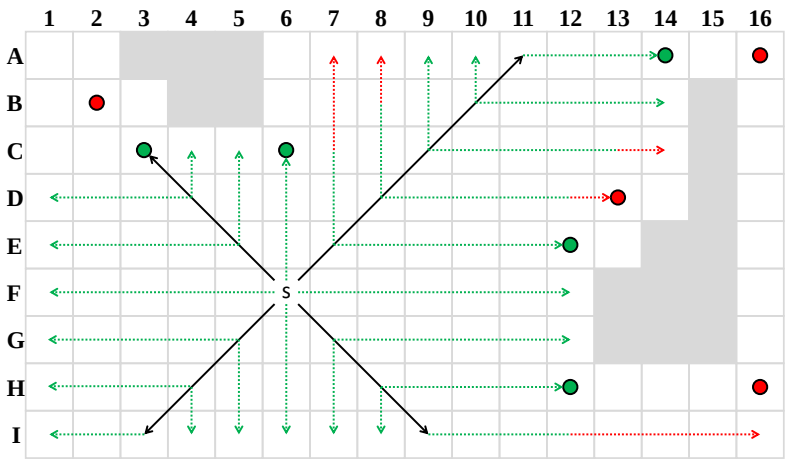


Figure 5.10: SF$^{\rightarrow}$-Connect scans the diagonal-first path to every cell that is SF$_{\mathcal{C}}$-reachable from s. Subgoals ($\mathcal{C}$) that are SF$_{\mathcal{C}}$-reachable from s are shown in green, and subgoals that are not SF$_{\mathcal{C}}$-reachable from s are shown in red. Clearance-value look-ups are shown as dotted lines, where the green parts sweep cells that are SF$_{\mathcal{C}}$-reachable from s and the red parts sweep cells that are not SF$_{\mathcal{C}}$-reachable from s.

SF$^{\rightarrow}$-Connect systematically explores these paths as follows: First, SF$^{\rightarrow}$-Connect explores all freespace-diagonal-first paths that consists of only cardinal moves, by performing *cardinal scans* (CardinalScan, lines 20–21) from s in every cardinal direction $\vec{c}$. A cardinal scan from s in direction $\vec{c}$ scans the grid from s in direction $\vec{c}$ until a subgoal or a blocked cell is reached (lines 1–6). Second, SF$^{\rightarrow}$-Connect explores all freespace-diagonal-first paths that contains at least one diagonal move, by performing a *diagonal-first scan* (DiagonalFirstScan, lines 22–23). A diagonal-first scan from s in direction $\vec{d}$ proceeds similarly to a cardinal scan, and scans the grid from s in direction $\vec{d}$ until a subgoal is reached or a $\vec{d}$ move is no longer possible due to blocked cells (lines 11–15), but also branches off into cardinal scans in directions $\vec{c}_1 + \vec{c}_2 = d$ (line 8) from every visited unblocked cell that is not a subgoal (lines 16–17).

- *Clearance values:* The results of cardinal scans for every starting cell n and cardinal direction $\vec{c}$ can be precomputed and stored as *clearance values*. That is, the clearance value $\mathbf{C}[n, \vec{c}]$ of a cell n in direction $\vec{c}$ is the number of moves that can be made from n in direction $\vec{c}$ to reach a subgoal or a blocked cell. Using precomputed clearance values, cardinal scans of SF$^{\rightarrow}$-Connect reduce to clearance-value look-ups (line 4), and the execution time of SF$^{\rightarrow}$-Connect can be measured by the length of its diagonal scans, shown as black lines in Figure 5.10. We discuss the complexity of SF$^{\rightarrow}$-Connect in more detail in Section 5.3.6.

- *Clearance limits:* Freespace-diagonal-first paths that originate at s and are not covered by $\mathcal{C}$ can reach cells that are not SF$_{\mathcal{C}}$-reachable from s. In Figure 5.10, paths that reach such cells are shown in red. For instance, D13 is reached from s with

a freespace-diagonal-first path that is not covered by $\mathcal{C}$, but is not $\text{SF}_{\mathcal{C}}$-reachable from s = F6 since the subgoal at E12 covers a shortest F6-D13 path. In order to ensure that it scans only $\text{SF}_{\mathcal{C}}$-reachable cells from s, $\text{SF}^{\rightarrow}$-Connect terminates some of its cardinal scans early by following two simple rules. Rule 1: When performing cardinal scans in direction $\vec{c}$ during a diagonal-first scan in direction $\vec{d}$, the ith cardinal scan cannot extend further than any of the previous cardinal scans (line 2), where the cardinal scan from s in direction $\vec{c}$ is assumed to be the 0th scan. Rule 2: If a cardinal scan finds a subgoal, its length is treated as one cell shorter for limiting the subsequent cardinal scans. $\text{SF}^{\rightarrow}$-Connect implements these two rules by maintaining a "clearance limit" (shown in blue text in Algorithm 14), which is initialized to $\mathbf{C}[s, \vec{c}]$ (minus one if $s + \vec{c} \cdot \mathbf{C}[s, \vec{c}]$ is a subgoal) (lines 9–10), and adjusted after each scan according to the two rules (lines 2,3, and 5). Any clearance value is then truncated to the clearance limit (line 2). We now explain the intuition behind these two rules.

By definition, a cell t is $\text{SF}_{\mathcal{C}}$-reachable from s if and only if all freespace-shortest s-t paths are unblocked (Definition 5.1), and none of these paths are covered by $\mathcal{C}$ (Definition 3.2). Since freespace-shortest s-t paths correspond to different orderings of a given set of diagonal and associated cardinal moves (Theorem 5.1), they form a "parallelogram area", as shown in Figure 5.2. Intuitively, for a cell t to be $\text{SF}_{\mathcal{C}}$-reachable from s, the parallelogram area between them cannot contain blocked cells that block freespace-shortest s-t paths, and cannot contain subgoals that cover freespace-shortest s-t paths. To ensure that it scans only those cells t that are $\text{SF}_{\mathcal{C}}$-reachable from s, $\text{SF}^{\rightarrow}$-Connect therefore has to verify that the parallelogram area between s and t satisfies these criteria. Therefore, it uses Rule 1 to make sure that parallelogram areas from s to scanned cells do not contain blocked cells, and uses both Rules 1 and 2 to make sure that parallelogram areas from s to scanned cells do not contain subgoals that cover freespace-shortest paths from s.

Algorithm 15 SF-Connect.

Input: Start vertex s, goal vertex t

Output: The freespace-diagonal-first s-t path π if π is unblocked; the set of edges $E^+ = \text{SF}_S^{s \rightarrow S} \cup \text{SF}_S^{S \rightarrow t}$ and edge lengths c^+ such that, $\forall (u, v) \in E^+$, $c^+(u, v) = d(u, v)$ otherwise

1: **if** the freespace-diagonal-first s-t path π is unblocked **then**
2: return π
3: $E^+ \leftarrow \emptyset$
4: $D \leftarrow \text{SF}^{\rightarrow}\text{-Connect}(s)$
5: **for all** $n \in D$ **do**
6: Add (s, n) to E^+ with $c^+(s, n) = \text{OctileDistance}(s, n)$
7: $D \leftarrow \text{SF}^{\rightarrow}\text{-Connect}(t)$
8: **for all** $n \in D$ **do**
9: Add (n, t) to E^+ with $c^+(n, t) = \text{OctileDistance}(n, t)$
10: return E^+, c^+

Although SF$^{\rightarrow}$-Connect can identify the set of SF$_{\mathcal{C}}$-reachable subgoals from s exactly, it cannot determine if the goal vertex t is SF$_{\mathcal{C}}$-reachable from s, due to its reliance on *precomputed* clearance values. SF-Connect (Algorithm 15) accounts for this shortcoming of SF$^{\rightarrow}$-Connect by first checking if the freespace-diagonal-first s-t path π is unblocked (line 1). If so, it returns π as a shortest s-t path (line 2). Otherwise, it has determined that t cannot be SF-reachable from s since a freespace-shortest s-t path is blocked, and proceeds to execute SF$^{\rightarrow}$-Connect from s and from t and returns the corresponding edges (lines 3–10).

5.3.6 Linear Time Preprocessing

In this section, we prove that subgoal graphs can be constructed in time linear in the size of the underlying grid by using SF$^{\rightarrow}$-Connect. Specifically, we prove that, if the underlying grid has width W and height H, $\mathcal{C}$ can be identified in $O(WH)$ time (Lemma 5.7), cardinal clearance values can be computed in $O(WH)$ time (Lemma 5.8), SF$^{\rightarrow}$-Connect can be executed from every cell $n \in \mathcal{C}$ to identify the edges of $G_{\mathcal{C}}$ in $O(WH)$ time (Lemma 5.9), and, therefore, $G_{\mathcal{C}}$ can be constructed in $O(WH)$ time (Theorem 5.10). This result also proves that the memory required to store all cardinal clearance values and $G_{\mathcal{C}}$ is $O(WH)$ as well, since the auxiliary information is generated by an $O(WH)$ time preprocessing operation and thus can be stored using $O(WH)$ memory.

Lemma 5.7. *$\mathcal{C}$ can be identified in $O(WH)$ time.*

Proof. For each cell, we can determine whether it is a convex corner cell in constant time, since, by Definition 5.2, we only need to consider its eight surrounding cells. To compute $\mathcal{C}$, we then only need to iterate over the $O(WH)$ cells once and perform a constant-time operation for each cell. □

Lemma 5.8. *All four cardinal clearance values for all unblocked cells can be computed in $O(WH)$ time.*

Proof. The Left clearance values for all unblocked cells can be computed by scanning each row of the grid from right to left once, while counting the number of consecutive unblocked cells visited since the last time a blocked cell or a convex corner cell is visited. This operation visits each cell once, and requires constant time per visited cell (assigning a clearance value to that cell and either incrementing or resetting the current count of unblocked cells). Therefore, it requires $O(WH)$ time. The Right, Up, and Down clearance values can be computed similarly by scanning the grid in different directions. □

Lemma 5.9. *The edges of $G_{\mathcal{C}}$ can be computed in $O(WH)$ time.*

Proof. To identify the edges of $G_{\mathcal{C}}$, we can run SF$^{\rightarrow}$-Connect (Algorithm 14) once from each $s \in \mathcal{C}$. We prove that the total execution time of running SF$^{\rightarrow}$-Connect from each $s \in \mathcal{C}$ is $O(WH)$.

The cardinal scans (lines 20–21) from each $s \in \mathcal{C}$ in all four cardinal directions $\vec{c}$ require $O(WH)$ time since SF$^{\rightarrow}$-Connect performs four constant-time clearance-value look-ups from each $s \in \mathcal{C} \subseteq \mathcal{G}$.

Consider two diagonal-first scans (lines 22–23) from any two different cells $u, v \in \mathcal{C}$ in the same diagonal direction $\vec{d}$. The diagonals iterated over by these two diagonal-first scans cannot overlap, since a diagonal-first scan terminates once it reaches a cell $n \in \mathcal{C}$ (lines 13–15). For instance, if the diagonal-first scan from u in direction $\vec{d}$ reaches $v \in \mathcal{C}$, it stops before iterating over the cells that the diagonal-first scan from v in direction $\vec{d}$ iterates over. Therefore, each one of the $O(WH)$ cells can be iterated over by at most one diagonal-first scan from a vertex $n \in \mathcal{C}$ in a given diagonal direction $\vec{d}$. Since SF$^{\rightarrow}$-Connect processes each visited cell it iterates over during a diagonal-first scan in constant time, by performing two cardinal clearance-value look-ups and updating clearance limits accordingly, the total time to perform all diagonal-first scans from all $s \in \mathcal{C}$ in direction $\vec{d}$ is $O(WH)$. Since there are only four diagonal directions, the total time to perform all diagonal-first scans from all $s \in \mathcal{C}$ is $O(WH)$.

□

Theorem 5.10. *$G_{\mathcal{C}}$ can be computed in $O(WH)$ time. All cardinal clearance values and $G_{\mathcal{C}}$ can be stored using $O(WH)$ memory.*

5.4 Jump-Point Graphs

In this section, we examine the similarities between subgoal graphs on grid graphs and jump-point search (Harabor & Grastien, 2011; Harabor et al., 2014), and show that jump-point search can be understood as a search on a subgoal graph constructed on the *direction-extended canonical grid graph.*

5.4.1 Jump-Point Search

As described in Section 2.3.3, jump-point search is an online (that is, does not perform preprocessing) path-planning algorithm that is specialized for grid graphs, uses a diagonal-first canonical ordering to consider only a small number of successors per expanded vertex, and performs "jumps" between "jump points" to expand only a small number of vertices. In this section, we describe the operation of jump-point search using our terminology.

Jump-point search only explores *diagonal-first* paths, that is, taut paths where no cardinal-to-diagonal turn can be replaced with a diagonal-to-cardinal turn (Harabor & Grastien, 2011, Definition 4). In this dissertation, we refer to such paths as *locally-diagonal-first* paths, since the canonicity constraints on these paths are defined "locally" on their 2-subpaths. To clarify, locally-diagonal-first paths differ from freespace-diagonal-first paths as follows: 1) Freespace-diagonal-first paths are shortest paths on $\mathcal{F}$ (that is, they are freespace shortest). Locally-diagonal-paths are not necessarily shortest paths on $\mathcal{F}$ or G. However, they are taut paths on G, that is, their 2-subpaths are shortest paths on G. 2) Freespace-diagonal-first paths are the lexically smallest shortest paths on $\mathcal{F}$. Locally-diagonal-first paths are not necessarily the lexically smallest paths on G or $\mathcal{F}$. However, all their 2-subpaths are lexically smallest shortest paths on G. 3) Between every pair of cells on G, there exists at most one freespace-diagonal-first path (if it is not blocked) but there might exists multiple locally-diagonal-first paths. In fact, as we prove in Lemma 5.11, between every pair of cells s an t on G, at least one shortest s-t path on

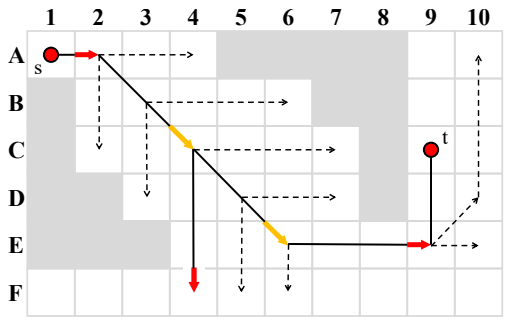


Figure 5.11: Operation of jump-point search. Straight jump points are shown as red arrows, and diagonal jump points are shown as orange arrows. Solid lines show freespace-diagonal-first paths that generated jump points, and dashed lines show freespace-diagonal-first paths that are scanned but do not generate jump points.

G is guaranteed to be locally-diagonal-first. Therefore, jump-point search is guaranteed to find shortest paths even though it only explores locally-diagonal-first paths.

Lemma 5.11. *For every shortest s-t path π on G, there exists a locally-diagonal-first shortest s-t path π' on G with $l(\pi) = l(\pi')$.*

Proof. Since π is a shortest path, it is taut. If π is not locally-diagonal-first, then there must be a cardinal-to-diagonal turn on π that can be replaced with a diagonal-to-cardinal turn that produces an unblocked path (that is lexically smaller than the cardinal-to-diagonal turn with respect to the diagonal-first canonical ordering). Recursively replacing such cardinal-to-diagonal turns in π with diagonal-to-cardinal turns until no such cardinal-to-diagonal turn remains results in a path π' with $l(\pi) = l(\pi')$ that is locally-diagonal-first. □

Jump-point search explores locally-diagonal-first paths by combining freespace-diagonal-first paths. When jump-point search expands the start vertex, it scans freespace-diagonal-first paths that extend from it to identify its *jump-point* successors, and inserts them into the OPEN list. Each jump point is associated with an *incoming direction*, which is the direction of the last move along the freespace-diagonal-first path that is used to reach that jump point. When jump-point search expands a jump-point, it scans freespace-diagonal-first paths that extend from the jump-point to identify its jump-point successors, but only in directions that form locally-diagonal-first 2-subpaths with the incoming direction of the jump point. Jump-point search distinguishes between straight and diagonal (intermediate) jump points. Straight jump points are always placed at convex corners cells, where the diagonal-first freespace path from the expanded vertex can be extended with a cardinal-to-cardinal turn that is taut (which is referred to as a "forced neighbor"). Diagonal jump points can be placed at any unblocked cell on the grid, and are only placed at a cell if there is a straight jump point that can be reached from that cell with a sequence of moves in a cardinal direction $\vec{c}$ that is associated with the incoming direction $\vec{d}$ of the diagonal jump point.

Figure 5.11 shows an example of the operation of jump-point search. When the start vertex s = A1 is expanded, a straight jump point (A2, Right) is identified as its successor, since the path ⟨A1, A2, B2⟩ forms a cardinal-to-cardinal turn that is taut. When the jump point (A2, Right) is expanded, jump-point search scans the freespace-diagonal-first paths that extend from A2 in directions Down, Right, and Down-Right, which form locally-diagonal-first 2-subpaths with the incoming direction Right of (A2, Right). The scan terminates when it generates (F4, Down) as a straight jump point (due to the path $\langle E4, F4, F3\rangle$ forming a taut cardinal-to-cardinal turn) and generates (C4, Down-Right) as a diagonal jump point that can reach (F4, Down) with a sequence of Down moves. Jump-point search proceeds by expanding (C4, Down-Right) to generate the straight jump point (E9, Right) and the intermediate diagonal jump point (E6, Down-Right), and finally expanding (E9, Right) to generate the goal vertex t = C9. When (E9, Right) is expanded, the freespace-diagonal-first paths that extend from E9 in direction Down-Right are not scanned. This is so, because, although Down-Right forms a taut turn with the incoming direction Right of (E9, Right) at E9, namely the turn ⟨E8, E9, F10⟩, it does not form a locally-diagonal-first 2-subpath, since it can be replaced with the diagonal-first path ⟨E8, F9, F10⟩.

So far, we have explained the operation of the offline version of jump-point search, abbreviated as JPS (Harabor & Grastien, 2011). After our publication of subgoal graphs on grid graphs (Uras, Koenig, & Hernandez, 2013), Harabor et al. (Harabor et al., 2014) introduced a preprocessing-based variant of jump-point search, called JPS+, that uses clearance values, similar to SF-Connect, to speed up the scanning of the grid for jump point successors of expanded vertices, and a variant called JPS+P that uses only straight jump points, which, as described earlier, are placed at the convex corner cells. JPS+P makes the similarities between jump-point search and answering queries using subgoal graphs clearer: both algorithms consider convex corner cells as "important" vertices and use clearance values to efficiently scan the grid for freespace-diagonal-first paths. We aim to formally characterize their similarities and differences in the following sections, by introducing direction-extended canonical grid graphs. We now describe JPS+ and JPS+P in further detail.

Figure 5.12 shows the operation of JPS+P. Similar to JPS, JPS+P expands s =A1 to generate the jump point (A2, Right). Unlike JPS, when JPS+ expands (A2, Right), it generates (F4, Down) and (E9, Right) as its successors, without generating (C4, Down-Right) and (E6, Down-Right) as diagonal (intermediate) jump points. The "P" in JPS+P stands for "improved pruning rules" (Harabor et al., 2014) since JPS+P can be considered to be immediately expanding diagonal jump points, effectively pruning them from the search.

JPS+P (and JPS+) uses clearance values, similar to SF-Connect (Algorithm 14), to efficiently scan the grid to generate jump-point successors of expanded vertices (rather than direct-SF-reachable subgoals from the start or the goal), with the following differences: 1) JPS+P does not use a "clearance limit" to limit its scans to $\mathrm{SF}_{\mathcal{C}}$-reachable cells (that is, the blue text in Algorithm 14 is omitted), since doing so would result in JPS+P identifying jump point successors incorrectly. As described in Section 5.3.5, without the clearance limit, SF-Connect explores all freespace-diagonal-first paths that originate at the start vertex, similar to the scans performed during a jump-point search.

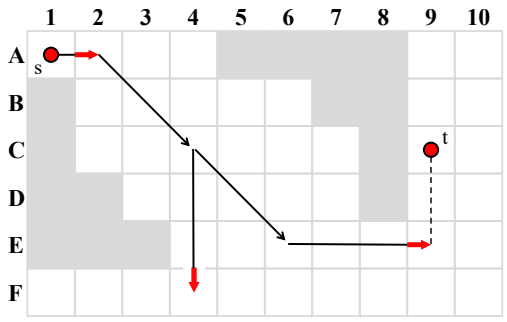


Figure 5.12: Operation of jump-point search using clearance values and without generating diagonal jump points (JPS+P). Solid lines correspond to clearance-value look-ups. Since clearance values do not account for the goal vertex, the goal vertex needs to be detected manually during scans.

2) The clearance values are computed to jump points with matching directions (that is, the clearance value in direction $\vec{c}$ indicates the distance in direction $\vec{c}$ to a jump point with incoming direction $\vec{c}$) rather than to convex corner cells that are closest. 3) Clearance values are stored in diagonal directions as well, to indicate distances to diagonal jump points. This is an optimization that can be used for SF-Connect as well, in order to "jump" along diagonals to rows or columns that contain subgoals. 4) Since *precomputed* clearance values cannot account for the goal vertex, JPS+ has to perform "target detection" during its scans to make sure that it does not "jump past" the goal vertex. Specifically, JPS+ checks, for each each diagonal jump, whether it crosses the row or the column of the goal vertex and, if so, whether the goal vertex can be reached by moving in a straight line along that row or column. 5) When scanning for jump-point successors of expanded jump points (and not the start vertex), JPS+P performs scans only in directions that form diagonal-first and taut turns with the direction of the expanded jump point. Figure 5.13 shows the differences in using SF-Connect to connect the start vertex to a subgoal graph and the scan performed by JPS+P to identify the jump-point successors of the start vertex.

5.4.2 Direction-Extended Canonical Grid Graph

Sturtevant and Rabin suggest that jump-point search can be broken down into three components: A best-first search (since jump-point search can be considered as a modification of A* search with further pruning and jumping rules), a canonical ordering, and a jumping policy (Sturtevant & Rabin, 2016). They also introduce a version of jump-point search without the jumping policy, namely an A* search that uses canonical orderings to prune successors, called *canonical A* search*. When expanding a vertex n that is reached with a move in direction $\vec{v}_1$, canonical A* search generates successors only in directions $\vec{v}_2$ such that $\langle \vec{v}_1, n, \vec{v}_2 \rangle$ is locally-diagonal-first. Similar to jump-point search, canonical

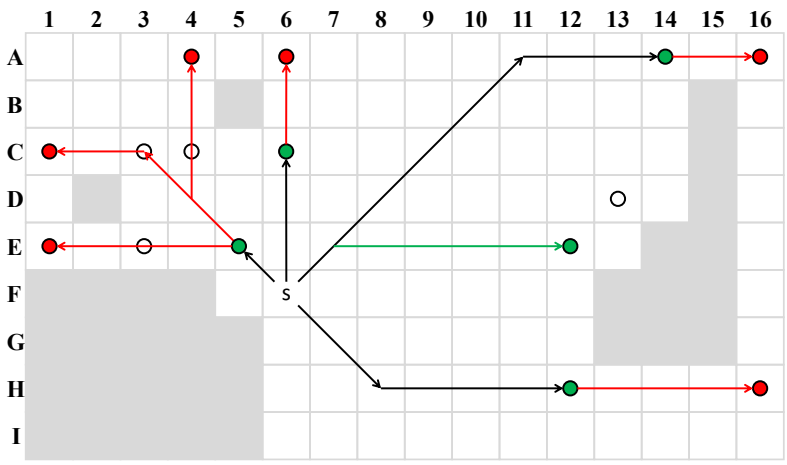


Figure 5.13: Differences between SF-Connect when connecting the start vertex to the subgoal graph and the scanning procedure of jump-point search when generating jump-point successors of the start vertex. Green disk: Subgoal that is direct-SF-reachable from s. Red disk: Cell containing a jump-point successors of s. Black line: Freespace-diagonal-first path explored by both SF-Connect and JPS+P. Green line: Freespace-diagonal-first path explored only by SF-Connect. Red line: Freespace-diagonal-first path explored only by JPS+P.

A* search is guaranteed to find shortest paths, since it does not prune locally-diagonal-first paths, one of which is guaranteed to exist between any given pair of start and goal vertices (Lemma 5.11).

We now formally introduce the *direction-extended canonical grid graph* G^* to characterize the search space of canonical A* search. That is, canonical A* search on G can be considered to be equivalent to a regular A* search on G^*.

Definition 5.3 (Direction-extended canonical grid graph.)**.** *The direction-extended canonical grid graph* $G^* = (V^*, E^*, c^*)$ *of* $G = (V, E, c)$ *is a graph where:*

- *For each vertex* $n \in V$ *and each direction* $\vec{v}$, $(n, \vec{v}) \in V^*$.
- *For each edge* $(n_1, n_2) \in E$ *and each pair of directions* $\vec{v}_1, \vec{v}_2$, $((n_1, \vec{v}_1), (n_2, \vec{v}_2)) \in E^*$ *if and only if:*
  - $n_2 = n_1 + \vec{v}_2$, *and*
  - $\langle \vec{v}_1, n_1, \vec{v}_2 \rangle$ *is locally-diagonal-first.*
- *For each edge* $((n_1, \vec{v}_1), (n_2, \vec{v}_2)) \in E^*$, $c^*((n_1, \vec{v}_1), (n_2, \vec{v}_2))) = c(n_1, n_2)$.

G^* contains eight vertices $(n, \vec{v})$ for each vertex n in G, annotated with one of the four cardinal or four diagonal directions $\vec{v}$ that encodes the incoming direction into n (that is, G^* is direction-extended). For every vertex on G^*, every pairing of an in-edge and an out-edge of that vertex form a path that corresponds to a locally-diagonal-first

path on G (that is, G^* is canonical). As a result, every path on G^* corresponds to a locally-diagonal-first path on G, since its 2-subpaths correspond to locally-diagonal-first paths on G. Furthermore, for every locally-diagonal-first path π on G, there exists a corresponding path π^* on G^*:

Lemma 5.12. *Let $\pi = \langle n_0, \ldots, n_k \rangle$ be a locally-diagonal-first path on G. Then, there exist $\vec{v}_0, \ldots, \vec{v}_k$ such that $\pi^* = \langle (n_0, v_0), \ldots, (n_k, v_k) \rangle$ is a path on G^*.*

Proof. The proof follows from Definition 5.3 by selecting, for $i = 1, \ldots, k$, $\vec{v}_i$ as the direction that satisfies $n_{i-1} + \vec{v}_i = n_i$, and selecting $\vec{v}_0 = \vec{v}_1$. □

For every $s, t \in V$, at least one shortest s-t path on G is locally-diagonal-first (Lemma 5.11), and this path is preserved in G^* (Lemma 5.12). Therefore, G^* can be used to find shortest s-t paths on G, by treating $(s, \vec{v})$ as start vertices and $(t, \vec{v})$ as goal vertices for all $\vec{v}$. Furthermore, since all paths on G^* correspond to locally-diagonal-first paths on G, searches on G^* can only explore paths that correspond to locally-diagonal-first paths on G, similar to canonical A* search and jump-point search.

5.4.3 Jump-Point Graph

We now show that jump-point search can be understood as a search on a subgoal graph constructed on G^* that uses straight jump points as subgoals. We refer to this subgoal graph as the *jump-point graph*.

We first extend our definitions on G to G^*:

- *Direction-extended canonical freespace grid graph:* We use $\mathcal{F}^*$ to denote the direction-extended canonical freespace grid graph. That is, all paths on $\mathcal{F}^*$ correspond to locally-diagonal-first paths on the freespace grid graph $\mathcal{F}$. On $\mathcal{F}$, between every pair of vertices, there is exactly one locally-diagonal-first path, namely the freespace-diagonal-first one: A locally-diagonal-first path on $\mathcal{F}$ can have at most one turn, namely a diagonal-to-cardinal turn. It cannot have a cardinal-to-diagonal turn, since that turn can always be replaced with a diagonal-to-cardinal turn. The only paths on $\mathcal{F}$ that contain at most one diagonal-to-cardinal turn are freespace-diagonal-first paths. Consequently, on $\mathcal{F}^*$, every s^*-t^* path is the unique s^*-t^* path, and corresponds to the freespace-diagonal-first s-t path, since $\mathcal{F}^*$ preserves only the locally-diagonal-first (freespace-diagonal-first) paths from $\mathcal{F}$.

- *Freespace-reachability:* We use F*-reachability to denote F-reachability on G^*. That is, for every $(s, \vec{v}_1), (t, \vec{v}_2) \in V^*$, $((s, \vec{v}_1), (t, \vec{v}_2)) \in$ F* if and only if the unique $(s, \vec{v}_1)$-$(t, \vec{v}_2)$ path on $\mathcal{F}^*$ is also an (unblocked) path on G^*. (Since, as discussed above, (freespace-)shortest paths on $\mathcal{F}^*$ are unique, CF*- and SF*-reachability on G^* are equivalent to F*-reachability.)

Definition 5.4 formally defines straight jump points as vertices of G^*, and Theorem 5.13 proves that the set of straight jump points form an F*-SPC on G^*. We only

provide a sketch of the proof, which is similar to our proof that the set of convex corner cells is an SF-SPC on G.[1] Figure 5.14a shows all straight jump points on a grid, and illustrates them covering a locally-diagonal-first (but not freespace-diagonal-first) s-t path. Figure 5.14b shows the set of subgoals (on G) as a comparison, and illustrates them covering at least one shortest s-t path.

Definition 5.4. *A vertex $(n, \vec{c}_1)$ of G^* is a straight jump point if and only if there exists $\vec{c}_2 \perp \vec{c}_1$ such that the cell $n - \vec{c}_1 - \vec{c}_2$ is blocked and $n - \vec{c}_1$ and $n - \vec{c}_2$ are unblocked (equivalently, the path $\langle \vec{c}_1, n, -\vec{c}_2 \rangle$ on G is taut).*

Theorem 5.13. *The set of straight jump points form an F*-SPC on G^*.*

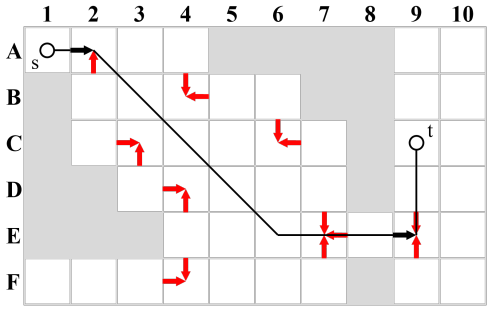
Proof. Sketch: Let π^* be any shortest s^*-t^* path on G^*, for some $s^* = (s, \vec{v}_1)$ and $t^* = (t, \vec{v}_2)$. By Definition 5.3, π^* corresponds to a locally-diagonal-first path on G. We show that, if $(s^*, t^*) \notin \mathrm{F}^*$, then π^* is covered by a straight jump point. For contradiction, assume that $(s^*, t^*) \notin \mathrm{F}^*$ and that π^* is not covered by a straight jump point. π^* cannot have a cardinal-to-cardinal turn because, otherwise, it is either non-taut or it is covered by a straight jump point. π^* cannot have a diagonal-to-diagonal turn because, otherwise, it is non-taut. π^* cannot have a cardinal-to-diagonal turn because, otherwise, it is either non-locally-diagonal-first or it is covered by a straight jump point. Therefore, π^* can only have diagonal-to-cardinal turns. There is only one such possible π^*, namely the one that corresponds to the freespace-diagonal-first s-t path. But then, the freespace-diagonal-first s^*-t^* path is unblocked on G^*, contradicting that $(s^*, t^*) \notin \mathrm{F}^*$.

□

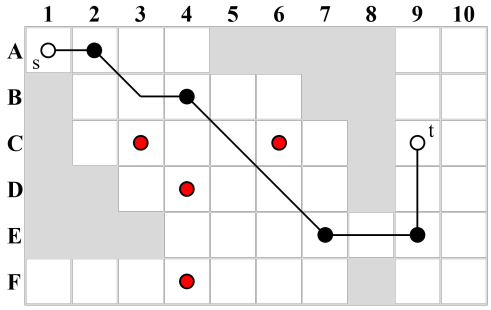
Theorem 5.13 suggests that we can construct an F* subgoal graph on G^* by using straight jump points as subgoals. We refer to this subgoal graph as the *jump-point graph*. Jump-point graphs can be used to answer queries on G as follows:

- **Connect:** The start vertex is connected to its jump-point successors, similar to jump-point search. The goal vertex, on the other hand, is connected to all jump-points from which the freespace-diagonal-first path to the goal vertex is unblocked, using a different connection algorithm. Namely, the algorithm that connects the goal vertex to the jump-point graph scans the freespace-diagonal-first paths that *terminate* at the goal vertex (equivalently, freespace-*cardinal*-first paths that *originate* from the goal vertex), by first scanning cardinally from the goal vertex and then branching off in diagonal directions (rather than first scanning diagonally and then branching off in cardinal directions, like SF-Connect). This scanning can be sped up by using a second set of clearance values in diagonal directions (similar to how SF-Connect can be sped up by using cardinal clearance values). We refer to the connection algorithm used to connect both the start and goal vertices to jump-point graphs as JP-Connect.

[1] G^* does not satisfy Assumption 1.2, that is, there might not be any paths between some pairs of vertices on G^*. For instance, in Figure 5.14a, no (locally-diagonal-first) path exists from (A2, Up) to (E9, Up). For our proof, we assume that a shortest path cover does not need to cover paths between such pairs of vertices.

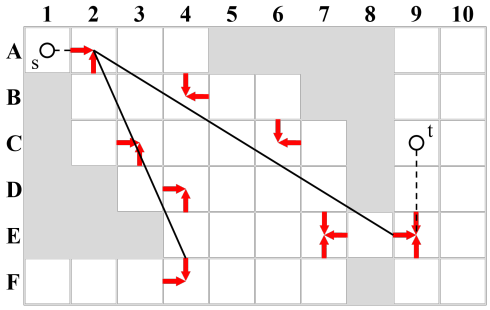


(a) Straight jump points (arrows) cover every s-t path on G^* that is not freespace-diagonal-first (that is, if $(s,t) \notin F^*$).

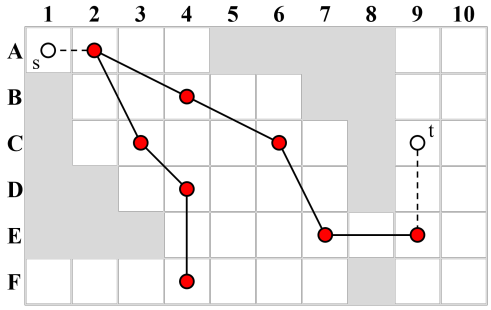


(b) Subgoals (disks) cover at least one shortest s-t path on G if $(s,t) \notin F$.

Figure 5.14: The set of straight jump points and the set of subgoals (convex corner cells) are F*-SPCs and F-SPCs on G^* and G, respectively.



(a) Search tree of a search on a (query) jump-point graph.



(b) Search tree of a search on a (query) subgoal graph.

Figure 5.15: Search trees of searches over (query) jump-point and subgoal graphs.

- **Search and refine:** Once the start and goal vertices are connected to the jump-point graph, the resulting *query* jump-point graph can be searched for a shortest path, which can then be refined into a path on G by replacing its edges with the corresponding freespace-diagonal-first path on G. We refer to the refinement algorithm used for paths found on jump-point graphs as JP-Refine. JP-Refine and CF-Refine are very similar, except that JP-Refine always finds freespace-diagonal-first paths, whereas, as discussed in Section 5.3.2, CF-Refine may find freespace-diagonal-first or freespace-cardinal-first paths, since we use the symmetric canonical ordering. (This distinction does not matter for answering queries using subgoal graphs, but will be relevant when we use CF-reachability to augment contraction hierarchies in Section 5.5.)

Figure 5.15 shows the search trees of searches over (query) jump-point and subgoal graphs. Although the jump-point graph has more vertices than the subgoal graph (two or more jump points at each convex corner cell rather than exactly one subgoal), the search on the jump-point graph expands fewer vertices, since it only considers jump

points $(n, \vec{c})$ that can be reached from other jump points (or the start vertex) through freespace-diagonal-first paths whose last move is in direction $\vec{c}$. We will refer back to this example when discussing our experimental results in Section 5.5.5.

Compared to jump-point search, answering queries using jump-point graphs has the following benefits: 1) The grid is scanned at most twice for connecting the start and goal vertices to the jump-point graph, rather than once for each expansion. 2) Since the jump-point graph is constructed explicitly, it can be combined easily with orthogonal techniques, such as contraction hierarchies. We explore this idea further in Section 5.5.5. However, unlike jump-point search that (optionally) uses precomputed clearance values only, answering queries using jump-point graphs requires explicitly storing the jump-point graph, as well as (optionally) *two* sets of precomputed clearance values, one for connecting the start vertex and one for connecting the goal vertex to the jump-point graph.

5.5 Experimental Evaluation

In this section, we experimentally evaluate answering queries using subgoal graphs, contraction hierarchies, their combinations, and their variants on grid graphs, with respect to their query times and memory requirements. We consider three different ways of augmenting contraction hierarchies with reachability relations: We have already described two of these augmentations in Section 3.4.2, namely constructing contraction hierarchies on subgoal graphs, or restricting the edges of contraction hierarchies to be R-reachable (R contraction hierarchies). The third augmentation is a simple modification that uses R-Refine rather than unpacking to refine R-reachable edges of contraction hierarchies.

This section is organized as follows: In Section 5.5.1, we provide an overview of the algorithms that we evaluate in this section and discuss implementation details. In Section 5.5.2, we describe the benchmark grids and problem instances that we use in our experimental evaluation. In Section 5.5.3, we describe our method of presenting the results. In Sections 5.5.4-5.5.5, we experimentally evaluate answering queries using subgoal graphs, jump-point graphs, and contraction hierarchies. In Sections 5.5.7 and 5.5.11, we experimentally evaluate answering queries using combinations and variants of subgoal graphs, jump-point graphs, and contraction hierarchies. In Section 5.5.12, we compare our algorithms against state-of-the-art path-planning algorithms on grid graphs that have been evaluated in the Grid-Based Path-Planning Competition (GPPC).

5.5.1 Algorithms and Implementation Details

We compare various algorithms that answer queries using subgoal graphs, contraction hierarchies, their combinations, and their variants. Table 5.1 provides a summary of these algorithms and how their connection, search, and refinement phases differ from one another. Figure 5.16 provides a summary of how these algorithms relate to each other. We now describe these algorithms and discuss implementation details.

- **A***: We use the Octile distance heuristic for our A* searches, and use a binary heap as the priority queue. We implement the searches for all other algorithms similarly, and preallocate memory for all required data structures and values maintained by the searches so that there is no memory allocation during a search.

Algorithm	Connect	Search	Refine
A*	-	A*	-
SG	SF-Connect	A*	CF-refine
JP	JP-Connect	A* (grouping)	JP-refine
CH-GPPC	-	Bidir. Dij.	Midpoint
CH	-	Bidir. A*	2-pointer
CH-SG	SF-Connect	Bidir. A*	2-pointer then CF-refine
CH-JP	JP-Connect	Bidir. A*	2-pointer then JP-refine
CH+Rr	-	Bidir. A*	R-refine or 2-pointer
CH-SG+Rr	SF-Connect	Bidir. A*	R-refine, or 2-pointer then CF-refine
R-CH	-	Bidir. A* (C)	R-refine
R-CH-SG	SF-Connect	Bidir. A* (C)	R-refine
R-N-SG	SF-Connect	Bidir. A* (CL)	R-refine

Table 5.1: Algorithms on grid graphs evaluated in this dissertation. "C" indicates that forward searches of bidirectional searches generate successors of expanded vertices using core edges. "L" indicates that forward searches of bidirectional searches use non-core same-level edges to check if the two searches meet.

- **SG**: SG constructs the subgoal graph $G_{\mathcal{C}}$ on G during preprocessing and uses it to answer queries. It uses precomputed clearance values from unblocked cells in all four cardinal directions, as explained in Sections 5.3.5 and 5.3.6.

  We store clearance values in a 1D array that can be accessed by linearizing the x and y coordinates of a cell (that is, the linearized coordinates l of the cell (x, y) is computed as $x + y \cdot W$, where y is the width of the grid). We store each entry in this array as a 4 byte integer, each byte representing a clearance value in one of the four cardinal directions. For clearance values above 255 (the highest value that can be represented using 1 byte), we simply store 255 as the clearance value. When looking up the clearance value $\mathbf{C}[n, \vec{c}]$ of a cell n in direction $\vec{c}$, if the stored clearance value is 255, we also look up the clearance value $\mathbf{C}[n + 255 \cdot \vec{c}, \vec{c}]$, and repeat this process until a cell $n' = n + k \cdot 255 \cdot \vec{c}$ is reached with $\mathbf{C}[n', \vec{c}] < 255$. The clearance value of n in direction $\vec{c}$ is then calculated as $255 \cdot k + \mathbf{C}[n', \vec{c}]$.

  Since all edges of subgoal graphs are F-reachable, their lengths can be calculated in constant time as the Octile distances between the vertices they connect, as described in Section 5.3.1. Therefore, we avoid storing the edge lengths and store each edge using 4 bytes only to indicate its destination.

- **JP**: JP constructs a jump-point graph on G^* during preprocessing and uses it to answer queries. It stores two sets of clearance values, a set of cardinal clearance values, similar to our SG implementation, to connect the start vertices to the jump-point graphs, and a set of diagonal clearance values to connect the goal vertices to the jump-point graphs. It therefore requires twice the memory to store the clearance values compared to SG.

Jump-point search has an optimization that we have not mentioned in Section 5.4.1. Namely, it expands at most one jump-point in each cell, the one that has the smallest g-value in that cell. All other jump-points in that cell are discarded, even if their g-values are the same as the g-value of the expanded one. We have included this optimization in JP as well. To the best of our knowledge, there is no formal proof that jump-point search with this optimization is still guaranteed to find shortest paths, but our JP implementation using this optimization has always found the shortest path in our experiments. We also optimize our JP implementation to store the g-values for cells rather than jump points, and preallocate memory for our data structures assuming that we are searching over the convex corner cells, rather than the jump points. That is, the jump-point graph is essentially used as a look-up table to determine the successors of the expanded "convex corner cells" based on their incoming directions.

- **CH and CH-GPPC**: CH constructs a contraction hierarchy on G during preprocessing and uses it to answer queries. It uses a bidirectional A* search, as described in Section 3.4.4, to search its contraction hierarchy for a shortest up-down path, and uses 2-pointer unpacking, as described in Section 3.4.4, to refine the path found into a shortest path on G. CH-GPPC instead uses bidirectional Dijkstra searches and midpointer unpacking.

  Our CH-GPPC implementation closely follows the implementation of the contraction hierarchy entry in GPPC. Namely: 1) It uses the same online top-down ordering for constructing the hierarchy, as described in Section 3.4.3.3. 2) It uses 12 bytes to store each edge, namely 4 bytes each for its destination, length, and midpoint (for midpoint unpacking). 3) It does not store downward edges since, in contraction hierarchies constructed on undirected graphs, there exists an upward edge (u, v) for every downward edge (v, u) and vice versa. 4) It uses stall-on-demand during searches. 5) It does not use a heuristic in searches. Our preliminary results show that the query times of our CH-GPPC implementation and the contraction hierarchy entry in the GPPC are very similar. Our implementation has shorter preprocessing times (almost by a factor of two) since it exploits the fact that contraction hierarchies on undirected graphs are also undirected, and performs only half of the witness searches.

  Our CH implementation differs from our CH-GPPC implementation in the following ways: 1) CH performs 2-pointer unpacking. 2) CH uses 16 bytes to store each edge, namely 4 bytes for its destination, 4 bytes for its length, and 8 bytes for two pointers to its unpacked edges. 3) CH uses the Octile distance heuristic during searches. All of our algorithms that use hierarchies follow an implementation similar to our CH implementation unless noted otherwise.

- **CH-SG:** CH-SG constructs a contraction hierarchy on the subgoal graph $G_{\mathcal{C}}$ during preprocessing and uses it to answer queries, as described in Section 3.4.4. CH-SG performs connection using SF-Connect, similar to SG; performs search using bidirectional A* searches, similar to CH; and performs refinement first by using 2-pointer unpacking, similar to CH, to refine the path found on the contraction

hierarchy to a path on the subgoal graph, and then by using CF-Refine, similar to SG, to refine it to a path on G. CH-SG stores clearance values in the four cardinal directions, similar to SG; and stores each edge using 16 bytes, similar to CH.

- **CH-JP:** CH-JP constructs a contraction hierarchy on the jump-point graph during preprocessing and uses it to answer queries. CH-JP operates similarly to CH-SG, with the following exceptions. Unlike CH-SG, but similarly to JP, CH-JP stores two sets of clearance values, uses JP-Connect for connection, and JP-Refine for refining paths on the jump-point graph to paths on G. Unlike CH and CH-SG, CH-JP stores upward edges and "reversed downward edges" of contraction hierarchies separately since jump-point graphs are directed graphs and, as a result, these two sets of edges are not necessarily equivalent. Furthermore, unlike JP, CH-JP treats jump points as individual vertices, rather than "grouping them" based on their locations. Our preliminary experiments have shown that, otherwise, CH-JP can find paths that are not shortest paths. We discuss further implementation details in Section 5.5.8.

- **CH+Rr and CH-SG+Rr:** CH+Rr (contraction hierarchies + R-refine) and CH-SG+Rr operate similarly to CH and CH-SG, respectively, but also "mark" shortcut edges that are R-reachable during preprocessing. During the refinement phases of queries, they refine the "marked" shortcuts using an R-refine algorithm instead of 2-pointer unpacking. Unmarked shortcut edges are refined using 2-pointer unpacking as usual. Whereas CH+Fr and CH-SG+Fr mark F-reachable shortcuts and refine them using F-Refine, CH+CFr and CH-SG+CFr mark CF-reachable shortcuts and refine them using CF-Refine. We do not perform experiments with CH+SFr and CH-SG+SFr since SF-Refine is implemented similarly to CF-Refine, and since a smaller number of shortcuts can be marked as SF-reachable rather than CF-reachable (SF $\subseteq$ CF). We mark each shortcut by storing a unique 8 byte value instead of the two pointers to its unpacked edges, and therefore do not use extra memory for marking shortcuts.

- **R-CH and R-CH-SG:** R-CH and R-CH-SG operate similarly to CH and CH-SG, respectively, but construct R contraction hierarchies instead of contraction hierarchies. The forward searches of their bidirectional searches also generate successors of expanded vertices that are reached by level edges, as described in Section 3.4.4. Since all their shortcuts are R-reachable, they refine shortcuts using R-refine rather than 2-pointer unpacking and store each edge using 4 bytes, similar to SG. Similar to CH+Rr, CH-SG+Rr, we use F-reachability and CF-reachability as R, but not SF-reachability.

- **R-N-SG** R-N-SG operates similarly to R-CH-SG, but constructs R N-level subgoal graphs instead of R contraction hierarchies. The forward searches of its bidirectional searches also use non-core same-level edges to check if the two searches meet, as described in Section 3.4.4. Similar to R-CH-SG, we use F-reachability and CF-reachability as R, but not SF-reachability.

Figure 5.16 provides a summary of the relationships between different algorithms and their associated trade-offs, and serves as a roadmap for our experiments. Namely: 1) In

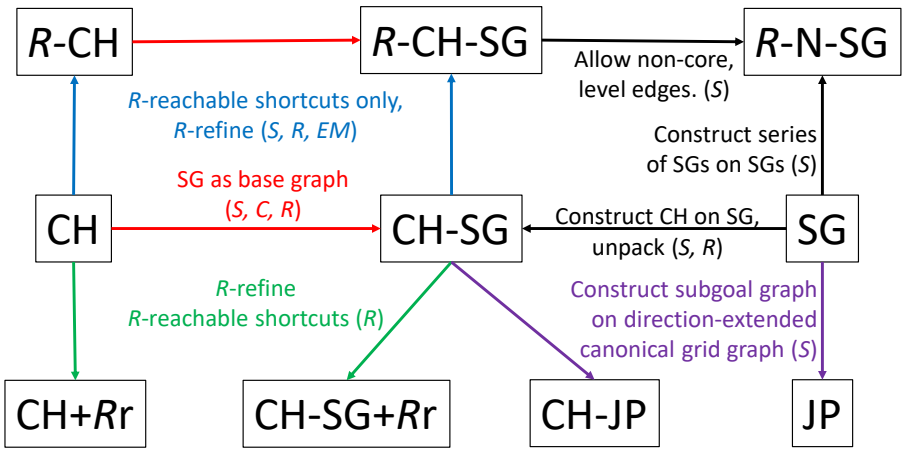


Figure 5.16: Relationships between variants and combinations of subgoal graphs, jump-point graphs, and contraction hierarchies. S: Structural differences in the graph/hierarchy can result in different query-time/memory trade-offs. C: Implementation of R-connect can affect the query-time/memory trade-off. R: Implementation of R-refine can affect the query-time/memory trade-off. EM: Memory required to store each edge can vary depending on R; unpacking information does not need to be stored for shortcuts.

Sections 5.5.4 and 5.5.6, we experimentally evaluate the query-time/memory trade-offs of SG and CH, respectively. Observe that these algorithms have no incoming arrows in Figure 5.16. 2) In Sections 5.5.7, 5.5.9, and 5.5.10, we experimentally evaluate the query-time/memory trade-offs associated with three different ways of augmenting contraction hierarchies with reachability relations, which are marked with red, green, and blue arrows in Figure 5.16, respectively. The purpose of this set of experiments is to demonstrate that it is possible to "specialize" contraction hierarchies to different types of graphs through reachability relations to exploit structure in them. 3) In Section 5.5.11, we experimentally evaluate the query-time/memory trade-off of R-N-SG compared to R-CH-SG. The purpose of this set of experiments, in conjunction with the previous set of experiments, is to systematically study the differences between CH and F-N-SG through the sequence of intermediate algorithms CH-SG and F-CH-SG. 4) In Sections 5.5.5 and 5.5.8, we experimentally evaluate the query-time/memory trade off of JP and CH-JP, relative to SG and CH-SG, respectively, which are marked as purple arrows in Figure 5.16. We discuss the query-time/memory trade-offs associated with each arrow in Figure 5.16 in the relevant sections, in conjunction with experimental results on grid graphs.

5.5.2 Benchmarks

We use two sets of benchmarks in our experiments: Nathan Sturtevant's benchmark grid maps and problem instances (Sturtevant, 2012a), which we refer to as the *MovingAI benchmarks*, and GPPC grid maps and problem instances (Sturtevant et al., 2015), which we refer to as the *GPPC benchmarks*. We use the MovingAI benchmarks for most of our experiments since they contain significantly more maps and instances than the GPPC

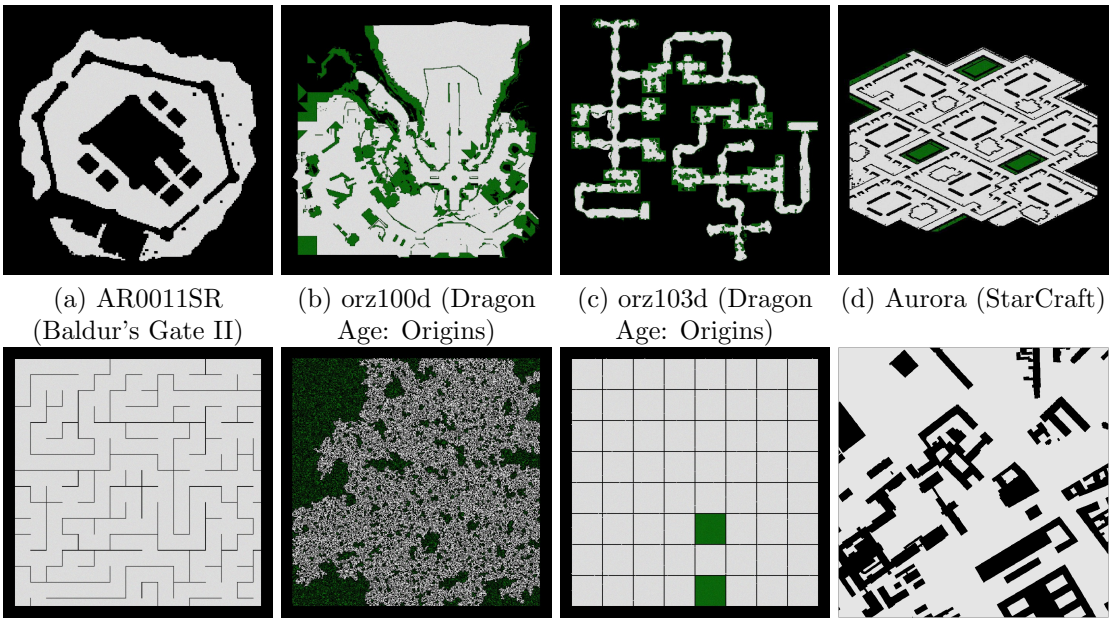
benchmarks, and use the GPPC benchmarks in Section 5.5.12 in order to compare our algorithms to other GPPC entries. We describe the MovingAI benchmarks below, and describe the GPPC benchmarks in Section 5.5.12.

The maps in the MovingAI benchmarks can be divided into 5 main categories (game, maze, random, room, and street), which can further be subdivided into a total of 26 subcategories, as summarized below:

- **Game:** The game maps are a collection of maps from different video games.
  - **bg:** 120 maps from the video game Baldur's Gate II, ranging in size from 50 $\times$ 50 to 320 $\times$ 320.
  - **bg-512:** 75 maps from the video game Baldur's Gate II, all scaled to size 512 $\times$ 512.
  - **dao:** 156 maps from the video game Dragon Age: Origins, ranging in size from 22 $\times$ 28 to 1260 $\times$ 1104.
  - **da2:** 67 maps from the video game Dragon Age 2, ranging in size from 37 $\times$ 37 to 770 $\times$ 770.
  - **sc:** 75 maps from the video game Starcraft II, ranging in size from 384 $\times$ 384 to 1024 $\times$ 1024.
  - **wc3-512** 36 maps from the video game Warcraft III, all scaled to size 512 $\times$ 512.
- **Maze:** The maze maps are artificially generated maps of size 512 $\times$ 512, with varying corridor widths of 1, 2, 4, 8, 16, and 32. There are 10 maps for each of the 6 corridor widths, for a total of 60 maps. We refer to maze maps with corridor width X as maze-X maps.
- **Random:** The random maps are artificially generated by randomly blocking a percentage of cells in a 512 $\times$ 512 grid. The percentage of blocked cells varies from 10 to 40 in increments of 5. There are 10 maps for each of the 7 percentages of blocked cells, for a total of 70 maps. We refer to random maps with X percent blocked cells as random-X maps.
- **Room:** The room maps are artificially generated by dividing a 512 $\times$ 512 grid into square rooms of fixed size, and randomly adding doors between adjacent rooms. The room sizes in room maps are 8 $\times$ 8, 16 $\times$ 16, 32 $\times$ 32, and 64 $\times$ 64. There are 10 maps for each of the 4 room sizes, for a total of 40 maps. We refer to room maps with room size X as room-X maps.
- **Street:** Street maps are generated by discretizing square areas within different cities into 256 $\times$ 256, 512 $\times$ 512, or 1024 $\times$ 1024 grids. There are 30 areas, each discretized into 3 different-sized grids, for a total of 90 maps. For 1024 $\times$ 1024 maps, we use only the first 10 maps rather than all 30, due to high preprocessing times on these maps. We refer to street maps with size $X \times X$ as street-X maps.

	Map	Instances		Path	Avg. G size		A* time
	Count	Count	Avg	length	$\|V\|$	$\|E\|$	(μs)
all	769	1,642,670	2,136	1,137	109,752	776,644	9,898
game-all	529	653,050	1,234	431	64,861	499,013	5,137
maze-all	60	627,000	10,450	2,265	207,941	1,338,316	16,732
random-all	70	155,750	2,225	483	185,864	937,004	5,508
room-all	40	84,350	2,109	422	232,785	1,691,743	8,645
street-all	70	122,520	1,750	454	218,423	1,710,030	6,738
bg	120	40,780	340	142	4,507	32,180	451
bg-512	75	122,600	1,635	361	73,930	574,509	2,359
dao	156	67,200	1,003	281	15,911	117,640	2,373
da2	67	155,620	998	418	21,322	159,548	1,049
sc	75	211,390	2,819	613	263,782	2,040,510	11,540
wc3-512	36	55,460	1,541	318	112,488	867,184	3,030
maze-1	10	145,490	14,549	2,986	131,071	262,140	6,947
maze-2	10	119,110	11,911	2,413	174,517	870,383	13,141
maze-4	10	104,220	10,422	2,150	209,268	1,356,891	18,461
maze-8	10	103,310	10,331	2,133	232,928	1,688,141	21,997
maze-16	10	91,210	9,121	1,877	246,042	1,871,727	24,349
maze-32	10	63,660	6,366	1,300	253,819	1,980,615	23,532
random-10	10	17,970	1,797	360	235,903	1,533,055	2,512
random-15	10	18,610	1,861	372	222,689	1,301,321	3,914
random-20	10	19,150	1,915	383	209,255	1,097,499	5,072
random-25	10	19,900	1,990	398	195,315	918,694	6,048
random-30	10	20,780	2,078	416	180,209	760,078	6,606
random-35	10	22,970	2,297	459	161,313	613,313	7,229
random-40	10	36,370	3,637	752	96,365	335,069	6,023
room-8	10	20,980	2,098	420	206,792	1,301,001	7,994
room-16	10	20,370	2,037	407	231,263	1,663,793	7,839
room-32	10	20,970	2,097	420	243,733	1,855,004	8,847
room-64	10	22,030	2,203	441	249,352	1,947,173	9,820
street-256	30	28,020	934	188	48,012	363,862	842
street-512	30	56,270	1,876	376	196,602	1,531,899	3,467
street-1024	10	38,230	3,823	765	795,122	6,282,926	15,874

Table 5.2: MovingAI benchmarks.



(a) AR0011SR (Baldur's Gate II) (b) orz100d (Dragon Age: Origins) (c) orz103d (Dragon Age: Origins) (d) Aurora (StarCraft)

(e) maze512-32-0 (maze-32) (f) random512-40-0 (random-40) (g) 64room-000 (room-64) (h) Berlin-0-1024 (street-1024)

Figure 5.17: MovingAI grid maps.

Each map in MovingAI benchmarks has an associated set of instances (start and goal pairs) that are uniformly distributed to cover various path lengths. Specifically, on a map where the maximum shortest path length is K, the interval $[0, K]$ is split into buckets $[4k, 4k+4)$ (where k is an integer), each bucket containing 10 instances where the distance between the start and goal vertices is in the range $[4k, 4k + 4)$ (Sturtevant, 2012a). As a result, maps with larger maximum shortest path lengths, such as maze maps, have more instances per map. Table 5.2 reports the total number of maps, the total and average number of instances, the average path length of the instances, the average number of vertices and edges of the corresponding grid graph G, and the average query time of A* searches, over the MovingAI benchmarks. We discuss how the averaging is performed in the next section, along with how the cells are colored in the table.

5.5.3 Result Presentation

We report our experimental results using tables that report statistics across different types of maps in MovingAI benchmarks. We use the following methods to make these tables easier to read:

- **Order of rows:** We organize the rows of the tables so that the first row provides an overview of the results on all maps in the MovingAI benchmarks, the next 5 rows provide an overview of the 5 main categories of maps, and the remaining rows provide results on the 26 subcategories of maps, as shown in Table 5.2. We average statistics that relate to preprocessing (such as preprocessing time, memory, and

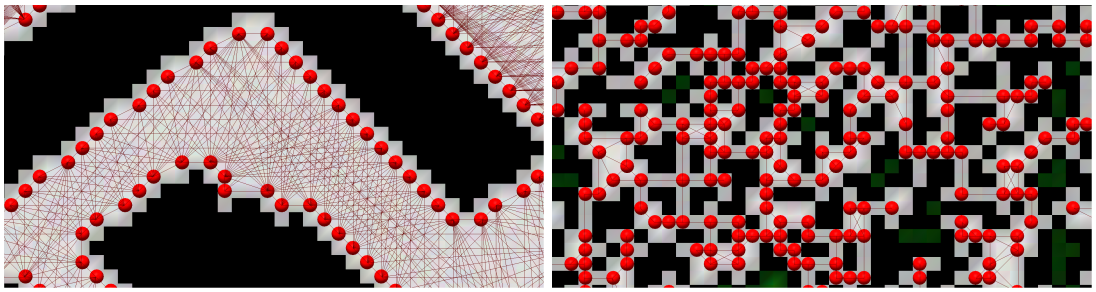
graph size) over all maps in each (main or sub) category, and average statistics that relate to queries (such as query time) over all instances associated with the maps in each category. This averaging strategy is also used in GPPC, and might skew the averages toward maps with larger numbers of instances or toward categories that contain more maps.

- **Abbreviations:** We use the following abbreviations for the various statistics that we report frequently in our tables: $|V|$ (number of vertices), $|E|$ (number of edges), PT (preprocessing time), Cn (connection time), Sr (search time), Rf (refinement time), DQ (distance query time, that is, the sum of the connection and search times), and PQ (path query time, that is, the sum of the connection, search, and refinement times). We use several other abbreviations which which we describe as necessary.

  When reporting statistics about an algorithm X relative to an algorithm Y, we mark the relevant part of the table with the syntax $Y \rightarrow X$ and mark the relevant columns of the table by adding a "%" or "*" at the end of their abbreviations. We typically use "%" to modify $|V|$ or $|E|$, indicating that we are reporting the number of vertices or edges of the graph used by algorithm X as a percentage of the number of vertices or edges, respectively, of the graph used by algorithm Y. We typically use "*" to modify the other statistics, indicating that we are reporting the *improvement factor* of switching from algorithm Y to X. Since shorter preprocessing and query times and lower memory requirements are always better, we calculate the improvement factor for those statistics by dividing the relevant statistic for algorithm Y by the relevant statistic for algorithm X.

- **Colors:** We color cells in our tables based on their values and types, in order to highlight trends in our results and distinguish between the different types of statistics that we report. We color cells in a column that reports statistics about a single algorithm on a scale from white to orange, where the cell with the lowest value in the column is colored white, the cell with the highest value is colored orange, and any other cell is colored in a shade of orange determined based on its value in relation to the lowest and highest values in the column. We color cells in a column that reports percentage statistics on a scale from white to blue, where a cell with value 0% is colored white, a cell with value 200% or higher is colored blue, and any other cell is colored in a shade of blue determined based on its value in relation to 0% and 200%. We color cells in a column that reports improvement factor statistics on a scale from red to white to green, where a cell with improvement factor 1 (no improvement or deterioration) is colored white, a cell with improvement factor 5 or above is colored green, a cell with an improvement factor from 1 to 5 is colored in a shade of green determined based on its value in relation to 1 and 5, and a cell with an improvement factor of less than 1 (that is, deterioration) is colored in a shade of red based on its value in relation to 0 and 1 (where 0 is red).

  The coloring of cells in tables does not convey additional information, and simply aims to make the tables easier to read. To that end, we make a couple of exceptions to the rules we described above. For some improvement factors (such as the query



(a) Staircase pattern (Berlin-0-256.map from street maps).

(b) Random pattern (random512-40-0.map).

Figure 5.18: Staircase and random patterns of subgoal graphs on grid graphs.

times of CH or SG relative to A*), the improvement is much higher than 5, resulting in all cells colored in the same shade of green. In these cases, we use the value 500 rather than 5 for determining the cell colors. We also make an exception when reporting the memory required by SG to store clearance values and edges in Table 5.3, by using the minimum and maximum values that appear in *both* columns to determine the colors of cells in either column, in order to make the two columns comparable.

5.5.4 Subgoal Graphs

In this section, we experimentally evaluate SG (answering queries using subgoal graphs) in terms of its preprocessing and path query times, and memory requirements. We start with a discussion of two types of patterns that we observe on grid graphs, namely the *staircase* and *random patterns*, which can negatively affect SG path query times. Figure 5.18 shows examples of these two patterns.

- **Staircase pattern:** The staircase pattern occurs on grids when the boundaries of obstacles in the environment do not align horizontally or vertically with the grid, as shown in Figure 5.18a. On grids with staircase patters, many cells are convex corner cells, resulting in many subgoals. We informally refer to the subgoals introduced by staircase patterns as *staircase subgoals*. When two staircases face each other, subgoal graphs typically have many edges between staircase subgoals along the two opposite staircases, as shown in Figure 5.18a. We informally refer to these edges as *staircase edges*.

  Most staircase subgoals and edges can be considered to be "unimportant" for answering queries unless the start or goal vertices are tucked away in the crevices of the staircases. That is, if most of the staircase subgoals were removed from the subgoal graph, the remaining subgoals can still be connected to each other through safe-freespace-reachable edges and be used to answer most path queries optimally. However, as discussed in Section 5.3.4, subgoal graphs need to include *all* convex corner cells as subgoals, for them to be useful for answering *all* path queries optimally. We refer to the problem of having to include staircase subgoals and edges

in subgoal graphs as the *staircase problem*. As discussed in Section 5.3.4, the staircase problem can be considered to be a direct consequence of the fact that the grid graphs that we consider in this dissertation exclude corner-cutting diagonal moves. We discuss in Section 5.5.5 how jump-point search avoids the staircase problem, and discuss in Section 5.5.7 how constructing hierarchies on subgoal graphs can mitigate this problem. As an interesting side note, the staircase problem was the motivation for our development of 2-level subgoal graphs (Uras et al., 2013), with the aim of ignoring most of the subgoals and edges along staircases during searches, by labeling such subgoals as level 1 subgoals. We eventually extended 2-level subgoal graphs to N-level subgoal graphs (Uras & Koenig, 2014).

Staircase patterns typically occur on game and street maps, where (the boundaries of) obstacles in the environment do not necessarily align horizontally or vertically with the grid. For instance, in the video games Baldur's Gate II and StarCraft, the viewpoint of the in-game camera is angled to reveal as much detail about the environment as possible, typically resulting in obstacles being aligned diagonally with the grid, as shown in Figures 5.17a and 5.17d, exacerbating the staircase problem. In the video games Dragon Age: Origins and Dragon Age 2, which have been released more recently than Baldur's Gate II and StarCraft, the players can rotate the camera as they please, and therefore maps in these games are typically designed so that obstacles are aligned horizontally or vertically with the grid, as shown in Figure 5.17c. However, maps in these games can still have obstacles that are curved or otherwise irregularly shaped, as shown in Figure 5.17b, resulting in staircase patterns. Street maps contain obstacles (that is, buildings) that are not necessarily aligned with the grid (that is, the geographic coordinate system), also resulting in staircase patterns. Maze and room maps do not have staircase patterns, as all obstacles are aligned horizontally or vertically with the grid, as shown in Figures 5.17f and 5.17g.

- **Random pattern:** The random pattern occurs on random maps, where the blocked cells of grids are scattered randomly around the grid, without forming large, contiguous areas of blocked cells. The random pattern therefore introduces, similar to the staircase pattern, many convex corners and, thus, subgoals, as shown in Figure 5.18b. Typically, random patterns introduce many more subgoals than staircase patterns, but do not introduce as many edges. Random patterns are detrimental to the freespace structure that we aim to exploit with subgoal graphs, since the random placement of blocked cells typically results in only nearby pairs of vertices being F-, CF-, or SF-reachable.

Table 5.3 reports the A* query times (also reported in Table 5.2), the SG preprocessing times (PT, including the time to calculate the clearance values), the numbers of vertices and edges of subgoal graphs relative to G ($|V|$% and $|E|$%), the memory required to the store clearance values and the edges of subgoal graphs (M(C) and M(E)), the SG connection, search, and refinement times (Cn, Sr, Rf), and the SG path query times relative to A* (PQ*).

We make the following observations:

	A*	A* → SG								
	PQ (μs)	PT (μs)	$\|V\|$ %	$\|E\|$ %	M($\mathbf{C}$) (MB)	M(E) (MB)	Cn (μs)	Sr (μs)	Rf (μs)	PQ *
all	9,898	13,898	8.29	7.28	0.78	0.24	5	527	8	18.33
game-all	5,137	6,510	2.19	4.29	0.70	0.08	7	123	2	39.10
maze-all	16,732	18,785	5.56	2.05	1.01	0.10	3	701	15	23.28
random-all	5,508	59,075	39.60	35.41	1.01	1.26	2	2,077	8	2.64
room-all	8,645	14,405	1.82	0.81	1.01	0.05	4	103	2	79.19
street-all	6,738	20,072	1.51	5.61	0.84	0.64	13	118	2	50.46
bg	451	781	9.96	13.38	0.07	0.01	3	39	1	10.28
bg-512	2,359	5,933	0.90	1.42	1.01	0.03	6	34	1	56.52
dao	2,373	3,067	4.20	6.26	0.52	0.03	5	91	2	24.21
da2	1,049	2,464	3.76	4.85	0.58	0.02	4	36	2	25.13
sc	11,540	25,936	2.16	5.08	1.71	0.40	10	265	3	41.54
wc3-512	3,030	8,793	1.06	1.56	1.01	0.05	6	31	1	79.61
maze-1	6,947	29,182	27.60	27.60	1.01	0.27	2	1,694	27	4.03
maze-2	13,141	25,995	12.58	6.58	1.01	0.22	2	1,138	20	11.32
maze-4	18,461	18,284	3.78	1.83	1.01	0.09	3	407	13	43.65
maze-8	21,997	13,784	1.03	0.44	1.01	0.03	3	115	9	174.02
maze-16	24,349	12,939	0.29	0.12	1.01	0.01	3	31	5	603.72
maze-32	23,532	12,528	0.07	0.03	1.01	0.00	4	8	3	1,553.45
random-10	2,512	60,289	28.30	30.23	1.01	1.77	3	788	4	3.16
random-15	3,914	65,124	36.04	34.79	1.01	1.72	3	1,392	5	2.80
random-20	5,072	66,074	40.92	37.24	1.01	1.56	3	1,943	6	2.60
random-25	6,048	65,298	43.66	38.20	1.01	1.34	2	2,384	7	2.53
random-30	6,606	62,953	44.97	38.30	1.01	1.11	2	2,604	7	2.53
random-35	7,229	57,697	45.14	37.68	1.01	0.88	2	2,811	8	2.56
random-40	6,023	36,087	45.05	37.11	1.01	0.47	2	2,201	14	2.72
room-8	7,994	18,724	6.18	3.15	1.01	0.15	3	327	4	23.91
room-16	7,839	13,928	1.40	0.64	1.01	0.04	3	66	2	109.25
room-32	8,847	12,860	0.32	0.14	1.01	0.01	4	18	2	371.68
room-64	9,820	12,109	0.07	0.03	1.01	0.00	5	5	1	928.87
street-256	842	5,052	3.88	7.39	0.25	0.11	5	41	1	17.60
street-512	3,467	18,542	1.89	6.27	1.01	0.36	10	92	2	33.66
street-1024	15,874	69,719	0.80	4.81	4.02	1.15	25	214	2	65.80

Table 5.3: Subgoal graphs on grid graphs.

- **PT:** As proven in Section 5.3.6, subgoal graphs on grid graphs can be constructed in time linear in the size of the underlying grid. Our experimental results agree with our theoretical results, and show that the SG preprocessing times are short, on the order of milliseconds, and comparable to the A* search times. On average, subgoal graphs can be constructed in the time that it takes to perform 1.40 A* searches on grid graphs. This value is lowest on maze-32 maps at 0.55, and highest on random-10 maps at 24.00.

- $|V|$**%:** On average, 8.29% of the vertices of G become subgoals. However this result is heavily skewed by the results on random maps (28.30%-45.14%) and maze maps with small corridor widths (27.60%-12.58%). Although game and street maps exhibit staircase patterns, they typically also have large contiguous areas of unblocked cells and, therefore, the percentage of subgoals on these types of maps is small, namely 2.19% and 1.51%, respectively. As the corridor width of maze maps increases, the number of corridors and convex corners formed by intersecting corridors decreases, and, therefore, the percentage of subgoals decreases from 27.6% on maze-1 maps to 0.07% on maze-32 maps. Similarly, as the room size of room maps increases, the number of rooms and the convex corners formed by "doors" between adjacent rooms decreases, and, therefore, the percentage of subgoals decreases from 6.17% on room-4 maps to 0.07% on room-32 maps.

- $|E|$**%:** On average, the number of edges in subgoal graphs is 7.29% of the number of edges in G and follows a pattern similar to the percentage of vertices of G that become subgoals, across different types of maps: When there are more subgoals, more edges are required to connect them. However, there are some deviations from this trend, which we explain by using the average number of edges per subgoal in subgoal graphs (not reported in Table 5.3): On game and street maps, where staircase patterns are prevalent, the average number of edges per subgoal is 15.06 and 29.04, respectively (up to a maximum of 47.37 on street-1024 maps). On maze, random, and room maps, which do not contain staircase patterns, the average number of edges per subgoal is significantly lower, namely 2.37, 4.51, and 3.23, respectively. Across all maps, the average number of edges per subgoal is 6.21, which is slightly smaller than 7.08, the average number of edges per vertex in G.

- **M (C and** E**):** On average, the memory required to store clearance values and subgoal graphs is very low, 0.78 and 0.24 megabytes, respectively, for a total of 1.02 megabytes. The memory required to store clearance values scales with the grid size, and is the highest on street-1024 and sc maps. The memory required to store subgoal graphs depends on the number of edges, and is the highest on random maps.

- **Cn:** The SG connection (SF-Connect) times are typically short, 5 microseconds on average. Recall that the the execution time of running SF-connect from a cell can be measured by the lengths of the diagonals that extend from the cell (Section 5.3.5). These diagonals extend further on game and street maps, which typically have large contiguous regions of unblocked cells. Therefore, SG connection times are significantly longer on game and street maps than they are on other types of maps.

- **Sr:** The SG search times contribute the most to the SG path query times ($\sim$ 105 times more than connection times), and are highly correlated with the number of subgoals, with a correlation coefficient of 0.897,[2] and less so with the number of edges of subgoal graphs, with a correlation coefficient of 0.638.

- **Rf:** The SG refinement (CF-Refine) times are typically short, 8 microseconds on average, but longer than the SG connection times. Recall that CF-Refine runs in time linear in the length of the resulting path on G (Section 4.5.5). Therefore, SG refinement times are highly correlated with instance path lengths (Table 5.2), with a correlation coefficient of 0.805. However, when refining a path π on the query subgoal graph, CF-Refine calculates, for every edge on π, the directions and the number of diagonal and cardinal moves on the corresponding freespace-shortest paths on G. Executing CF-Refine for paths that consist of longer sequences of shorter edges incurs this overhead more frequently, thereby increasing its execution times. For instance, the SG refinement times on random maps, where subgoal graphs typically have short edges, are disproportionately long (with respect to instance path lengths), namely $\sim$4 times longer than they are on game, room, and street maps, even though the instance path lengths are similar on these four types of maps.

- **PQ*:** The SG path query times are 18.33 times shorter than the A* path query (search) times across all maps, only 2.64 times shorter on random maps, and up to $\sim$1,550 times shorter on maze-32 maps. Since the SG search times contribute significantly more to the SG path query times than SG connection and refinement times, and since the SG search times are highly correlated with the numbers of vertices and edges of subgoal graphs, the speed-up achieved by using subgoal graphs over A* searches on G are highly correlated with the numbers of vertices and edges of the subgoal graphs relative to G, with correlation coefficients of 0.956 and 0.957, respectively. The results over all maps are heavily skewed by the results on random maps and maze maps with small corridor widths, where the percentage of subgoals is higher than it is on other types of maps.

- **Scaling:** As discussed in Section 5.5.2, the street-256, street-512, and street-1024 maps are generated by discretizing the same environments using different resolution grids. As our results show, SG achieves a higher speed-up over A* searches on street-1024 maps (65.80) than it does on street-256 and street-512 maps (17.60 and 33.66, respectively). The reason for this is as follows: As larger resolution grids are used for discretization, the number of vertices of G, and therefore the number of vertices expanded by A* searches on G, grows *quadratically* in the dimensions of the grid. However, the number of convex corners in G, and therefore the number of vertices expanded by SG searches, grow only *linearly* in the dimensions of the grid, since they can only appear along the boundaries of obstacles. Furthermore, since the path lengths and the lengths of the diagonals that extend from unblocked cells grow linearly in the dimensions of the grid, the SG connection and refinement times also grow only linearly in the dimensions of the grid.

[2]We calculate correlation coefficients based on the averaged statistics for the 26 subcategories of maps.

As discussed in Section 5.5.2, the bg-512 and wc3-512 maps are generated by scaling the (smaller) maps used in the games Baldur's Gate II and Warcraft III, respectively, to size 512 × 512. However, this scaling is not performed by discretizing an environment using a higher resolution grid, but instead by replacing each blocked or unblocked cell on the original grid with an appropriate-sized square area of blocked or unblocked cells. Therefore, this scaling does not introduce additional convex corners to the grid (that is, this scaling can be considered as "zooming-in" without enhancing the details), and, therefore, does not increase the SG search times. As a result, the staircase problem is less pronounced, and SG achieves higher speed-ups over A* searches on bg-512 and wc3-512 maps, compared to other types of game maps.

Finally, maze maps with large corridor widths and room maps with large room sizes can be considered to be scaled up versions of *parts* of maze maps with smaller corridor widths and room maps with smaller room sizes, respectively. That is, although all maze and room maps have the same size, maze maps with larger corridor widths and room maps with larger room sizes can be considered to be using higher resolution grids to discretize environments. As a result, SG achieves higher speed-ups over A* searches on maze maps with large corridor widths and room maps with large room sizes.

To summarize, on grid graphs, SG requires little preprocessing time and memory, and achieves a significantly higher speed-up over A* than on state lattices (Section 4.6.5). SG achieves a small speed-up over A* on random maps and maze maps with small corridor widths, higher speed-ups on game and street maps despite their staircase patterns, due to them typically containing large contiguous regions of unblocked cells, and significant speed-ups on maze maps with large corridor widths and room maps with large room sizes. Furthermore, SG achieves higher speed-ups over A* when using "higher resolution" grids or when scaling grids to larger sizes, as can be observed on bg-512, wc3-512 maps, maze maps with large corridor widths, room maps with large room sizes, and street maps with large grid resolutions. Although the SG connection and refinement times contribute little to the SG path query times on most types of maps, they make up a significant percentage of the path query times on maze maps with large corridor widths and room maps with large room sizes, and a non-trivial percentage of the path query times on game and street maps. As we will discuss in Section 5.5.7, combining subgoal graphs with contraction hierarchies can significantly reduce the search times, making the connection and refinement times more important for the path query times across all types of maps.

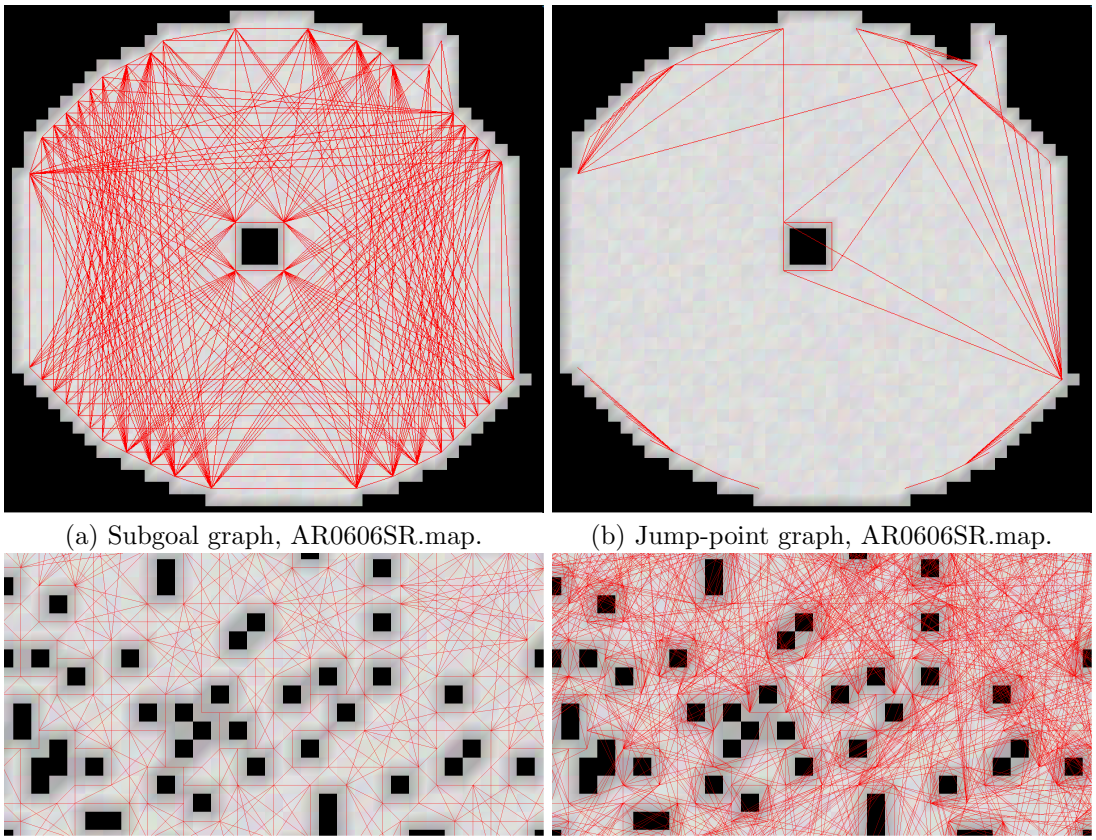
5.5.5 Jump-Point Graphs

In this section, we compare JP (answering queries using jump-point graphs) with SG (answering queries using subgoal graphs) with respect to their preprocessing and query times, and memory requirements.

Table 5.5 reports the JP preprocessing, connection, search, refinement, and path query times relative to SG (PT*, Cn*, Sr*, Rf*, and PQ*), the number of vertices and edges in jump-point graphs relative to subgoal graphs ($|V|$% and $|E|$%), the memory required to

	SG → JP									
	PT	$\|V\|$%	$\|E\|$%	M*	Exp*	Succ*	Cn*	Sr*	Rf*	PQ*
all	0.29	226.77	95.58	0.56	1.41	4.09	0.64	1.34	1.08	1.32
game-all	0.45	202.67	46.63	0.55	3.99	33.05	0.68	5.43	1.54	3.90
maze-all	0.34	217.74	108.77	0.52	1.31	3.03	0.61	1.24	1.01	1.22
random-all	0.22	230.90	138.98	0.60	1.45	2.63	0.49	1.21	1.24	1.21
room-all	0.36	278.81	131.82	0.51	1.22	2.03	0.61	1.02	1.14	1.00
street-all	0.27	202.60	21.86	0.64	4.49	93.16	0.64	8.66	1.47	3.72
bg	0.29	205.35	43.00	0.57	5.05	28.77	0.75	6.34	1.69	3.86
bg-512	0.47	200.30	31.52	0.51	4.58	32.92	0.73	6.30	1.45	2.86
dao	0.42	204.02	51.40	0.52	3.13	14.98	0.70	3.85	1.57	3.06
da2	0.38	202.51	44.20	0.51	4.35	27.36	0.72	5.80	1.55	3.24
sc	0.48	202.27	46.72	0.58	4.30	45.17	0.67	6.01	1.56	4.55
wc3-512	0.36	200.08	54.67	0.52	3.37	17.27	0.62	4.03	1.39	2.06
maze-1	0.28	233.99	139.01	0.53	1.12	2.24	0.67	1.03	0.91	1.03
maze-2	0.29	200.00	88.94	0.56	1.48	3.84	0.72	1.46	1.00	1.44
maze-4	0.36	200.00	78.42	0.53	2.40	7.49	0.67	2.40	1.21	2.29
maze-8	0.43	199.96	79.53	0.51	2.37	7.29	0.62	2.52	1.19	2.19
maze-16	0.48	200.00	79.26	0.50	2.36	7.17	0.56	2.66	1.21	1.80
maze-32	0.50	200.00	80.53	0.50	2.34	6.87	0.49	2.44	1.17	1.03
random-10	0.23	217.67	157.61	0.58	1.36	1.89	0.46	1.09	1.57	1.08
random-15	0.21	224.54	144.71	0.61	1.29	2.02	0.47	1.07	1.47	1.07
random-20	0.21	229.84	136.67	0.62	1.28	2.19	0.48	1.04	1.37	1.04
random-25	0.21	233.63	131.71	0.62	1.33	2.42	0.49	1.07	1.28	1.07
random-30	0.22	236.31	128.60	0.62	1.40	2.69	0.49	1.14	1.23	1.14
random-35	0.22	237.73	126.18	0.60	1.51	3.08	0.51	1.29	1.19	1.29
random-40	0.23	238.26	124.95	0.57	1.62	3.49	0.52	1.53	1.16	1.53
room-8	0.27	275.55	132.59	0.52	1.20	1.98	0.69	1.00	1.15	1.00
room-16	0.38	286.74	129.94	0.51	1.27	2.15	0.65	1.09	1.14	1.06
room-32	0.46	296.45	128.15	0.50	1.34	2.31	0.60	1.18	1.14	1.02
room-64	0.49	291.40	127.04	0.50	1.42	2.48	0.56	1.25	1.13	0.81
street-256	0.23	204.18	41.32	0.65	3.37	24.65	0.71	4.54	1.43	2.71
street-512	0.24	202.45	24.76	0.65	3.92	58.89	0.63	6.74	1.46	3.43
street-1024	0.31	201.44	13.90	0.63	5.55	169.37	0.63	12.49	1.49	4.17

Table 5.4: Jump-point graphs.



(a) Subgoal graph, AR0606SR.map. (b) Jump-point graph, AR0606SR.map.

(c) Subgoal graph, random512-10-0.map. (d) Jump-point graph, random512-10-0.map.

Figure 5.19: Jump-point and subgoal graphs on grids with staircase and random patterns.

store the edges of jump-point graphs and the clearance values in both the cardinal and diagonal directions relative to the memory required to store the edges of subgoal graphs and the clearance values in only cardinal directions (M*), and the number of vertices expanded and successors evaluated by JP searches relative to SG searches (Exp* and Succ*).

We make the following observations:

- **PT:** The JP preprocessing times are 3.45 times longer than the SG preprocessing times since JP calculates two sets of clearance values and often scans longer diagonals when identifying edges, as we discuss later. However, the JP preprocessing times are still very short, namely 48 milliseconds on average, especially compared to other algorithms that we discuss in the following sections.

- $|V|$**%:** As discussed in Section 5.4.3, convex corner cells typically contain two or more jump points, compared to exactly one subgoal. Our results show that jump-point graphs have 2.27 times more vertices on average than subgoal graphs.

- $|E|$**%:** As discussed in Section 5.4.3, even if the freespace-diagonal-first path π between two jump points is unblocked, jump-point graphs may not have an edge

between them if the direction of the last move on π does not match the direction of the second jump point. Therefore, jump-point graphs typically avoid adding edges between staircase subgoals, resulting in jump-point graphs having 53.37% and 78.14% fewer edges than subgoal graphs on game and street maps, respectively, which frequently contain staircase patterns. Figures 5.19a and 5.19b show an example.

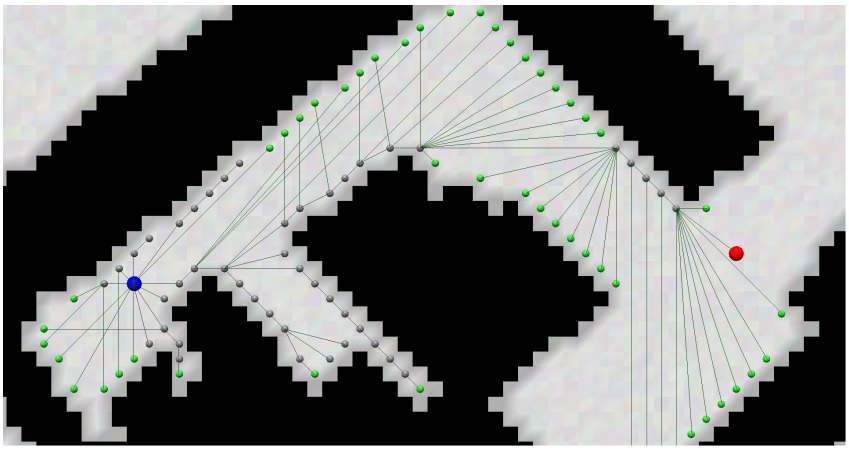
Whereas subgoal graphs have edges between subgoals that are direct-reachable with respect to the set of subgoals on G, jump-point graphs have edges between jump points that are direct-reachable with respect to the set of jump-points on G^*. If a path on G passes through a convex corner cell, it would be covered by the subgoal at that cell and, therefore, an edge that corresponds to that path does not need to be added to the subgoal graph. On the other hand, if a path on G^* passes through a convex corner cell, a jump point $(n, \vec{c})$ at that cell only covers that path if the last move on the path when reaching n is in direction $\vec{c}$. That is, jump points typically cover fewer paths on G^* than subgoals do on G. As a result, jump-point graphs might have more edges than subgoal graphs. Our results show that jump-point graphs have 38.93% and 31.82% more edges than subgoal graphs on random and room maps, respectively. Figures 5.19c and 5.19d show an example.

- **M*:** As discussed in Section 5.5.4, SG requires more memory to store clearance values than the edges of subgoal graphs. Since JP stores twice the number of clearance values than SG and roughly the same number of edges, JP requires on average 1.79 times more memory than SG.

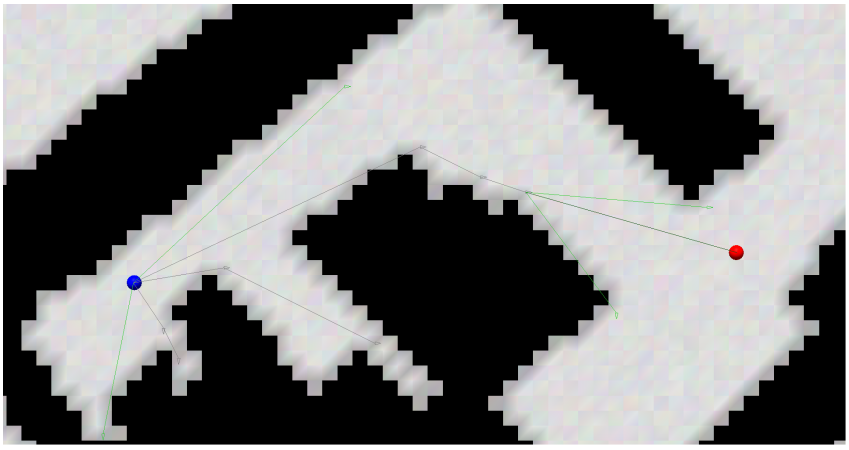
- **Cn*:** The JP connection (JP-Connect) times are 1.56 times longer than the SG connection (SF-Connect) times, because, since jump-point graphs do not have diagonal jump-points, the diagonal scans in JP-Connect terminate only when movement is no longer possible in that direction. In contrast, as discussed in Section 5.3.5, the diagonal scans in SF-Connect terminate at convex corner cells.

  We have also experimented with storing additional clearance values to scan diagonals (rows and columns, when connecting the goal vertex to JP) more efficiently, as discussed in Section 5.4.1, by "jumping" to those rows and columns that contain subgoals or jump points, for SG and JP, respectively. This modification decreases the average SG connection times from 5.14 to 4.62 microseconds, and the JP connection times from 7.99 to 4.95 microseconds, making SG and JP connection times very similar. However, it increases the SG and JP memory requirements by factors of 1.78 and 1.89, respectively, while reducing the SG and JP path query times by factors of only 1.001 and 1.008, respectively.

- **Sr*, Exp*, and Succ*:** JP searches explore only locally-diagonal-first paths on G, since jump-point graphs contain only locally-diagonal-first paths, as discussed in Section 5.4.3. When we consider the number of vertices and edges in jump-point graphs relative to subgoal graphs, we observe that the average out-degree of vertices is 2.37 times smaller in jump-point graphs compared to subgoal graphs. As a result, JP searches expand 1.41 times fewer vertices, process 4.09 times fewer successors, and are 1.34 times faster than SG searches. These numbers are higher on game and



(a) Search tree of SG.



(b) Search tree of JP.

Figure 5.20: Search trees of SG and JP on Berlin-0-256.map (street-256). Edges are colored based on their destination vertices, namely gray (expanded) and green (generated but not expanded). The start and goal vertices are shown as large blue and red circles, respectively.

street maps, where jump-point graphs also contain significantly fewer edges than subgoal graphs due to jump-point graphs avoiding staircase problems, as discussed earlier. For instance, JP searches process 93.16 times fewer successors than SG searches on street maps. Figure 5.20 shows an example.

- **Rf*:** The JP refinement (JP-Refine) times are 1.08 times shorter than the SG refinement (CF-Refine) times on average, and 1.54 times and 1.47 times shorter on game and street maps, respectively. As discussed in Section 5.4.3, JP-Refine and CF-Refine can be considered to be equivalent. As discussed in Section 5.5.4, the CF-refinement times are typically shorter when refining paths that consist of shorter sequences of longer edges. Jump-point graph edges are typically longer than subgoal-graph edges, as shown in Figures 5.19d and 5.20b, resulting in CF-refinement times being slightly shorter on paths found on jump-point graphs compared to those found on subgoal graphs.

- **PQ*:** The JP path query times are 1.32 times shorter than the SG path query times on average, since the JP search times are shorter than the SG search times and, as discussed in Section 5.5.4, the SG search times are the dominant factor in the SG path query times (that is, the SG connection times being shorter than the JP connection times does not affect the path query times too much).

To summarize, by exploring only locally-diagonal-first paths, JP searches avoid the staircase problem that negatively affects the SG search times, and run significantly faster than SG searches on game and street maps, and slightly faster on other types of maps. Although jump-point graphs have more edges than subgoal graphs, they have significantly fewer edges on game and street maps, but more edges on other types of maps, which will become more relevant in Section 5.5.8 when we compare answering queries using contraction hierarchies constructed on subgoal and jump-point graphs. Since JP scans the grid differently when connecting goal vertices to jump-point graphs, it stores an additional set of clearance values and therefore requires more memory than SG.

5.5.6 Contraction Hierarchies

In this section, we compare CH (answering queries using contraction hierarchies on G) with A* (answering queries using A* searches on G), SG (answering queries using subgoal graphs on G), and CH-GPPC (a variant of CH that does not use a heuristic in its searches, and performs midpoint unpacking instead of 2-pointer unpacking), with respect to their preprocessing and query times, and memory requirements.

Recall that contraction hierarchies are constructed by forming hierarchies among the vertices of a graph and adding shortcut edges to ensure that, between every pair of vertices s and t, the hierarchies contain at least one up-down s-t path with length $d(s, t)$ (Section 3.4.2). As a result, contraction hierarchies can be searched for a shortest up-down s-t path with bidirectional searches, where the forward search constructs the upward part of a shortest up-down s-t path and the backward search constructs the downward part. Any vertex that cannot be reached from s with an upward path, or cannot reach t with a downward part is therefore effectively eliminated from these searches (Section 3.4.4).

Table 5.5 reports the SG path query times relative to A* (PQ*, also reported in Table 5.3), the CH preprocessing times (PT), the number of edges in contraction hierarchies relative to G ($|E|$%), the memory required to store these edges (M), the CH search and refinement times (Sr and Rf), the CH path query times relative to A* (PQ*), and the CH search and refinement times relative to CH-GPPC (Sr* and Rf*).

We make the following observations:

- **PT:** The CH preprocessing times are significantly longer than the SG preprocessing times (Table 5.3), by a factor of 7,555 over all maps. CH preprocessing times are the longest on the street-1024 maps, followed by the street-512 and sc maps. These maps are also the largest maps in the MovingAI benchmarks and contain large contiguous regions of unblocked cells, which might be the reason why the CH preprocessing times are longer on these types of maps.

- **$|E|$%:** On average, contraction hierarchies have 2.3% fewer edges than G: Since contraction hierarchies do not have level edges, every edge (u, v) of G becomes

	A* → SG	A* → CH					CH-GPPC → CH	
	PQ *	PT (s)	$\lvert E\rvert$ %	M (MB)	Sr (μs)	Rf (μs)	PQ *	Sr *	Rf *
all	18.33	105	97.70	12.13	53	19	136.41	2.15	2.32
game-all	39.10	66	92.01	8.28	70	10	64.27	1.44	2.43
maze-all	23.28	33	118.52	17.23	24	33	292.55	0.85	2.25
random-all	2.64	74	110.18	12.98	43	12	100.79	2.23	2.30
room-all	79.19	44	113.59	22.73	44	10	160.29	1.03	2.50
street-all	50.46	528	87.01	29.99	130	17	45.67	5.69	2.62
bg	10.28	1	112.14	0.44	14	2	28.23	1.16	2.42
bg512	56.52	110	89.80	9.76	57	8	36.00	1.99	2.38
dao	24.21	9	101.43	2.40	35	8	55.28	1.09	2.58
da2	25.13	4	107.44	1.67	24	4	37.25	1.00	2.54
sc	41.54	281	90.21	34.52	129	17	79.26	1.17	2.39
wc3-512	79.61	99	91.98	14.39	66	9	40.39	3.06	2.48
maze-1	4.03	0	116.49	3.43	4	38	163.27	0.85	1.90
maze-2	11.32	2	161.10	8.24	6	35	314.06	0.87	2.10
maze-4	43.65	5	152.55	13.57	10	32	443.60	0.86	2.33
maze-8	174.02	19	124.20	20.74	19	32	427.71	0.83	2.53
maze-16	603.72	52	107.19	26.64	45	30	325.52	0.84	2.60
maze-32	1,553.45	119	98.28	30.75	107	22	182.35	0.86	2.53
random-10	3.16	242	95.50	24.50	61	14	33.66	3.94	2.47
random-15	2.80	161	101.11	19.64	75	13	44.55	2.71	2.42
random-20	2.60	75	107.98	15.51	73	13	59.55	2.05	2.38
random-25	2.53	27	116.89	11.99	60	12	84.51	1.66	2.31
random-30	2.53	7	127.45	9.10	41	11	128.97	1.40	2.28
random-35	2.56	2	139.86	6.69	23	10	224.79	1.13	2.21
random-40	2.72	1	150.09	3.41	7	12	323.67	0.89	2.14
room-8	23.91	6	143.29	13.85	29	10	205.01	1.32	2.45
room-16	109.25	18	123.41	20.57	24	9	235.48	1.10	2.57
room-32	371.68	47	107.69	26.28	39	9	182.84	0.94	2.54
room-64	928.87	106	98.41	30.19	82	10	106.11	0.95	2.44
street-256	17.60	28	96.74	5.74	34	4	22.33	3.28	2.39
street-512	33.66	290	88.90	26.29	91	13	33.30	4.54	2.51
street-1024	65.80	2739	84.23	113.82	258	34	54.37	6.52	2.71

Table 5.5: Contraction hierarchies on grid graphs. The CH-GPPC and CH preprocessing times are the same. The CH-GPPC memory requirements are 75% of those of CH.

either an upward or a downward edge in the hierarchy. Furthermore, since G is undirected, for every edge (u, v) of G that becomes an upward edge in the hierarchy, there exists an edge (v, u) of G that becomes a downward edge. Since CH does not store downward edges (Section 5.5.1), exactly half of the edges of G are present in contraction hierarchies, and the other half are discarded. Our results show that, on average, the number of shortcut edges added to contraction hierarchies is slightly smaller than the number of edges discarded, resulting in contraction hierarchies having a slightly smaller number of edges than G. Similar results have also been observed for contraction hierarchies constructed on road networks (Geisberger et al., 2008). This result holds mainly for game and street maps, but not other types of maps. A discussion for why this is the case is beyond the scope of this dissertation. Compared to subgoal graphs (Table 5.3), contraction hierarchies have 14.06 times more edges on average.

- **M:** Since CH uses 16 bytes to store each edge of a contraction hierarchy (4 times higher than SG, as discussed in Section 5.5.1), and since contraction hierarchies on grid graphs have significantly more edges than subgoal graphs, CH requires 11.89 times more memory on average to store the edges of contraction hierarchies than SG requires to store the edges of subgoal graphs plus the clearance values. Since CH-GPPC uses 12 bytes to store each edge of a contraction hierarchy, it requires 25% less memory than CH (not reported in the table).

- **Sr:** The CH search times are short, namely 9.93 times shorter than the SG search times and 186.30 times shorter than the A* search times across all maps. As the corridor width and the room size increases in maze and room maps, respectively, the CH search times increase, similarly to the number of vertices of G and the A* search times (Table 5.2). As the percentage of blocked cells increases in random maps, the CH search times decrease, similarly to the number of vertices of G, but contrary to the A* search times. The increase in the A* search times despite the decrease in the number of vertices of G suggests that the Octile distance heuristic becomes less informed on random maps as the percentage of blocked cells increases. However, as our results suggest, less informed heuristics do not seem to affect the CH search times as much as they affect the A* search times. We now explain why this is the case.

  1) Heuristics typically help to avoid expanding vertices that are "far away" from the start and goal vertices: An A* search for a shortest s-t path does not expand vertices n with $d(s, n) + h(n, t) > d(s, t)$. Adding $d(n, t)$ to both sides of this inequality and rearranging the terms, we obtain the equivalent inequality $d(s, n) + d(n, t) - d(s, t) > d(n, t) - h(n, t)$. Intuitively, the right-hand side of this inequality $(d(n, t) - h(n, t))$ corresponds to how much h underestimates the n-t distance on G. If the underestimation is less than the value of the left-hand side $(d(s, n) + d(n, t) - d(s, t))$, then the A* search is guaranteed to avoid expanding n. As the heuristic becomes more informed, A* searches avoid expanding vertices n with smaller values of $d(s, n) + d(n, t) - d(s, t)$, that is, vertices n for which $d(s, n)$ and/or $d(n, t)$ are large. 2) Forward and backward CH searches typically avoid expanding vertices that are "far away" from the start and the goal vertices, respectively: Intuitively,

as discussed in Section 2.3.2, CH searches expand vertices that are far away from the start and the goal vertex only if they are "important", that is, if their levels are high. More formally, high-level shortcut edges are typically longer than low-level (shortcut) edges in contraction hierarchies, since high level shortcut edges are formed by combining two lower-level edges and, thus, their lengths are determined as the sum of the two edges. As a result, the successors of expanded vertices are typically further away than the expanded vertex, the higher the level of the expanded vertex. Figure 5.21c shows an example of CH search trees, which we discuss in further detail in Section 5.5.7. Since CH searches "climb upward" in the hierarchy, the branches of CH search trees are typically "sparser" the further away they are from the start (for the forward search) or the goal (for the backward search) vertex. Therefore, the further away a vertex is from the start and the goal, the less likely it is to be expanded by CH searches. Consequently, CH searches benefit less from heuristics, since CH searches already avoid expanding most vertices that are far away from the start and the goal, even without the use of a heuristic.

Compared to the SG search times, the CH search times are typically shorter across all types of maps, with the following exceptions: On maze maps with large corridor widths and room maps with large room sizes, the SG search times are significantly shorter, namely by factors of 13.63 and 17.56 on maze-32 and room-64 maps, respectively (not reported in the table). On bg-512, wc3-512, and street-1024 maps, the SG search times are slightly shorter, namely by factors of 1.67, 2.17, and 1.20, respectively. As discussed in Section 5.5.4, this can be explained by the fact that all five types of maps can be considered to be "higher resolution" discretizations of environments, where the SG search times are typically very short. On bg-512, wc3-512, and street-1024 maps, SG search times are only slightly shorter than the CH search times, since these maps contain staircase patterns, which negatively affect the SG search times. In addition to searches, SG queries also include a connection phase. The results reported above change only slightly when the SG connection times are taken into account, and the SG path query times are shorter than the CH path query times on these five types of maps.

- **Sr*:** The CH search times are 2.15 times shorter than the CH-GPPC search times, since CH searches use the Octile distance heuristic. On street maps, where the Octile distance heuristic can be very informative, CH searches expand or stall 6.48 times fewer vertices than CH-GPPC searches (not reported in Table 5.5), and the CH search times are 5.69 times shorter. On maze maps, where the Octile distance heuristic is not very informative, CH searches expand or stall only 1.02 times fewer vertices than CH-GPPC searches, but the CH search times are 1.18 times *longer*, due to the overhead of computing the Octile distances.

- **Rf:** The CH refinement (2-pointer unpacking) times make up 26.78% of the CH path query times on average, and are 2.38 times longer than the SG refinement (CF-Refine) times. Recall that 2-pointer unpacking runs in time linear in the number of edges of the resulting path (Section 3.4.4). Therefore, similar to the SG refinement times, the CH refinement times are highly correlated with the instance path lengths, with a correlation coefficient of 0.905. However, 2-pointer unpacking

frequently accesses memory to look up the two edges that correspond to a shortcut edge, whereas CF-Refine does not need to access memory (besides the paths found by the SG searches and the resulting paths after refinement), since it simply generates freespace-diagonal-first or freespace-cardinal-first paths between CF-reachable vertices. Therefore, 2-pointer unpacking is typically slower than CF-Refine. An exception to this rule occurs on random maps, where CF-Refine has to calculate the directions and number of moves along freespace-shortest paths frequently, as discussed in Section 5.5.4. On random-40 maps, CF-Refine is 1.19 times slower than 2-pointer unpacking. On all other types of maps, CF-Refine is faster.

- **Rf*:** The CH refinement (2-pointer unpacking) times are 2.32 times shorter than the CH-GPPC refinement (midpoint unpacking) times, since midpoint unpacking performs the same sequence of look-ups as 2-pointer unpacking, but also has to scan the incident edges of vertices to find the corresponding two edges. The ratios of the CH refinement times to the CH-GPPC refinement times are similar across the different types of maps.

- **PQ*:** The CH path query times are on average 2.20 times shorter than the CH-GPPC path query times, since the CH search and refinement times are shorter than the CH-GPPC search and refinement times, respectively. The CH path query times are 136.41 times shorter than the A* search times across all maps, and 7.44 times shorter than the SG path query times. The relative path query times of CH and SG across the different types of maps depend on their relative search and refinement times, as well as the SG connection times. We discuss the CH and SG path query times in more detail in Section 5.5.7, when we construct contraction hierarchies on subgoal graphs.

To summarize, CH has shorter path query times than SG, especially on random maps, but longer preprocessing times, higher memory requirements, and longer refinement times. On grids with "higher resolutions", the SG path query times are typically shorter than the CH path query times, but less so when staircase patterns are present. On random maps, the SG path query times are long, and the CH path query times are significantly shorter. In this sense, SG and CH seem to have complementary strengths, which we explore in more detail in Section 5.5.7. Furthermore, the SG refinement times are typically much shorter than the CH refinement times, and SG uses less memory to store each edge, which we aim to exploit in Sections 5.5.9 and 5.5.10 when we augment CH with reachability relations.

5.5.7 Contraction Hierarchies on Subgoal Graphs

In this section, we compare CH-SG (answering queries using contraction hierarchies on subgoal graphs) to CH and SG with respect to their preprocessing and query times, and memory requirements. Recall that CH-SG performs connection using SF-Connect, similar to SG; performs search using bidirectional A* searches, similar to CH; and performs refinement first by using 2-pointer unpacking, similar to CH, to refine the path found on the contraction hierarchy to a path on the subgoal graph, and then by using CF-Refine,

similar to SG, to refine it into a path on G. Furthermore, CH-SG stores clearance values in the four cardinal directions, similar to SG; and stores each edge using 16 bytes, similar to CH.

Table 5.5 reports the CH-SG preprocessing times (PT*) relative to CH, the number of edges of contraction hierarchies on subgoal graphs ($|E|$%) relative to subgoal graphs and contraction hierarchies on G, respectively, and the CH-SG memory requirements (M*), distance query times (DQ*, that is, connection plus search times), refinement times (Rf*), and path query times (PQ*), all relative to SG and CH, respectively.

We make the following observations:

- **PT*:** The CH-SG preprocessing times are shorter than the CH preprocessing times on all types of maps except for the street-512, street-1024, random-10, and random-15 maps. The CH-SG preprocessing times are significantly shorter than the CH preprocessing times on maze maps (by a factor of 700 on all maze maps and up to a factor of 8,669 on the maze-32 maps), room maps (by a factor of 138 on all room maps and up to a factor of 7,417 on room-64 maps), and game maps (by a factor of 5 on all game maps and up to factors of 143 and 78 on the bg-512 and wc3-512 maps, respectively). The types of maps for which the CH-SG preprocessing times are significantly shorter than the CH preprocessing times correspond to the types of maps that are scaled-up, as discussed in Section 5.5.4, except for the street maps. The scaling on street maps also introduces additional staircase subgoals and edges, unlike the scaling on game, maze, and room maps. Therefore, as the grid resolution of street maps increases, the average number of neighbors per subgoal increases, and contracting each subgoal requires many more witness searches. This issue can be addressed by avoiding witness searches between those pairs of neighbors whose corresponding edges form "non-taut" paths. Whether two edges incident to a subgoal form a taut path can easily be determined in constant time simply by checking the directions of the edges and the direction of the corner that introduced the subgoal. Our implementation does not have this optimization and, as a result, performs poorly on street maps. Since the CH preprocessing times are already long on street maps and the CH-SG preprocessing times are longer, the CH-SG preprocessing times appear to be longer than the CH preprocessing times across all maps.

- **$|E|$%:** Contraction hierarchies on subgoal graphs have 8.28% fewer edges than subgoal graphs across all maps, similar to contraction hierarchies on G, which have 2.3% fewer edges than G, as reported in Section 5.5.6. Contraction hierarchies on subgoal graphs, on average, have 93.48% fewer edges than contraction hierarchies on G, since they are constructed on subgoal graphs, which have, on average, 92.72% fewer edges than G (Table 5.3).

- **$|M|$%:** Since CH-SG stores 92.72% fewer edges than CH, it requires less memory than CH, by a factor of 7.63 on average, despite its additional memory overhead for storing clearance values. Although CH-SG stores each edge using four times more memory than SG, it requires only 1.56 times more memory than SG, since the memory required to store clearance values is higher than the memory required

	SG → CH-SG					CH → CH-SG					
	$\|E\|$%	M*	DQ*	Rf*	PQ*	PT*	$\|E\|$%	M*	DQ*	Rf*	PQ*
all	91.72	0.64	30.54	0.63	18.37	0.82	6.52	7.63	3.05	1.62	2.47
game-all	56.84	0.89	6.34	0.49	5.29	5.07	2.25	9.41	3.43	2.25	3.22
maze-all	78.16	0.83	126.83	0.70	26.90	700.23	1.90	12.86	4.38	1.55	2.14
random-all	119.55	0.32	49.94	0.50	36.04	0.95	46.64	1.84	1.03	0.72	0.94
room-all	126.31	0.83	8.13	0.48	6.05	138.39	1.16	17.90	3.38	1.94	2.99
street-all	54.82	0.77	3.79	0.43	3.44	0.43	2.67	15.62	3.75	4.19	3.80
bg	60.72	0.73	4.34	0.42	3.41	3.48	9.11	4.00	1.41	0.71	1.24
bg512	58.08	0.96	3.07	0.46	2.58	143.39	0.74	9.04	4.39	2.59	4.04
dao	59.94	0.90	6.14	0.51	4.83	6.92	3.81	3.93	2.27	1.60	2.11
da2	58.57	0.95	3.74	0.48	2.91	13.58	3.05	2.65	2.22	1.22	1.96
sc	55.60	0.81	7.85	0.50	6.79	3.25	2.55	13.33	3.68	2.82	3.56
wc3-512	59.78	0.94	2.64	0.44	2.24	78.57	0.86	12.73	4.77	2.78	4.41
maze-1	85.02	0.66	366.85	0.69	39.66	3.95	27.34	1.76	0.92	0.99	0.98
maze-2	75.58	0.74	197.58	0.69	33.04	18.39	8.01	4.93	1.12	1.20	1.19
maze-4	68.30	0.87	69.36	0.71	17.40	117.21	1.90	10.69	1.61	1.75	1.71
maze-8	68.31	0.95	20.78	0.74	7.21	810.40	0.38	19.03	3.39	2.71	2.93
maze-16	67.73	0.99	6.07	0.72	3.03	3,280.81	0.09	25.86	7.84	3.94	5.63
maze-32	65.96	1.00	1.96	0.64	1.39	8,669.57	0.02	30.45	17.20	4.78	11.86
random-10	140.83	0.25	13.35	0.37	11.27	0.90	40.66	2.23	1.03	1.20	1.06
random-15	135.89	0.26	19.51	0.38	16.48	0.95	47.80	1.89	1.04	0.99	1.03
random-20	126.35	0.29	27.86	0.40	23.10	1.02	50.81	1.74	1.04	0.86	1.01
random-25	114.22	0.33	41.73	0.43	33.03	1.12	51.01	1.68	1.05	0.77	0.99
random-30	100.99	0.39	65.71	0.47	47.44	1.30	49.30	1.66	1.03	0.68	0.93
random-35	87.24	0.46	126.69	0.52	73.68	1.93	45.98	1.64	1.01	0.60	0.84
random-40	76.91	0.60	286.93	0.62	74.15	2.98	42.84	1.38	0.91	0.52	0.62
room-8	128.73	0.64	14.34	0.52	10.84	6.00	5.82	7.65	1.27	1.26	1.26
room-16	122.39	0.87	5.35	0.48	3.98	115.91	0.96	17.00	1.89	1.74	1.85
room-32	111.45	0.97	2.26	0.44	1.73	1,538.68	0.17	25.03	4.02	2.29	3.51
room-64	92.55	0.99	1.31	0.45	1.06	7,417.83	0.03	29.60	11.63	3.54	9.28
street-256	59.51	0.72	3.02	0.39	2.58	2.13	4.26	11.48	2.17	1.35	2.03
street-512	55.44	0.75	3.62	0.43	3.23	0.77	3.09	14.45	3.26	3.32	3.27
street-1024	52.98	0.80	4.07	0.45	3.76	0.37	2.15	17.62	4.40	6.30	4.56

Table 5.6: Contraction hierarchies on subgoal graphs.

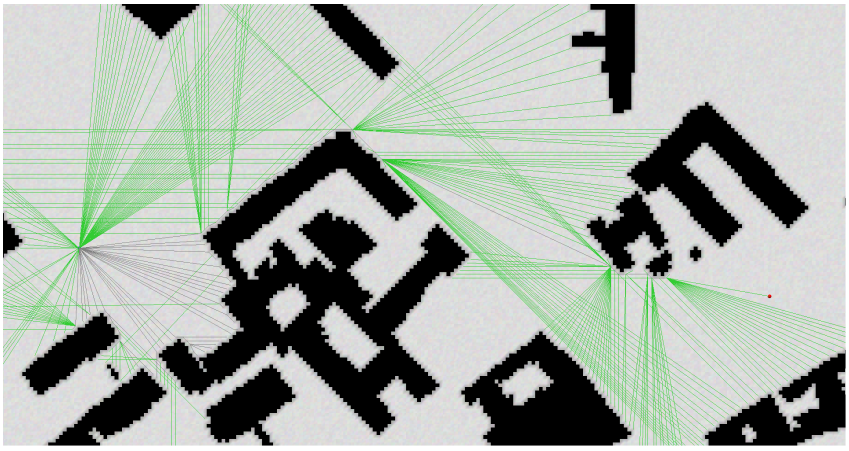
to store edges, as reported in Section 5.5.4, and, to a lesser extent, since CH-SG stores 8.28% fewer edges than SG.

- **DQ* vs. SG:** The CH-SG distance query (connection plus search) times are 30.54 times shorter than the SG distance query times on average. The factor of improvement is lowest on maze maps with large corridor widths and room maps with large room sizes, where SG already achieves large speed-up over A* searches, and highest on maze maps with small corridor widths and random maps with high percentages of blocked cells, where SG achieves small speed-ups over A*. The factor of improvement is relatively small on game and street maps, which we think is due to the staircase patterns that appear frequently on these types of maps, as we explain below.

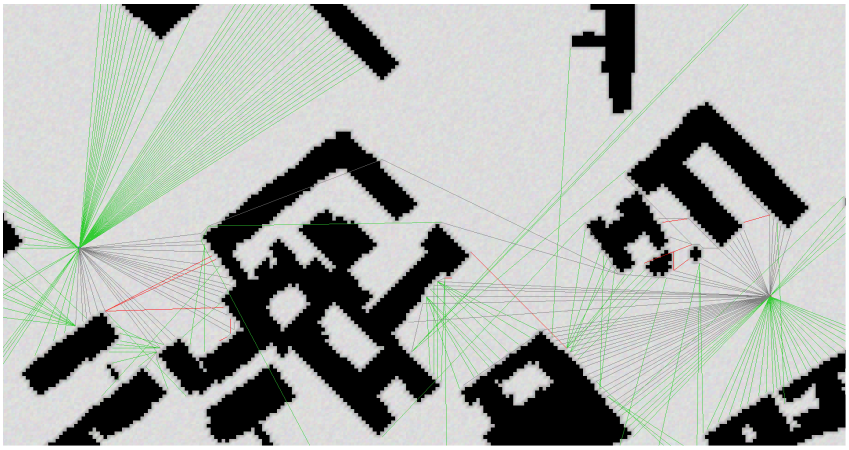
  Figure 5.21a shows an example of a SG search tree (the figure only shows the parents of vertices in the search tree, and not all successors considered per expansion). SG expands many staircase subgoals and uses many staircase edges to generate successors that cannot appear on any shortest path between the start and goal vertices (the staircase problem). CH-SG mitigates the staircase problem to some degree, by assigning lower levels to staircase subgoals and thus removing them from consideration as successors for higher-level vertices. However, CH-SG still connects the start and goal vertices to staircase subgoals, generates staircase subgoals as successors when the start and goal vertices are expanded, and eventually can expand staircase subgoals, which might generate more staircase subgoals as successors (although, only the ones with lower levels). Figure 5.21b shows an example of a CH-SG search tree. Most of the expanded or generated vertices are staircase subgoals to which the start or goal vertices are connected.

  The following statistics (not reported in the table) support this explanation: On game maps, CH-SG connects the start and goal vertices to a total of 57.60 subgoals, and expands or stalls 81.9 vertices. On street maps, CH-SG connects the start and goal vertices to a total of 125.72 subgoals, and expands or stalls 81.08 vertices. That is, on game and street maps, the number of subgoals to which the start or goal vertices are connected are comparable to the number vertices expanded or stalled by CH-SG. On maze, random, and room maps, CH-SG connects the start and goal vertices to only 4.78, 7.69, and 5.81 subgoals, respectively, and expands or stalls 26.72, 126.84, and 49.15 vertices. That is, on maze, random, and room maps, the number of subgoals to which the start or goal vertices are connected is significantly smaller than the number vertices expanded or stalled by CH-SG searches.

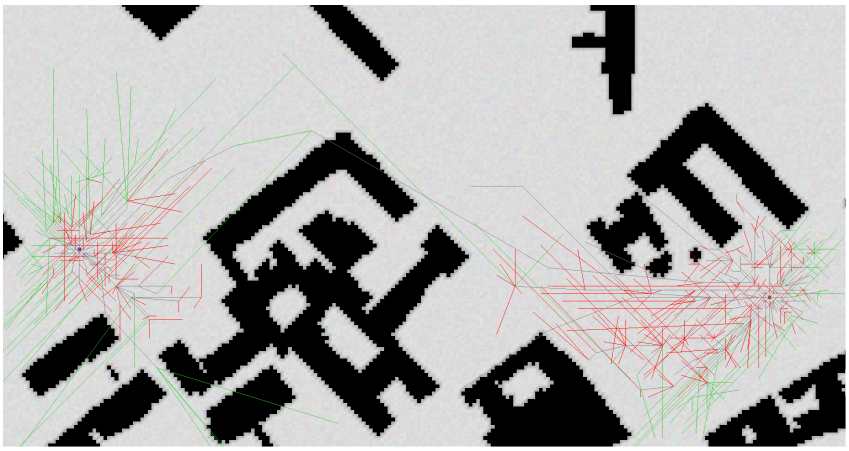
- **DQ* vs. CH:** The CH-SG distance query times are 3.05 times shorter than the CH distance query times on average, and shorter across all types of maps except for the random-10 and maze-1 maps, where they are 1.09 times longer. There are two competing factors that determine whether the CH-SG distance query times are shorter than the CH distance query times. Namely, the CH-SG search times relative to the CH search times, and the CH-SG connection times: The CH-SG search times are typically shorter than the CH search times since contraction hierarchies



(a) The search tree of SG. The start and goal vertices are connected to many subgoals along staircases, whose expansions also generate many successors along staircases.



(b) The search trees of CH-SG. The start and goal vertices are connected to many subgoals along staircases, whose expansions typically do not generate many successors along staircases.



(c) The search trees of CH. Most vertices are generated and expanded or stalled in freespace, which is avoided by CH-SG. In this example, CH expands or stalls ∼4.5 times more vertices than CH-SG.

Figure 5.21: The search trees of SG, CH, and CH-SG on Berlin-0-256.map (street-256). For CH and CH-SG, the search trees in both the forward and backward searches are shown. The edges are colored based on their destination vertices, namely gray (expanded), green (generated but not expanded), and red (stalled).

constructed on subgoal graphs are typically smaller than contraction hierarchies constructed on G. However, in order for the CH-SG distance query times to be shorter than the CH distance query times, the CH-SG connection times should not exceed the difference between the CH and CH-SG search times. On the maze-1 and random-40 maps, where the SG search times are only slightly shorter than the A* search times, the CH-SG search times are also only slightly shorter than the CH search times (not reported in the table), and the difference in the CH-SG and CH search times is shorter than CH-SG connection times, resulting in the CH-SG distance query times being longer than the CH distance query times.

The factor of improvement of the distance query times for switching from CH to CH-SG is the highest on the scaled-up maps, that is, the wc-512 maps, the bg-512 maps, the maze maps with large corridor widths, the room maps with large room sizes, and the street maps with large grid resolutions. As discussed in Section 5.5.6, CH searches are more likely to expand vertices that are "close" to the start or goal vertices. CH-SG can avoid most of these expansions since it typically connects the start and goal vertices directly to "far away" subgoals. Since scaled-up maps typically contain large contiguous regions of unblocked cells, CH-SG can connect the start and goal vertices to subgoals that are farther away than on other types of maps. Figures 5.21b and 5.21c show an example, where, despite the staircase problem, CH-SG expands or stalls 4.5 times fewer vertices than CH.

- **Rf* vs. SG:** The CH-SG refinement times are strictly longer than the SG refinement times, by a factor of 1.59 on average, since both CH-SG and SG refine paths on subgoal graphs into paths on grids, but CH-SG first has to refine paths on contraction hierarchies into paths on subgoal graphs.

- **Rf* vs. CH:** The CH-SG refinement times are 1.62 times longer than the CH refinement times on average, but shorter on random maps. Recall that (2-pointer) unpacking a path π on a contraction hierarchy recursively replaces shortcut edges on π with two (shortcut) edges until a path on G is generated. CH-SG refinement differs from unpacking by using CF-Refine for edges of subgoal graphs. As discussed in Section 5.5.6, CF-Refine is typically faster than 2-pointer unpacking when refining long edges, but may be slower when refining short edges. Therefore, CH-SG refinement is typically faster than CH refinement, except on random maps where subgoal graph edges are typically short.

- **PQ*:** Since both the CH-SG distance query and refinement times are shorter than the CH distance query and refinement times, respectively, their sums, the CH-SG path query times, are also shorter than the CH path query times, by a factor of 2.47 on average. Since the CH-SG distance query times are significantly shorter than the SG distance query times and the CH-SG refinement times are only slightly longer than the SG refinement times, the CH-SG path query times are shorter than the SG path query times, by a factor of 18.37 on average.

To summarize, compared to CH, CH-SG typically has shorter path query times and lower memory requirements, since CH-SG stores and searches smaller hierarchies, and

since it arguably pays only a small "cost" for using smaller hierarchies, by using efficient connection and refinement algorithms that exploit the freespace structure of grid graphs. Compared to SG, CH-SG has shorter path query times but higher memory requirements, since using a contraction hierarchy allows searches to ignore more vertices (in addition to the non-subgoal vertices ignored by SG), and since CH-SG uses more memory to store each edge.

5.5.8 Contraction Hierarchies on Jump-Point Graphs

In this section, we compare CH-JP (answering queries using contraction hierarchies on jump-point graphs) to CH-SG (answering queries using contraction hierarchies on subgoal graphs) with respect to their preprocessing and query times, and memory requirements.

We have remarked in Section 5.4.3 that constructing an explicit graph of jump-points, namely the jump-point graph, allows for the combination of jump-point search with other orthogonal techniques, such as contraction hierarchies. We now point out a drawback of the straightforward way of combining these two techniques, and discuss how our implementation addresses this drawback.

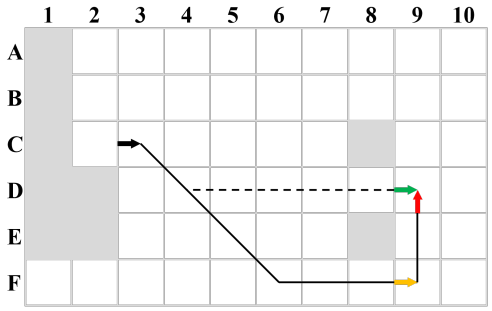


Figure 5.22: Shortest paths on jump-point graphs are not necessarily shortest paths on G. Contraction hierarchies constructed on jump-point graphs preserve these paths, which can be considered to be redundant when answering queries on G.

Consider the example shown in Figure 5.22. The only locally-diagonal-first path from (C3, Right) to (D9, Up) on the jump-point graph (shown as a solid line) does not correspond to a shortest path on G. That is, the (C3, Right)-(D9-Up) distance on the jump-point graph is not equal to the C3-D9 distance on G. When constructing a contraction hierarchy on the jump-point graph, if (F9, Right) is contracted before (C3, Right) and (D9, Up), the shortcut edge ((C3, Right), (D9, Up)) would be introduced to preserve the (C3, Right)-(D9-Up) distance on the jump-point graph. However, since this edge does not correspond to a shortest path on G, it cannot be used for finding shortest paths on G and is therefore redundant. We have observed in our preliminary experiments that the straightforward way of combining contraction hierarchies and jump-point graphs resulted in very long preprocessing times (more than several hours) on random maps, due to redundant edges resulting in more witness searches during contractions.

One method of addressing this problem is to avoid adding shortcut edges whose lengths do not represent distances on G. However, this requires determining distances on G to check whether each shortcut edge is redundant, which can also result in long preprocessing times. Our current implementation of CH-JP instead performs the following, approximate, check to determine whether a shortcut edge $((n_1, \vec{c}_1), (n_2, \vec{c}_2))$ is redundant: It performs a search on the jump-point graph, treating all jump points at cell n_1 as start vertices and all jump points at cell n_2 as goal vertices, in order to determine an upper bound on the n_1-n_2 distance on G (to determine the exact n_1-n_2 distance, all pairings of n_1 and n_2 with the eight directions need to be considered). It avoids adding the shortcut edge $((n_1, \vec{c}_1), (n_2, \vec{c}_2))$ if its length is greater than this bound. For instance, in the example in Figure 5.22, it finds that the length of the path from (C3, Right) to (D9, Right) is smaller than the length of the shortcut edge ((C3,Right),(C9,Up)), and therefore avoids adding ((C3,Right),(C9,Up)) as a shortcut edge.

Our preliminary experiments on a 400 × 400 grid with 33% blocked cells showed that, without the elimination of redundant edges, contraction hierarchies on jump-point graphs have 475,401 shortcut edges, of which 121,890 (25.6%) are redundant. Our approximate method of eliminating redundant edges eliminate all but 19,456 of these redundant edges, reducing the CH-JP path query times by a factor of 1.42 and the preprocessing times by a factor of 2.06 (since fewer witness searches need to be performed). Eliminating all redundant edges further reduces the CH-JP path query times by a factor of 1.07, but increases the preprocessing times by a factor of 3.31 (due to the more expensive checks needed to determine whether each shortcut is redundant). In our experiments, we use the approximate method of eliminating redundant edges, whose preprocessing times are still high on random maps (∼1 hour on average, and up to ∼4.5 hours on the random10 maps).

Table 5.5 reports the CH-JP preprocessing, connection, search, refinement, and path query times relative to CH-SG (PT*, Cn*, Sr*, Rf*, and PQ*), the number of edges of contraction hierarchies on jump-point graphs relative to contraction hierarchies on subgoal graphs ($|E|$%), the CH-JP memory requirements relative to CH-SG (M*), and the number of vertices expanded and successors evaluated by CH-JP searches relative to CH-SG searches (Exp* and Succ*).

We make the following observations:

- **PT*:** As discussed in Section 5.5.5, jump-point graphs have significantly fewer edges than subgoal graphs on game and street maps. As a result, the construction of contraction hierarchies on jump-point graphs requires significantly fewer witness searches and, thus, runs 8.04 and 23.82 times faster than the construction of contraction hierarchies on subgoal graphs on game and street maps, respectively. The opposite case happens on maze, random, and room maps, since jump-point graphs typically have more edges than subgoal graphs on these types of maps. The CH-JP preprocessing times are 45.30 times longer than the CH-SG preprocessing times on random maps, despite the technique discussed above that we use to reduce preprocessing times.

- **$|E|$%:** The number of edges of contraction hierarchies on jump-point graphs is smaller than the number of edges of contraction hierarchies on subgoal graphs on

	CH-SG → CH-JP						
	PT	$\|E\|$%	M*	Exp*	Succ*	Sr*	PQ*
all	0.39	133.35	0.43	1.17	1.57	0.79	0.83
game-all	8.04	45.80	0.56	2.19	6.25	2.15	1.28
maze-all	0.33	104.36	0.50	1.01	1.66	0.92	0.89
random-all	0.02	169.85	0.31	0.51	0.48	0.35	0.43
room-all	0.14	123.54	0.48	0.70	0.76	0.57	0.67
street-all	23.82	23.24	0.74	3.54	12.48	3.49	1.26
bg	3.34	40.36	0.69	2.20	5.24	2.17	1.34
bg-512	8.25	31.75	0.52	2.39	6.35	2.41	1.20
dao	2.13	51.57	0.54	1.63	3.50	1.58	1.19
da2	8.84	42.81	0.52	2.06	5.57	2.13	1.26
sc	8.97	45.71	0.61	2.54	8.29	2.48	1.39
wc3-512	3.02	54.08	0.53	1.73	3.09	1.57	0.97
maze-1	0.23	128.72	0.44	0.79	1.11	0.59	0.80
maze-2	0.36	87.26	0.53	1.01	1.68	0.95	0.93
maze-4	0.50	72.43	0.53	1.11	1.90	1.11	1.08
maze-8	0.50	73.21	0.51	1.13	1.97	1.16	1.02
maze-16	0.47	72.96	0.50	1.18	2.08	1.22	0.91
maze-32	0.46	73.40	0.50	1.23	2.10	1.23	0.73
random-10	0.02	209.66	0.25	0.44	0.36	0.29	0.34
random-15	0.03	184.26	0.28	0.45	0.41	0.31	0.36
random-20	0.04	163.52	0.32	0.47	0.47	0.33	0.38
random-25	0.07	147.94	0.35	0.51	0.56	0.37	0.43
random-30	0.09	136.60	0.38	0.55	0.64	0.41	0.50
random-35	0.10	127.73	0.41	0.60	0.77	0.48	0.62
random-40	0.21	122.37	0.44	0.67	0.89	0.55	0.87
room-8	0.13	126.08	0.45	0.61	0.67	0.49	0.59
room-16	0.23	116.84	0.49	0.75	0.86	0.61	0.70
room-32	0.39	110.15	0.50	0.86	1.05	0.81	0.82
room-64	0.45	108.05	0.50	1.01	1.26	0.93	0.73
street-256	4.76	42.75	0.69	2.07	3.89	1.84	1.18
street-512	13.67	26.18	0.75	3.04	8.11	2.79	1.25
street-1024	27.53	14.45	0.74	5.30	24.75	5.45	1.29

Table 5.7: Contraction hierarchies on jump-point graphs.

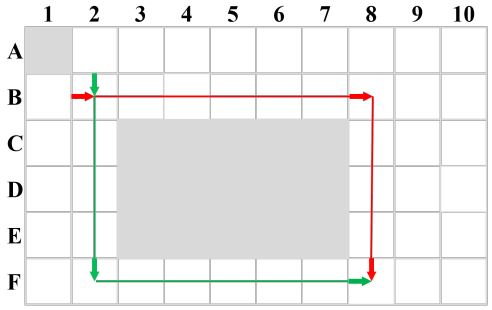


Figure 5.23: Symmetric paths on jump-point graphs with different topologies. Contraction hierarchies constructed on jump-point graphs might need to preserve all such paths to guarantee the finding of shortest paths on G.

game and street maps, but greater on maze, random, and room maps. These results are similar to the relative numbers of edges of subgoal graphs and jump-point graphs discussed in Section 5.5.5.

- **M*:** CH-JP requires 2.33 times more memory than CH-SG on average, since it stores two sets of clearance values, stores more edges than CH-SG, and cannot avoid storing "reversed downward edges" (since, as discussed in Section 5.5.1, contraction hierarchies on jump-point graphs are directed graphs and their reversed downward edges are not equivalent to their upward edges).

- **Sr*, Exp*, and Succ*:** The CH-JP search times are 2.15 times and 3.49 times shorter than the CH-SG search times on game and street maps, respectively, but 1.09, 2.88, and 1.75 times longer on maze, random, and room maps. That is, unlike the JP search times which are always shorter than the SG search times, the CH-JP search times are not always shorter than the CH-SG search times and, even when they are, the ratio of them is smaller than the ratio of the JP and SG search times. Similar trends hold for the number of expanded vertices and processed successors. Although we do not have a complete explanation for why this is the case, we suspect that the following factors contribute to these results:

  1. As discussed in Section 5.5.5, each convex corner cell typically contains between two to four jump points, but exactly one subgoal. Unlike JP searches, CH-JP searches do not "group together" jump-points that share the same cell, and might expand each of them separately. We have experimented with a version of CH-JP that prunes jump points with provably suboptimal g-values based on the g-values of other jump-points at the same cell. However, this approach slightly increased the CH-JP search times due to the overhead of performing this check, which typically prunes only one or two jump-points from the search. We suspect that stall-on-demand already prunes most of these jump points from the search.

2. As discussed earlier in this section, contraction hierarchies on jump-point graphs may include redundant edges that correspond to suboptimal paths on G. We suspect that a similar problem occurs where contraction hierarchies on jump-point graphs preserve multiple *shortest* paths with different topologies between two cells, as shown in Figure 5.23. However, we do not think that these paths can be considered to be redundant: For instance, if the red path in Figure 5.23 is not preserved and the start vertex is connected to the red jump point at B2 but not the green one, then it might not be possible to find a shortest path from the start vertex to F8. We reiterate that our preprocessing eliminates most of the redundant (suboptimal) edges from contraction hierarchies on jump-point graphs, and that our preliminary results show that even if all redundant edges are eliminated, the CH-JP search times are still longer than CH-SG search times on random and room maps.
3. Although CH-JP searches expand fewer vertices and process fewer successors than CH-SG searches on maze maps, CH-JP search times are longer than CH-SG search times on maze maps. We suspect that this is due to CH-SG maintaining smaller data structures that account for a smaller set of vertices (subgoals, as opposed to jump points) and edges (since CH-SG discards "reversed downward edges", unlike CH-JP). As a result, cache misses might be less frequent during CH-SG searches compared to CH-JP searches.

- **PQ*:** The relative path query times of CH-JP and CH-SG mirror the relative search times of CH-JP and CH-SG. The CH-JP path query times are shorter than the CH-SG path query times on game and street maps, but 1.20 times longer across all maps. We have also experimented with versions of CH-SG and CH-JP that store additional clearance values to speed up their connection phases, which reduce the CH-SG path query times by a factor of 1.02 and the CH-JP path query times by a factor of 1.09, resulting in CH-JP path query times being only 1.11 times longer than the CH-SG path query times across all maps.

To summarize, CH-JP addresses the staircase problem of CH-SG discussed in Section 5.5.7, and therefore has shorter preprocessing and path query times on game and street maps, but higher memory requirements. However, across all maps, CH-JP has longer preprocessing and query times than CH-SG, and higher memory requirements. We do not clearly understand why the CH-JP path query times are longer than the CH-SG path query times even though the JP path query times are shorter than the SG path query times, but suspect that having multiple copies of vertices that represent different incoming directions to cells results in contraction hierarchies on jump-point graphs having to preserve shortest paths between more pairs of vertices compared to contraction hierarchies on subgoal graphs. We have identified one such problem where contraction hierarchies on jump-point graphs preserve paths that are shortest paths on jump-point graphs but not on G. However, addressing this problem did not significantly change the results.

5.5.9 R-Refining R-Reachable Shortcut Edges

In this section, we compare CH+Rr (answering queries using contraction hierarchies and an R-refine algorithm) and CH-SG+Rr (answering queries using contraction hierarchies on subgoal graphs and an R-refine algorithm) with CH and CH-SG, respectively, with respect to their query times, for $R = \text{CF}$ and $R = \text{F}$. Recall that, during preprocessing, CH+Rr and CH-SG+Rr mark R-reachable shortcut edges in contraction hierarchies and, during 2-pointer unpacking, use an R-refine algorithm to refine R-reachable shortcut edges. The main idea behind CH+Rr and CH-SG+Rr is to capitalize on the efficient refinement algorithms on grid graphs to try and improve CH and CH-SG refinement times, without increasing their memory requirements.

Table 5.8 reports the percentage of shortcut edges that are R-reachable in contraction hierarchies (R%), and the CH+Rr refinement and path query times (Rf* and PQ*) relative to CH, for $R = \text{CF}$ and $R = \text{F}$. It also reports the same statistics for CH-SG+Rr relative to CH-SG. Since marking R-reachable edges during preprocessing requires little time, the preprocessing times are not reported. Since marking R-reachable edges does not require extra memory, the memory requirements are not reported either.

We make the following observations:

- **CH → CH+CFr:** The CH+CFr refinement times are 1.48 times shorter than the CH refinement times on average, but longer on maze maps with small corridor widths and random maps with large percentages of blocked cells. As discussed in Section 5.5.6, CF-Refine is typically faster than 2-pointer unpacking, especially when refining long edges. Therefore, the CH+CFr refinement times are typically shorter than the CH refinement times. However, CF-reachable edges are typically short on maze maps with small corridor widths and random maps with large percentages of blocked cells. Therefore, on these types of maps, the CH+CFr refinement times are longer than the CH refinement times. A significant percentage, on average 95.54%, of shortcut edges of contraction hierarchies are CF-reachable, which means that CF-Refine is used frequently during refinement. This percentage is lower (at 55.00%) on random maps, and decreases as the percentage of blocked cells increases, due to the random pattern discussed in Section 5.5.4. Since the CH+CFr refinement times are shorter than the CH refinement times, the CH+CFr path query times are also shorter than the CH path query times.

- **CH → CH+Fr:** The CH+Fr refinement times are 1.19 times *longer* than the CH refinement times on average. Recall that F-Refine, similar to CF-Refine, first calculates the directions and the number of diagonal and cardinal moves along the freespace-shortest paths between two vertices. However, unlike CF-Refine, which simply orders these moves according to a canonical ordering, F-Refine performs a depth-first search to find an ordering of these moves that corresponds to an unblocked freespace-shortest path on G. If the freespace-diagonal-first path between two F-reachable vertices is unblocked, the depth-first search of F-Refine simply generates this path, without backtracking (this is due to our implementation of the depth-first search, which generates successors using diagonal moves first). However, even if this is the case, F-Refine still has to verify that this path is unblocked on the

	CH → CH+Rr						CH-SG → CH-SG+Rr					
	R = CF			R = F			R = CF			R = F		
	R%	Rf*	PQ*	R%	Rf*	PQ*	R%	Rf*	PQ*	R%	Rf*	PQ*
all	95.54	1.48	1.10	96.81	0.84	0.95	31.16	1.11	1.04	49.91	0.56	0.76
game-all	99.74	4.51	1.11	99.92	1.70	1.05	71.01	1.28	1.04	91.00	0.64	0.91
maze-all	97.09	1.16	1.09	97.35	0.68	0.79	10.40	1.04	1.03	12.46	0.51	0.57
random-all	55.00	1.04	1.01	67.61	0.77	0.94	29.34	1.26	1.06	48.66	0.86	0.96
room-all	97.82	2.09	1.10	98.45	1.15	1.02	19.80	1.10	1.03	32.05	0.58	0.84
street-all	99.82	8.98	1.12	99.93	2.94	1.08	70.85	1.27	1.02	90.83	0.60	0.93
bg	98.89	1.81	1.06	99.49	0.88	0.98	75.92	1.24	1.05	89.59	0.84	0.96
bg-512	99.86	4.55	1.11	99.96	1.61	1.05	68.64	1.17	1.03	89.09	0.58	0.88
dao	99.33	3.16	1.14	99.75	1.32	1.04	69.59	1.29	1.06	89.12	0.70	0.91
da2	99.50	2.60	1.11	99.81	1.12	1.02	73.57	1.20	1.04	89.75	0.69	0.90
sc	99.78	5.79	1.11	99.93	2.00	1.06	71.52	1.36	1.04	92.51	0.62	0.92
wc3-512	99.73	5.20	1.10	99.93	2.01	1.06	65.61	1.11	1.02	87.00	0.57	0.88
maze-1	50.62	0.72	0.74	50.62	0.47	0.49	4.23	1.02	1.02	4.23	0.54	0.56
maze-2	79.39	0.86	0.88	80.38	0.52	0.56	14.56	1.04	1.03	14.56	0.54	0.59
maze-4	96.86	1.46	1.32	98.26	0.81	0.85	25.08	1.09	1.07	39.84	0.54	0.61
maze-8	99.37	2.12	1.49	99.73	1.06	1.04	25.40	1.08	1.06	40.80	0.46	0.55
maze-16	99.79	2.98	1.36	99.91	1.29	1.10	27.11	1.05	1.03	41.94	0.38	0.52
maze-32	99.94	4.60	1.16	99.98	1.49	1.06	28.57	1.03	1.01	43.96	0.36	0.57
random-10	65.45	1.67	1.08	81.50	1.27	1.04	33.44	1.44	1.05	63.11	0.93	0.99
random-15	55.41	1.34	1.04	69.65	1.17	1.02	29.37	1.40	1.05	51.07	1.06	1.01
random-20	49.26	1.15	1.02	60.61	0.99	1.00	27.25	1.32	1.04	42.97	1.02	1.00
random-25	46.14	1.04	1.01	55.40	0.83	0.97	26.40	1.28	1.05	38.74	0.95	0.99
random-30	45.34	0.97	0.99	53.09	0.73	0.93	26.74	1.25	1.06	37.00	0.89	0.97
random-35	46.96	0.90	0.97	53.54	0.64	0.85	28.32	1.22	1.08	37.47	0.83	0.92
random-40	49.52	0.81	0.87	55.19	0.52	0.64	30.86	1.18	1.13	39.41	0.74	0.79
room-8	85.00	1.27	1.06	88.90	0.86	0.96	19.84	1.13	1.03	31.83	0.68	0.89
room-16	97.59	1.85	1.14	98.40	1.08	1.02	19.65	1.08	1.02	32.08	0.59	0.84
room-32	99.52	2.70	1.14	99.72	1.29	1.04	19.63	1.12	1.03	34.80	0.54	0.80
room-64	99.88	4.13	1.09	99.93	1.56	1.04	22.04	1.03	1.01	42.45	0.45	0.74
street-256	99.32	3.15	1.08	99.69	1.43	1.03	64.50	1.15	1.02	85.61	0.74	0.94
street-512	99.76	7.00	1.12	99.90	2.54	1.08	69.57	1.23	1.02	90.61	0.63	0.93
street-1024	99.93	13.25	1.12	99.98	3.60	1.09	78.53	1.38	1.02	95.64	0.53	0.93

Table 5.8: R-refining R-reachable edges of contraction hierarchies.

grid graph because, different from CF-reachable vertices, the freespace-diagonal-first or freespace-cardinal-first path between two F-reachable vertices is not necessarily unblocked. That is, F-Refine is slower than CF-Refine even if its depth-first search does not backtrack, since it needs to access memory to determine whether certain paths are unblocked on G. As a result of this overhead (memory access and possible backtracking), unlike CF-Refine, F-Refine is typically slower than 2-pointer unpacking and, therefore, the CH+Fr refinement times are typically longer than the CH refinement times. An exception to this trend occurs on scaled-up maps and most game maps, since these maps typically contain large contiguous regions of unblocked cells and, therefore, the freespace-diagonal-first paths that correspond to F-reachable edges are typically unblocked.

- **CH-SG → CH-SG+CFr:** The CH-SG+CFr refinement times are 1.11 times shorter than the CH-SG refinement times on average, and consistently shorter on each type of map. The improvement factor of the refinement times for using CH-SG+CFr over CH-SG (1.11) is smaller than it is for using CH+CFr over CH (1.48), since, as discussed in Section 5.5.7, the CH-SG refinement times are already shorter than the CH refinement times. However, unlike the CH+CFr refinement times compared to the CH refinement times, the CH-SG+CFr refinement times are *consistently* shorter than the CH-SG refinement times, across all benchmarks. This is due to the following reason. Since shortcut edges in contraction hierarchies on subgoal graphs (and also on G) are formed by combining two (shortcut) edges, they are typically longer than the edges of subgoal graphs. Therefore, the (shortcut) edges refined by CF-Refine during the CH-SG+CFr refinement are typically longer than the edges refined by CF-Refine during the CH-SG refinement. That is, the overhead of calculating the directions and number of diagonal and cardinal moves along freespace-shortest paths are incurred less frequently during the CH-SG+CFr refinement than the CH-SG refinement and, therefore, the CH-SG+CFr refinement times are consistently shorter than the CH-SG refinement times.

- **CH-SG → CH-SG+Fr:** CH-SG+Fr refinement is 1.78 times slower than CH-SG refinement on average. As discussed earlier, this is due to 2-pointer unpacking typically being faster than F-Refine, and also due to CH-SG refinement being already fast.

To summarize, marking shortcut edges for R-refinement has no downsides if R-refinement can be performed faster than 2-pointer unpacking. This idea can be improved in several ways: 1) Marking only those edges for R-refinement where R-refinement runs faster than 2-pointer unpacking would guarantee that there is no deterioration in refinement times (such as the CH+CFr refinement times on the random-40 and maze-1 maps). 2) Using several reachability relations and marking shortcut edges accordingly could help with refining more edges using efficient refinement procedures. For instance, vertices between which the freespace-diagonal-first path is unblocked and between which the freespace-cardinal-first path is unblocked can be marked separately, and refined using CF-Refine accordingly. 3) The bits that are no longer used for storing 2-pointer unpacking information could be utilized to store additional information that can be used to refine

the edges. For instance, the directions and number of moves along freespace-shortest paths can be stored (instead of computed during refinement). 4) Storing the R-reachable and non-R-reachable shortcut edges in hierarchies separately could allow for the storage of R-reachable edges using less memory (for instance, with 4 bytes for CF- or F-reachable edges rather than 16 bytes).

5.5.10 R Contraction Hierarchies

In this section, we compare R-CH (answering queries using R contraction hierarchies) and R-CH-SG (answering queries using R contraction hierarchies on subgoal graphs) with CH and CH-SG, respectively, with respect to their query times and memory requirements, for R = CF and R = F. Recall that R contraction hierarchies are constructed similarly to contraction hierarchies, except that no vertex is contracted whose contraction introduces a shortcut edge that is not R-reachable. Therefore, R contraction hierarchies have R-reachable shortcut edges only, but may contain a core of uncontracted vertices. As discussed in Section 3.4.4, the bidirectional searches used for searching contraction hierarchies can be modified so that the core vertices are expanded by the forward (or the backward) search but not the backward (or forward) search. The main idea behind R-CH and R-CH-SG is to: 1) capitalize on efficient refinement algorithms for grid graphs to try and improve the CH and CH-SG refinement times, even more so than for CH+Rr and CH-SG+Rr, by ensuring that all edges in the hierarchies can be refined using R-Refine; and 2) reduce the memory requirements by storing each edge using less memory (4 bytes instead of 16 bytes). The main downside of R-CH and R-CH-SG compared to CH and CH-SG, respectively, is that their hierarchies are restricted to use only R-reachable edges and might have uncontracted cores, which can result in longer search times.

Table 5.9 reports the percentage of vertices of R contraction hierarchies that remain in their cores (Core%), the R-CH memory requirements (M*), the distance query times (DQ*), refinement times (Rf*), and path query times (PQ*) relative to CH, for R = CF and R = F. Table 5.10 reports the same statistics for R-CH-SG relative to CH-SG. The R-CH preprocessing times are similar to the CH preprocessing times, and the R-CH-SG preprocessing times are similar to the CH-SG preprocessing times, and thus not reported.

We make the following observations:

- **Core%, CH → R-CH:** The cores of CF and F contraction hierarchies are typically small, containing on average 4.17% and 3.50% of the vertices of G, respectively. Since F ⊇ CF, F contraction hierarchies have fewer constraints on how they can be constructed than CF contraction hierarchies, and therefore consistently have smaller cores than CF contraction hierarchies across all types of maps. As discussed in Section 5.5.4, the random patterns of random maps typically result in only nearby pairs of vertices to be CF- or F-reachable, therefore imposing even more constraints on how CF and F contraction hierarchies can be constructed. As a result, the cores of CF and F contraction hierarchies on random maps are larger, containing 25.42% and 21.36% of the vertices of G, respectively. In maze maps, especially those with small corridor widths, it is also typically the case that only nearby pairs of vertices are CF- or F-reachable. Interestingly, CF and F contraction

	CH → CF-CH					CH → F-CH				
	Core%	M*	DQ*	Rf*	PQ*	Core%	M*	DQ*	Rf*	PQ*
all	4.17	4.03	0.27	3.24	0.35	3.50	4.03	0.30	1.09	0.37
game-all	0.16	4.00	0.86	6.39	0.97	0.11	4.00	0.93	1.77	0.80
maze-all	0.00	4.03	0.48	2.76	0.91	0.00	4.03	0.49	0.93	0.22
random-all	25.42	4.36	0.03	1.89	0.04	21.36	4.30	0.04	0.84	0.04
room-all	1.46	4.02	0.29	3.89	0.35	1.29	4.01	0.31	1.36	0.16
street-all	0.13	4.00	0.89	11.34	1.00	0.10	4.00	0.94	3.29	0.90
bg	0.91	4.00	0.82	2.29	0.90	0.67	4.00	0.87	0.88	0.83
bg-512	0.09	4.00	0.92	5.88	1.03	0.06	4.00	0.98	1.65	0.89
dao	0.46	4.00	0.76	4.61	0.89	0.27	4.00	0.87	1.41	0.76
da2	0.21	3.98	0.88	3.45	1.00	0.12	3.98	0.94	1.06	0.89
sc	0.12	4.00	0.87	8.78	0.97	0.08	4.00	0.93	2.12	0.78
wc3-512	0.14	4.00	0.89	6.36	0.98	0.10	4.00	0.94	2.04	0.87
maze-1	0.00	4.45	0.07	1.73	0.51	0.00	4.45	0.07	0.69	0.09
maze-2	0.00	4.10	0.13	2.20	0.63	0.00	4.10	0.13	0.83	0.15
maze-4	0.00	4.03	0.37	3.51	1.19	0.00	4.01	0.43	1.08	0.43
maze-8	0.00	4.00	0.68	4.68	1.47	0.00	4.00	0.75	1.24	0.60
maze-16	0.00	4.00	0.92	5.95	1.38	0.00	4.00	0.95	1.30	0.77
maze-32	0.00	4.00	0.97	7.05	1.14	0.00	4.00	1.01	1.39	0.91
random-10	24.56	4.38	0.07	3.77	0.09	18.55	4.26	0.09	1.67	0.10
random-15	29.03	4.47	0.05	3.03	0.06	23.82	4.38	0.06	1.34	0.07
random-20	30.12	4.46	0.04	2.54	0.05	25.76	4.41	0.05	1.12	0.05
random-25	28.52	4.36	0.03	2.18	0.04	24.88	4.33	0.03	0.95	0.04
random-30	25.02	4.23	0.02	1.83	0.03	21.98	4.21	0.03	0.82	0.03
random-35	19.57	4.10	0.01	1.54	0.02	17.21	4.08	0.02	0.68	0.02
random-40	13.26	4.06	0.01	1.18	0.03	11.65	4.06	0.01	0.53	0.01
room-8	4.91	4.06	0.08	2.58	0.10	4.35	4.05	0.08	1.01	0.09
room-16	1.16	4.01	0.26	3.51	0.35	1.02	4.01	0.28	1.30	0.23
room-32	0.26	4.00	0.67	4.73	0.80	0.23	4.00	0.71	1.54	0.55
room-64	0.05	4.00	0.94	6.41	1.04	0.05	4.00	0.98	1.84	0.81
street-256	0.53	3.99	0.84	3.74	0.91	0.41	3.99	0.89	1.45	0.85
street-512	0.17	4.00	0.89	8.92	1.00	0.12	4.00	0.95	2.87	0.88
street-1024	0.04	4.00	0.90	17.04	1.01	0.03	4.00	0.94	4.08	0.92

Table 5.9: R contraction hierarchies.

	CH-SG → CF-CH-SG					CH-SG → F-CH-SG				
	Core%	M*	DQ*	Rf*	PQ*	Core%	M*	DQ*	Rf*	PQ*
all	51.75	1.64	0.12	2.01	0.20	43.18	1.64	0.14	0.70	0.21
game-all	11.88	1.19	0.64	2.94	0.74	5.41	1.19	0.80	0.87	0.81
maze-all	0.01	1.25	0.22	1.81	0.72	0.01	1.25	0.22	0.62	0.45
random-all	65.07	3.21	0.04	2.39	0.05	55.34	3.14	0.04	1.13	0.06
room-all	81.29	1.20	0.14	2.17	0.19	67.68	1.20	0.16	0.78	0.20
street-all	18.26	1.45	0.75	3.01	0.82	6.60	1.45	0.90	0.95	0.91
bg	11.58	1.38	0.76	3.56	0.93	7.35	1.38	0.83	1.36	0.92
bg-512	13.36	1.05	0.81	2.77	0.93	7.06	1.05	0.89	0.78	0.86
dao	14.29	1.13	0.60	2.92	0.74	7.03	1.13	0.76	0.96	0.79
da2	7.36	1.05	0.81	3.05	1.00	3.68	1.05	0.89	0.98	0.91
sc	10.96	1.34	0.59	2.96	0.67	4.15	1.34	0.78	0.82	0.79
wc3-512	16.01	1.09	0.80	2.79	0.92	9.64	1.09	0.87	0.89	0.88
maze-1	0.00	1.68	0.09	1.69	0.58	0.00	1.68	0.09	0.70	0.40
maze-2	0.00	1.48	0.15	1.84	0.66	0.00	1.48	0.15	0.71	0.44
maze-4	0.01	1.20	0.36	2.13	0.97	0.01	1.19	0.43	0.65	0.58
maze-8	0.04	1.07	0.52	1.93	1.02	0.04	1.06	0.60	0.49	0.52
maze-16	0.14	1.02	0.72	1.76	1.08	0.14	1.02	0.77	0.35	0.46
maze-32	0.54	1.00	0.89	1.73	1.13	0.54	1.00	0.91	0.29	0.48
random-10	87.72	3.93	0.08	2.87	0.10	69.95	3.73	0.10	1.38	0.11
random-15	81.73	3.81	0.06	2.83	0.07	68.28	3.70	0.07	1.33	0.08
random-20	74.76	3.51	0.05	2.71	0.05	64.32	3.46	0.05	1.28	0.06
random-25	66.29	3.14	0.03	2.55	0.04	57.80	3.11	0.04	1.18	0.05
random-30	56.37	2.75	0.03	2.42	0.04	49.36	2.73	0.03	1.13	0.04
random-35	43.86	2.34	0.02	2.33	0.03	38.40	2.34	0.02	1.10	0.03
random-40	29.75	1.82	0.01	2.08	0.05	25.96	1.80	0.01	0.99	0.05
room-8	81.14	1.55	0.08	2.04	0.11	67.92	1.55	0.09	0.86	0.12
room-16	82.76	1.15	0.20	2.14	0.27	68.20	1.15	0.23	0.81	0.29
room-32	80.10	1.03	0.48	2.40	0.62	64.77	1.03	0.55	0.76	0.59
room-64	70.43	1.01	0.78	2.28	0.97	56.45	1.01	0.81	0.63	0.75
street-256	22.36	1.56	0.71	3.14	0.82	10.72	1.56	0.85	1.27	0.90
street-512	19.36	1.49	0.70	2.99	0.78	6.44	1.50	0.88	1.02	0.90
street-1024	12.70	1.39	0.81	2.99	0.86	3.26	1.40	0.92	0.81	0.91

Table 5.10: R contraction hierarchies on subgoal graphs.

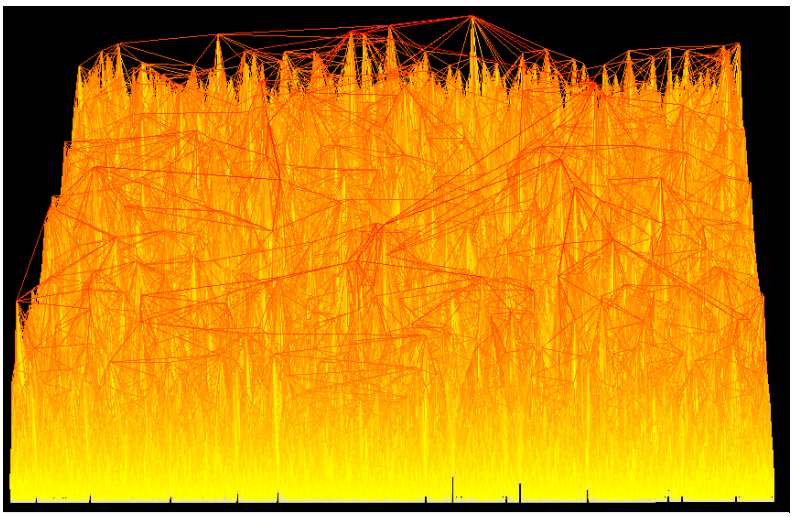
hierarchies constructed on maze maps do not have cores (except for the highest-level vertex), that is, all vertices are contracted when forming these hierarchies, similarly to contraction hierarchies. However, CF and F contraction hierarchies on maze maps have very different structures than contraction hierarchies, as shown in Figure 5.24. Observe that up-down paths on the F contraction hierarchy in Figure 5.24b use longer sequences of shorter edges than on the contraction hierarchy in Figure 5.24a. As a result, the F contraction hierarchy in Figure 5.24b has more levels to guarantee that the levels of vertices on up-down paths first strictly increase and then strictly decrease.

- **M*, CH → *R*-CH:** Both the CF-CH and F-CH memory requirements are 4.03 times lower than the CH memory requirements on average, since CF-CH and F-CH require 4 times less memory to store each edge than CH. This result also indicates that CF and F contraction hierarchies have roughly the same number of edges as contraction hierarchies.

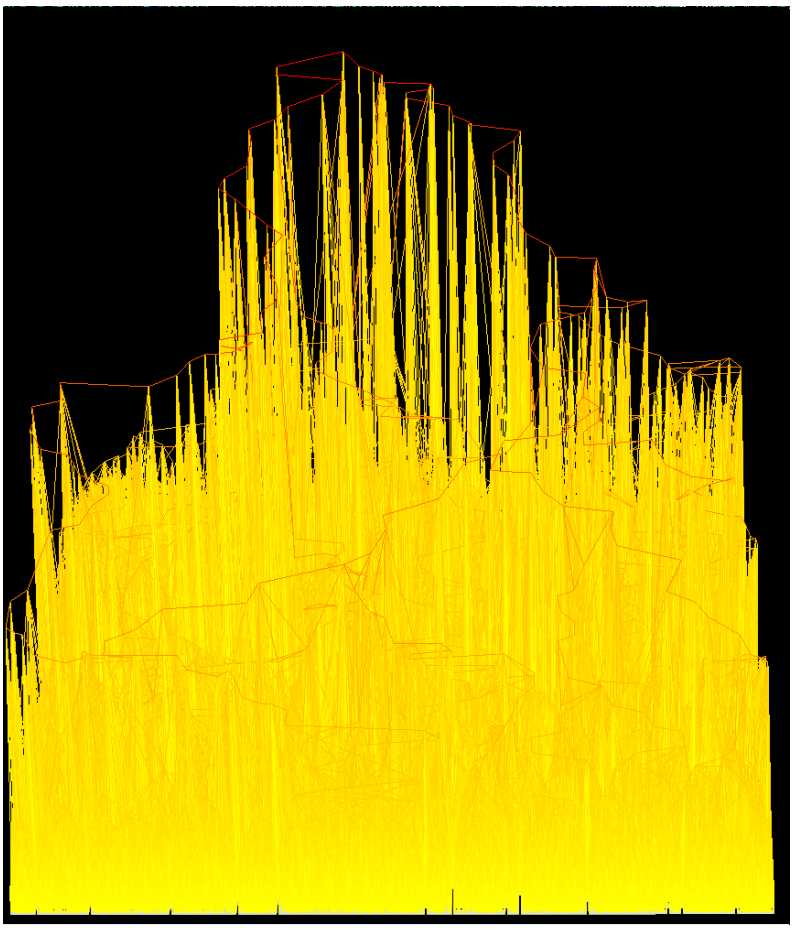
- **DQ*, CH → *R*-CH:** The CF-CH and F-CH distance query (search) times are significantly longer than the CH distance query times, on average by factors of 3.75 and 3.37, respectively, and on random maps by factors of 31.68 and 27.74, respectively. As discussed earlier, the deterioration in path query times is worse for CF-CH since CF contraction hierarchies have more constraints on how they can be constructed compared to F contraction hierarchies, resulting in larger cores. The deterioration in path query times for both CF-CH and F-CH is typically small on scaled-up maps, which, as discussed in Section 5.5.4, have large contiguous areas of unblocked cells that allow distant pairs of vertices to be CF or F-reachable.

- **Rf*, CH → *R*-CH:** The CF-CH refinement times are 3.24 times shorter than the CH refinement times on average. Furthermore, they are 2.19 times shorter than the CH+CF refinement times and 1.27 times shorter than SG refinement times, making CF-CH, along with CF-CH-SG and CF-N-SG (that we discuss in the next section), all algorithm with the shortest refinement times among the algorithms listed in Table 5.1 (except for A*, which does not perform refinement). The CF-CH refinement times are shorter than the SG refinement times since the shortcut edges of CF contraction hierarchies can be longer than the edges of subgoal graphs (which contain only SF-reachable edges, where SF $\subseteq$ CF), requiring CF-Refine to calculate the directions and number of moves along freespace-shortest paths less frequently when refining paths found on CF contraction hierarchies. The F-CH refinement times are 1.09 times shorter than the CH refinement times on average, and 2.98 times longer than the CF-CH refinement times, due to the overhead of having to access memory and perform depth-first searches, as discussed in Section 5.5.9.

- **PQ*, CH → *R*-CH:** The CF-CH and F-CH path query times are longer than the CH path query times, on average by factors of 0.35 and 0.37, respectively. On game, maze, and street maps, the CF-CH path query times are similar to the CH path query times, indicating that the reduction in the refinement times can make up for the increase in the path query times. The CF-CH path query times are also shorter



(a) Contraction hierarchies can have long shortcut edges that are not freespace reachable, allowing for the uniform contraction of vertices.



(b) F contraction hierarchies can only have F-reachable shortcut edges, which limits how they can be constructed: Vertices whose contraction would introduce non-F-reachable shortcut edges are not contracted, interfering with the contraction order and typically resulting in an uncontracted core. In this example, although all the vertices are contracted, the resulting hierarchy is significantly different from the contraction hierarchy shown in (a).

Figure 5.24: Contraction and F contraction hierarchies constructed on maze-512-8-0.map (maze map with corridor width 8). The higher-level vertices are shown higher from the grid. The edges are colored based on the level of their source vertex, on a scale from yellow (low level) to red (high level).

than the F-CH path query times on these types of maps, but longer on random and room maps, making them longer across all maps.

- **CH-SG $\rightarrow$ R-CH-SG:** The trends in distance query times and memory requirements of CF-CH-SG and F-CH-SG relative to CH-SG are similar to the trends in distance query times and memory requirements of CF-CH and F-CH relative to CH. However, the improvements in refinement times and memory requirements and the deterioration in distance and path query times of CF-CH-SG and F-CH-SG relative to CH-SG are worse than they are for CF-CH and F-CH relative to CH, since The reason for this result is that CH-SG has shorter refinement, path, and distance query times, and lower memory requirements than CH, as discussed in Section 5.5.7. We discuss three cases where this fact results in different CF-CH-SG and F-CH-SG improvement factors over the CH-SG compared to CF-CH and F-SG improvement factors over CH: 1) The numbers of vertices in the cores of CF and F contraction hierarchies on subgoal graphs are similar to the numbers of vertices in the cores of CF and F contraction hierarchies (on G), respectively (not reported in the table). However, since subgoal graphs are smaller than G, the sizes of their cores relative to G are larger for hierarchies constructed on subgoal graphs. 2) The CF-CH-SG and F-CH-SG refinement times are similar to the CF-CH and F-CH refinement times, respectively (not reported in the table). However, since the CH-SG refinement times are shorter than the CH refinement times, the improvement in the refinement times of CF-CH-SG and F-CH-SG relative to CH-SG is less than the improvement in the refinement times of CF-CH and F-CH relative to CH, respectively. 3) CF-CH-SG, F-CH-SG, CF-CH, and F-CH all store shortcut edges using 4 times less memory than CH-SG and CH. However, CF-CH-SG, F-CH-SG, and CH-SG also store clearance values and, therefore, the improvement in the CF-CH-SG and F-CH-SG memory requirements relative to CH-SG is less than a factor of 4.

To summarize, using R contraction hierarchies rather than contraction hierarchies for answering path-queries trades off lower memory requirements for longer distance and path query times. This trade-off is typically worse (that is, reduces memory requirements less but increases query times more) when subgoal graphs are used as base graphs, worse when distant pairs of vertices are typically not R-reachable (for instance, on random maps), and better when they are (for instance, on scaled-up maps).

5.5.11 N-Level Subgoal Graphs

In this section, we compare R-N-SG (answering queries using R N-level subgoal graphs) with R-CH-SG with respect to their path query times and memory requirements, for $R = \text{CF}$ and $R = \text{F}$. Recall that, similar to R contraction hierarchies (on subgoal graphs), R N-level subgoal graphs have R-reachable edges only. Different from R contraction hierarchies, R N-level subgoal graphs can have same-level edges between non-core vertices, with the caveat that any arching path on them can use at most one non-core same-level edge. As discussed in Section 3.4.4, the bidirectional searches used for searching R contraction hierarchies can be modified so that the non-core same-level edges are considered only by the forward (or the backward) search but not the backward (or forward)

search, and successors generated through non-core same-level edges are never placed in the OPEN list. That is, the non-core same-level edges are used only to check if the forward and backward searches meet.

Our entry into the GPPC uses F N-level subgoal graphs, and is non-dominated with respect to its query-time/memory trade-off among all entries that are guaranteed to find shortest paths. As discussed in Section 5.5.1, F N-level subgoal graphs can be related to contraction hierarchies through three separate modifications to contraction hierarchies: 1) Constructing (contraction) hierarchies on subgoal graphs rather than on G (CH $\rightarrow$ CH-SG). As shown in Section 5.5.7, this modification can reduce both the query times and memory requirements of answering queries using contraction hierarchies. 2) Constructing hierarchies with F-reachable edges only (CH-SG $\rightarrow$ F-CH-SG). As shown in Section 5.5.6, this modification can reduce the refinement times and memory requirements, but can increase the search times and, ultimately, the path query times. 3) Allowing non-core same-level edges in the hierarchy (F-CH-SG $\rightarrow$ F-N-SG). As we show in this section, this modification can result in fewer expansions during the searches but more successors evaluated per expansion, ultimately resulting in slightly longer query times.

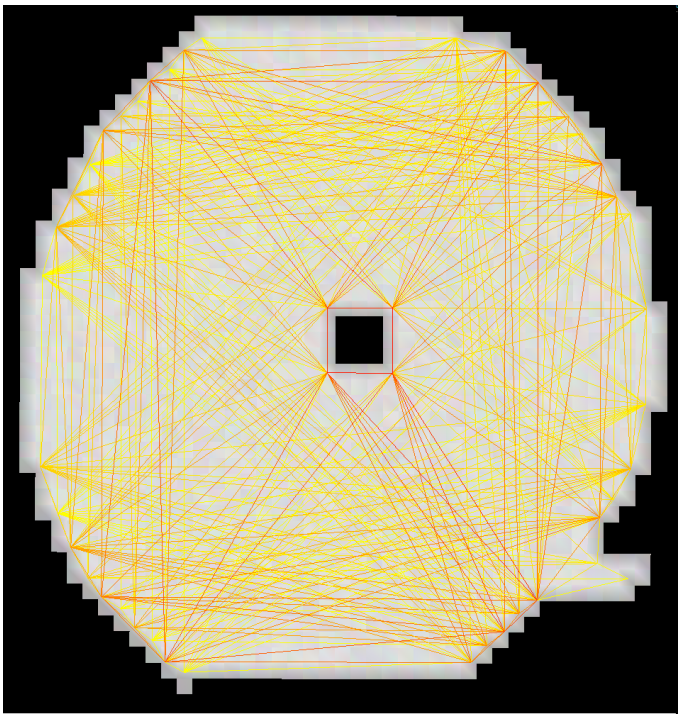
Table 5.11 reports the number of edges of R N-level subgoal graphs relative to R contraction hierarchies constructed on subgoal graphs ($|E|$%), the maximum level of vertices in R N-level subgoal graphs relative to R contraction hierarchies constructed on subgoal graphs ($\mathbb{L}$%), the number of vertices expanded (or stalled) and the number of successors evaluated by R-N-SG searches relative to R-CH-SG searches (Exp* and Succ*), and the R-N-SG search times relative to R-CH-SG (Sr*), for R = CF and R = F. The R-N-SG preprocessing times are similar to the R-CH-SG preprocessing times and thus not reported.

We make the following observations:

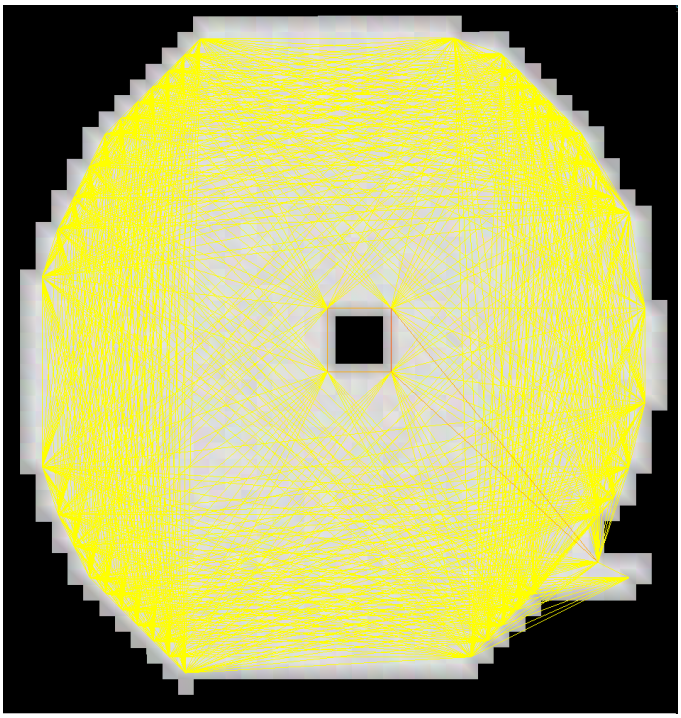
- **E%:** CF and F N-level subgoal graphs have on average 37.08% and 70.83% more edges than CF and F contraction hierarchies on subgoal graphs, respectively. We explain this result using a hypothetical example and a concrete one that is shown in Figure 5.25. Recall that N-level subgoal (overlay) graphs consist of $N-1$ extended overlay graphs (Section 3.4.1). Also recall that an extended overlay graph $G_{S,T}$ has an edge (u, v) if and only if $u, v \in T$ and v is direct-reachable from u with respect to S. That is, as S gets smaller, more pairs of vertices in T become direct-reachable with respect to S, and, therefore, more edges are included in $G_{S,T}$. In the extreme case, when $S = \emptyset$, the edges of $G_{S,T}$ form a clique of T, that is, $G_{S,T}$ has the maximum set of edges possible that can be added between the vertices in T. This hypothetical case can only occur if all pairs of vertices in T are R-reachable (otherwise, the N-level subgoal graph that contains $G_{S,T}$ would have non-R-reachable edges), and therefore rarely occurs in practice. However, it can still occur "locally", where non-core same-level edges can form cliques with many vertices. As the results show, this problem typically occurs on game and street maps, where staircase subgoals along one or multiple staircases can all be connected to each other through F- or CF-reachable edges. Figure 5.25b shows an example, where level 1 edges in an F 4-level subgoal graph form large cliques. Figure 5.25a shows an F contraction hierarchy constructed on a subgoal graph for comparison.

	CF-CH-SG → CF-N-SG					F-CH-SG → F-N-SG				
	E%	𝕃%	Exp*	Succ*	Sr*	E%	𝕃%	Exp*	Succ*	Sr*
all	137.08	86.24	1.01	0.97	0.93	170.83	81.49	1.01	0.94	0.94
game-all	183.92	65.42	1.08	0.90	0.84	253.50	49.34	1.23	0.79	0.83
maze-all	121.42	99.66	1.00	1.00	0.95	126.52	99.51	1.00	1.00	0.95
random-all	105.23	87.43	1.00	1.00	0.94	110.75	84.24	1.00	1.00	0.95
room-all	100.95	89.26	1.00	1.00	0.94	103.09	87.04	1.00	1.00	0.95
street-all	247.59	72.17	1.08	0.77	0.77	416.22	43.03	1.33	0.54	0.72
bg	315.22	55.02	1.16	0.79	0.87	397.66	44.47	1.32	0.70	0.88
bg-512	191.31	59.28	1.12	0.89	0.87	261.26	43.46	1.29	0.79	0.88
dao	191.68	65.19	1.08	0.91	0.87	257.43	53.71	1.19	0.79	0.85
da2	226.49	67.83	1.15	0.85	0.88	290.53	51.45	1.30	0.74	0.87
sc	171.66	73.48	1.07	0.91	0.83	243.59	48.94	1.23	0.80	0.81
wc3-512	160.63	67.16	1.07	0.95	0.86	194.86	52.24	1.16	0.91	0.87
maze-1	103.32	100.01	1.00	1.00	0.95	103.32	100.01	1.00	1.00	0.95
maze-2	128.61	99.85	1.00	1.00	0.95	128.61	99.85	1.00	1.00	0.95
maze-4	145.45	99.31	1.00	1.00	0.92	166.21	98.53	1.01	1.00	0.95
maze-8	145.24	98.70	1.00	1.00	0.94	166.48	97.70	1.01	0.99	0.93
maze-16	143.74	97.13	1.01	0.99	0.94	163.73	95.44	1.02	0.98	0.94
maze-32	142.33	94.38	1.02	0.99	0.95	162.79	89.00	1.04	0.97	0.95
random-10	101.03	79.26	1.00	1.00	0.94	107.11	77.53	1.00	1.00	0.94
random-15	102.07	87.02	1.00	1.00	0.94	107.28	68.75	1.00	1.00	0.96
random-20	103.64	82.86	1.00	1.00	0.94	108.39	86.01	1.00	1.00	0.93
random-25	106.00	85.81	1.00	1.00	0.96	110.92	86.75	1.00	1.00	0.97
random-30	108.99	85.79	1.00	1.00	0.94	114.61	81.97	1.00	1.00	0.96
random-35	113.41	89.96	1.00	1.00	0.93	120.34	85.93	1.00	1.00	0.95
random-40	118.50	91.67	1.00	1.00	0.95	127.10	92.84	1.00	1.00	0.96
room-8	100.96	90.24	1.00	1.00	0.94	102.94	86.17	1.00	1.00	0.95
room-16	100.79	95.89	1.00	1.00	0.95	103.16	90.00	1.00	1.00	0.96
room-32	101.17	89.19	1.00	1.00	0.93	104.70	88.57	1.00	1.00	0.94
room-64	102.76	81.16	1.01	1.00	0.95	106.00	83.75	1.01	1.00	0.95
street-256	202.12	71.66	1.05	0.91	0.86	306.82	48.32	1.18	0.79	0.88
street-512	223.80	73.63	1.06	0.86	0.81	371.63	40.71	1.28	0.65	0.80
street-1024	285.98	69.97	1.11	0.70	0.71	494.26	37.79	1.45	0.47	0.64

Table 5.11: R N-level subgoal graphs.



(a) An F contraction hierarchy on subgoal graph with 12 levels and 366 edges. Since F contraction hierarchies cannot have non-core same-level edges, the non-core vertices cannot form cliques.



(b) An F 4-level subgoal graph with 1,456 edges. Almost all staircase subgoals are level 1 vertices, and the edges between them form four large cliques, on the up, down, left, and right side of the obstacle in the middle. If the obstacle in the middle were to be removed, the edges between the staircase subgoals would form one large clique.

Figure 5.25: F contraction hierarchy on subgoal graph (with 12 levels) and F 4-level subgoal graph constructed on AR0606SR.map. The edges are colored based on the level of their source vertices, on a scale from yellow (low level) to red (high level).

Since R contraction hierarchies cannot have non-core same-level edges, they cannot contain such cliques.

Our algorithm for constructing N-level subgoal graphs amplifies the problem, since it generates every extended overlay graph $G_{S,T}$ in an N-level subgoal graph by starting with the overlay graph G_T and identifying S as a minimal (R,T)-SPC (Section 3.3.5). That is, it tries to minimize the number of vertices in S and, as a consequence, maximize the number of edges in $G_{S,T}$. Since F $\supseteq$ CF, our algorithm is able to identify a smaller S from a given T when constructing F N-level subgoal graphs, compared to when constructing CF N-level subgoal graphs. Therefore, the F N-level subgoal graphs that it constructs typically have more edges than the CF N-level subgoal graphs that it constructs.

- **$\mathbb{L}$%:** The maximum levels of CF and F N-level subgoal graphs are smaller than the maximum levels of CF and F contraction hierarchies on subgoal graphs, respectively, namely 13.76% and 19.51% smaller on average, 34.58% and 50.66% smaller on game maps, and 27.83% and 56.97% smaller on street maps. These results have an explanation similar to the larger numbers of edges in N-level subgoal graphs compared to R contraction hierarchies on subgoal graphs: The cliques formed by non-core same-level edges of R N-level subgoal graphs are not allowed in R contraction hierarchies (on subgoal graphs), where many pairs of vertices have to be assigned different levels to avoid such edges.

- **Exp*:** CF-N-SG and F-N-SG searches expand (or stall) fewer vertices on game and street maps than CF-CH-SG and F-CH-SG searches, respectively, by factors of 1.08 an 1.23 on game maps, and 1.08 and 1.33 on street maps. There is no noticeable difference on the other types of maps. Recall that neither R-N-SG nor R-CH-SG searches expand non-core vertices that cannot be reached with an upward path from the start vertex and cannot reach the goal vertex with a downward path. Since R N-level subgoal graphs typically have smaller maximum levels than R contraction hierarchies on subgoal graphs, especially on game and street maps with staircase patterns, more vertices have the *same* or a lower level than the start or goal vertices and can thus be ignored by searches.

- **Succ*:** CF-N-SG and F-N-SG searches evaluate more successors on game and street maps than CF-CH-SG and F-CH-SG searches, respectively, by factors of 1.11 an 1.26 on game maps, and 1.31 and 1.86 on street maps. This is a direct consequence of CF and F N-level subgoal graphs having more edges than CF and F contraction hierarchies on subgoal graphs, respectively: Although successors generated by non-core same-level edges are not placed in the OPEN list, they are still evaluated during searches to check if the forward and backward searches meet.

- **Sr*:** The CF-N-SG and F-N-SG search times are longer than the CF-CH-SG and F-CH-SG search times, respectively, by factors of 1.19 and 1.21 on game maps, 1.30 and 1.39 on street maps, and 1.06 and 1.05 on the other types of maps. The slight increase in search times across all types of maps is due to our R-N-SG implementation performing a check for every successor v of an expanded vertex u to determine

Benchmark	Map Type	Map Count	Instances Count	Instances Avg	Path length	Avg. G size $\|V\|$	Avg. G size $\|E\|$	A* time (μs)
GPPC	all	132	347,868	2,635	2,095	216,261	1,491,276	57,086
	game	105	142,534	1,357	488	48,250	367,992	4,086
	maze	9	145,976	16,220	4,180	939,083	7,008,834	120,289
	random	9	32,228	3,581	888	663,094	2,628,585	36,172
	room	9	27,130	3,014	749	1,006,724	7,941,382	20,309
MovingAI	all	769	1,642,670	2,136	1,137	109,752	776,644	9,898
	game	529	653,050	1,234	431	64,861	499,013	5,137
	maze	60	627,000	10,450	2,265	207,941	1,338,316	16,732
	random	70	155,750	2,225	483	185,864	937,004	5,508
	room	40	84,350	2,109	422	232,785	1,691,743	8,645
	street	70	122,520	1,750	454	218,423	1,710,030	6,738

Table 5.12: GPPC benchmarks.

whether (u, v) is a same-level edge. The increase in search times on game and street maps is due to the increased number of successors evaluated during the searches, as discussed above.

To summarize, allowing non-core same-level edges in R N-level subgoal graphs seem to negatively affect both the search times and memory requirements (due to larger numbers of edges), especially on game and street maps due to their staircase patterns.

5.5.12 Grid-Based Path-Planning Competition

In the previous sections, we have evaluated answering queries using subgoal graphs, jump-point graphs, contraction hierarchies, and their combinations and variants, with respect to their path query times, memory requirements, and other characteristics on the Moving-AI benchmarks, typically by comparing a small number of algorithms at a time and discussing their query-time/memory trade-offs across different types of maps. In this section, we evaluate the query-time/memory trade-offs of these algorithms on Grid-Based Path-Planning Competition (GPPC) benchmarks, both to summarize our results from earlier sections and to compare these algorithms to state-of-the-art path-planning algorithms on grid graphs that have been evaluated on GPPC benchmarks.

5.5.12.1 Competition Benchmarks

The GPPC benchmarks include 105 game maps from the MovingAI benchmarks (namely, 27 from dao, 67 from dao2, and 11 from sc), 9 new maze maps, 9 new random maps, and new room maps, for a total of 132 maps. Each maze, random, and room map in the GPPC benchmarks has a different size, varying from 100 $\times$ 100 to 1550 $\times$ 1550. The corridor widths in maze maps are equal to 1% of the map dimensions. For instance, on the 100 $\times$ 100 maze map, the corridor width is 1. The room sizes in room maps are equal to 10% of the map dimensions. That is, each room map contains exactly 100 rooms.

The random maps have 33% blocked cells. The GPPC benchmarks do not include street maps.

Table 5.12 reports the number of maps and instances, instance path lengths, graph sizes, and A* search times across different types of maps in the GPPC and MovingAI benchmarks (the statistics for the MovingAI benchmarks are the same as reported in Table 5.2). The maze, random, and room maps are significantly larger in the GPPC benchmarks then they are in the MovingAI benchmarks and, as a result, the instance path lengths are 1.84 times longer, the graphs have 1.97 times more vertices and 1.92 times more edges, and the A* search times are 5.77 times longer on GPPC benchmarks compared to the MovingAI benchmarks. Similar to the MovingAI benchmarks, the majority of maps in the GPPC benchmarks are game maps and the majority of instances are from game and maze maps.

Table 5.13 reports the preprocessing times (PT), memory requirements (M), and path query times (PQ) of the algorithms listed in Table 5.1 on the MovingAI and GPPC benchmarks. (It also reports normalized statistics for the GPPC benchmarks, which we discuss later.) We observe the following differences in results on the GPPC and MovingAI benchmarks:

- **PT:** The preprocessing times are generally longer on the GPPC benchmarks due to their larger graph sizes. However, since the GPPC benchmarks do not include street maps, the preprocessing times of the algorithms that use hierarchies on subgoal graphs (CH-SG, CH-SG+R, R-CH, and R-N-SG) are significantly shorter on the GPPC benchmarks. As discussed in Section 5.5.6, these algorithms typically have long preprocessing times on street maps due to their staircase patterns. Similarly, the CH-JP preprocessing times are shorter on the GPPC benchmarks, which include only random maps with 33% blocked cells. As discussed in Section 5.5.8, the CH-JP preprocessing times are longer on random maps with small percentages of blocked cells. The differences in the preprocessing times of the algorithms on the MovingAI and GPPC benchmarks are significant and affect their relative preprocessing times: For instance, the CH-SG preprocessing times are 1.22 times longer than the CH preprocessing times on the MovingAI benchmarks, but 10.72 times *shorter* on the GPPC benchmarks. That is, the relative preprocessing times of CH and CH-SG differ by a factor of 13.08 between the MovingAI and GPPC benchmarks.

- **M:** The memory requirements of each algorithm are higher on the GPPC benchmarks compared to the MovingAI benchmarks, due to the larger graph sizes in the GPPC benchmarks. However, the increase in memory requirements is about the same for each algorithm, ranging from a factor of 1.64 (for CH-SG) to a factor of 1.85 (for R-CH).

- **PQ:** The path query times are longer on the GPPC benchmarks due to the increased graph sizes and instance path lengths, and the increase in the path query times is significantly different for different algorithms: The R-CH, R-CH-SG, and R-N-SG path query times are increased by factors ranging from 4.72 to 5.43, the SG path query times are increased by a factor of 3.03, and the CH, CH+Rr, CH-SG, and CH-SG+Rr path query times are increased by factors ranging from 1.09 to 1.36.

	MovingAI			GPPC			GPPC (Normalized)		
	PT (s)	M (MB)	PQ (μs)	PT (s)	M (MB)	PQ (μs)	PT (s)	M (MB)	PQ (μs)
A*	-	-	9,898	-	-	57,086	-	-	110,501
SG	0	1.02	540	0	1.78	1,638	0	1.93	3,170
JP	0	1.81	409	0	3.26	1,397	0	3.54	2,705
CF-CH	98	3.01	205	125	5.57	1,016	241	6.04	1,967
F-CH	98	3.01	197	126	5.57	930	243	6.04	1,800
CF-N-SG	86	1.03	158	10	1.77	857	19	1.92	1,659
CF-CH-SG	95	0.97	148	7	1.71	813	13	1.86	1,574
F-N-SG	132	1.09	149	16	1.82	783	31	1.97	1,515
F-CH-SG	119	0.97	142	9	1.71	742	18	1.86	1,437
CH-GPPC	105	9.10	160	130	16.76	187	252	**18.18**	**362**
CH	105	12.13	73	130	22.35	99	252	24.25	191
CH+Fr	105	12.13	74	130	22.35	85	252	24.25	164
CH+CFr	105	12.13	65	130	22.35	71	252	24.25	138
CH-SG+Fr	127	1.59	39	12	2.61	47	24	2.83	90
CH-JP	325	3.72	35	65	5.73	43	125	6.22	82
CH-SG	127	1.59	29	12	2.61	34	24	2.83	66
*CH-SG+CFr	127	1.59	28	12	2.61	32	24	2.83	62
*JPS	-	-	-	-	-	-	-	-	62,524
JPS+(Harabor)	-	-	-	-	-	-	34	22.73	20,874
*BL-JPS	-	-	-	-	-	-	0	0.15	14,453
JPS+(Rabin)	-	-	-	-	-	-	0	7.17	7,732
*BL-JPS2	-	-	-	-	-	-	0	0.36	7,444
JPS+Bucket(Rabin)	-	-	-	-	-	-	0	7.17	1,616
*2-SG	-	-	-	-	-	-	0	0.70	1,429
*N-SG-GPPC	-	-	-	-	-	-	1	2.22	773
CH-GPPC	-	-	-	-	-	-	440	18.18	362
JPS+BB	-	-	-	-	-	-	1,386	15.15	149
SRC	-	-	-	-	-	-	5,605	393.94	145

Table 5.13: Results on MovingAI and GPPC benchmarks. The GPPC (Normalized) column contains the official GPPC results, as well as our results normalized to match the GPPC results, treating CH-GPPC as a baseline. During normalization, the preprocessing and path query times are multiplied by 1.94 and the memory requirement is multiplied by 1.08. The cells in each column are colored from red (worst) to white (median) to green (best), based on the logarithms of the values they contain. Algorithms with non-dominated query-time/memory trade-offs on the GPPC benchmarks are marked with asterisks.

The difference in the increase of the path query times of different algorithms can be explained by their query times on random maps: 1) Since SG, R-CH, R-CH-SG, and R-N-SG all use graphs or hierarchies with R-reachable edges only, they have long query times on random maps, as discussed in Sections 5.5.4, 5.5.6, and 5.5.10. 2) Since the GPPC benchmarks contain large random maps, the longer path query times on these maps can significantly affect the average path query times across all GPPC maps. Our results on the path query times on different types of maps (not reported in the table) confirm this observation: For instance, the SG path query times on GPPC game, maze, and room maps are 1.02, 2.44, and 1.32 times *shorter*, respectively, than the MovingAI benchmarks (the GPPC maze and room maps are large, but typically contain fewer convex corners). However, the SG path query times are 7.52 times longer on the GPPC random maps and, as a result they are 3.03 times longer across all GPPC maps.

To summarize, the results on the MovingAI and GPPC benchmarks are similar, except for: 1) the relatively short preprocessing times for hierarchies constructed on subgoal graphs, due to the exclusion of street maps from the GPPC benchmarks; and 2) the relatively long query times for SG, R-CH, R-CH-SG, and R-N-SG, due to their long query times on the larger random maps of the GPPC benchmarks.

5.5.12.2 Competition Results

We now compare our algorithms on grid graphs to the state-of-the-art path-planning algorithms of the GPPC, which we refer to as the *GPPC algorithms*. Seven of these algorithms have non-dominated query-time/memory trade-offs among the twenty-one algorithms that find shortest paths in GPPC. These algorithms include two of our entries based on subgoal graphs, namely 2-SG (called "Subgoal Graph (Low Memory)" in the GPPC) and N-SG-GPPC (called "NSubgoal" in the GPPC); three variants of jump-point search, namely JPS, BLJPS, and BLJPS2; the contraction hierarchy entry CH-GPPC (called "CH" in the GPPC); and the single-row compression entry SRC. 2-SG and N-SG-GPPC answer queries using F 2-level and N-level subgoal graphs, respectively, and were implemented in a different framework than the one that we use for our experiments. We explain the implementation differences and give a brief summary of how the other algorithms operate, during the discussion of our results. We also include three implementations of JPS+ in our comparison, whose query-time/memory trade-offs are all dominated by N-SG-GPPC in the GPPC. Finally, we include JPS+BB in our comparison, which combines JPS+ with bounding boxes, and was evaluated on the GPPC server using the GPPC benchmarks after GPPC was held for the last time. That is, it can be considered to be a GPPC entry whose results have not yet been officially announced. To the best of our knowledge, these eleven algorithms form a comprehensive list of state-of-the-art path-planning algorithms on grid graphs.

Our experiments were run on a different server (a 3.6GHz Intel Core i7-7700 with 32 GB of RAM) than the GPPC server (a 2.4 GHz Intel Xeon E5620 with 12 GB of RAM). In order to establish a meaningful comparison between our algorithms and the GPPC algorithms, we normalize the results on our server to match the results on the GPPC server by treating CH-GPPC as a baseline:

- We inflate the path query (and preprocessing) times measured on our server by a factor of 1.94, so that the path query time of our implementation of CH-GPPC matches the path query time of the CH-GPPC entry measured on the GPPC server. As discussed in Section 5.5.1, our implementation of CH-GPPC closely follows the implementation of the CH-GPPC entry. The main differences are that our CH-GPPC implementation uses floating point values for the edge lengths instead of integers, and keeps track of various statistics during queries. Our normalization of results also accounts for these minor implementation differences. Our preliminary results suggest that both implementations expand or stall similar numbers of vertices (within 1% of each other) and have similar query times on our server, with our implementation being slightly slower (by a factor of 1.18).
- We inflate our memory requirement estimations by a factor of 1.08, so that our estimated memory requirement for CH-GPPC matches the memory requirement measured by the GPPC. Recall that our estimation considers only the memory required to store the edges of graphs and, for algorithms that use subgoal graphs, the clearance values. Specifically, our estimation leaves out the amount of memory required to store the source vertices, which can be stored compactly by storing together edges that share the same source vertex. This normalization of the memory requirements also inflates the amount of memory required to store the clearance values, which does not require such inflation. However, we allow for this inflation to keep our normalized results as conservative as possible.

Table 5.13 reports the normalized preprocessing times, memory requirements, and path query times of our algorithms on the GPPC benchmarks, as well as the same statistics for the GPPC algorithms as measured in the GPPC. We now summarize these results by discussing the query-time/memory trade-offs of these algorithms, as shown in Figure 5.26, in decreasing order of their path query times. For brevity, we refer to the path query times simply as the query times, since we do not discuss the distance query times.

- **JPS, BL-JPS, BL-JPS2, and JPS+ variants:** JPS is the online version of jump-point search, and has a non-dominated query-time/memory trade-off since it does not perform preprocessing and does not store any information (Figure 5.26 reports its memory requirements as 0.1 megabytes since it uses a logarithmic scale for the memory requirements). JPS+(Harabor) is an earlier implementation of JPS+, which, as discussed in Section 5.4.1, uses precomputed clearance values to identify the jump-point successors of expanded vertices efficiently. JPS+(Rabin) is a more recent implementation of JPS+ and has a dominating query-time/memory trade-of compared to JPS+(Harabor). BL-JPS and BL-JPS2 can be considered to be variants of JPS+ that use "boundary-lookup" values, which can be considered as "compressed clearance values". BLJPS2 has slightly shorter query times compared to JPS+(Rabin) and requires significantly less memory, and, therefore, has a dominating query-time/memory trade-off compared to JPS+(Rabin). Both BL-JPS and BL-JPS2 have non-dominated query-time/memory trade-offs since they use very little memory. Finally, JPS+Bucket(Rabin) is a variant of JPS+(Rabin) that implements its OPEN list using buckets rather than a binary heap and runs 4.78 times

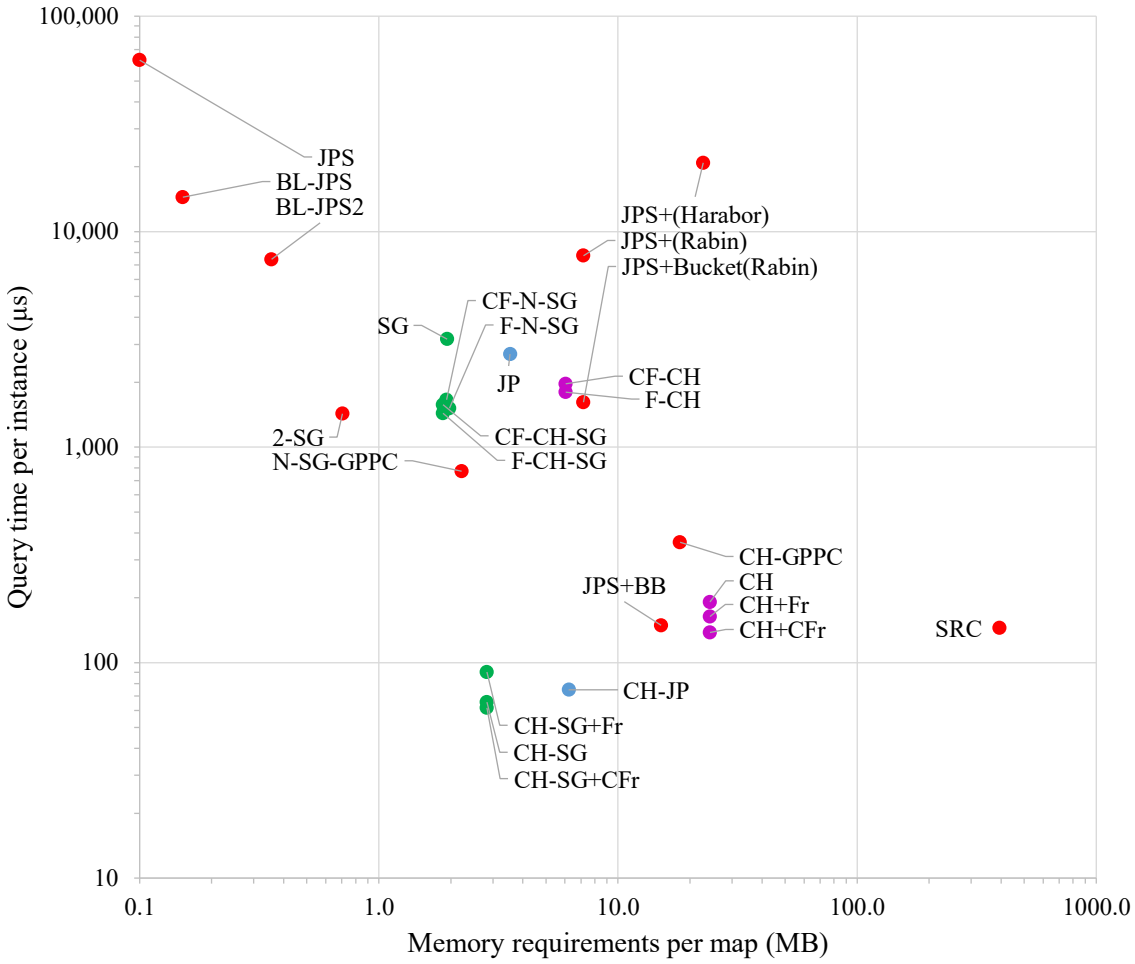


Figure 5.26: Query-time/memory trade-off of various algorithms on the GPPC benchmarks. Red dots: Official results from the GPPC. Green dots: Algorithms evaluated in this dissertation that use subgoal graphs. Blue dots: Algorithms evaluated in this dissertation that use jump-point graphs. Purple dots: Algorithms evaluated in this dissertation that use variants of contraction hierarchies. The query times and memory requirements of the algorithms evaluated in this dissertation are normalized the query times and memory requirements of CH-GPPC reported in the GPPC and our evaluation, as shown in Table 5.13.

faster than JPS+(Rabin) while using the same amount of memory, and, therefore, has a dominating query-time/memory trade-off compared to JPS+(Rabin). However, its query-time/memory trade-off is dominated by N-SG-GPPC.

- **SG, R-CH-SG, R-N-SG, and R-CH:** CF-CH-SG, F-CH-SG, CF-N-SG and F-N-SG all have very similar query times and memory requirements, as we have also observed in Sections 5.5.10 and 5.5.11, with F-CH-SG having the dominating query-time/memory trade-off compared to the other three algorithms. F-CH-SG also has a dominating query-time/memory trade-off compared to SG, CF-CH, and F-CH. Although we have not directly compared F-CH-SG with SG, CF-CH, or F-CH in our experiments, the results are consistent with our comparison of CH-SG with SG or CH: Queries can be answered faster by using a combination of subgoal graphs and contraction hierarchies, instead of using them individually.

- **JP:** JP, which explicitly constructs jump-point graphs, runs faster than JPS+(Rabin), which scans the grid whenever vertices are expanded. However, JP is slower than JPS+Bucket(Rabin). We suspect that using a bucket implementation of the OPEN list significantly decreases the query times on random maps, where JP and (presumably) all JPS variants have long query times due to a large number of expansions. On random maps, we expect the scanning of the grid to identify jump-point successors of expanded vertices to be quite fast since diagonals that extend from cells are typically very short. The query-time/memory trade-off of JP is dominated by F-CH-SG, 2-SG, and N-SG-GPPC. As discussed in Section 5.5.5, neither JP nor SG dominate each other with respect to their query-time/memory trade-offs.

- **2-SG and N-SG-GPPC:** 2-SG has slightly shorter query times compared to F-CH-SG but uses significantly less memory, and, therefore, has a dominating query-time/memory trade-off compared to F-CH-SG. N-SG-GPPC has shorter query times than F-N-SG but uses more memory, although they both use F N-level subgoal graphs to answer queries. These differences can be explained by the differences in their implementations and algorithmic design decisions: We have developed 2-SG and N-SG-GPPC as entries in the GPPC, and, therefore, they have very optimized implementations. On the other hand, we have developed F-CH-SG and F-N-SG (which can be considered to be the most similar algorithm to 2-SG and N-SG-GPPC) as augmentations of CH (CH-GPPC) with reachability relations, and, therefore, their implementations closely follow the CH-GPPC implementation. We now outline some of the important differences in the implementations and algorithmic design decisions of 2-SG, N-SG-GPPC, and F-N-SG and comment on how they might affect the resulting query-time/memory trade-offs:

  - 2-SG and N-SG-GPPC perform backward searches before forward searches and use breadth-first searches rather than A* searches for the backward searches. Since backward searches over N-level (2-level) subgoal graphs do not expand core vertices, they typically exhaust (the vertices in) their OPEN lists before the bidirectional searches terminate. 2-SG and N-SG-GPPC commit to exhausting the OPEN lists of the backward searches, with the following benefits:

1) The backward searches are run before the forward searches, which can avoid accessing the memory of data structures by two different searches at once. 2) The backward searches are implemented as breadth-first searches that expand vertices in increasing order of levels rather than A* searches, which reduce the time to perform each expansion by avoiding the calculation of heuristic distances and performing insertions and removals in the OPEN list in O(1) time. 3) Non-core same-level edges are considered by the more efficient backward searches, which can reduce the number of vertices in the OPEN lists of the forward searches.

- 2-SG and N-SG-GPPC use a "fast stack" to complement the binary heap used for maintaining the OPEN lists of their forward searches: Whenever a successor is generated with the same f-value as the expanded vertex, the successor is inserted into the fast stack rather than the binary heap. Vertices are selected for expansion first from the fast stack, until the fast stack is empty, and only then from the binary heap. This optimization results in fewer insertion and removal operations in binary heaps, and is particularly impactful on random maps where subgoal graphs have shorter edges, since successors generated through shorter edges are more likely to have f-values similar to the ones of the expanded vertices.
- 2-SG and N-SG-GPPC both store edge lengths, which increases their memory requirements (not storing the edge lengths was an optimization we have thought about later). 2-SG stores each clearance value using 4 bits rather than 1 byte. That is, it can store clearance values up to 16 rather than 256, which means that determining the clearance value of a cell in a given direction might require more clearance-value look-ups.
- When there are more than 65,536 subgoals on a grid graph, such as on most random maps, 2-SG does not use subgoal graphs for answering queries and instead uses an A* search with buckets.[3] This inadvertently reduces its memory requirements further, since it avoids storing subgoal graphs (or clearance values) for maps with more than 65,536 subgoals.
- 2-SG and N-SG-GPPC do not determine the order of heavy F contractions online when constructing F N-level (2-level) subgoal graphs, and instead heavy F contract them in the order of their subgoal IDs. Therefore, their preprocessing times are shorter than the N-F-SG preprocessing times.

An evaluation of these optimizations is beyond the scope of this dissertation. However, we believe that the N-SG-GPPC query times and memory requirements may be improved by using F contraction hierarchies instead of F N-level subgoal graphs,

[3]The reason for this is as follows: Clearance values for convex corner cells are never used by SF-Connect when connecting the start and goal vertices to subgoal graphs, since the diagonal scans of SF-Connect terminate when they encounter subgoals. Our 2-SG implementation leverages this fact by using the memory allocated for storing the clearance values to store subgoal IDs instead, to associate convex corner cells with the subgoals they contain. However, since 2-SG uses 4 bits to store each clearance value, it allocates only 16 bits to store a subgoal ID at each convex corner cell, and can therefore store subgoal IDs only up to 65,536.

as our results from Section 5.5.11 suggest. Similarly, we believe that applying the memory optimizations of 2-SG to N-SG-GPPC can result in an algorithm that uses less memory than 2-SG while having shorter query times.

- **CH-GPPC, CH, and CH+*R*r:** As discussed in Sections 5.5.1 and 5.5.6, CH uses 2-pointer unpacking rather than midpoint unpacking and uses the Octile distance heuristic in its bidirectional searches, which results in higher memory requirements but shorter query times compared to CH-GPPC. Therefore, neither algorithm has a dominating query-time/memory trade-off compared to the other one. Contrary to our results in Section 5.5.9, CH+Fr has shorter query times than CH due to shorter refinement times on the larger room and maze maps in the GPPC benchmarks. CH+CFr has shorter query times than CH+Fr and therefore has a dominating query-time/memory trade-off compared to CH and CH+Fr. Interestingly, CH+CFr has shorter query times than SRC while requiring significantly less memory and, therefore, has a dominating query-time/memory trade-off compared to SRC. However, the query-time/memory trade-offs of CH-GPPC, CH, CH+CFr and CH+Fr are all dominated by the query-time/memory trade-off of CH-SG(+CFr) since, as we have shown in Section 5.5.7, constructing hierarchies on subgoal graphs rather than G can significantly reduce the query times and memory requirements of algorithms that use them to answer queries.

- **JPS+BB:** JPS+BB is a variant of JPS+ that associates each edge (u, v) with a bounding box that contains all cells that can be reached optimally from u via a path whose first edge is (u, v). During searches, JPS+BB ignores those edges whose bounding boxes do not contain the goal vertex. Section 2.3.3 provides a more detailed description of the "goal-directed" pruning of JPS+BB. JPS+BB uses the "fast stack" implementation that we have used in 2-SG and N-SG-GPPC. With the combined pruning of jump-point search and bounding boxes, and by using the fast stack to avoid binary heap operations, JPS+BB performs an average of 3.01 and 3.91 insertions into the binary heap on the dao and wc3-512 maps, respectively (Rabin & Sturtevant, 2016). However, these numbers are typically higher on most other types of maps (for instance 200.41 on the sc maps). The JPS+BB query times are shorter than the CH-GPPC, CH, CH+Fr, CH+CFr query times, and JPS+BB requires less memory than these algorithms. Therefore, it has a dominating query-time/memory trade-off compared to them. However, its query-time/memory trade-off is dominated by CH-SG(+CFr).

- **SRC:** Single-row compression (SRC) precomputes and stores compressed pairwise next-move tables, and answers queries by repeatedly looking up the next move along a shortest path from the start vertex to the goal vertex. It has long preprocessing times (5,604 seconds) and large memory requirements (394 megabytes), but also the shortest query times in the GPPC (145 microseconds). Among our new algorithms, CH+CFr and CH-SG (+CFr, +Fr) all have shorter query times than SRC, while requiring significantly less memory and preprocessing time, and therefore have a dominating query-time/memory trade-off compared to SRC.

- **CH-JP:** CH-JP has a dominating query-time/memory trade-off compared to JPS+ BB and SRC. The CH-JP query times are 1.82 times shorter than the JPS+BB query times. However, it is unclear whether this result is due to JPS+BB performing scans during query time, or whether the combination of jump-point search with contraction hierarchies results in shorter query times compared to the combination of jump-point search with bounding boxes. However, the query-time/memory trade-off of CH-JP is dominated by the one of CH-SG, whose query times are 1.24 times shorter and whose memory requirements are 2.20 times lower. Although we have not experimented with a version of CH-JP that uses CF-Refine to refine any CF-reachable shortcuts (that is, CH-JP+CFr), we suspect that its query times would only be slightly shorter than the CH-JP query times.

- **CH-SG and CH-SG+*R*r:** CH-SG+CFr has the shortest query times in our comparison, which are 2.23 times shorter than the CH+CFr query times, 2.34 times shorter than the SRC query times, 2.40 times shorter than the JPS+BB query times, and 5.83 times shorter than the CH-GPPC query times. It also requires less memory than these algorithms, namely 8.56 times less memory than CH+CFr, 139.13 times less memory than SRC, 5.35 times less memory than JPS+BB, and 6.42 times less memory than CH-GPPC. It therefore has a dominating query-time/memory trade-off compared to these algorithms. Since CH-SG+CFr has the shortest query times, it also has a non-dominated query-time/memory trade-off in our comparison.

To summarize, we observe that implementation details play a significant role in determining the query-time/memory trade-offs of different algorithms. We think that our implementation of the algorithms listed in Table 5.1 within the same framework, which follows the CH-GPPC implementation, minimizes the effects of implementation details on the *relative* query-time/memory trade-offs of these algorithms; and our normalization of the results on our server with respect to the results on the GPPC server allows for a meaningful comparison of all algorithms.

Among the algorithms listed in Table 5.1, CH-SG+CFr and F-CH-SG have undominated query-time/memory trade-offs. Both algorithms combine subgoal graphs with (variants of) contraction hierarchies, where F-CH-SG restricts its hierarchies to use only F-reachable edges that can be stored compactly, and therefore has lower memory requirements, and CH-SG+CFr has no restrictions on the edges of its hierarchies, and therefore can achieve significantly shorter query times. Although the query-time/memory trade-off of F-CH-SG is dominated by our GPPC entry 2-SG, which is also based on subgoal graphs, we think that, with similar implementations, F-CH-SG can achieve a dominating query-time/memory trade-off compared to 2-SG and N-SG-GPPC, based on our results from Section 5.5.11.

Our implementation of jump-point search within the subgoal graph framework achieves a dominating query-time/memory trade-off compared to other JPS+ variants except for JPS+Bucket(Rabin), and its combination with contraction hierarchies achieves a dominating query-time/memory trade-off compared to JPS+BB. Although the query-time/memory trade-off of CH-JP is dominated by CH-SG (and CH-SG+*R*r), the combination of jump-point graphs and contraction hierarchies is not well understood, and there might be a version of CH-JP that can achieve shorter query times than CH-SG.

5.6 Conclusions

In this chapter, we have applied the subgoal graph framework to grid graphs by using safe-freespace-reachability as reachability relation, discussed the similarities and differences of subgoal graphs with jump-point search, and augmented contraction hierarchies with reachability relations in various ways. Specifically, we have shown that the freespace structure (Octile property) of grid graphs allows for the construction of safe-freespace-reachability subgoal graphs by using only convex corner cells as subgoals, introduced a connection algorithm for safe-freespace-reachability that scans the grid efficiently by using clearance values, and proved that this algorithm can be used to construct subgoal graphs in time linear in the size of the underlying grid. We have shown that jump-point search can be understood as a search on a jump-point graph, which is a freespace-reachability subgoal graph on the direction-extended canonical grid graph. We have experimentally demonstrated that answering queries using contraction hierarchies on subgoal graphs achieves a *dominating query-time/memory trade-off* compared to answering queries using contraction hierarchies on G or jump-point graphs. Our results further suggest (through interpolation) that answering queries using contraction hierarchies on subgoal graphs and performing freespace-based refinement is *2.34 times faster* than single-row compression, the fastest entry in the Grid-Based Path-Planning Competition, while requiring *139.06 times less memory*. These results validate the hypothesis of this dissertation that one can develop preprocessing-based path-planning algorithms for grid graphs that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.

As future work, we consider combining subgoal graphs with other techniques, such as geometric containers or hub labeling. As discussed in Section 2.3.2, storing geometric containers for each edge of G allows for goal-directed pruning during searches, and storing hubs for each vertex of G allows for answering queries without performing search. Storing geometric containers or hubs only for the edges or vertices of subgoal graphs rather than G would require significantly less memory, and we think can still be used to answer queries in a very short time. Another direction that we consider for future work is to better understand the combination of jump-point graphs with contraction hierarchies. As discussed in Section 5.5.8, a straightforward combination results in redundant shortcut edges in contraction hierarchies on jump-point graphs. Although we have (partially) addressed this issue, we still do not clearly understand why queries can be answered faster with a combination of subgoal graphs and contraction hierarchies than a combination of jump-point graphs and contraction hierarchies.

Chapter 6

Conclusions

Preprocessing-based path-planning algorithms compute and store auxiliary information about a graph during a preprocessing phase, and then use the auxiliary information to efficiently answer path or distance queries on that graph. Preprocessing-based path-planning algorithms that achieve non-dominated query-time/memory trade-offs on road networks have been shown to exploit their hierarchical structure and small highway dimensions. State lattices and grid graphs, on the other hand, have larger highway dimensions than road networks. When some of these algorithms are applied to grid graphs, they achieve smaller speed-ups relative to Dijkstra searches and require more memory relative to the size of the graph. However, state lattices and grid graphs have other properties that can be exploited, namely, their freespace structure. In this dissertation, we have hypothesized that one can develop preprocessing-based path-planning algorithms for state lattices and grid graphs that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms. We made the following contributions to validate this hypothesis:

- In Chapter 3, we have introduced the subgoal graph framework, that can be specialized to exploit structure in different types of graphs, by choosing a reachability relation that captures structure in that type of graph and developing specialized connection and refinement algorithms that exploit that structure. We have proved that subgoal graphs can be used to answer path queries optimally with the Connect-Search-Refine algorithm, and that it is possible to construct locally-sparse subgoal graphs with respect to bounded-distance reachability on graphs with small highway dimensions. We have introduced a hierarchical variant of subgoal graphs, called N-level subgoal graphs, and introduced variants of contraction hierarchies within this framework. We have introduced a suboptimal variant of subgoal graphs, called strongly connected subgoal graphs, that can be used to answer path queries without the guarantee of optimality. We have introduced algorithms for constructing subgoal graphs, N-level subgoal graphs, and strongly connected subgoal graphs.

- In Chapter 4, we have applied the subgoal graph framework to state lattices, by using freespace-reachability and canonical-freespace-reachability as reachability relations to capture the freespace structure of state lattices, and developing efficient connection and refinement algorithms that exploit this structure. Specifically, we have characterized the freespace structure of state lattices as the *translation-invariance*

of freespace distances and freespace-canonical paths, and showed that it can be exploited to efficiently compute and compactly store freespace information, such as pairwise distances or shortest path trees on freespace state lattices. We have introduced freespace-reachability and canonical-freespace-reachability as reachability relations to distinguish those pairs of vertices on state lattices between which the freespace information is accurate, and developed connection and refinement algorithms for these reachability relations that use freespace information to efficiently explore the freespace-shortest and freespace-canonical paths, respectively. We have experimentally demonstrated that answering queries using freespace-reachability or canonical-freespace-reachability strongly connected subgoal graphs achieves a *dominating query-time/path-suboptimality trade-off* compared to answering queries using bounded-distance-reachability strongly connected subgoal graphs, and a *non-dominated query-time/path-suboptimality* trade-off compared to answering queries using weighted A* searches. These results validate the hypothesis of this dissertation that one can develop preprocessing-based path-planning algorithms for state lattices that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.

- In Chapter 5, we have applied the subgoal graph framework to grid graphs by using safe-freespace-reachability as a reachability relation, discussed the similarities and differences of subgoal graphs with jump-point search, and augmented contraction hierarchies with reachability relations in various ways. Specifically, we have shown that the freespace structure (Octile property) of grid graphs allows for the construction of safe-freespace-reachability subgoal graphs by using only convex corner cells as subgoals, introduced a connection algorithm for safe-freespace-reachability that scans the grid efficiently by using clearance values, and proved that this algorithm can be used to construct subgoal graphs in time linear in the size of the underlying grid. We have shown that jump-point search can be understood as a search over a jump-point graph, which is a freespace-reachability subgoal graph on the direction-extended canonical grid graph. We have experimentally demonstrated that answering queries using contraction hierarchies on subgoal graphs achieves a *dominating query-time/memory trade-off* compared to answering queries using contraction hierarchies alone, or on jump-point graphs. Our results further suggest (through interpolation) that answering queries using contraction hierarchies on subgoal graphs and performing freespace-based refinement is *2.34 times faster* than single-row compression, the fastest entry in the Grid-Based Path-Planning Competition, while requiring *139.06 times less memory*. These results validate the hypothesis of this dissertation that one can develop preprocessing-based path-planning algorithms for grid graphs that exploit their freespace structure to improve the query-time/memory/path-suboptimality Pareto frontier of the state-of-the-art algorithms.

Our work on using subgoal graphs to exploit freespace structure in state lattices and grid graphs has resulted in preprocessing-based path-planning algorithms that achieve non-dominated query-time/path-suboptimality trade-offs on state lattices, and non-dominated query-time/memory trade-offs on grid graphs. As of the time of the publication of

this dissertation, subgoal graphs on grid graphs have been applied by other researchers to moving target search (Nussbaum & Yorukcu, 2015), adapted to 2^k-neighbor grid graphs (Hormazábal et al., 2017), and used for planning high-level paths for agents maneuvering in continuous and uncertain environments (Zeng et al., 2019). The clearance-based scanning that we developed for the connection phases of queries answered using subgoal graphs on grid graphs have been adapted and used in preprocessing-based variants of jump-point search, previously an online path-planning algorithm on grid graphs (Harabor et al., 2014; Rabin & Sturtevant, 2016).

Going forward, we think that there are several research directions in which our work can be applied to other types of graphs or improved in various ways:

- Throughout this dissertation, we have shown that various algorithms can be understood and augmented in the (N-level) subgoal graph framework. For instance, we have shown that contraction hierarchies can be understood as N-level subgoal graphs without same-level edges or a reachability relation and shown that they can be augmented within this framework to use a reachability relation in various ways. We think that it might be possible to use this understanding to augment contraction hierarchies with reachability relations on other types of graphs as well. For instance, one might be able to improve query times on road networks by identifying various reachability relations on road networks, developing connection and refinement algorithms for these reachability relations, and using these algorithms in various ways to augment contraction hierarchies (as outlined in this dissertation or in other novel ways). Our results on grid graphs suggest that simply marking some shortcut edges for R-refinement can shorten query times without increasing memory requirements, given that an efficient R-refine algorithm is available. One candidate for such an R-refine algorithm on road networks might be to greedily move in the direction of the target vertex (the endpoint of the R-reachable edge) by always picking the next move as the move whose direction is closest to the direction of the target vertex from the current vertex.

- Although the paths found by using strongly connected subgoal graphs are not necessarily bounded-suboptimal, our experimental results on state lattices suggest that they are typically not much longer than optimal. As we have discussed in Sections 3.6 and 4.7, we think that there are several ways in which our current algorithm of constructing strongly connected subgoal graphs can be improved. We also think that developing a variant of strongly connected subgoal graphs that can be used for finding bounded-suboptimal paths would be an interesting research direction.

- Our experimental results on grid graphs suggest that queries can be answered faster with a combination of contraction hierarchies and subgoal graphs than with either method by itself. We think that a similar case might hold for the combinations of subgoal graphs with other speed-up techniques, such as geometric containers or hub labeling, and might be an interesting direction for future research.

- Our experimental results on grid graphs suggest that queries can be answered faster with a combination of contraction hierarchies and subgoal graphs than with a combination of contraction hierarchies and jump-point graphs, despite the fact that they

also suggest that queries can be answered faster with jump-point graphs than with subgoal graphs. As we have mentioned in Section 5.5.8, we do not completely understand the combination of contraction hierarchies and jump-point graphs, and we think that further research on this subject might result in algorithms with shorter query times.

To summarize, the research presented in this dissertation has resulted in new state-of-the-art path-planning algorithms on grid graphs and state lattices, and we think can be applied to other types of graphs and be improved in various ways.

Bibliography

Abraham, I., Delling, D., Goldberg, A. V., & Werneck, R. F. (2011). A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the International Symposium on Experimental Algorithms*, pp. 230–241.

Abraham, I., Delling, D., Goldberg, A. V., & Werneck, R. F. (2012). Hierarchical hub labelings for shortest paths. In *Proceedings of the European Symposium on Algorithms*, pp. 24–35.

Abraham, I., Fiat, A., Goldberg, A. V., & Werneck, R. F. (2010). Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 782–793.

Antsfeld, L., Harabor, D., Kilby, P., Walsh, T., et al. (2012). TRANSIT routing on video game maps. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 2–7.

Arz, J., Luxen, D., & Sanders, P. (2013). TRANSIT node routing reconsidered. In *Proceedings of the International Symposium on Experimental Algorithms*, pp. 55–66.

Bast, H., Delling, D., Goldberg, A. V., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., & Werneck, R. F. (2016). Route planning in transportation networks. In *Algorithm Engineering: Selected Results and Surveys*, pp. 19–80. Springer.

Bast, H., Funke, S., & Matijević, D. (2006). TRANSIT— ultrafast shortest-path queries with linear-time preprocessing. In *Proceedings of the DIMACS Implementation Challenge – Shortest Paths*.

Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., & Wagner, D. (2010). Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *Journal of Experimental Algorithmics*, *15*, 1–31.

Björnsson, Y., & Halldórsson, K. (2006). Improved heuristics for optimal path-finding on game maps. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 9–14.

Botea, A. (2011). Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 122–127.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development*, *1*(1), 7–28.

Botea, A., & Harabor, D. (2013). Path planning with compressed all-pairs shortest paths data. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 288–292.

Botea, A., Strasser, B., & Harabor, D. (2015). Complexity results for compressing optimal paths.. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 1100–1106.

Butzke, J., Sapkota, K., Prasad, K., MacAllister, B., & Likhachev, M. (2014). State lattice with controllers: augmenting lattice-based path planning with controller-based motion primitives. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pp. 258–265. IEEE.

Canny, J. (1988). *The Complexity of Robot Motion Planning*. MIT press.

Choset, H., Lynch, K., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L., & Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementation*. MIT press.

Delling, D., Goldberg, A. V., Pajor, T., & Werneck, R. F. (2015). Customizable route planning in road networks. *Transportation Science*, *51*(2), 566–591.

Delling, D., Goldberg, A. V., Razenshteyn, I., & Werneck, R. F. (2011). Graph partitioning with natural cuts. In *Proceedings of the Parallel and Distributed Processing Symposium*, pp. 1135–1146.

Delling, D., Holzer, M., Müller, K., Schulz, F., & Wagner, D. (2009). High-performance multi-level routing. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, *74*, 73–92.

Delling, D., & Nannicini, G. (2012). Core routing on dynamic time-dependent road networks. *INFORMS Journal on Computing*, *24*(2), 187–201.

Demetrescu, C., Goldberg, A. V., & Johnson, D. (2009). *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, Vol. 74. American Mathematical Society.

Demetrescu, C., & Italiano, G. (2004). A new approach to dynamic all pairs shortest paths. *Journal of the ACM*, *51*(6), 968–992.

Dibbelt, J., Strasser, B., & Wagner, D. (2014). Customizable contraction hierarchies. In *Proceedings of the International Symposium on Experimental Algorithms*, pp. 271–282.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271.

Eppstein, D., & Goodrich, M. (2008). Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 16:1–16:10. ACM.

Felner, A., Sturtevant, N., & Schaeffer, J. (2009). Abstraction-based heuristics with true distance computations. In *Proceedings of the Symposium on Abstraction, Reformulation and Approximation*.

Floyd, R. (1962). Algorithm 97: Shortest path. *Communications of the ACM*, *5*(6), 345.

Foux, G., Heymann, M., & Bruckstein, A. (1993). Two-dimensional robot navigation among unknown stationary polygonal obstacles. *IEEE Transactions on Robotics and Automation*, *9*(1), 96–102.

Fredman, M., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, *34*(3), 596–615.

Funke, S., Nusser, A., & Storandt, S. (2014). On k-path covers and their applications. *Proceedings of the VLDB Endowment*, *7*(10), 893–902.

Geisberger, R. (2008). *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks*. Ph.D. thesis, Institut für Theoretische Informatik Universität Karlsruhe.

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the International Workshop on Experimental and Efficient Algorithms*, pp. 319–333.

Goldberg, A. V., & Harrelson, C. (2005). Computing the shortest path: A* search meets graph theory. In *Proceedings of the ACM-SIAM Symposium on Discrete algorithms*, pp. 156–165.

Goldberg, A. V., Kaplan, H., & Werneck, R. F. (2009). Reach for A*: shortest path algorithms with preprocessing. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, *74*, 93–139.

Goldenberg, M., Felner, A., Sturtevant, N., & Schaeffer, J. (2010). Portal-based true-distance heuristics for path finding. In *Proceedings of the Symposium on Combinatorial Search*.

Goldenberg, M., Sturtevant, N., Felner, A., & Schaeffer, J. (2011). The compressed differential heuristic. In *Proceedings of the Symposium on Combinatorial Search*.

Gutman, R. J. (2004). Reach-based routing: a new approach to shortest path algorithms optimized for road networks. In *Proceedings of the Workshop on Algorithm Engineering and Experiments and the Workshop on Analytic Algorithmics and Combinatorics*, pp. 100–111.

Harabor, D., Botea, A., & Kilby, P. (2011). Path symmetries in uniform-cost grid maps. In *Proceedings of the Symposium on Abstraction, Reformulation and Approximation*.

Harabor, D., & Grastien, A. (2011). Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 1114–1119.

Harabor, D., Grastien, A., et al. (2014). Improving jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *2*, 100–107.

Hilger, M., Köhler, E., Möhring, R. H., & Schilling, H. (2009). Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, *74*, 41–72.

Holte, R., & Burch, N. (2014). Automatic move pruning for single-agent search. *AI Communications*, *27*(4), 363–383.

Holzer, M., Schulz, F., & Wagner, D. (2009). Engineering multilevel overlay graphs for shortest-path queries. *Journal of Experimental Algorithmics*, *13*, 5.

Hormazábal, N., Díaz, A., Hernández, C., & Baier, J. A. (2017). Fast and almost optimal any-angle pathfinding using the 2^k neighborhoods. In *Proceedings of the Symposium on Combinatorial Search*.

Jung, S., & Pramanik, S. (2002). An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, *14*(5), 1029–1046.

Kalapala, V., Sanwalani, V., Clauset, A., & Moore, C. (2006). Scale invariance in road networks. *Physical Review E*, *73*(2), 026130.

Koenig, S., & Smirnov, Y. (1997). Sensor-based planning with the freespace assumption. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 4, pp. 3540–3545.

Kushleyev, A., & Likhachev, M. (2009). Time-bounded lattice for efficient planning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 1662–1668.

Lauther, U. (2004). An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background.. *Geoinformation und Mobilität - Von der Forschung zur praktischen Anwendung*, *22*, 219–230.

Likhachev, M., & Ferguson, D. (2009). Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, *28*(8), 933–945.

Lozano-Pérez, T., & Wesley, M. A. (1979). An Algorithm for Planning Collision-free Paths Among Polyhedral Obstacles. *Commun. ACM*, *22*(10), 560–570.

Nourbakhsh, I. R., & Genesereth, M. R. (1996). Assumptive planning and execution: a simple, working robot architecture. *Autonomous Robots*, *3*(1), 49–67.

Nussbaum, D., & Yorukcu, A. (2015). Moving target search with subgoal graphs. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 179–187.

Pearl, J. (1985). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pivtoraiko, M., & Kelly, A. (2005a). Efficient constrained path planning via search in state lattices. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, pp. 1–7.

Pivtoraiko, M., & Kelly, A. (2005b). Generating near minimal spanning control sets for constrained motion planning in discrete state spaces. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3231–3237.

Pochter, N., Zohar, A., Rosenschein, J., & Felner, A. (2010). Search space reduction using swamp hierarchies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 155–160.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, *1*(3-4), 193–204.

Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell Labs Technical Journal*, *36*(6), 1389–1401.

Rabin, S., & Sturtevant, N. (2016). Combining bounding boxes and JPS to prune grid pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 746–752.

Reynolds, C. (1999). Steering behaviors for autonomous characters. In *Proceedings of the Game Developers Conference*, Vol. 1999, pp. 763–782.

Sanders, P., & Schultes, D. (2005). Highway hierarchies hasten exact shortest path queries. In *Proceedings of the European Symposium on Algorithms*, pp. 568–579.

Sanders, P., & Schultes, D. (2006). Engineering highway hierarchies. In *Proceedings of the European Symposium on Algorithms*, pp. 804–816.

Schultes, D., & Sanders, P. (2007). Dynamic highway-node routing. In *Proceedings of the International Workshop on Experimental and Efficient Algorithms*, pp. 66–79.

Schulz, F., Wagner, D., & Weihe, K. (2000). Dijkstra's algorithm on-line: an empirical case study from public railroad transport. *Journal of Experimental Algorithmics*, *5*, 12.

Schulz, F., Wagner, D., & Zaroliagis, C. (2002). Using multi-level graphs for timetable information in railway systems. In *Workshop on Algorithm Engineering and Experimentation*, pp. 43–59.

Sommer, C. (2014). Shortest-path queries in static networks. *ACM Computing Surveys*, *46*(4), 45.

Stentz, A. (1994). Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 94, pp. 3310–3317.

Strasser, B., Botea, A., & Harabor, D. (2015). Compressing optimal paths with run length encoding. *Journal of Artificial Intelligence Research*, *54*, 593–629.

Sturtevant, N., Felner, A., Barrer, M., Schaeffer, J., & Burch, N. (2009). Memory-based heuristics for explicit state spaces. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 609–614.

Sturtevant, N. (2012a). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(2), 144–148.

Sturtevant, N. (2012b). Grid-based path-planning competition..

Sturtevant, N., & Buro, M. (2005). Partial pathfinding using map abstraction and refinement. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 1392–1397.

Sturtevant, N., & Geisberger, R. (2010). A comparison of high-level approaches for speeding up pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 76–82.

Sturtevant, N., & Rabin, S. (2016). Canonical orderings on grids. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 683–689.

Sturtevant, N., Traish, J., Tulip, J., Uras, T., Koenig, S., Strasser, B., Botea, A., Harabor, D., & Rabin, S. (2015). The grid-based path planning competition: 2014 entries and results. In *Proceedings of the Symposium on Combinatorial Search*.

Svec, P., Shah, B., Bertaska, I., Alvarez, J., Sinisterra, A., Von Ellenrieder, K., Dhanak, M., & Gupta, S. (2013). Dynamics-aware target following for an autonomous surface vehicle operating under COLREGs in civilian traffic. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3871–3878.

Tao, Y., Sheng, C., & Pei, J. (2011). On k-skip shortest paths. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 421–432.

Taylor, L., & Korf, R. (1993). Pruning duplicate nodes in depth-first search. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pp. 756–761.

Thakur, D., Likhachev, M., Keller, J., Kumar, V., Dobrokhodov, V., Jones, K., Wurz, J., & Kaminer, I. (2013). Planning for opportunistic surveillance with multiple robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5750–5757.

Traish, J., Tulip, J., & Moore, W. (2016). Optimization using boundary lookup jump point search. *IEEE Transactions on Computational Intelligence and AI in Games*, *8*(3), 268–277.

Uras, T., Koenig, S., & Hernandez, C. (2013). Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 224–232.

Uras, T., & Koenig, S. (2014). Identifying hierarchies for fast optimal search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 878–884.

Urmson, C., Anhalt, J., Bagnell, D., Baker, C., Bittner, R., Clark, M., Dolan, J., Duggins, D., Galatali, T., Geyer, C., et al. (2008). Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, *25*(8), 425–466.

Wagner, D., Willhalm, T., & Zaroliagis, C. (2005). Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics*, *10*, 1–3.

Warshall, S. (1962). A theorem on boolean matrices. *Journal of the ACM*, *9*(1), 11–12.

Zelinsky, A. (1992). A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation*, *8*(6), 707–717.

Zeng, J., Qin, L., Hu, Y., Hu, C., & Yin, Q. (2019). Combining subgoal graphs with reinforcement learning to build a rational pathfinder. *Applied Sciences*, *9*(2), 323.