ANY-ANGLE PATH PLANNING

by

Alex Nash

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)

August 2012

Table of Contents

List of Tables

List of Figures

Abstract

Navigating an agent from a given start coordinate to a given goal coordinate through a continuous environment is one of the most important problems faced by roboticists and video game developers. A key part of navigation is path planning. Path planning is the process of generating a path, defined by a sequence of waypoints, that begins at a given start coordinate, ends a given goal coordinate and avoids obstacles in the continuous environment. Path planning is typically composed of two parts: the generate-graph problem, which is solved by discretizing a continuous environment into a graph and the find-path problem, which is solved by searching this graph for a path from a given start vertex to a given goal vertex. Roboticists and video game developers use many different techniques for solving the generate-graph problem, but the find-path problem is typically solved using a traditional edge-constrained find-path algorithm, such as A*. The reason for this is that traditional edge-constrained find-path algorithms have the following desirable properties: Simplicity Property: They are simple to implement and understand. Efficiency Property: They provide a good tradeoff with respect to the runtime of the search and the length of the resulting path. Generality Property: They can be used to search any Euclidean graph. However, these desirable properties come with a penalty. Traditional edge-constrained find-path algorithms propagate information along graph edges *and* constrain paths to be formed by graph edges. This constraint is artificial and causes the paths found by traditional edge-constrained find-path algorithms to be both longer and less realistic looking than the true shortest paths (that is, the shortest paths in the continuous environment). While this fact is well known by roboticists and video game developers two important questions remain: Question 1: How much longer can the paths found by traditional edge-constrained find-path algorithms be than the true shortest paths? Question 2: Can more sophisticated find-path algorithms be developed that find shorter and more realistic looking paths than traditional edge-constrained find-path algorithms, while maintaining the desirable properties of traditional edge-constrained find-path algorithms? This dissertation addresses these two questions and therefore our hypotheses are as follows: Hypothesis 1: Analytical bounds can be introduced which compare the lengths of the paths found by traditional edge-constrained find-path algorithms on certain types of graphs with the lengths of the true shortest paths. Hypothesis 2: A new class of any-angle find-path algorithms, that propagate information along graph edges, without constraining paths to be formed by graph edges, can be used to quickly find paths that are shorter than the paths found by traditional edge-constrained find-path algorithms, while maintaining the Simplicity and Generality Properties of traditional edge-constrained find-path algorithms.

To validate Hypothesis 1, we introduce a comprehensive set of eight new analytical bounds which compare the lengths of the paths found by traditional edge-constrained find-path algorithms on grid graphs constructed from 2D and 3D regular grids with the lengths of the true shortest paths.

To validate Hypothesis 2, we introduce a new class of any-angle find-path algorithms that propagate information along graph edges (to achieve a short runtime) without constraining paths to be formed by graph edges (to find any-angle paths). We introduce new members to this class and evaluate each member in one of three types of continuous environments, namely continuous 2D environments in which agents have complete knowledge of the environment (known 2D environments), continuous 3D environments in which agents have complete knowledge of the environment (known 3D environments), and continuous 2D environments in which agents do not have complete knowledge of the environment (unknown 2D environments). For each new any-angle find-path algorithm, we use the Simplicity, Efficiency and Generality Properties as our evaluation criteria. Specifically, we introduce three new any-angle find-path algorithms, namely Basic Theta*, Lazy Theta* and Incremental Phi*. In known 2D environments, Basic Theta* satisfies the Simplicity and Generality Properties, provides a good tradeoff, relative to traditional edge-constrained find-path algorithms, with respect to the runtime of the search and the length of the resulting path and a dominating tradeoff over existing any-angle find-path algorithms with respect to the runtime of the search and the length of the resulting path (Efficiency Property). Lazy Theta* is a variant of Basic Theta* that is designed for path planning in known 3D environments. In known 3D environments, Lazy Theta* satisfies the Simplicity and Generality Properties, provides a good tradeoff, relative to traditional edge-constrained find-path algorithms, with respect to the runtime of the search and the length of the resulting path and a dominating tradeoff over Basic Theta* with respect to the runtime of the search and the length of the resulting path (Efficiency Property). Incremental Phi* is a variant of Basic Theta* that is designed for path planning in unknown 2D environments. In unknown 2D environments, Incremental Phi* satisfies the Simplicity Property and provides a dominating tradeoff over Basic Theta* with respect to the runtime of the search and the length of the resulting path (Efficiency Property). These contributions demonstrate that any-angle find-path algorithms represent a promising new technique for path planning in robotics and video games.

Chapter 1

Introduction

Navigating an agent from a given start coordinate to a given goal coordinate through a continuous environment is one of the most important problems faced by roboticists and video game developers. A key part of navigation is path planning. Path planning is the process of generating a path, defined by a sequence of waypoints, that begins at a given start coordinate, ends a given goal coordinate and avoids obstacles in the continuous environment. Path planning is typically composed of two parts: the generate-graph problem, which is solved by discretizing a continuous environment into a graph and the find-path problem, which is solved by searching this graph for a path from a given start vertex to a given goal vertex. Roboticists and video game developers use many different techniques for solving the generate-graph problem, but the find-path problem is typically solved using a traditional edge-constrained find-path algorithm, such as A*. The reason for this is that traditional edge-constrained find-path algorithms have the following desirable properties: Simplicity Property: They are simple to implement and understand. Efficiency Property: They provide a good tradeoff with respect to the runtime of the search and the length of the resulting path. Generality Property: They can be used to search any Euclidean graph. However, these desirable properties come with a penalty. Traditional edge-constrained find-path algorithms propagate information along graph edges *and* constrain paths to be formed by graph edges. This constraint is artificial and causes the paths found by traditional edge-constrained find-path algorithms to be both longer and less realistic looking than the true shortest paths (that is, the shortest paths in the continuous environment). While this fact is well known by roboticists and video game developers two important questions remain: Question 1: How much longer can the paths found by traditional edge-constrained find-path algorithms be than the true shortest paths? Question 2: Can more sophisticated find-path algorithms be developed that find shorter and more realistic looking paths than traditional edge-constrained find-path algorithms, while maintaining the desirable properties of traditional edge-constrained find-path algorithms? This dissertation addresses these two questions and therefore our hypotheses are as follows: Hypothesis 1: Analytical bounds can be introduced which compare the lengths of the paths found by traditional edge-constrained find-path algorithms on certain types of graphs with the lengths of the true shortest paths. Hypothesis 2: A new class of any-angle find-path algorithms, that propagate information along graph edges, without constraining paths to be formed by graph edges, can be used to quickly find paths that are shorter than the paths found by traditional edge-constrained find-path algorithms, while maintaining the Simplicity and Generality Properties of traditional edge-constrained find-path algorithms.

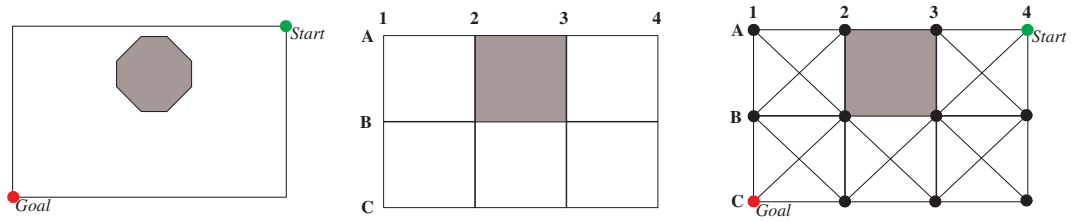
To validate Hypothesis 1, we introduce a comprehensive set of eight new analytical bounds which compare the lengths of the paths found by traditional edge-constrained find-path algorithms on grid graphs constructed from 2D and 3D regular grids with the lengths of the true shortest paths.

To validate Hypothesis 2, we introduce a new class of any-angle find-path algorithms that propagate information along graph edges (to achieve a short runtime) without constraining paths to be formed by graph edges (to find any-angle paths). We introduce new members to this class and evaluate each member in one of three types of continuous environments, namely continuous 2D environments in which agents have complete knowledge of the environment (known 2D environments), continuous 3D environments in which agents have complete knowledge of the environment (known 3D environments), and continuous 2D environments in which agents do not have complete knowledge of the environment (unknown 2D environments). Specifically, we introduce Basic Theta*, Lazy Theta* and Incremental Phi* and evaluate them in known 2D environments, known 3D environments and unknown 2D environments, respectively, using the Simplicity, Efficiency and Generality Properties as our evaluation criteria. Basic Theta* and Lazy Theta* satisfy the Simplicity and Generality Properties and provide a good tradeoff with respect to the runtime of the search and the length of the resulting path (Efficiency Property). Incremental Phi* satisfies the Simplicity Property and provides a good tradeoff with respect to the runtime of the search and the length of the resulting path (Efficiency Property).

1.1 Path Planning

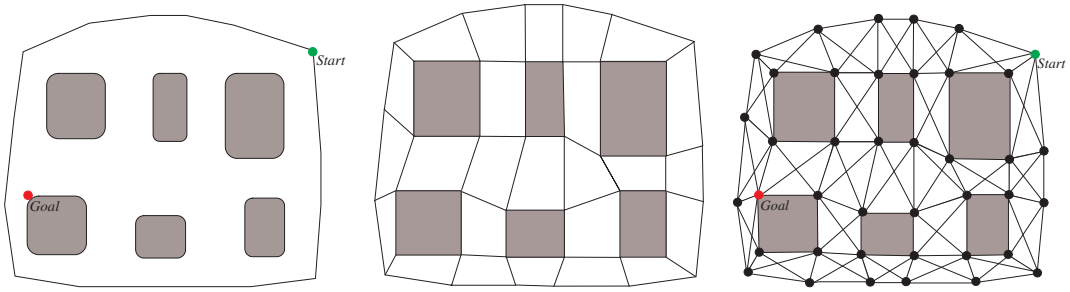
Navigating an agent from a given start coordinate to a given goal coordinate through a continuous environment is one of the most important problems faced by roboticists and video game developers (Deloura, 2000; Rabin, 2002, 2004; Latombe, 1991; Murphy, 2000; Choset, Lynch, Hutchinson, Kantor, Burgard, Kavraki, & Thrun, 2005). Agents must be able to navigate between coordinates in a realistic manner without getting lost, getting stuck or bumping into obstacles. When navigating an agent from a given start coordinate to a given goal coordinate one must address both **path planning**, which is the process of generating a path, defined by a sequence of waypoints, that begins at a given start coordinate, ends a given goal coordinate and avoids obstacles in the continuous environment and **movement**, which is the process of taking a path and following it in a such a manner that takes into account the kinematic actions and dynamic constraints of the agent. Addressing path planning and movement simultaneously is extremely difficult (Latombe, 1991; Reif, 1979; Canny, 1988) and thus it is common to treat path planning and movement as two separate problems (Patel, 2000; Ferguson, 2006). We are interested in path planning. Path planning is important because it allows an agent to plan ahead rather than just reacting to the environment. When an agent reacts to an environment (that is, movement) without planning ahead (that is, path planning) it is less likely to follow a short and realistic looking path.

Path planning is typically composed of two parts: the **generate-graph problem**, which is solved by discretizing a continuous environment into a graph and the **find-path problem**, which is solved by searching this graph for a path from a given start vertex to a given goal vertex (Wooden, 2006; Murphy, 2000).



(a) Continuous Environment (b) Square Grid (c) 8-Neighbor Square Grid Graph

Figure 1.1: 8-Neighbor Square Grid Graph Constructed from a Continuous Environment Discretized into a Square Grid

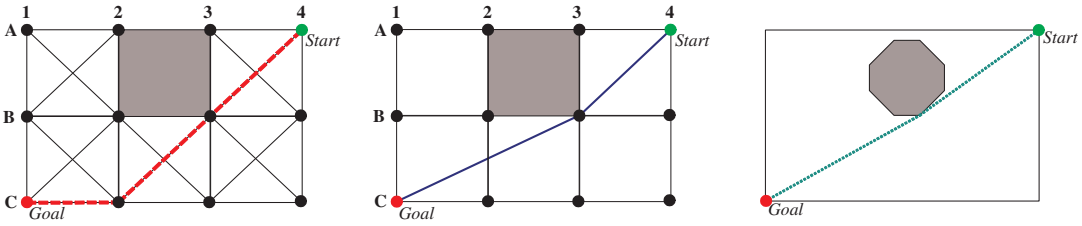


(a) Continuous Environment (b) NavMesh (c) 8-Neighbor Nav Graph

Figure 1.2: 8-Neighbor Nav Graph Constructed from a Continuous Environment Discretized into a NavMesh (adapted from (Patel, 2000))

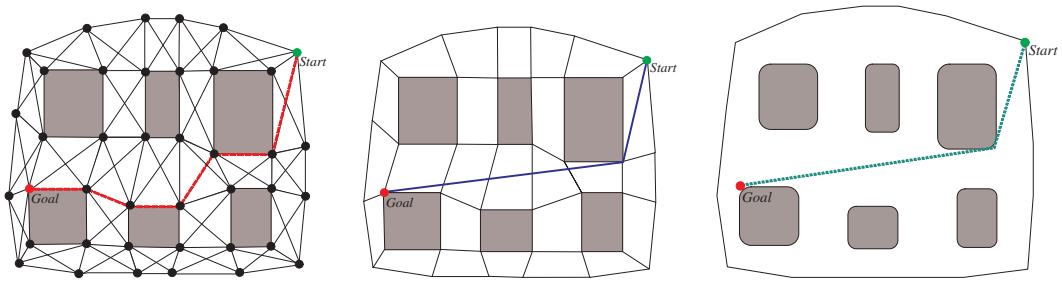
1.1.1 The Generate-Graph Problem

Roboticists and video game developers use many different techniques for solving the generate-graph problem: two-dimensional regular grids composed of squares (square grids), hexagons (hexagonal grids) or triangles (triangular grids), three-dimensional regular grids composed of cubes (cubic grids) (Björnsson, Enzenberger, Holte, Schaeffer, & Yap, 2003; Tozour, 2004; Carsten, Ferguson, & Stentz, 2006; Rabin, 2000b; Chrpa & Komenda, 2011; Choi, Curry, & Elkaim, 2010), visibility graphs (Lozano-Pérez & Wesley, 1979), circle based waypoint graphs (Tozour, 2004), navigation meshes (NavMeshes) (Snook, 2000; OŃeil, 2004; Tozour, 2002), hierarchical techniques such as quad trees (Samet, 1982, 1988) or split waypoint graphs (Tozour, 2004), probabilistic road maps (PRMs) (Kavraki, Svestka, Latombe, & Overmars, 1996) and rapidly exploring random trees (RRTs) (LaValle & Kuffner, 2001; LaValle, Branicky, & Lindemann, 2004). For example, in Figures 1.1 and 1.2, an 8-neighbor square grid graph and an 8-neighbor nav graph are constructed from a continuous environment, respectively, as follows: **(1)** the continuous environment is discretized into a square grid or NavMesh, respectively **(2)** grid cells or polygons are labeled blocked (grey) if they contain part of an obstacle and all other grid cells or polygons are labeled unblocked (white) and **(3)** a grid graph or nav graph is constructed by placing vertices in the corners of grid cells or polygons, and adding an edge between a vertex and up to 8 of its adjacent or closest vertices, respectively (thus defining an 8-neighbor square grid graph and an 8-neighbor nav graph, respectively).



(a) Grid Path (b) Any-Angle Path (c) True Shortest Path

Figure 1.3: Classification of Paths on a Grid Graph Constructed from a Square Grid



(a) Edge-Constrained Path (b) Any-Angle Path (c) True Shortest Path

Figure 1.4: Classification of Paths on a Nav Graph Constructed from a NavMesh

1.1.2 Notation and Definitions

We define different classes of paths that are formed by the vertices and edges in a graph constructed using any of the aforementioned generate-graph techniques. All of these paths are sequences of straight-line segments. **Any-angle paths** are paths formed by line segments whose end points are vertices on the graph (Figures 1.3(b) and 1.4(b)). **Edge-constrained paths** are paths formed by edges on the graph (Figure 1.4(a)). If the graph is a grid graph constructed from a regular grid then we use the more specific term **grid path** in place of edge-constrained path (Figure 1.3(a)). The relationships between the lengths of these paths and the lengths of **true shortest paths**, which are the shortest paths in the continuous environment (Figures 1.3(c) and 1.4(c)), is simple: true shortest paths are at most as long as shortest any-angle paths and shortest any-angle paths are at most as long as shortest edge-constrained paths. **Unblocked paths** are paths such that each vertex on the path has line-of-sight to its successor on the path. If the vertices are members of a grid path then line-of-sight is defined as follows: Two vertices on a grid graph constructed from a 2D regular grid have **line-of-sight** iff the straight-line segment connecting the two vertices neither traverses blocked grid cells nor passes between blocked grid cells that share a side. Two vertices on a grid graph constructed from a 3D regular grid have **line-of-sight** iff the straight-line segment connecting the two vertices neither traverses blocked grid cells nor passes between blocked grid cells that share a face. A straight-line segment traverses a grid cell iff it crosses into its interior. For simplicity, we allow a straight line to pass between diagonally touching blocked grid cells. If the vertices are members of a graph constructed using any of the other aforementioned generate-graph techniques then we use the following definition for line-of-sight:

Figure 1.5: Screen Shot from Starcraft II (Blizzard Entertainment)

Two vertices on the graph have line-of-sight iff the straight-line segment connecting the two vertices is unobstructed. A straight-line segment is unobstructed iff it does not cross into the interior of an obstacle. The definition of line-of-sight between two vertices on a grid graph constructed from a regular grid is intentionally more rigorous than the definition of line-of-sight between two vertices on a graph constructed using any of the other aforementioned generate-graph techniques because the former definition of line-of-sight is actually implemented while the latter definition is only used when describing different techniques for solving the generate-graph and find-path problems.

1.1.3 The Find-Path Problem

Roboticists and video game developers typically solve the find-path problem with traditional edge-constrained find-path algorithms (Choset et al., 2005; Murphy, 2000; Stout, 2000; Matthews, 2002; Higgins, 2002). **Traditional edge-constrained find-path algorithms**, such as Dijkstra's algorithm (Dijkstra, 1959) or A* (Hart, Nilsson, & Raphael, 1968), are graph search algorithms that find edge-constrained paths by propagating information along graph edges *and* constraining paths to be formed by graph edges. These algorithms are used because they have the following desirable properties:

- **Simplicity Property:** *They are simple to implement and understand.* Algorithms that are simple to implement can typically be described with a short snippet of pseudocode which can easily be converted to actual source code in a small amount of time by a computer programmer that has some nominal amount of experience. Typically, the pseudocode for an algorithm that is simple to implement uses mostly data types (for example, integers, arrays, strings etc.), programming statements (for example, while loops, assignments, if/then operators etc.) and basic library functions (for example, queues, heaps, basic I/O etc.) which are available to all programming languages (for example, C++, Fortran, C# etc.). It is important that find-path algorithms are simple to implement for several reasons: It suggests

that, in a short amount of time and with a high probability, the algorithm can be implemented correctly. Furthermore, algorithms that are simple to implement typically have a small amount of code, which means that both the human and the compiler can generate more efficient code. Finally, because there is a small amount of code there will likely be fewer bugs and those bugs that do fall through the cracks can be more cleanly debugged (Vykruta, 2002). Therefore, find-path algorithms that are simple to implement can significantly reduce the level of risk to a robotics or video game company that is producing a product with strict budget limitations and schedule deadlines.

Algorithms that are simple to understand are designed so that it is easy to understand how the algorithm operates (that is, what is going on in the code). Typically, an algorithm that is simple to understand can easily be explained with a short lecture or online tutorial. It is important that find-path algorithms are simple to understand for several reasons: If an algorithm is simple to understand then a programmer not familiar with the algorithm or the code will understand it more clearly and maintain it more cleanly. Furthermore, algorithms that are simple to understand can be more readily modified or extended to conform to the particular requirements of a given path-planning application.

Therefore, it is important that find-path algorithms are simple to implement and understand.

- **Efficiency Property:** *They provide a good tradeoff with respect to the runtime of the search and the length of the resulting path.* When solving the find-path problem there is an inherent tradeoff with respect to two conflicting criteria, namely the runtime of the search and the length of the resulting path. Roboticists and video game developers want find-path algorithms to quickly find short paths.

  It is important that find-path algorithms are fast for several reasons: The find-path problem is only part of the navigation problem, which is only part of the overall functionality required by robotics and video game applications: video games today require thousands of Non-Game Characters (NGCs) to interact with both the Game Character (GC) and the environment in a realistic manner (Figure 1.5), find-path algorithms are often run on low powered computing devices such as those found in robots (for example, the radiation hardened processors of the Mars rovers Spirit and Opportunity, the QRIO and AIBO robots and unmanned ground/aerial/underwater vehicles used by the military etc.) and those found in hand held gaming devices (for example, iPhone, Nintendo 3DS and Playstation Vita etc.).

  It is important that find-path algorithms find short paths for several reasons: In robotics, shorter paths are more efficient and thus can lead to significant cost savings or overall mission effectiveness (for example, a Mars rover can examine more of the planet over the course of its life time and unmanned aerial vehicles can complete their daily missions using less fuel resulting in significant cost savings). In video games, if the paths traversed by the NGCs are unnecessarily long it degrades the overall quality of game play because it undermines the believability of the game's artificial intelligence.

  Therefore, both the runtime of the search and the length of the resulting path are important. However, path length and runtime are conflicting criteria. There is an inherent tradeoff with respect to the two and, as a result, find-path algorithms must intelligently search for short paths. If one find-path algorithm finds shorter paths faster than another find-path algorithm, then the former find-path algorithm **dominates** the latter. If one find-path algorithm does

not dominate another then it is difficult to make precise scientific statements as to which tradeoff is better because different path-planning applications balance these tradeoffs in different ways. For example, fast paced video games may place a premium on short runtimes while slow moving robots may place a premium on short paths. As a result, we use the concept of a good tradeoff, which we define using the following two benchmarks: the length of a true shortest path and the time it takes to find a true shortest path and the length of a shortest edge-constrained path and the time it takes to find a shortest edge-constrained path. A find-path algorithm provides a good tradeoff if it finds a path that is shorter than a shortest edge-constrained path faster than the time it takes to find a true shortest path. For precision, in some cases we use the term **nearly dominates** in place of dominates. We say that one find-path algorithm nearly dominates another when the former find-path algorithm has a shorter runtime than the latter, but the lengths of the paths found by the two find-path algorithms are nearly identical (that is, the lengths of the paths found by the former find-path algorithm are within a couple tenths of a percent of the lengths of the paths found by the latter).

- **Generality Property:** *They can be used to search any Euclidean graph.* Roboticists and video game developers use different generate-graph techniques for discretizing a continuous environment into a graph. The reason for this is that any generate-graph technique that discretizes a continuous environment into a graph has to balance many tradeoffs. For example, graphs with fewer vertices and edges typically require less memory. However, if the graph has too few vertices and edges then it does not provide a complete representation of the traversable space in the continuous environment and thus the paths found when searching the graph are long and unrealistic looking (Tozour, 2004). Furthermore, the ideal balance between these tradeoffs can vary significantly for different robotics and video game applications because they have different considerations and requirements (Tozour, 2004, 2008; Stout, 2000; Latombe, 1991; Choset et al., 2005): are the agents navigating sparse natural environments or cluttered man made environments, what are their capabilities (some agents may be able to cross barbed wire or ice while others may not), do agents walk/run/fly/swim/jump, do agents have complete or incomplete knowledge of the traversable space in the continuous environment, are these graphs created automatically from sensors or the full 3D mesh representation of the environment, or are they created by hand with manual placement of graph edges and vertices?

  It is important that find-path algorithms can be used to search any Euclidean graph for several reasons: While many different generate-graph techniques are used to discretize a continuous environment into a graph, the continuous environment is typically Euclidean and thus the resulting graphs are typically Euclidean as well. A **Euclidean graph** is a graph in which the vertices represent coordinates in a continuous environment, and the edges are assigned lengths equal to the Euclidean distance between the two vertices they connect. The laws of Euclidean geometry are guaranteed to hold in Euclidean graphs. For example, if three edges in a Euclidean graph form a triangle then the lengths of the three edges obey the triangle inequality. Traditional edge-constrained find-path algorithms can be used to find paths on any Euclidean graph because they can be used to search any graph that has non-negative edge lengths. This allows roboticists and video game developers to use one find-path algorithm to find paths on graphs that are constructed using any of the aforementioned

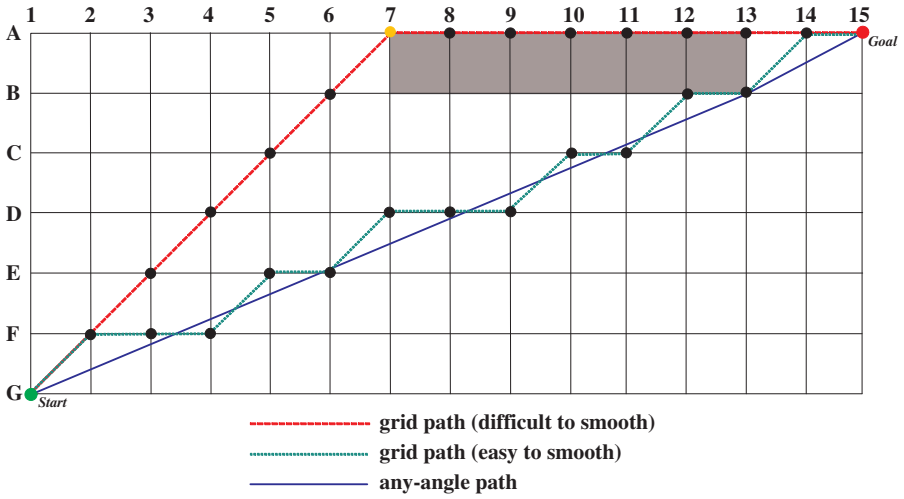


Figure 1.6: Post-Processing Technique

generate-graph techniques that are used to discretize a continuous environment into a graph. This greatly simplifies path planning because roboticists and video game developers only have to implement and maintain a single find-path algorithm.

Therefore, it is important that find-path algorithms can be used to search any Euclidean graph.

While traditional edge-constrained find-path algorithms have many desirable properties, these properties come with a penalty. Traditional edge-constrained find-path algorithms propagate information along graph edges *and* constrain paths to be formed by graph edges. This constraint is artificial and causes the paths found by traditional edge-constrained find-path algorithms to be both longer and less realistic looking than the true shortest paths.[1] This is true even for traditional edge-constrained find-path algorithms, such as Dijkstra's algorithm or A* (with consistent h-values), that are guaranteed to find shortest edge-constrained paths. The differences between edge-constrained paths and true shortest paths can be seen in Figures 1.3 and 1.4. The edge-constrained path found when a traditional edge-constrained find-path algorithm searches the nav graph in Figure 1.2(c) can be seen in Figure 1.4(a) while the true shortest path can be seen in Figure 1.4(c). Similarly, the grid path found when a traditional edge-constrained find-path algorithm searches the grid graph in Figure 1.1(c) can be seen in Figure 1.3(a) while the true shortest path can be seen in Figure 1.3(c). The grid path depicted in Figure 1.3(a) is longer and less realistic looking because of the unnecessary heading change at vertex C2.

While it is well known that traditional edge-constrained find-path algorithms can find long and unrealistic looking paths (Rabin, 2000a; Ferguson & Stentz, 2006) two important questions remain:

[1] A realistic, natural or aesthetic looking path is a subjective criterion which is distinct from a short path. A short path can be unrealistic looking if a character "ziggs and zags" or moves with a restricted set of headings while following the path.

- **Question 1:** *How much longer can the paths found by traditional edge-constrained find-path algorithms be than the true shortest paths?* No comprehensive analysis has been performed which compares the lengths of the paths found by traditional edge-constrained find-path algorithms with the lengths of the true shortest paths. Traditional edge-constrained find-path algorithms can find shortest edge-constrained paths and the only existing path length analysis results that we know of show that shortest grid paths on 8-neighbor square grid graphs can be $\approx 8\%$ longer than true shortest paths (Ferguson & Stentz, 2006) and that shortest grid paths on 6-neighbor triangular grid graphs can be $\approx 15\%$ longer than true shortest paths (Nagy, 2003). However, neither of these results is general enough to be useful for most path-planning applications because both of them completely ignore the potential effects of blocked grid cells on path lengths.

- **Question 2:** *Can more sophisticated find-path algorithms be developed that find shorter and more realistic looking paths than traditional edge-constrained find-path algorithms, while maintaining the desirable properties of traditional edge-constrained find-path algorithms?* It is well known to roboticists and video game developers that traditional edge-constrained find-path algorithms can find paths that are long and unrealistic looking (Ferguson & Stentz, 2006; Rabin, 2000a; Nash, Daniel, Koenig, & Felner, 2007). The most common approach used to find shorter and more realistic looking paths is to apply post-processing techniques to the paths found by traditional edge-constrained find-path algorithms. However, choosing a post-processing technique that efficiently "smooths" the paths found by traditional edge-constrained find-path algorithms into paths that are shorter and more realistic looking can be difficult. One reason for this is that traditional edge-constrained find-path algorithms find one of many edge-constrained paths, some of which can be smoothed more effectively than others. For example, in Figure 1.6, the dashed red path and the dotted green path depict two grid paths formed by the edges of an 8-neighbor square grid graph, the solid blue path depicts the any-angle path and the solid circles depict the vertices on the three paths. The dashed red path and the dotted green path have different topologies because they circumnavigate blocked grid cells differently. The dotted green path can easily be smoothed into the any-angle path by removing vertices on the path that are between two vertices on the path that have line-of-sight. For example, in Figure 1.6, vertex F2 is removed from the dotted green path because it is between vertices G1 and F3, which have line-of-sight and removing vertex F2 shortens the dotted green path. However, due to its topology, the dashed red path cannot easily be smoothed into the any-angle path. Applying the post-processing technique that we applied to the dotted green path does not shorten the dashed red path at all. For example, in Figure 1.6, vertex F2 is removed from the dashed red path because it is between vertices G1 and E3 which have line-of-sight. However, removing vertex F2 does not shorten the dashed red path. Therefore, while these so called aesthetic optimizations (Rabin, 2000a) can make paths shorter and more realistic looking, they are often ineffective (Nash et al., 2007; Nash, 2010) because they do not address the fundamental issue, namely that during the search, traditional edge-constrained find-path algorithms only consider edge-constrained paths and thus cannot make informed decisions about other paths. As a result, traditional edge-constrained find-path algorithms can find paths with topologies that cannot easily be made shorter and more realistic looking with a post-processing technique.

Recently, a more sophisticated approach to path planning was developed, which finds paths that are not constrained to be formed by graph edges by focusing on a specific path-planning problem. Field D* (Ferguson & Stentz, 2006) is an incremental find-path algorithm designed specifically to find paths on 8-neighbor square grid graphs constructed from unknown 2D environments that have been discretized into square grids. It uses a closed form linear interpolation equation to find paths that are not constrained to be formed by the edges of 8-neighbor square grid graphs. While Field D* has proven to be useful to roboticists (Ferguson, 2006; Ferguson & Stentz, 2006; Carsten, Rankin, Ferguson, & Stentz, 2009), it has significant shortcomings, which we discuss in Sections 2.2.4 and 4.3.3 and it was not designed to be a more sophisticated approach to solving the general find-path problem.

To summarize, it is well known that traditional edge-constrained find-path algorithms can find long and unrealistic looking paths, but there does not currently exist a good solution to this problem. Therefore, there is a need for more sophisticated find-path algorithms.

Contribution	Dimension	Regular Grid	Grid Graph	% Longer Than True Shortest Path
1	2D	Triangular grid	3-Neighbor	≈ 100
			6-Neighbor	≈ 15
		Square grid	4-Neighbor	≈ 41
			8-Neighbor	≈ 8
		Hexagonal grid	6-Neighbor	≈ 15
			12-Neighbor	≈ 4
	3D	Cubic grid	6-Neighbor	≈ 73
			26-Neighbor	≈ 13

Table 1.1: Summary of Contribution 1

Contribution	Environment Type	Any-Angle Find-Path Algorithm	Simplicity	Efficiency	Generality
2	Known 2D Environment	Basic Theta*	✓	✓	✓
3	Known 3D Environment	Lazy Theta*	✓	✓	✓
4	Unknown 2D Environment	Incremental Phi*	✓	✓	

Table 1.2: Summaries of Contributions 2-4

1.2 Hypotheses and Contributions

This dissertation uses the following two hypotheses and four contributions to address the aforementioned two questions:

1.2.1 Hypothesis 1

- **Hypothesis 1:** *Analytical bounds can be introduced which compare the lengths of the paths found by traditional edge-constrained find-path algorithms on certain types of graphs with the lengths of the true shortest paths.*

To validate Hypothesis 1 we introduce a simple, unified proof structure which can be used to compare the lengths of the paths found by traditional edge constrained find-path algorithms on grid graphs constructed from 2D and 3D regular grids with the lengths of the true shortest paths. Specifically, we make the following contribution:

Contribution 1: Traditional edge-constrained find-path algorithms can find shortest grid paths and thus we compare the lengths of shortest grid paths with the lengths of true shortest paths. First, we examine regular grids that discretize continuous *2D* environments. We perform a comprehensive path length analysis on the shortest grid paths formed by the edges of grid graphs constructed from all three types of regular grids that can be used to tessellate continuous 2D environments, namely triangular, square and hexagonal grids. We show that shortest grid paths can be longer than true shortest paths as follows: 100% for 3-neighbor triangular grid graphs, $\approx 41\%$ for 4-neighbor square grid graphs, $\approx 15\%$ for 6-neighbor hexagonal grid graphs, $\approx 15\%$ for 6-neighbor triangular grid graphs, $\approx 8\%$ for 8-neighbor square grid graphs and $\approx 4\%$ for 12-neighbor hexagonal grid graphs. Second, we examine regular grids that discretize continuous *3D* environments. We perform a comprehensive path length analysis on the shortest grid paths formed by the edges of grid graphs constructed from the only type of regular grid that can be used to tessellate continuous 3D environments, namely a cubic grid (Gosset, 1900). We show that shortest grid paths can be longer than true shortest paths as follows: $\approx 73\%$ for 6-neighbor cubic grid graphs and $\approx 13\%$ for 26-neighbor cubic grid graphs (Nash, Koenig, & Tovey, 2010). A summary of these bounds can be seen in Table 1.1.

1.2.2 Hypothesis 2

- **Hypothesis 2:** *A new class of any-angle find-path algorithms, that propagate information along graph edges, without constraining paths to be formed by graph edges, can be used to quickly find paths that are shorter than the paths found by traditional edge-constrained find-path algorithms, while maintaining the Simplicity and Generality Properties of traditional edge-constrained find-path algorithms.*

Robotics and video game applications require that agents perform path planning in many different types of continuous environments. We define a **continuous environment type** by the dimensionality of the continuous environment and how much agents know about the traversable space in the continuous environment. For example, in known 2D environments, agents have complete knowledge of the traversable space and freedom of movement in two dimensions; in known 3D environments, agents have complete knowledge of the traversable space and freedom

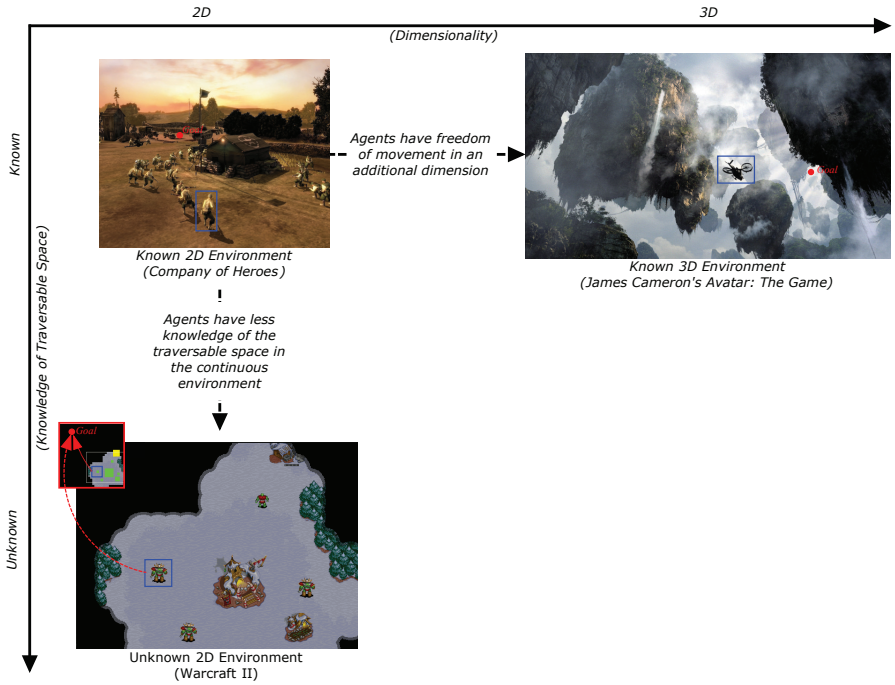


Figure 1.7: Different Types of Continuous Environments

of movement in three dimensions; and, in unknown 2D environments, agents have incomplete knowledge of the traversable space and freedom of movement in two dimensions. All three types of continuous environments are widely used by roboticists and video game developers (Carsten et al., 2009; Ferguson & Stentz, 2006; Stentz & Hebert, 1995; Hebert, MacLachlan, & Chang, 1999; Tozour, 2004; Patel, 2000), and thus it is important to evaluate find-path algorithms in all three types of environments. Known 2D environments are the simplest type of environment in which path planning is performed. Known 3D environments and unknown 2D environments make path planning more difficult for two orthogonal reasons (Figure 1.7). In the former case, the fact that agents have freedom of movement in three dimensions instead of two dimensions makes path planning more difficult (for reasons explained in Section 1.2.2.2) and, in the latter case, the fact that agents have less knowledge of the traversable space in the continuous environment makes path planning more difficult (for reasons explained in Section 1.2.2.3).

To validate Hypothesis 2, we introduce a new class of **any-angle find-path algorithms**, that propagate information along graph edges (to achieve a short runtime) without constraining paths to be formed by graph edges (to find any-angle paths). We introduce new members to this class and evaluate each member in one of the aforementioned three types of continuous environments using the Simplicity, Efficiency and Generality Properties as our evaluation criteria. Table 1.2 provides a summary of the (most significant) contributions made during this evaluation. We begin by introducing Basic Theta*, a new any-angle find-path algorithm that we evaluate in the simplest type of continuous environment, namely known 2D environments. We build on the key ideas from Basic Theta* to develop Lazy Theta*, a new any-angle find-path algorithm that we evaluate in known 3D environments and Incremental Phi*, a new incremental any-angle find-path

algorithm that we evaluate in unknown 2D environments. Specifically, we make the following three contributions:

1.2.2.1 Basic Theta*: Any-Angle Path Planning in Known 2D Environments

Contribution 2: We demonstrate that an any-angle find-path algorithm can satisfy the Simplicity, Efficiency and Generality Properties when path planning in the simplest type of continuous environment, namely known 2D environments. To that end, we introduce Basic Theta (Nash et al., 2007; Nash, Daniel, Koenig, & Felner, 2010), a new any-angle find-path algorithm. Basic Theta* is an any-angle variant of A*, that is, it propagates information along graph edges, without constraining paths to be formed by graph edges. Basic Theta* does this by intelligently using line-of-sight-checks to consider shortcuts that take advantage of the triangle inequality during an A* search. Basic Theta* is extremely similar to A*, and thus it is as simple to implement and understand as A* (Simplicity Property). We evaluate Basic Theta* on 8-neighbor square grid graphs both because they are widely used when path planning in known 2D environments (Murphy, 2000; Ferguson & Stentz, 2006; Stout, 2000; Tozour, 2004) and because it allows for comparisons with both Field D*, the current state-of-the-art any-angle find-path algorithm and traditional edge-constrained find-path algorithms. We compare Basic Theta* experimentally with A*, A* with a post-processing technique and Field D*. We show that Basic Theta* provides a dominating tradeoff over Field D* and A* with a post-processing technique with respect to the runtime of the search and the length of the resulting path. We show that Basic Theta* provides a good tradeoff with respect to the runtime of the search and the length of the resulting path. The runtime of Basic Theta* is similar to that of A*, but Basic Theta* finds paths which are $\approx 4 - 5\%$ shorter than the paths found by A* on average (Efficiency Property).[2] The lengths of the paths found by Basic Theta* are nearly identical to the lengths of the true shortest paths. We use metrics which correlate with realism to show that the paths found by Basic Theta* are more realistic looking than those found by Field D*. The key ideas behind Basic Theta* depend only on the triangle inequality and, since the triangle inequality is guaranteed to hold in any Euclidean environment, Basic Theta* can be used to search any Euclidean graph (Generality Property). The worst-case complexity of Basic Theta* is greater than that of A* because the runtime of each line-of-sight check can be linear in the number of vertices. We introduce a new any-angle find-path algorithm, Angle-Propagation Theta*, which reduces the worst-case complexity of Basic Theta* so that it is the same as that of A*. Angle-Propagation Theta* does this by intelligently using angle ranges (instead of line-of-sight checks) to consider shortcuts that take advantage of the triangle inequality during an A* search. Angle-Propagation Theta* is similar to Basic Theta* and has a better worst-case complexity, but is more difficult to implement and understand, is not as fast, finds slightly longer paths and cannot be used to search any Euclidean graph.

1.2.2.2 Lazy Theta*: Any-Angle Path Planning in Known 3D Environments

Contribution 3: We demonstrate that an any-angle find-path algorithm can satisfy the Simplicity, Efficiency and Generality Properties when path planning in known 3D environments. Path

[2] In our experiments, A* is guaranteed to find shortest grid paths and thus the paths found by Basic Theta* are $\approx 4 - 5\%$ shorter than shortest grid paths on average.

planning in known 3D environments is more difficult than path planning in known 2D environments. True shortest paths in known 2D environments with polygonal obstacles can be found in polynomial time (Lozano-Pérez & Wesley, 1979), but finding true shortest paths in known 3D environments with polyhedral obstacles is NP-hard (Canny & Reif, 1987). Basic Theta* can be applied to known 3D environments without any changes to the pseudocode. However, Basic Theta* is less efficient in known 3D environments than it is in known 2D environments. This is because Basic Theta* can perform up to one line-of-sight check for each neighbor of each vertex that is expanded during the search and a vertex can have far more neighbors on a 26-neighbor *cubic* grid graph than on an 8-neighbor *square* grid graph. Thus, in order for an any-angle find-path algorithm to satisfy the Efficiency Property in known 3D environments, it must be smarter about when it performs line-of-sight checks. To that end, we introduce Lazy Theta* (Nash et al., 2010), a new any-angle find-path algorithm that is a more efficient variant of Basic Theta* designed for path planning in known 3D environments. Lazy Theta* is similar to Basic Theta* and thus is simple to implement and understand (Simplicity Property). We evaluate Lazy Theta* on 26-neighbor cubic grid graphs because they are widely used when path planning in known 3D environments (Carsten et al., 2006; Cohen, Subramanian, Chitta, & Likhachev, 2011). We compare Lazy Theta* experimentally with A*, A* with a post-processing technique and Basic Theta*. We show that Lazy Theta* and Basic Theta* provide a dominating tradeoff over A* with a post-processing technique with respect to the runtime of the search and the length of the resulting path. We show that Lazy Theta* provides a nearly dominating tradeoff over Basic Theta* with respect to the runtime of the search and the length of the resulting path. Lazy Theta* is more efficient than Basic Theta* because it frequently performs more than one order of magnitude fewer line-of-sight checks than Basic Theta* while finding paths whose lengths are nearly identical to the lengths of the paths found by Basic Theta*. We show that Lazy Theta* and Basic Theta* both find paths that are $\approx 7 - 8\%$ shorter than the paths found by A* on average.[3] We show that, in certain types of known 3D environments Lazy Theta* with optimizations can provide a dominating tradeoff over A* with a post-processing technique and Basic Theta* with respect to the runtime of the search and the length of the resulting path. Finally, we show that, in certain types of known 3D environments, Lazy Theta* with optimizations can finds paths that are $\approx 15\%$ shorter than the paths found by A* on average, but with a similar runtime (Efficiency Property). Like Basic Theta*, the key ideas behind Lazy Theta* depend only on the triangle inequality and, since the triangle inequality is guaranteed to hold in any Euclidean environment, Lazy Theta* can be used to search any Euclidean graph (Generality Property).

1.2.2.3 Incremental Phi*: Any-Angle Path Planning in Unknown 2D Environments

Contribution 4: We demonstrate that an any-angle find-path algorithm can satisfy the Simplicity and Efficiency Properties when path planning in unknown 2D environments. Path planning in unknown 2D environments is more difficult than path planning in known 2D environments because an agent's knowledge of the traversable space in the continuous environment can change as it navigates towards a given goal coordinate. An agent can account for this using one of two approaches: **(1)** the agent can generate a contingency plan for everything that it could potentially learn about the unknown regions of traversable space in the continuous environment. It has been

[3] In our experiments, A* is guaranteed to find shortest grid paths and thus the paths found by Basic Theta* and Lazy Theta* are $\approx 7 - 8\%$ shorter than shortest grid paths on average.

shown that this approach is extremely difficult because the agent must generate very large conditional plans and thus it is too time consuming to be used by roboticists and video game developers (Koenig, Smirnov, & Tovey, 2003). On the other hand, **(2)** the agent can find an initial path by making assumptions about the regions of traversable space in the continuous environment that are unknown to the agent. For example, roboticists often use the **freespace assumption**, that is, they assume that the agent is able to traverse areas that are unknown to it (Stentz & Hebert, 1995; Hebert et al., 1999). While the agent follows the path it learns more about the traversable space in the continuous environment. Each time the agent learns that an assumption it made about the unknown regions of traversable space in the continuous environment is incorrect (for example, the agent learns that its current path is blocked by an obstacle that was previously in an unknown region of traversable space in the continuous environment) it updates the graph representation of the traversable space in the continuous environment to account for its new knowledge and solves a new find-path problem. Thus, in order for an any-angle find-path algorithm to satisfy the Efficiency Property in unknown 2D environments using the free space assumption, it must be able to efficiently solve a series of similar find-path problems. Incremental find-path algorithms are a class of algorithms which are commonly used when planning with the freespace assumption because they efficiently solve a series of similar find-path problems by reusing information from previous find-path problems to find a solution to the next find-path problem more quickly. However, incremental find-path algorithms such as Differential A* (Trovato & Dorst, 2002) and D* Lite (Koenig & Likhachev, 2002a) are edge-constrained find-path algorithms and thus there is a need for incremental any-angle find-path algorithms. To that end, we introduce Incremental Phi* (Nash, Koenig, & Likhachev, 2009), a new *incremental* any-angle find-path algorithm. Incremental Phi* is a new incremental any-angle find-path algorithm that is a more efficient variant of Basic Theta* designed for path planning in unknown 2D environments. Incremental Phi* is similar to both Basic Theta* and existing incremental find-path algorithms and thus is simple to implement and understand (Simplicity Property). However, incremental find-path algorithms, including Incremental Phi*, are more difficult to implement and understand than traditional edge-constrained find-path algorithms because they solve a more difficult problem. We evaluate Incremental Phi* on 8-neighbor square grid graphs using the freespace assumption because this experimental setup is widely used by roboticists when path planning in unknown 2D environments (Koenig & Likhachev, 2002a, 2002b; Koenig, Likhachev, Liu, & Furcy, 2004; Ferguson & Stentz, 2006; Trovato & Dorst, 2002). We compare Incremental Phi* experimentally with Basic Theta*. We show that Incremental Phi* provides a nearly dominating tradeoff over repeated Basic Theta* searches with respect to the runtime of the search and the length of the resulting path. Incremental Phi* solves a series of similar find-path problems one order of magnitude faster than repeated Basic Theta* searches from scratch by reusing information from previous searches to speed up the next one and it does so while finding paths whose lengths are nearly identical to the lengths of the paths found by Basic Theta* (Efficiency Property). Making a variant of Basic Theta* incremental is non-trivial in that any-angle find-path algorithms do not conform to the standard assumption of incremental find-path algorithms, namely that the parent of a vertex in the search tree must also be its neighbor. Due to the lack of this property, Incremental Phi* requires that graphs have additional properties other than just being Euclidean. In other words, Incremental Phi* does not satisfy the Generality Property.

1.3 Dissertation Structure

This dissertation is structured as follows: In Chapter 2, we give a more in-depth overview of path planning. In Chapter 3, we compare the lengths of the paths found by traditional edge-constrained find-path algorithms with the lengths of the true shortest paths (Contribution 1). In the following three chapters we introduce the following any-angle find-path algorithms: In Chapter 4, we introduce Basic Theta*, which we evaluate in known 2D environments (Contribution 2). In Chapter 5, we introduce Lazy Theta*, which we evaluate in known 3D environments (Contribution 3). In Chapter 6, we introduce Incremental Phi*, which we evaluate in unknown 2D environments (Contribution 4). Finally, in Chapter 7, we summarize the contributions made in this dissertation.

Chapter 2

Path Planning

This chapter begins by motivating path planning as a key part of navigation in Section 2.1. We then provide an overview of existing path-planning research in Section 2.2.

2.1 Navigation

Navigating an agent from a given start coordinate to a given goal coordinate through a continuous environment is one of the most important problems faced by roboticists and video game developers (Deloura, 2000; Rabin, 2002, 2004; Latombe, 1991; Murphy, 2000; Choset et al., 2005). Agents must be able to navigate between coordinates in a realistic manner without getting lost, getting stuck or bumping into obstacles. When navigating an agent from a given start coordinate to a given goal coordinate one must address both path planning and movement. Addressing path planning and movement simultaneously is extremely difficult (Latombe, 1991; Reif, 1979; Canny, 1988) and thus it is very common to treat path planning and movement as two separate, but equally important parts of navigation (Patel, 2000; Ferguson, 2006).

Path planning is important because it allows an agent to plan ahead rather than just reacting to the environment. When an agent reacts to an environment (that is, movement) without planning ahead (that is, path planning) it is less likely to follow a short and realistic looking path. This can be seen by examining the Bug algorithm (Lumelsky & Stepanov, 1990), a simple approach to navigation in which the agent heads toward the goal coordinate and takes into account obstacles when it finds them. Thus, when an agent navigates using the Bug algorithm it performs movement with very little path planning. The agent heads toward the goal coordinate until it is about to bump into an obstacle, at which point it follows the boundary of the obstacle while keeping track of the closest coordinate p to the goal coordinate. After the agent has traversed the entire circumference of the obstacle, it returns to p and again heads toward the goal coordinate. This process repeats until the agent reaches the goal coordinate at which point the algorithm terminates. The Bug algorithm only requires that the agent know its own position, the goal coordinate and whether or not it is in contact with an obstacle and thus it works well when a simple agent has no knowledge of the traversable space in the continuous environment and only has basic sensors. Furthermore, the Bug algorithm is simple and guaranteed to find a path to the goal coordinate if such a path exists (Lumelsky & Stepanov, 1990).[1] However, because the Bug algorithm greedily heads directly toward the goal coordinate it is susceptible to local minima. This can be seen in

[1] This property holds if every obstacle is bounded by a simple closed curve of finite length and any straight line intersects only a finite number of obstacles.

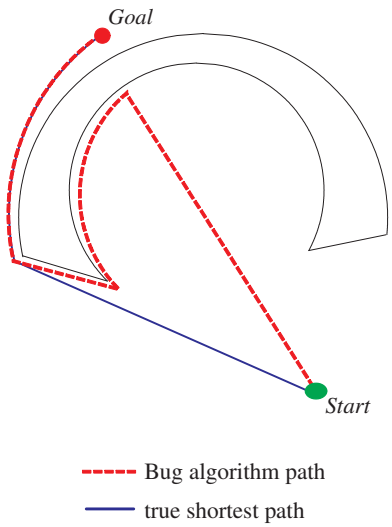


Figure 2.1: Movement without Path Planning

Figure 2.1, in which the Bug algorithm follows a path that is much longer than a true shortest path.

While path planning is important, it must be performed in concert with movement. Path-planning techniques discretize the traversable space in the continuous environment and ignore kinematic actions and dynamic constraints. Approximating the continuous environment and ignoring kinematic actions and dynamic constraints makes the problem simpler and far more manageable (Choset et al., 2005; Ferguson, 2006). However, as a consequence, the agent may not be able to follow the paths that it finds. For example, path planning typically assumes that agents can turn in place, which is certainly not the case for fast moving agents. As a result, path-planning techniques, which are often referred to as *global* planners, must be used in concert with *local* planners, which account for movement by taking into account these kinematic actions and dynamic constraints (Howard & Kelly, 2005; Howard, Knepper, & Kelly, 2006; Patel, 2000).

This dissertation examines path planning.

2.2 Path Planning

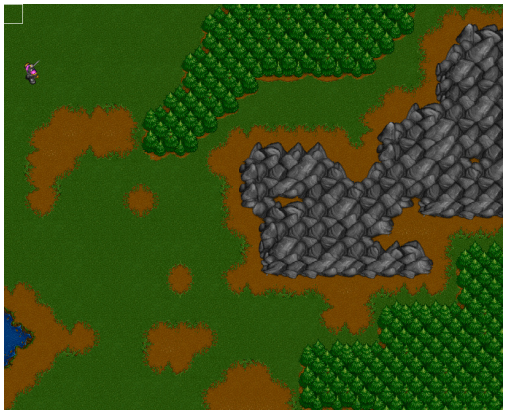
Path planning is typically composed of two parts: the generate-graph problem, which is solved by discretizing a continuous environment into a graph and the find-path problem, which is solved by searching this graph for a path from a given start vertex to a given goal vertex. For example, one approach to solving the generate-graph problem is to construct an 8-neighbor square grid graph from the continuous environment. First, the continuous environment is discretized into a square grid by laying the regular grid over the continuous environment. Grid cells are labeled blocked (grey) if they contain part of an obstacle and all other grid cells are labeled unblocked (white). An 8-neighbor square grid graph is then constructed by placing vertices in the corners of grid cells and adding an edge between a vertex and up to 8 of its adjacent vertices. In Figure 2.2, we have depicted this process beginning with a continuous environment (Figure 2.2(a)) and

concluding with an 8-neighbor square grid graph (Figure 2.2(d)). Once a graph has been constructed from the continuous environment, the find-path problem is solved using one of many traditional edge-constrained find-path algorithms such as Dijkstra's algorithm or A*, which find an edge-constrained path between a given start vertex and a given goal vertex. In Figure 2.2(e), the dashed red path depicts a grid path between the start vertex (green circle) and the goal vertex (red circle). In Figure 2.2(f), the dashed red path depicts that same path in the continuous environment.

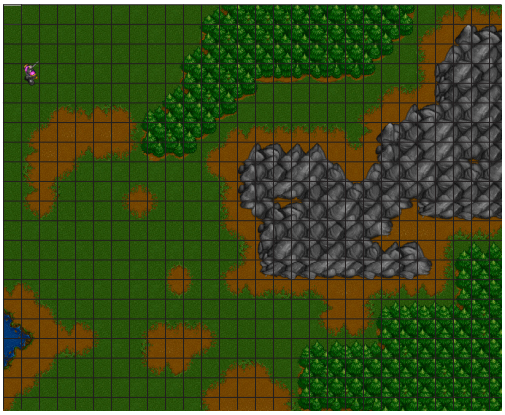
In the following sections, we provide an overview of existing path-planning research: Section 2.2.1 provides an overview of existing generate-graph techniques that are used to discretize known 2D environments, known 3D environments and unknown 2D environments into graphs (that is, solve the generate-graph problem). We highlight the tradeoffs that each generate-graph technique provides and demonstrate that there is no generate-graph technique that is superior to all others for every path-planning application. Section 2.2.2 provides an overview of traditional edge-constrained find-path algorithms that are used to find edge-constrained paths (that is, solve the find-path problem) in known 2D environments, known 3D environments and unknown 2D environments. We highlight that, while traditional edge-constrained find-path algorithms maintain the Simplicity, Efficiency and Generality Properties, the paths that they find can be longer and less realistic looking than true shortest paths. Section 2.2.3 provides an overview of post-processing techniques that can be used to make edge-constrained paths shorter and more realistic looking. We highlight that, while post-processing techniques can improve the paths found by traditional edge-constrained find-path algorithms, they are not always effective at making paths shorter and more realistic looking and, as a result, more sophisticated approaches to solving the find-path problem, such as any-angle find-path algorithms, are needed. Finally, Section 2.2.4 provides a brief introduction to Field D*, the current state-of-the art any-angle find-path algorithm. We highlight that, while Field D* is an any-angle find-path algorithm, it is designed for a very specific path-planning problem and has substantial shortcomings.

2.2.1 The Generate-Graph Problem

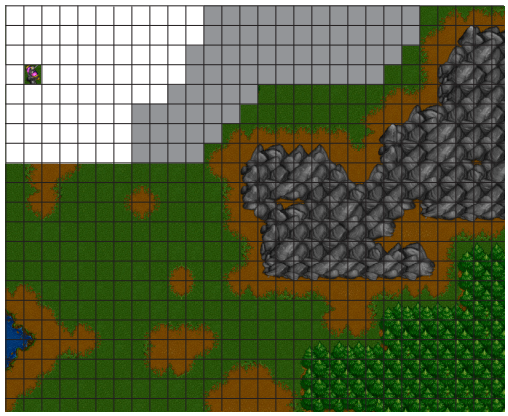
Robotics and video game applications require that path planning be performed in many different types of continuous environments, such as known 2D environments, known 3D environments and unknown 2D environments. In this section, we introduce several different generate-graph techniques that can be used to discretize all three types of environments into finite Euclidean graphs. Each generate-graph technique that we introduce can be classified as either a skeletonization technique or a cell decomposition technique (Choset et al., 2005; Russel & Norvig, 1995). We define each class and introduce examples in Section 2.2.1.1 and 2.2.1.2. A generate-graph technique is composed of two parts, the discretization of the continuous environment and the graph constructed from the discretization. For skeletonization techniques these two parts coincide and thus we do not distinguish between them. We introduce each generate-graph technique in the context of known 2D environments both because path planning is most commonly performed in known 2D environments and because known 2D environments are easy to visualize. Once we have described how the generate-graph technique can be used when path planning in known 2D environments we describe how it can be used when path planning in known 3D environments. In Section 2.2.1.4, we describe a general technique for modifying each of the introduced generate-graph techniques so that they can be used to discretize unknown 2D environments.



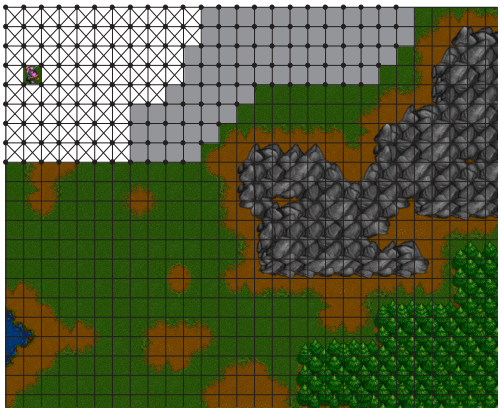
(a) Continuous Environment



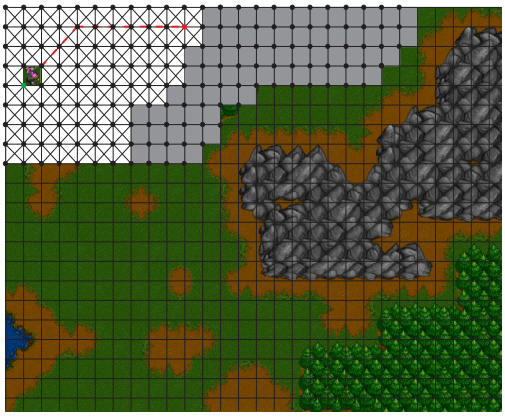
(b) Continuous Environment Discretized into a Square Grid



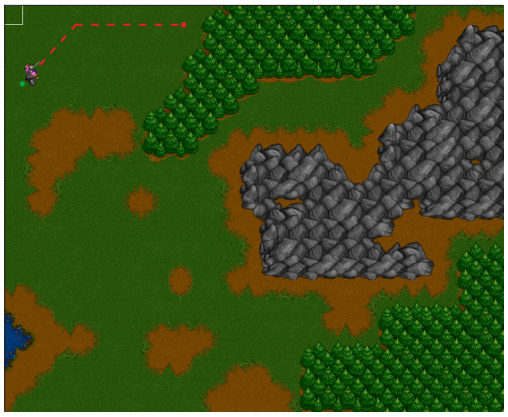
(c) Square Grid with Blocked and Unblocked Grid Cells



(d) 8-Neighbor Square Grid Graph Constructed from the Square Grid



(e) Grid Path on the 8-Neighbor Square Grid Graph

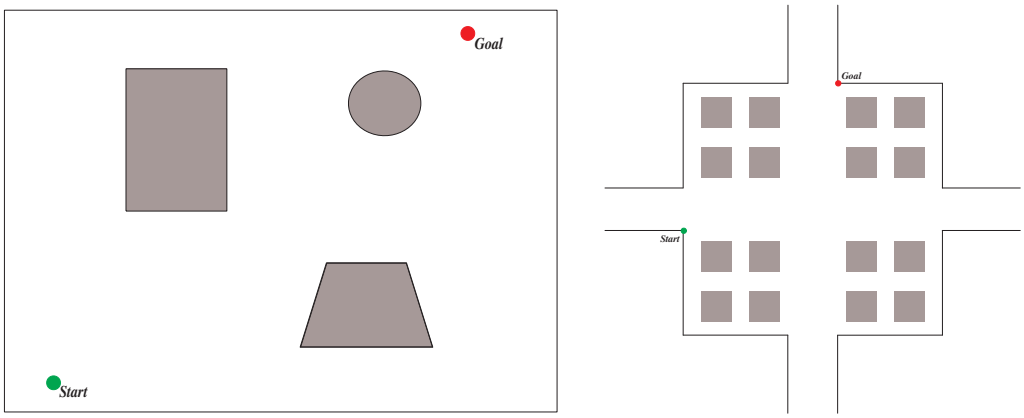


(f) Grid Path in the Continuous Environment

Figure 2.2: Screen Shots from Warcraft II (Blizzard Entertainment)

2.2.1.1 Skeletonization Techniques

The first class of generate-graph techniques is based on the idea of skeletonization. **Skeletonization techniques** extract a skeleton from the continuous environment which captures the salient



(a) Natural Environment

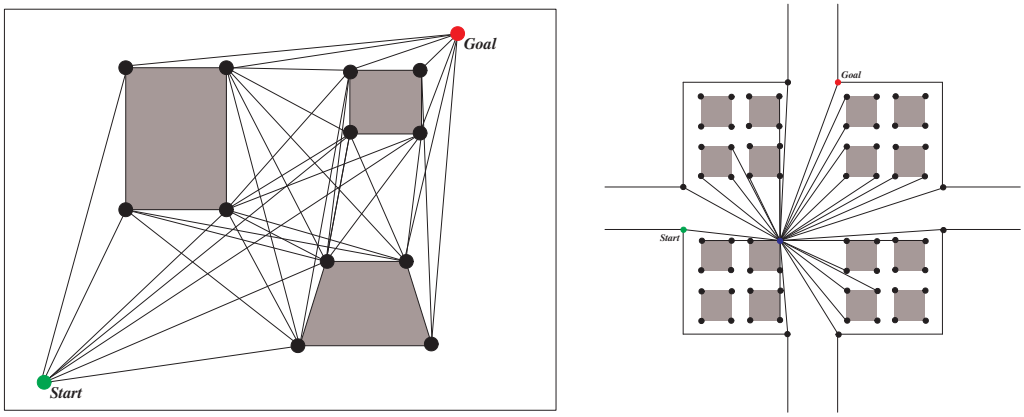
(b) Man Made Environment (adapted from (Tozour, 2004))

Figure 2.3: Continuous 2D Environments

topology of the traversable space in the continuous environment. Skeletonization techniques define a graph $G = (V, E)$, where V is a set of vertices such that each vertex maps to a coordinate in the continuous environment and E is a set of edges which connect vertices in V that have line-of-sight.

We examine two skeletonization techniques, namely visibility graphs and waypoint graphs. First, we describe how each skeletonization technique can be used to discretize known 2D environments and then we discuss the tradeoffs that each technique provides when it is used to discretize two types of known 2D environments, namely man made environments (that is, environments that are constructed by man) and natural environments (that is, environments that are found in nature). Man made environments often have rectangular obstacles, such as buildings, and passageways, such as hallways, bridges and roads. Natural environments are typically sparsely populated with arbitrarily shaped obstacles that are placed randomly throughout the continuous environments. We examine these two types of environments because they highlight some of the tradeoffs that roboticists and video game developers consider when choosing a generate-graph technique. We evaluate each generate-graph technique on both man made and natural environments based on the tradeoff that it provides with respect to the number of vertices and edges in the resulting graphs and the length and realism of the paths that the graphs contain (that is, the lengths and realism of the paths formed by the edges in the graphs). Roboticists and video game developers typically prefer graphs with fewer vertices and edges because such graphs require less memory and can be searched for paths more quickly. Roboticists and video game developers typically prefer graphs that contain shorter and more realistic looking paths because traditional edge-constrained find-path algorithms find paths that are formed by graph edges. Thus, if the paths contained in the graphs are shorter and more realistic looking, then the paths that agents follow are shorter and more realistic looking.

- **Visibility Graphs:** Visibility graphs (sometimes referred to as corner graphs) are widely used by roboticists (Latombe, 1991; Choset et al., 2005) and video game developers (Tozour, 2004; Rabin, 2000b). We begin by discussing how visibility graphs can be used to

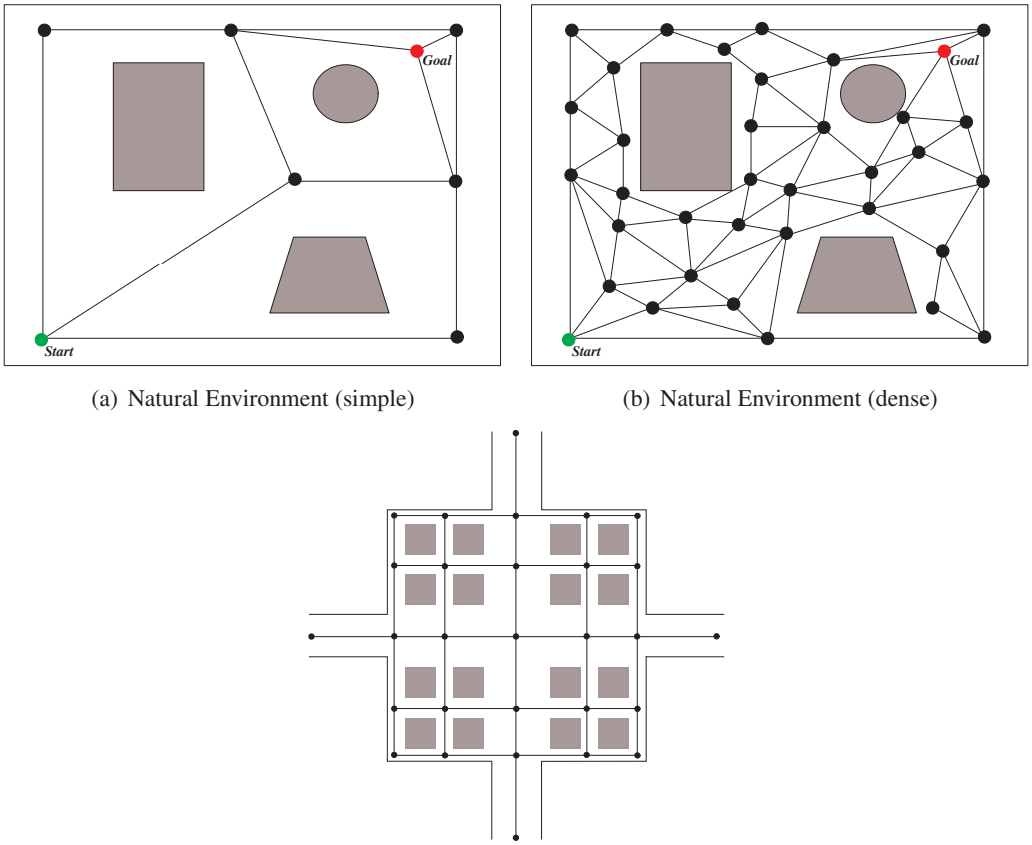


(a) Natural Environment (b) Man Made Environment

Figure 2.4: Visibility Graphs

discretize generic known 2D environments. Visibility graphs discretize continuous 2D environments into graphs by placing a vertex at the start coordinate (that is, the start vertex), the goal coordinate (that is, the goal vertex) and in the corners of each polygonal obstacle. An edge is then added to the graph for each pair of vertices that have line-of-sight (Lozano-Pérez & Wesley, 1979; Lee, 1978). Each vertex can have edges connecting it to all other vertices and thus the number of edges in visibility graphs can be quadratic in the number of their vertices. The number of edges can be reduced by using reduced visibility graphs (Liu & Arimoto, 1992). However, reduced visibility graphs are difficult to implement and are often ineffective because the number of edges in reduced visibility graphs can still be quadratic in the number of their vertices. Visibility graphs (and reduced visibility graphs) are guaranteed to contain true shortest paths in known 2D environments with polygonal obstacles (Lozano-Pérez & Wesley, 1979) and therefore shortest edge-constrained paths between any two vertices in visibility graphs are guaranteed to be short and realistic looking.

We now discuss some of the tradeoffs considered when visibility graphs are used to discretize natural and man made environments. Natural environments are typically sparsely populated with obstacles which means that, when visibility graphs are used to discretize such environments, they typically have a small number of vertices (and thus a smaller number of edges) (Figure 2.4(a)). Man made environments are typically more densely populated with obstacles which means that, when visibility graphs are used to discretize such environments, they typically have a larger number of vertices (and thus a larger number of edges) (Figure 2.4(b)). For example, Figure 2.4(b) only depicts the edges in the visibility graph that have the blue vertex as an endpoint. The complete visibility graph would have far more edges. In both man made and natural environments, visibility graphs are guaranteed to contain true shortest paths. Therefore, in natural environments, visibility graphs typically provide a better tradeoff than in man made environments with respect to the number of vertices and edges in the graphs and the length and realism of the paths that the graphs contain.



(a) Natural Environment (simple)

(b) Natural Environment (dense)

(c) Man Made Environment

Figure 2.5: Waypoint Graphs

Visibility graphs can be constructed from known 3D environments without any changes. However, visibility graphs constructed from known 3D environments are not guaranteed to contain true shortest paths (Choset et al., 2005). Furthermore, polyhedral obstacles typically have more vertices than polygonal obstacles and thus, when visibility graphs are used to discretize known 3D environments, the resulting graphs typically have more vertices and edges. Therefore, the tradeoff with respect to the number of vertices and edges in the graphs and the lengths and realism of the paths that the graphs contain is less desirable for both natural environments and man made environments in known 3D environments than it is in known 2D environments.

- **Waypoint Graphs:** Waypoint graphs are widely used by video game developers (for example, the video game Killzone 2 (Champandard, 2010)) and **Voronoi graphs** (Aurenhammer, 1991), which are similar in spirit to waypoint graphs, are widely used by roboticists. We begin by discussing how waypoint graphs can be used to discretize generic known 2D environments. Waypoint graphs discretize continuous 2D environments into graphs by placing a vertex at the start coordinate (that is, the start vertex), the goal coordinate (that is, the goal vertex) and in open spaces or the middle of passageways. An edge is then added to the graph connecting a vertex with up to k of its closest vertices that have line-of-sight.

The value of k can be different for each vertex, but it is typically relatively constant and much smaller than the number of vertices. There is no rigorous definition as to where vertices are placed in the continuous environment or what value to use for k. The specific requirements of a particular path-planning application determines how waypoint graphs are constructed. In fact, the placement of vertices and the values of k are typically customized by a level designer in order to account for the nuances of a particular continuous environment (for example, the desired game play on a particular level of a video game) (Tozour, 2002, 2004). However, automated techniques for discretizing known 2D environments into waypoint graphs do exist. For example, the space between each pair of obstacles (including the boundaries of the continuous environment) can be modeled as $m \times n$ rectangles where $m > n$. Each rectangle has a central axis defined by a straight line connecting the midpoints of the two sides of the rectangle that have length n (Stout, 2000). Stout (2000) refers to the rectangle and its central axis as a 2D cylinder. The endpoints and intersection points of these central axes determine where vertices are placed in the continuous environment and the axes themselves define the edges.

We now discuss some of the tradeoffs considered when waypoint graphs are used to discretize natural and man made environments. Man made environments have passageways which constrain an agent's movement and, as a result, a waypoint graph with a small number of well placed vertices and edges can contain realistic looking paths (Figure 2.5(c)). For example, the waypoint graph in Figure 2.5(c) has fewer vertices and edges than the visibility graph in Figure 2.4(b) and yet still contains realistic looking paths. Natural environments typically have large open spaces and as a result the paths contained in waypoint graphs artificially constrain an agent's movement to the middle of open spaces which often results in long and unrealistic looking paths (Figure 2.5(a)). This problem can be mitigated by adding more vertices and edges to the waypoint graph (Figure 2.5(b)), but it is difficult to represent a short and realistic looking path between any pair of coordinates in the continuous environment with a discrete number of paths. Therefore, in man made environments, waypoint graphs typically provide a better tradeoff with respect to the number of vertices and edges in the graphs and the length and realism of the paths that the graphs contain than they provide in natural environments.

Waypoint graphs can be constructed from known 3D environments without any changes because vertices can be placed anywhere in continuous 3D environments. However, there are far more short and realistic looking paths that an agent can follow between any pair of coordinates in a (natural or man made) known 3D environment than there are in a known 2D environment, and thus waypoint graphs constructed from known 3D environments typically have more vertices and edges than waypoint graphs constructed from known 2D environments. For example, in a man made environment, an agent may be able to walk and fly (at different altitudes) down a passageway, which would require that lots of edges and vertices be placed in the passageway. Therefore, the tradeoff that waypoint graphs provide with respect to the number of vertices and edges in the graphs and the length and realism of the paths that the graphs contains is less desirable for both natural and man made environments in known 3D environments than it is in known 2D environments.

We have argued that, in man made environments, waypoint graphs often provide a better tradeoff with respect to the number of vertices and edges in the graphs and the length and realism

of the paths that the graphs contain than visibility graphs provide, where as in natural environments, visibility graphs often provide a better tradeoff with respect to the number of vertices and edges in the graphs and the length and realism of the paths that the graphs contain than waypoint graphs provide.

Skeletization techniques such as visibility graphs and waypoint graphs are simple to understand, but have two major shortcomings:

- The path-planning system has no information about the traversability of the continuous environment other than what is defined by the graph G. This can be problematic for the following reason: Determining whether or not two vertices have line-of-sight requires a check that must be performed on the raw geometry of the continuous environment. The continuous environment can be large and intricate and thus determining whether or not two vertices have line-of-sight can require lots of floating point computations, which can be slow. For example, in video games, the continuous environment is typically defined by a set of meshes which define polyhedrons that have thousands of vertices and hundreds of faces. Thus, line-of-sight checks require a large amount of computation. This makes two important parts of path planning slower:

  - Determining whether or not an edge can be added to the graph requires a line-of-sight check and thus constructing graphs can be time consuming. For example, the complete visibility graph for the known 2D environment in Figure 2.4(b) (without a specific start and goal coordinate) contains 72 vertices and thus requires $2,556 = (72 \times 71)/2$ line-of-sight checks. Several skeletization techniques have been developed to minimize the time required to determine whether or not an edge can be added to the graph: Probabilistic Roadmaps (PRMs) (Kavraki et al., 1996) can be constructed using lazy evaluation, which means that these checks are delayed until they are absolutely necessary (Bohlin & Kavraki, 2000; Sanchez & Latombe, 2002). We make use of this technique in Chapter 5. Rapidly-exploring random trees (RRTs) (LaValle & Kuffner, 2001; LaValle et al., 2004), Adriadne's Clew Algorithm (Mazer, Ahuactzin, & Bessire, 1996) and Expansive-Space Trees (ESTs) (Hsu, Latombe, & Motwani, 1997) are constructed by interleaving the generate-graph technique with the find-path algorithm in order to ensure that only edges that are needed to find a path (typically not a true shortest path) from the start vertex to the goal vertex are added to the graph.[2] However, these techniques must be used with care because examining fewer edges can result in graphs that contain longer and less realistic looking paths.
  - Post-processing techniques often need to check whether or not an agent can follow a straight line between non-consecutive vertices on the path that are not connected by an edge in the graph. Since these checks must be performed on the raw geometry of the continuous environment, post-processing techniques are often too slow to be used in conjunction with skeletization techniques (Tozour, 2004). In fact, some video games are designed such that the path-planning system does not have access to the raw geometry of the continuous environment and thus post-processing techniques cannot be used with skeletonization techniques (Tozour, 2008).

[2]This technique is often used in the context of visibility graphs in order to reduce the total runtime spent checking whether or not two vertices have line-of-sight.

	Traversable Space	Memory	Difficulty	3D
Regular Grids	$\checkmark^{+}$	$\checkmark$	$\checkmark^{++}$	$\checkmark$
Circle Based Waypoints	$\checkmark$	$\checkmark^{++}$	$\checkmark^{+}$	
NavMeshes	$\checkmark^{++}$	$\checkmark^{+}$	$\checkmark$	

Table 2.1: Cell Decomposition Techniques

- The only way to make the paths contained in the graph shorter and more realistic looking is to increase the number of vertices and edges in the graph which is not an efficient mechanism for representing the traversable space in a continuous environment.

In general, a path-planning system is constantly computing whether or not an agent can navigate between any two given coordinates. For many path-planning applications it is impossible to explicitly represent a short and realistic looking path between any two given coordinates using a graph G constructed using a skeletonization technique because of memory and computation limitations. Therefore, a discretization of the continuous environment that attempts to represent more of the traversable space in the environment is often required.

2.2.1.2 Cell Decomposition Techniques

The second class of generate-graph techniques is based on the idea of cell decomposition. **Cell decomposition techniques** decompose the traversable space in the continuous environment into cells. Each cell, which is typically defined by a circle or convex polygon, represents a region of traversable space without obstacles. Because cells are circles or convex polygons that do not contain obstacles, agents can travel in a straight line between any two coordinates within the same cell (without path planning). Cell decomposition techniques define a graph $G = (V, E)$ where V is a set vertices such that each vertex maps to both a coordinate in the continuous environment and one or more cells and E is a set of edges connecting adjacent vertices in V that have line-of-sight. Two vertices are adjacent to one another if the cells that they map to share all or part of a boundary.

In other words, cell decomposition techniques, unlike skeletonization techniques, are composed of two distinct parts:

- The cell decomposition, which is a discretization of the continuous environment that represents regions of traversable space in the continuous environment. The cell decomposition acts as an intermediate representation of the traversable space in the continuous environment in that it does not contain as much information about the traversable space in the continuous environment as the raw geometry, but contains more information about the traversable space in the continuous environment than a visibility graph or waypoint graph. Cell decompositions can be used for both path planning and as a simple data structure which supports other parts of a path-planning system. For example, information can be associated with the region of the continuous environment that corresponds to a cell, such as the amount of gold hidden in the region or a rendering of the region when displaying the terrain. A more detailed discussion of these properties can be found in Chapter 3.

- A graph, which is constructed from the cell decomposition rather than the raw geometry of the continuous environment and used by the find-path algorithm to search for paths.

It is for this reason that we have been making distinctions between regular grids and grid graphs and NavMeshes and nav graphs. The former represents the cell decomposition and the latter represents the graph constructed from the cell decomposition, respectively.

We examine three cell decomposition techniques, namely regular grids, circle based waypoints and NavMeshes. We provide an example of each generate-graph technique in Figures 2.6, 2.7 and 2.8. The blue regions depict regions in the continuous environment that are traversable but are not represented by a region of traversable space in the cell decomposition. We evaluate each cell decomposition technique by the tradeoff that it provides with respect to the following four criteria:

- The amount of traversable space in the continuous environment that is not represented by some region of traversable space in the cell decomposition. This is important because agents should be able to traverse all of the traversable space in the continuous environment, and if they cannot then the paths that agents follow can appear long and unrealistic looking (called **Traversable Space** in Table 2.1).

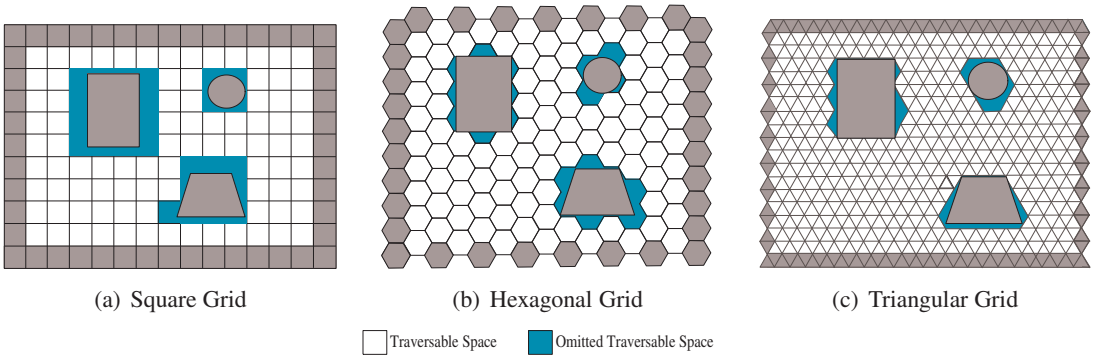
- The amount of memory each cell decomposition technique requires. The generate-graph problem is one of many different problems that must be addressed by robotics and video game applications. The solutions to each of these different problems require memory and thus roboticists and video game developers prefer cell decomposition techniques that use memory efficiently (called **Memory** in Table 2.1).

- The implementation difficulty of the cell decomposition technique. For example, circles are simpler to define than a set of arbitrary m-sided convex polygons. Cell decomposition techniques that are simple are desirable for the same reasons we specified when discussing the Simplicity Property in Chapter 1 (called **Difficulty** in Table 2.1).

- Whether or not each cell decomposition technique can be used to discretize known 2D environments and known 3D environments (called **3D** in Table 2.1).

Table 2.1 summarizes our analysis of cell decomposition techniques. In the columns labeled Traversable Space, Memory and Difficulty more $+$'s indicate that the cell decomposition technique represents more traversable space, that the cell decomposition technique uses a smaller amount of memory and that the cell decomposition technique is simpler to implement, respectively. In the column labeled 3D a checkmark indicates that the cell decomposition technique can be used when path planning in known 3D environments. To summarize, more $+$'s always indicate that the cell decomposition technique is "better." Finally, in addition to evaluating each cell decomposition technique using the aforementioned four criteria, we highlight desirable properties that are unique to regular grids since they are the focus of this dissertation.

- **Regular Grids:** Regular grids are widely used by roboticists (for example, the Mars rovers Spirit and Opportunity (Carsten et al., 2009)) and video game developers (for example, the video games Dawn of War 1 and 2, Civilization V and Company of Heroes (Champandard, 2010)). We begin by discussing how regular grids can be used to discretize known 2D environments. Regular grids discretize continuous environments into tessellations of regular



(a) Square Grid (b) Hexagonal Grid (c) Triangular Grid

Figure 2.6: Regular Grids

polygons (that is, equilateral and equiangular polygons). Only three types of regular polygons can be used to tessellate continuous 2D environments, namely squares, hexagons and triangles (Figures 2.6(a)-2.6(c), respectively) (Patel, 2000; Deloura, 2000; Murphy, 2000; Tozour, 2004; Lengyel, Reichert, Donald, & Greenberg, 1990; Chrpa & Komenda, 2011; Chrpa, 2011). Throughout this dissertation we refer to these regular polygons as grid cells. In this section, we focus on square grids because they are the simplest type of regular grid to implement.[3] Every grid cell that contains part of an obstacle is labeled blocked, and all other grid cells are labeled unblocked. A graph is constructed by placing vertices in either the centers or corners of grid cells. A vertex maps to a grid cell c if it is in the center or upper right corner of c, respectively. Edges are added between all pairs of adjacent vertices that have line-of-sight. For regular grids, there are two commonly used definitions of adjacent vertices: **(1)** two vertices are adjacent iff the grid cells that they map to share a side (in which case a vertex can be adjacent with up to four vertices) or **(2)** two vertices are adjacent iff the grid cells that they map to share either a corner or a side (in which case a vertex can be adjacent with up to eight vertices). This chapter assumes the latter definition. The start and goal vertices, are the vertices that are closest to the start and goal coordinates, respectively. We refer to graphs constructed from regular grids as grid graphs. If a vertex can be adjacent with up to n vertices then we refer to it as an n-neighbor grid graph. Therefore, Figures 1.1(c) and 2.2(d) depict 8-neighbor square grid graphs with vertices placed in the corners of grid cells.

Regular grids, especially square grids, are the simplest cell decomposition technique to define and implement because the grids are implicitly defined by the length of a side of a grid cell and a reference coordinate. However, regular grids suffer from **digitization bias** in that grid cells that contain only a small portion of an obstacle must be labeled blocked in order to ensure that agents can travel in a straight line between any two coordinates in a grid cell and, as a result, agents are artificially prevented from traversing some regions of traversable space in the continuous environment (Murphy, 2000). It is important to minimize digitization bias in order to ensure that if there is a path between two coordinates in the continuous environment then there is also a path on the grid graph. Digitization bias can be reduced by decreasing the size of the grid cells, but this increases the memory

[3] A detailed description of triangular and hexagonal grids can be found in Chapter 3.

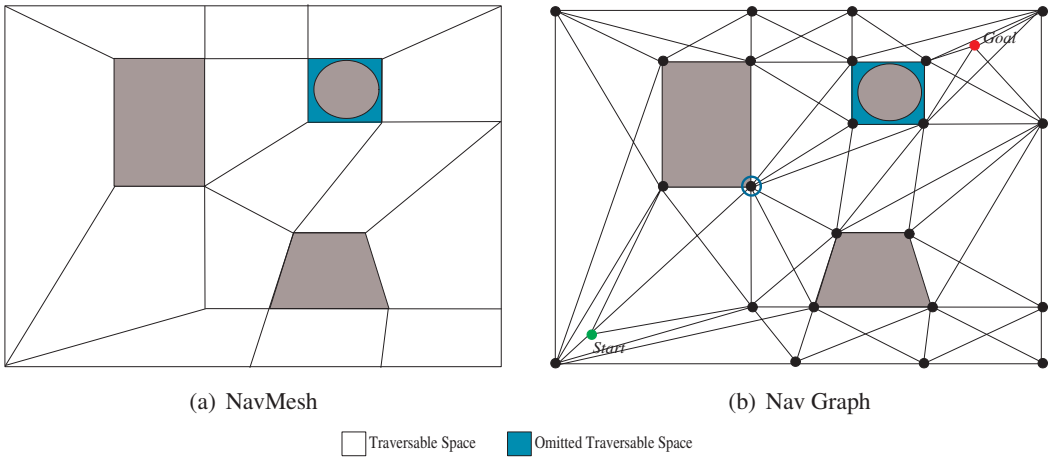


Figure 2.7: NavMesh

required to store the regular grid and makes searching the resulting grid graph slower. Typically, one needs to use grid cells of small sizes and thus the memory required to store the regular grid and the resulting grid graph is large.

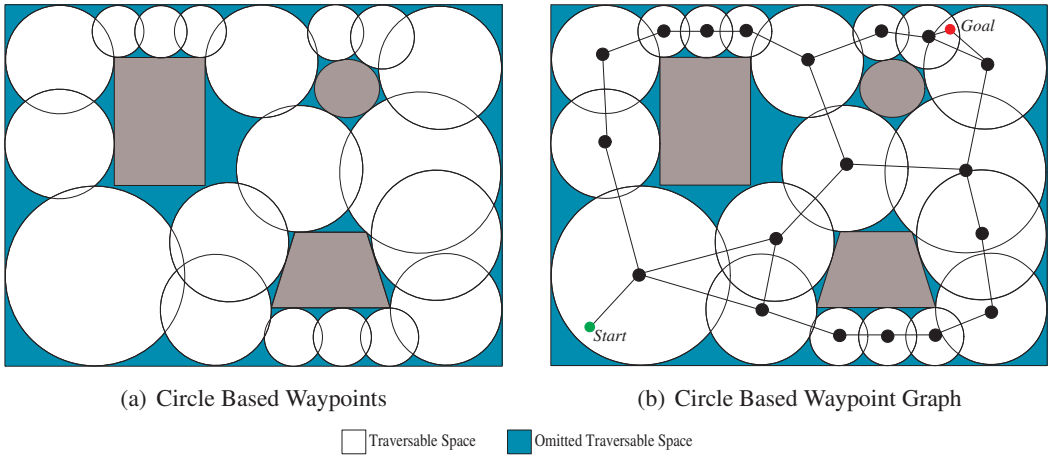
Regular grids also have several unique properties: **(1)** Regular grids support random access lookup. It is possible to determine the grid cell and any information associated with it at any given coordinate in constant time (Tozour, 2004). This is useful to roboticists and video game developers because a grid cell may hold information about the animation to display when traversing that grid cell, the cost to traverse that grid cell, or the amount of gold that resides in that grid cell. **(2)** Regular grids can implement post-processing techniques more efficiently than any other cell decomposition technique. Determining whether or not two vertices have line-of-sight on a square grid is similar to determining which points to plot on a raster display when drawing a straight line between two points because the plotted points correspond to grid cells that the straight line passes through. Thus, two vertices have line-of-sight iff none of the plotted points correspond to blocked grid cells. This allows roboticists and video game developers to perform the line-of-sight checks that are often required by post-processing techniques using only fast logical and integer computations. **(3)** The grid cells in regular grids are rigorously defined, which allows for the determination of upper bounds on the ratios between the lengths of grid paths and the lengths of true shortest paths that apply to any grid graph constructed from a regular grid.

Regular grids can be used to discretize known 3D environments with the only change being that the space is tessellated with regular polyhedrons instead of regular polygons. Only one type of regular polyhedron can be used to tessellate a continuous 3D environment, namely a cube (Gosset, 1900). Hexagons and triangles with a height parameter, that is prismatic hexagons and triangles, can be used to tessellate a continuous 3D environment as well, but they are not regular polyhedrons. Cubic grids maintain all of the desirable properties of square grids and thus are widely used by roboticists and video games developers when path planning in known 3D environments (Nash et al., 2009; Carsten et al., 2006; Cohen et al., 2011).

- **NavMeshes:** NavMeshes are widely used by video game developers (for example, the video games Halo 2, Counter-Strike: Source and Metroid Prime (Tozour, 2008)) and **trapezoidal decompositions** (Chazelle & Dobkin, 1979), which are similar in spirit to NavMeshes, are widely used by roboticists. We begin by discussing how NavMeshes can be used to discretize known 2D environments. A NavMesh discretizes a continuous 2D environment into a set of adjacent m-sided convex polygons (Figure 2.7(a)) (Snook, 2000; Tozour, 2002; OŃeil, 2004; Rabin, 2006, 2008). The polygons in a NavMesh are typically generated by hand and/or with the help of middleware programs such as Xaitment (http://www.xaitment.com/), NavPower (http://www.navpower.com/), PathEngine (http://www.pathengine.com/) or Kynapse (http://usa.autodesk.com/). A graph is constructed by placing vertices at the start and goal coordinates and in the corners, centers or on the sides of a polygon. A vertex maps to a polygon p if it is in the center, one of the corners of p, or one of the sides of p, respectively. Edges are added between all pairs of adjacent vertices that have line-of-sight. For NavMeshes, like regular grids, there are two commonly used definitions of adjacent vertices: **(1)** two vertices are adjacent iff the polygons that they map to share a side or **(2)** two vertices are adjacent iff the polygons that they map to share either a corner or a side. This chapter assumes the latter definition.[4] We refer to graphs constructed from NavMeshes as nav graphs. Unlike a grid cell in a regular grid, a polygon in a NavMesh can share a corner with an arbitrary number of other polygons. Thus, under the second definition of adjacency there is not necessarily a relationship between the value on m and the maximum number of vertices that a vertex can be adjacent to. For example, in Figure 2.7(b), the circled vertex is adjacent to 9 vertices even though each polygon in the NavMesh has 4 sides. However, for simplicity, NavMeshes are often constructed such that there is a relationship between m and the maximum number of vertices that a vertex can be adjacent to. If this relationship exists then we use the same terminology that we use for grid graphs: if a vertex can be adjacent with up to n vertices then we refer to it as an n-neighbor nav graph. For example, Figure 1.2(c) depicts an 8-neighbor nav graph with vertices placed in the corners of the 4-sided polygons that were used to discretize a continuous environment into a NavMesh.

  NavMeshes, more so than other cell decomposition techniques, can provide a nearly complete representation of the traversable space in the continuous environment. In Figure 2.7(a), one can see that the NavMesh represents almost all of the traversable space in the continuous environment. This follows from the fact that a NavMesh can be defined by a set of arbitrary m-sided convex polygons. However, this flexibility has drawbacks as well: **(1)** Because the m-sided convex polygons can be defined arbitrarily, determining the value of m and defining the set of polygons can be time consuming even with the help of middleware programs (Tozour, 2010). **(2)** Because the m-sided convex polygons can be defined arbitrarily, the coordinates of the vertices that define the polygons can require single or even double precision floating point numbers. This makes all computations that involve the polygons more mathematically involved than those involving the grid cells in a regular grid (Latombe, 1991). For example, determining whether or not two vertices on a nav graph have line-of-sight is typically slower than determining whether or not two

[4] Typically there are fewer edges connecting the start and goal vertices to their adjacent vertices. In some cases, an edge is only added between the start and goal vertices and the vertices they are closest to.



(a) Circle Based Waypoints (b) Circle Based Waypoint Graph

Figure 2.8: Circle Based Waypoints

vertices on a grid graph have line-of-sight (Champandard, 2010). **(3)** Because the m-sided convex polygons can be defined arbitrarily, NavMeshes are not rigorously defined, which makes it difficult to determine upper bounds on the ratios between the lengths of edge-constrained paths formed by the edges of nav graphs and the lengths of true shortest paths that apply to any nav graph constructed from any NavMesh. In general, these drawbacks make algorithms that discretize continuous environments into NavMeshes more difficult to implement than algorithms that discretize continuous environments into regular grids (Latombe, 1991). NavMeshes can use memory more efficiently than regular grids because they can use large polygons to represent large regions of traversable space and smaller polygons near obstacles.

NavMeshes were not designed to be used to discretize known 3D environments. However, NavMeshes are extensible to path planning in 2.5 dimensions, namely environments in which the agent only has freedom of movement in two dimensions and the terrain introduces movement in the third dimension (for example, stairs, hills or bridges), without any changes. This is because a polygon represents traversable space and thus it does not matter whether or not a given polygon introduces movement in a third dimension (for example, a polygon that represents traversable space that is flat can be adjacent to a polygon that represents traversable space that is at an incline or decline). Furthermore, edges can be added to or removed from a nav graph to control whether or not an agent can move from a polygon that is flat to a polygon that introduces movement in the third dimension. NavMeshes can also handle more difficult cases in continuous 2.5D environments, such as the case in which one region of traversable space is underneath another (for example, a road that passes under a bridge) by adding a height parameter that indicates the amount of vertical clearance between the two regions of traversable space.

- **Circle Based Waypoints:** Circle based waypoints are widely used by video game developers (for example, the video game MechWarrior 4: Vengeance (Tozour, 2010)). We begin by discussing how circle based waypoints can be used to discretize known 2D environments. Circle based waypoints discretize a continuous 2D environment into a set of

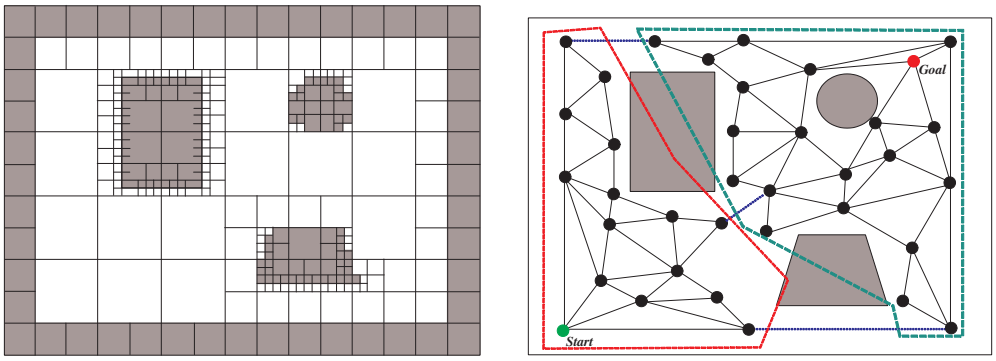
overlapping circles (Figure 2.8(a)). Circles are typically placed by hand. The graph is constructed by placing vertices at the start and goal coordinates and in the centers of circles. A vertex maps to a circle l if it is in the center of l. Edges are added between all pairs of adjacent vertices. For circle based waypoint graphs, two vertices are adjacent if the two circles that they map to overlap. An edge is also added between the start and goal vertices and every vertex that is the center of a circle within which the start and goal vertices reside (Figure 2.8(b)). We refer to graphs constructed from circle based waypoints as circle based waypoint graphs.

Circle based waypoints are simpler to implement than NavMeshes, but more difficult to implement than a regular grid. This follows from the fact that a circle is easier to define than an arbitrary m-sided polygon, but more difficult to define than a regular grid which is implicitly defined by a reference coordinate and the length of a side of a grid cell. Circle based waypoints do not represent as much of the traversable space as regular grids or NavMeshes because circle based waypoints can only represent traversable space with circles and, in most continuous environments, obstacles are polygonal. Circle based waypoints are more memory efficient than regular grids because they can use large circles to represent large regions of traversable space and lots of smaller circles in hallways/corridors/bridges and more memory efficient than NavMeshes because a circle can be defined with less memory than an arbitrary m-sided convex polygon.

Circle based waypoints were not designed to be used to discretize known 3D environments. However, circle based waypoints, like NavMeshes, are extensible to known 2.5D environments without any changes because a circle represents traversable space and thus it does not matter whether or not a circle introduces movement in the third dimension (for example, a circle that represents traversable space that is flat can overlap with a circle that represents traversable space that is at an incline or decline). Furthermore, edges can be added or removed from circle based waypoint graphs to control whether or not an agent can move from a circle that is flat to a circle that introduces movement in the third dimension. Circle based waypoints can also add a height parameter to each circle (cylinder based waypoints), in order to handle the more difficult cases in which one region of traversable space (that is, a road) is underneath another (that is, a bridge).

Cell decomposition techniques address many of the shortcomings of skeletization techniques: **(1)** Post processing can be performed much more efficiently because line-of-sight checks can be performed using the cell decomposition as opposed to the raw geometry of the continuous environment. **(2)** It is easier to determine whether or not there is an edge between two vertices. The edges in a graph constructed from a cell decomposition can be inferred with very little computation. For example, in an 8-neighbor square grid graph with vertices placed in the centers of grid cells an edge exists between a vertex and each of its 8 adjacent vertices that map to unblocked grid cells and therefore the existence of an edge can be determined simply by checking whether or not a grid cell is blocked. As a result, some path-planning systems only maintain the cell decomposition in memory and construct the graph from the cell decomposition while the find-path algorithm is searching for a path. Such an approach requires less memory than storing both the cell decomposition and the graph, but can make solving the find-path algorithm slow.

Cell decomposition techniques have drawbacks as well. They can be more difficult to implement and use more memory than skeletization techniques because the cell decomposition must



(a) Quadtree (b) Split Waypoint Graph

Figure 2.9: Hierarchical Generate-Graph Techniques

be defined and stored in memory. For large intricate continuous environments, constructing the cell decomposition can be time consuming even with the help of middleware solutions. Finally, our descriptions of many of the cell decomposition techniques is notional and should not be interpreted as a set of rigid rules. For example, there is nothing that prevents one from defining a nav graph in which vertices are placed in the corners, on the sides *and* in centers of a polygon.

2.2.1.3 Hierarchical Generate-Graph Techniques

The memory limitations of waypoint graphs and regular grids can be mitigated by using hierarchical techniques. Hierarchical techniques allow the continuous environment to be discretized with a finer granularity in regions where a more accurate representation of the traversable space in the continuous environment is necessary (for example, near obstacles) and for the continuous environment to be discretized with a coarser granularity in regions where a more accurate representation of the traversable space in the continuous environment is not necessary (for example, in wide open spaces). Split waypoint graphs provide an example of applying the idea of hierarchical discretization to a skeletonization technique, namely waypoint graphs and quadtrees provide an example of applying the idea of hierarchical discretization to a cell decomposition technique, namely square grids.

- **Quadtrees:** Quadtrees (Samet, 1982, 1988) are constructed by recursively dividing the grid cells in a square grid into four equally sized grid cells until either a grid cell contains no obstacles or the size of a grid cell is the smallest allowable. A grid graph can then be constructed from the quadtree using the same technique used to construct a grid graph from a square grid. Quadtrees are useful in environments with a small number of obstacles because smaller grid cells are used only in the regions near obstacles. This results in fewer grids cells, which in turn reduces the amount of memory required, the size of the grid graph and the time it takes to search the grid graph. In environments with a large number of obstacles more of the continuous environment requires smaller grid cells. This results in a larger number of grid cells which in turn increases the amount of memory required, the size of the grid graph and the time it takes to search the grid graph. In Figure 2.9(a), the environment from Figure 2.3(a) has been discretized into a quadtree. The key ideas behind

quadtrees can be applied to other cell decomposition techniques as well. For example, one can construct rectangular NavMeshes from a square grid using a technique that is very similar to the one used to construct quadtrees (Board & Ducker, 2002).

- **Split Waypoint Graphs:** Split waypoint graphs are constructed by recursively dividing a waypoint graph into smaller subgraphs that represent regions or traversable space which are naturally or geometrically related to one other. Figure 2.9(b) depicts a simple example in which the waypoint graph in Figure 2.5(b) has been divided into two low level subgraphs G' (dashed red) and G'' (dashed green) which are then connected to one another at a higher level (dotted blue edges). Split waypoint graphs can reduce the number of edges in waypoint graphs because edges exist between a vertex and k of its adjacent vertices within a lower level subgraph, but the higher level graphs are connected with only a sparse collection of edges. This reduces the amount of memory required, the size of the waypoint graph and the time it takes to search the waypoint graph. Paths between two vertices in the same subgraph are the same as the paths in waypoint graphs. However, paths between two vertices in different subgraphs can be longer and less realistic looking than the paths found on waypoint graphs because there are only a few places where paths can pass between subgraphs.

2.2.1.4 The Generate-Graph Problem in Unknown 2D Environments

When an agent performs path planning in an unknown 2D environment it has incomplete knowledge of the traversable space in the continuous environment. Typically, an agent begins with some knowledge of the traversable space in the continuous environment and makes an assumption about the traversability of the regions of the traversable space in the continuous environment that are unknown to it. This dissertation makes the freespace assumption, which is widely used by roboticists (Stentz & Hebert, 1995; Hebert et al., 1999) and thus the agent assumes that any region of traversable space in the continuous environment that it does not know about is traversable. The agent performs path planning, that is, solves the generate-graph and find-path problems, given its current knowledge of the traversable space in the continuous environment and the freespace assumption. The agent's knowledge of the traversable space in the continuous environment increases as the agent follows its current path toward the goal coordinate. If the agent finds its current path to be blocked as it follows its current path toward the goal coordinate then it has to perform path planning again, that is, solve a new instance of the generate-graph and find-path problems. This process is repeated until the agent either reaches the goal coordinate or determines that there is no path from its current coordinate to the goal coordinate. In this section, we examine the generate-graph problem.

The agent can solve each instance of the generate-graph problem by applying any of the generate-graph techniques that we have already introduced from scratch (that is, to the entire continuous environment) each time it finds its current path to be blocked. However, an agent's knowledge of the traversable space in the continuous environment typically does not increase significantly each time it finds its current path to be blocked because the range that an agent can sense from its current coordinate is usually small relative to the size of the continuous environment. Therefore, the agent's knowledge of the traversable space in the continuous environment before it finds its current path to be blocked is similar to the agent's knowledge of the traversable space in the continuous environment after it finds its current path to be blocked (Koenig et al.,

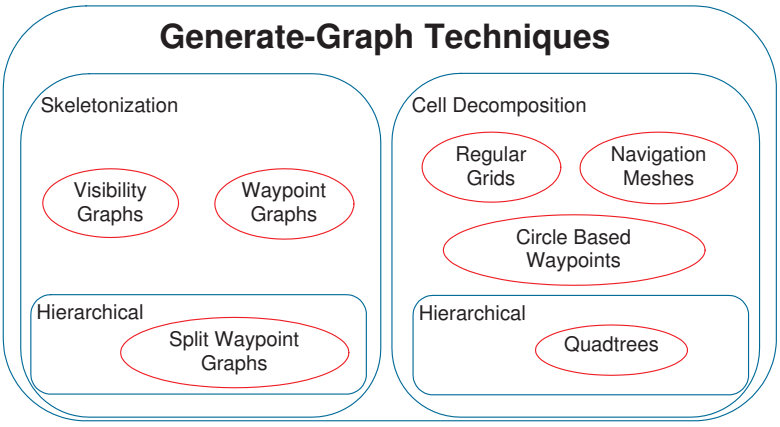


Figure 2.10: Classification of Generate-Graph Techniques

2004). As a result, one would expect the graph constructed from the continuous environment before the agent finds its current path to be blocked to be similar to the graph constructed from the continuous environment after the agent finds its current path to be blocked.

Therefore, rather than solving the generate-graph problem from scratch each time the agent finds its current path to be blocked, it can simply update its current graph. However, updating the current graph is more efficient for some generate-graph techniques than it is for others. For example, updating a visibility graph requires the following steps: since the agent has moved we remove the old start vertex and all edges that have the start vertex as an endpoint, remove each edge from the current graph that no longer has line-of-sight due to the newly found obstacle, add a new vertex to the graph for each corner of each newly found polygonal obstacle, add a start vertex at the current coordinate of the agent and, finally, add new edges between each new vertex and every other vertex in the graph between which there is line-of-sight. On the other hand, updating an 8-neighbor square grid graph can be done using much simpler steps: label any grid cells that contain part of a newly found obstacle as blocked, label the vertex that is closest to the agent's current coordinate as the start vertex and, finally, remove edges that connect vertices that no longer have line-of-sight due to the newly blocked cell. For example, if vertices are mapped to the corners of grid cells then each edge that needs to be removed has one endpoint that is also a corner of a newly blocked cell. This simplicity is one of the reasons that 8-neighbor square grid graphs are widely used when path planning in unknown 2D environments (Koenig & Likhachev, 2002a; Ferguson & Stentz, 2006; Nash et al., 2009) and one of the reasons that this dissertation uses 8-neighbor square grid graphs when path planning in unknown 2D environments.

A summary of our classification of different generate-graph techniques is depicted in Figure 2.10. The rounded rectangles represent different classes of generate-graph techniques and ovals represent generate-graph techniques. If an oval is within a rounded rectangle then it is a member of that class. For example, regular grids are a **1** cell decomposition **2** generate-graph technique. The rounded rectangles and ovals in Figure 2.10 were introduced in this section as follows: First, we separated different generate-graph techniques into two classes, namely skeletonization techniques (Section 2.2.1.1) and cell decomposition techniques (Section 2.2.1.2), each of which is

represented by a rounded rectangle in Figure 2.10. We then introduced several different generate-graph techniques from both classes, each of which is represented by an oval in Figure 2.10. Finally, we introduced classes of hierarchical generate-graph techniques (Section 2.2.1.3), each of which is represented by a rounded rectangle in Figure 2.10, as well as hierarchical generate-graph techniques, each of which is represented by an oval in Figure 2.10.

In this section, we introduced some of the tradeoffs that roboticists and video game developers consider when choosing a generate-graph technique to discretize a continuous environment into a graph. Given these tradeoffs it is clear that there is no single generate-graph technique that dominates all others for every path-planning application (Murphy, 2000). The requirements of a particular path-planning application determine which generate-graph technique is best and thus it is extremely desirable that solutions to the find-path problem satisfy the Generality Property and can be used to search any Euclidean graph. P. Tozour provides a very concise description of this dilemma: "There is no single correct way to handle pathfinding and navigation for [path planning]. It is critical to understand the benefits and drawbacks of all of the available approaches when selecting a [generate-graph technique]. The best representation ultimately depends on the layout of the [continuous environment] and design of your [applications path-planning systems], and the memory and performance requirements of the target platform (Tozour, 2004)." Finally, we highlighted the desirable properties of regular grids, which are the generate-graph technique that this dissertation focuses on for Contributions 1-4.

	Simplicity		Efficiency	Generality
	Simple to Understand	**Simple to Implement**	**Runtime/Path Length**	**Euclidean Graph Independent**
Dijkstra's algorithm	✓$^{++++}$	✓$^{+++}$	✓/✓	✓
A*	✓$^{+++}$	✓$^{+++}$	✓$^{+}$/✓	✓

Single-Shot Find-Path Algorithms

Differential A*	✓$^{++}$	✓$^{+}$	✓$^{+}$/✓	✓
D* Lite	✓$^{+}$	✓$^{++}$	✓$^{+}$/✓	✓

Incremental Find-Path Algorithms

Field D*/ 3D Field D*	✓	✓	✓/✓$^{+}$	

Incremental Any-Angle Find-Path Algorithms

Table 2.2: Find-Path Algorithms

2.2.2 The Find-Path Problem

In this section, we provide an overview of traditional edge-constrained find-path algorithms that are widely used by roboticists and video game developers. We introduce several traditional edge-constrained find-path algorithms that are able to find shortest edge-constrained paths from a given start vertex to a given goal vertex. In this section, for brevity, we use the term path in place of edge-constrained path because all traditional edge-constrained find-path algorithms find edge-constrained paths. Each traditional edge-constrained find-path algorithm that we introduce can be classified as either a single-shot find-path algorithm, which is used to find paths when path planning in known 2D environments and known 3D environments, or an incremental find-path algorithm, which is used to find paths when path planning in unknown 2D environments. We define each class and introduce examples in Sections 2.2.2.1 and 2.2.2.2, respectively. In Section 2.2.3, we demonstrate that post processing the paths found by traditional edge-constrained find-path algorithms is often unable to improve paths. We then introduce the only existing find-path algorithm that can be classified as any-angle in Section 2.2.4. Each find-path algorithm is evaluated based on the tradeoff it provides with respect to the Simplicity, Efficiency and Generality Properties introduced in Chapter 1: Simplicity Property: They are simple to implement and understand. Efficiency Property: They provide a good tradeoff with respect to the runtime of the search and the length of the resulting path. Generality Property: They can be used to search any Euclidean graph.

Table 2.2 summarizes our analysis of traditional edge-constrained find-path algorithms. For the column labeled **Simplicity**, more $+$'s indicate that the find-path algorithm is either easier to understand or easier to implement. The entries in these columns are absolute because all of the find-path algorithms can be fairly evaluated against one another. In the column labeled **Efficiency**, we use a tuple, namely <runtime/path length> and more $+$'s indicate that the find-path algorithm has a shorter runtime and finds shorter paths, respectively. The path length entries in this column are absolute because all of the algorithms can be fairly evaluated against one another. The runtime entries in this column are not absolute. Incremental find-path algorithms cannot be evaluated fairly against single-shot find-path algorithms because they are designed to solve different problems. Therefore, the runtime entries compare incremental find-path algorithms against one another and single-shot find-path algorithms against one another. For the column labeled **Generality**, a check mark indicates that the algorithm can be used to search any Euclidean graph. To summarize, more $+$'s always indicate that the find-path algorithm is "better."

The find-path algorithm descriptions provided in this section are correct, but they are intended to be general algorithm descriptions that provide intuition as to how the find-path algorithms operate and how the find-path algorithms are different. These descriptions should not be used as a basis for an implementation. In subsequent chapters more detailed descriptions and pseudocode are provided for the find-path algorithms that this dissertation builds on.

2.2.2.1 Single-Shot Find-Path Algorithms

When path planning in known 2D or 3D environments the traversable space in the continuous environment never changes and therefore, once a continuous environment has been discretized into a graph, the graph never changes and therefore a path in the continuous environment can be found with a single search. Therefore, we refer to find-path algorithms that are used when path planning in known 2D and 3D environments as "single-shot" find-path algorithms. Single-shot

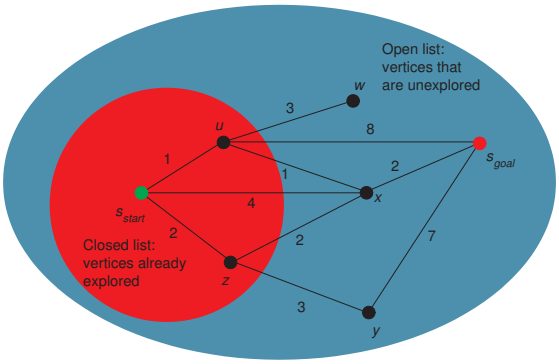


Figure 2.11: Snapshot of Dijkstra's Algorithm

find-path algorithms find a shortest edge-constrained path in a (static) graph $G = (V, E)$ between a given start vertex $s_{start} \in V$ and a given goal vertex $s_{goal} \in V$. This problem is difficult because there are typically a large number of paths between the start vertex and the goal vertex, only a small fraction of which are shortest paths. Therefore, single-shot find-path algorithms need to efficiently search through a large number of paths for a shortest path. We examine two single-shot find-path algorithms, namely Dijkstra's algorithm, an uniformed find-path algorithm, and A*, an informed find-path algorithm.

- **Dijkstra's Algorithm: An Uninformed Find-Path Algorithm.** Dijkstra's algorithm (Dijkstra, 1959) is a search algorithm for finding a shortest path on a graph $G = (V, E)$ from a given start vertex to a given goal vertex when no information other than the graph is given. All of the find-path algorithms that we discuss in this section build on Dijkstra's algorithm.

  Dijkstra's algorithm maintains two global data structures:

  - The closed list (*closed*) is a set which contains vertices. We say that the vertices in the closed list are in the "explored" part of the graph.
  - The open list (*open*) is a set which contains vertices that are in V but not in the closed list, that is, $open = V - closed$. We say that the vertices in the open list are in the "unexplored" part of the graph.

  Dijkstra's algorithm maintains two values for every vertex s:

  - The g-value $g(s)$ is the length of the shortest path from the start vertex to vertex s found so far.
  - The parent $parent(s)$ is used to extract a path from the start vertex to the goal vertex after Dijkstra's algorithm terminates. At any point during the search, the parents form a **search tree** rooted at the start vertex.

  Dijkstra's algorithm maintains the following property:

  - **Closed List Property:** When a vertex s is added to the closed list for the first time, a shortest path from the start vertex to vertex s has been found.

Dijkstra's algorithm begins searching the graph by setting $closed = \emptyset$, $g(s) = \infty$ and $parent(s) = NULL$ for every vertex s in V, $g(s_{start}) = 0$, $parent(s_{start}) = s_{start}$ and adding the start vertex to the closed list. Next, Dijkstra's algorithm performs the **expansion process:** for every vertex s' in the open list, the algorithm computes the lengths of shortest paths from the start vertex through a sequence of vertices in the closed list to some vertex s in the closed list followed by a single edge $(s, s') \in E$. That is, the algorithm computes the following **expansion equation:** $g'(s') = min_{(s,s'):s \in closed}\, g(s) + c(s, s')$ where $c(s, s')$ is the length of the edge (s, s'). The vertex s' for which this quantity is minimized is **expanded**, that is, it is removed from the "unexplored" part of the graph (that is, the open list) and added to the "explored" part of the graph (that is, the closed list), the g-value of vertex s' is set to $g'(s')$ and the parent of vertex s' is set to vertex s. Figure 2.11 shows a snapshot of Dijkstra's algorithm where $V = \{s_{start}, u, z, w, x, y, s_{goal}\}$, edges are labeled with their lengths, $closed = \{s_{start}, u, z\}$, $open = \{w, x, y, s_{goal}\}$, the g-values of the vertices in V are $g(s_{start}) = 0$, $g(u) = 1$, $g(z) = 2, g(w) = \infty, g(x) = \infty, g(y) = \infty, g(s_{goal}) = \infty$, the parents of the vertices in V are $parent(s_{start}) = s_{start}$, $parent(u) = s_{start}$, $parent(z) = s_{start}$, $parent(w) = NULL$, $parent(x) = NULL$, $parent(y) = NULL$, $parent(s_{goal}) = NULL$ and the next vertex expanded is vertex x due to the path from the start vertex to vertex u and from vertex u to vertex x (and thus the parent of vertex x is vertex u and the g-value of vertex x is 2). The expansion process is repeated until either the goal vertex is expanded, in which case Dijkstra's algorithm returns that a path from the start vertex to the goal vertex exists, or the open list is empty in which case Dijkstra's algorithm returns that a path from the start vertex to the goal vertex does not exist. If Dijkstra's algorithm returns that a path from the start vertex to the goal vertex exists then a path from the start vertex to the goal vertex in reverse, can be retrieved using **path extraction**, that is, by following parents starting at the goal vertex and finishing at the start vertex. Dijkstra's algorithm is **correct**, that is, if it returns that a path exists from the start vertex to the goal vertex then path extraction retrieves a path from the start vertex to the goal vertex in reverse. Dijkstra's algorithm is **complete**, that is, it returns that there is a path from the start vertex to the goal vertex if one exists and returns that there is not a path from the start vertex to the goal vertex if one does not exist. Dijkstra's algorithm is **optimal**, that is, if it returns that there is a path from the start vertex to the goal vertex then path extraction retrieves a shortest path from the start vertex to the goal vertex in reverse.

Dijkstra's algorithm is simple to implement and understand. The pseudocode for Dijkstra's algorithm is ≈ 10 lines and it is taught in many introductory undergraduate algorithms courses (for example, Dijkstra's algorithm is covered in both Kleinberg and Tardos (2005) and Cormen, Stein, Rivest, and Leiserson (2001), two of the most widely used text books in introductory algorithms courses). The only requirement for Dijkstra's algorithm to be correct, complete and optimal is that the lengths of all edges in the graph are non-negative and thus it can be used to search any Euclidean graph.

- **A*: An Informed Find-Path Algorithm.** A* (Hart et al., 1968) is a simple variant of Dijkstra's algorithm that is designed to reduce the runtime of Dijkstra's algorithm without an increase in the length of the resulting path. It does this by using information other than just the graph to help focus the search so that the goal vertex can be found with a shorter runtime. A* maintains a user provided h-value to give it an additional clue as to which

areas of the graph are more likely to contain a shortest path from the start vertex to the goal vertex. This h-value $h(s)$ is an estimate of the length of a shortest path from vertex s to the goal vertex. A* uses the h-value to calculate an f-value. The f-value $f(s) = g(s) + h(s)$ is an estimate of the length of a shortest path from the start vertex via vertex s to the goal vertex. Therefore, unlike Dijkstra's algorithm, which only has information about the length of a path from the start vertex to a vertex (that is, its g-value), A* has information about the length of a path from the start vertex to a vertex (that is, its g-value) and the length of the path from that vertex to the goal vertex (that is, its h-value). If the h-value of every vertex is zero then A* is identical to Dijkstra's algorithm.

A* is identical to Dijkstra's algorithm except that the expansion equation that it computes during the expansion process is different. Unlike Dijkstra's algorithm, which chooses the vertex with the smallest g-value, A* chooses the vertex with the smallest f-value. A* performs the following expansion process: Like Dijkstra's algorithm, for every vertex s' in the open list, A* computes the lengths of shortest paths from the start vertex through a sequence of vertices in the closed list to some vertex s in the closed list followed by a single edge $(s, s') \in E$. However, A* also computes the h-value of vertex s' and adds that to the g-value of vertex s'. That is, A* computes the following expansion equation: $f'(s') = min_{(s,s'):s \in \mathit{closed}}\, g(s) + c(s, s') + h(s')$. The vertex s' in the open list for which this quantity is minimized is expanded, just like Dijkstra's algorithm would do.

As long as the h-values provide accurate estimates of the lengths of shortest paths from a vertex to the goal vertex, the expansion equation computed by A* is more informed than the one computed by Dijkstra's algorithm. Thus, an A* search is able to avoid examining a larger number of paths in the graph than a Dijkstra search, which often results in shorter runtimes. There are two commonly used classes of h-values, namely admissible and consistent. The h-values of an A* search are **admissible** if they never overestimate the length of the shortest path from a vertex to the goal vertex. The h-values of an A* search are **consistent** if they satisfy the triangle inequality, that is, the h-value of the goal vertex is zero and for every vertex s and vertex s' such that $(s, s') \in E$, the h-value of vertex s is no greater than the length of the edge from vertex s to vertex s' plus the h-value of vertex s' (Dechter & Pearl, 1985). In other words, $h(s) \leq c(s, s') + h(s')$. Consistency is a stronger property than admissibility and thus consistent h-values are admissible, but admissible h-values are not necessarily consistent. On Euclidean graphs the straight-line Euclidean distance, is both admissible and consistent. The vertices in the closed list have different properties for an A* search with consistent h-values than they do for an A* search with admissible h-values. If h-values are consistent then A* maintains the Closed List Property. If h-values are admissible then the Closed List Property no longer holds for all vertices, but is guaranteed to hold for the goal vertex (when it is added to the closed list for the first time). An A* search with admissible h-values, unlike an A* search with consistent h-values, can iterate on the same vertex more than once during the expansion process.[5] This dissertation focuses on informed searches with consistent h-values.

[5]This implies that an A* search with consistent h-values might have a slightly different implementation than an A* search with admissible h-values.

A* is just as simple to implement as Dijkstra's algorithm and is only slightly more difficult to understand. The pseudocode for A* is about the same length as that of Dijkstra's algorithm and it is taught in many introductory undergraduate artificial intelligence, robotics and video game courses (for example, A* is covered in Russel and Norvig (1995), one of the most widely used text books in artificial intelligence courses, Choset et al. (2005), one the most widely used text books in robotics courses and Deloura (2000), one the most widely used text books in video games courses). The simplicity and efficiency of A* make it the find-path algorithm of choice when path planning in known 2D or 3D environments (Murphy, 2000; Matthews, 2002; Rabin, 2000a; Higgins, 2002; Rabin, 2000b). The only requirement for A* to be complete, correct and optimal is that the lengths of all the edges in the graph are non-negative and that the h-values are admissible. A* is at the core of every new find-path algorithm that is introduced in this dissertation. In Chapter 4, we introduce an implementation of A* with consistent h-values and discuss the algorithm in greater detail.

2.2.2.2 Incremental Find-Path Algorithms

As we described in Section 1.2.2.3, our approach to path planning in unknown 2D environments is planning with the freespace assumption. When path planning in unknown 2D environments with the freespace assumption, an agent may find its current path to be blocked as it follows it toward the goal coordinate, in which case it must perform path planning again. Therefore, in unknown 2D environments, a path from a given start vertex to a given goal vertex is often found incrementally by solving a series of similar path-planning problems. Therefore, we refer to find-path algorithms that are used when path planning in unknown 2D environments as "incremental" find-path algorithms.

In unknown 2D environments, an agent is forced to find a new path every time it finds its current path to be blocked. Thus, an agent must solve a new instance of both the generate-graph problem and the find-path problem each time it finds its current path to be blocked. In Section 2.2.1.4, we examined techniques that an agent can use to update the graph to account for the information it learns about the traversable space in the continuous environment (that is, solve the generate-graph problem) when it finds its current path to be blocked. In this section, we examine techniques that an agent can use to efficiently search this new graph for a new path from its current vertex to the goal vertex (that is, solve the find-path problem) when it finds its current path to be blocked. This problem is difficult because solving the find-path problem from scratch using a single-shot find-path algorithm each time the graph changes can be time consuming (Stentz, 1994). Therefore, in order for an incremental find-path algorithm to provide a good tradeoff with respect to the runtime of the search and the length of the resulting path it must efficiently reuse information from previous find-path problems to find a solution to the next find-path problem more quickly. In Section 2.2.1.4, we showed that the graph constructed from the continuous environment prior to the agent finding its current path to be blocked is typically very similar to the graph constructed from the continuous environment after the agent has found its current path to be blocked. It turns out that the search trees that are generated when searching either of the two graphs are also very similar (Koenig et al., 2004). Thus, an agent should be able to reuse information from previous find-path problems to find a solution to the next find-path problem more quickly. This is precisely what incremental find-path algorithms do. We examine

two incremental find-path algorithms, namely Differential A* (Trovato & Dorst, 2002), an eager informed incremental find-path algorithm, and D* Lite (Koenig & Likhachev, 2002a), a lazy informed incremental find-path algorithm:

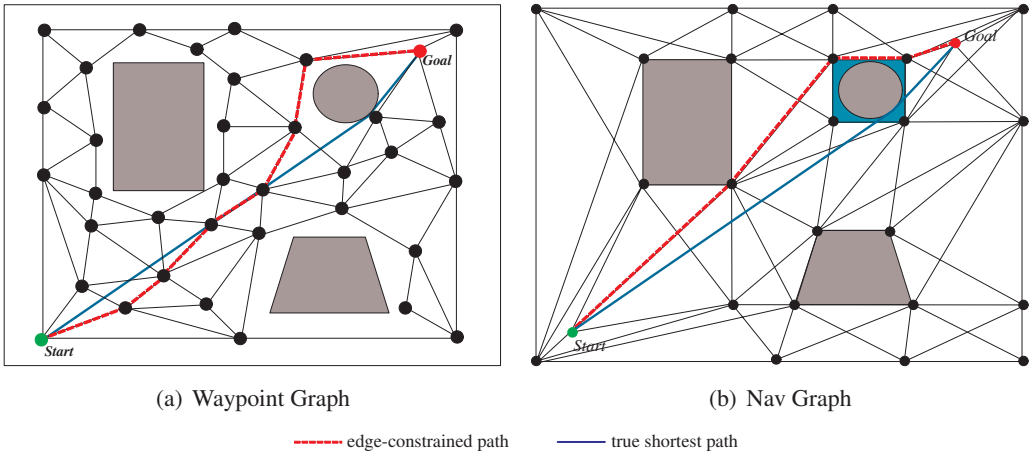
- **Differential A*: An Eager Informed Incremental Find-Path Algorithm.** Differential A* (Trovato & Dorst, 2002) is an eager informed incremental find-path algorithm that is designed to solve a series of similar find-path problems. Differential A* does this by performing a series of standard A* searches. However, the A* searches are not performed from scratch because between each A* search, the search tree from previous A* searches is repaired to account for the changes to the graph, which allows the subsequent A* search to be started on the (repaired) search tree from the previous A* searches. In other words, Differential A* preprocesses the search tree from previous searches so that the subsequent search can reuse information from previous searches, which makes the subsequent search faster than performing an A* search from scratch.

  We say that Differential A* is an eager informed incremental find-path algorithm because it often updates portions of the search tree from previous searches that do not need to be repaired in order to find a shortest path on the graph in the subsequent search. The new incremental find-path algorithm that we introduce in Chapter 6 is an eager incremental find-path algorithm that is similar in spirit to Differential A*.

  Differential A* is more difficult to implement and understand than repeated A* searches because it performs a series of A* searches between which the search tree must be repaired in order to account for changes to the graph while ensuring that the subsequent A* search is complete, correct and optimal. Updating the search tree between successive A* searches is done using a difference engine, which is essentially an extensive case based analysis that handles all the different ways in which the search tree must be repaired. Each A* search performed during a Differential A* search finds a shortest path. However, when path planning in unknown 2D environments with the freespace assumption it can be faster than repeated A* searches (Trovato & Dorst, 2002). Differential A* has the same requirements for completeness, correctness and optimality as A* and thus it can be used to search any Euclidean graph.

- **D* Lite: A Lazy Informed Incremental Find-Path Algorithm.** D* Lite (Koenig & Likhachev, 2002a) is a lazy incremental find-path algorithm that is designed to solve a series of similar find-path problems. Unlike Differential A*, which performs a series of A* searches between which the search tree is preprocessed, D* Lite performs a series of searches without any preprocessing between successive searches. A D* Lite search is very similar to an A* search, but it performs additional computations during the expansion process in order to maintain invariants. These invariants allow D* Lite to repair the search tree from previous D* Lite searches during the subsequent D* Lite search.

  We say that D* Lite is a lazy incremental find-path algorithm because it lazily repairs the search tree, that is, it is able to identify and repair only the portions of the search tree from previous searches that must be repaired in order for the subsequent search to be complete, correct and optimal. To the best of our knowledge no experimental evaluation has been performed which compares Differential A* with D* Lite (and thus we do not make a claim as to which has a shorter runtime).



(a) Waypoint Graph (b) Nav Graph

Figure 2.12: Edge-Constrained Path Versus True Shortest Path

The basic version of D* Lite is only slightly more difficult to implement than A* and the pseudocode for D* Lite is not much longer than the pseudocode for A*. However, D* Lite is more difficult to understand than either A* or Differential A* because the invariants that D* Lite maintains are difficult to understand. Each D* Lite search is guaranteed to find a shortest path, however when path planning in unknown 2D environments with the freespace assumption it can be one order of magnitude faster than repeated A* searches (from scratch) (Koenig & Likhachev, 2002a). D* Lite has the same requirements for completeness, correctness and optimality that A* has and thus it can be used to search any Euclidean graph.

Incremental find-path algorithms solve a more difficult problem than single-shot find-path algorithms and as a result they are more difficult to understand and implement than single-shot find-path algorithms. However, incremental find-path algorithms can be faster than repeated single-shot searches when path planning in unknown 2D environments with the freespace assumption. Furthermore, like single-shot find-path algorithms, incremental find-path algorithms can find shortest paths on the graphs that they search. This does not mean that incremental find-path algorithms are ideal for path planning in all unknown 2D environments. Incremental find-path algorithms require that the search tree be kept in memory. For large environments, such large memory usage may not be feasible. Furthermore, if the agent has to find new paths frequently, it may be more efficient to use a simple technique for navigating an agent to the goal coordinate such as the Bug algorithm (which may find longer paths as we argued in Section 2.1).

In this section, we showed that traditional edge-constrained find-path algorithms maintain the Simplicity, Efficiency and Generality Properties. However, maintaining these properties comes with a penalty because traditional edge-constrained find-path algorithms propagate information along graph edges *and* constrain paths to be formed by graph edges. This constraint is artificial and causes the paths found by traditional edge-constrained find-path algorithms to be both longer and less realistic looking than the true shortest paths. This can be seen in Figures 1.3, 2.12(a) and

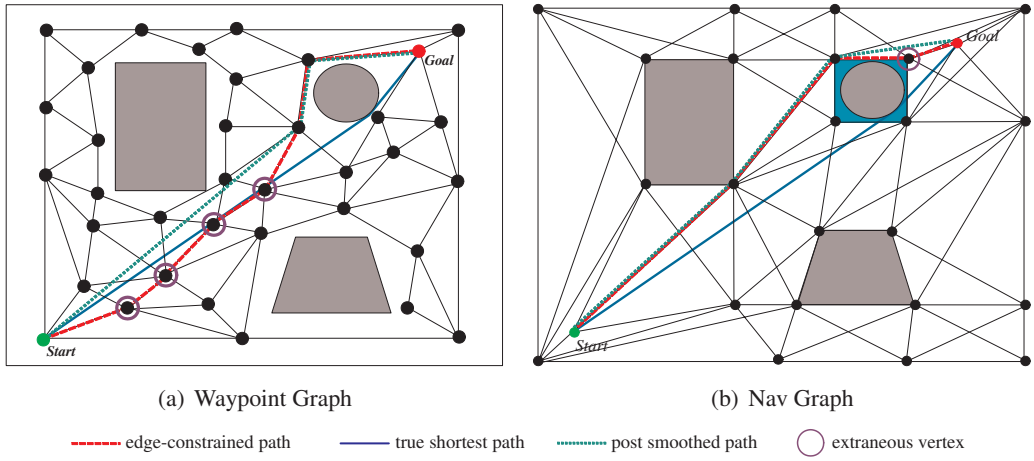


Figure 2.13: Post-Processing Techniques

2.12(b) in which the shortest path formed by the edges of a grid graph, a waypoint graph and a nav graph, respectively, are depicted by the dashed red path and the true shortest path is depicted by the solid blue path.

2.2.3 Post-Processing Techniques

In this section, we introduce two post-processing techniques that are widely used by roboticists and video game developers to address the fact that traditional edge-constrained find-path algorithms can find long and unrealistic looking paths (Rabin, 2000a; Thorpe, 1984; Botea, Müller, & Schaeffer, 2004; Ferguson & Stentz, 2006; Tozour, 2008; Ian Millington, 2009). These two post-processing techniques demonstrate two different approaches to post processing, the key ideas of which are at the core of every post-processing technique that we know of.

- **Post-Smoothing (Botea et al., 2004; Ian Millington, 2009)** is a post-processing technique that uses the triangle-inequality to find paths that are shorter and more realistic looking than edge-constrained paths. Post Smoothing makes a path shorter and more realistic looking by repeatedly removing extraneous vertices from the path. An extraneous vertex is any vertex that is on the path and lies between two vertices on the path which have line-of-sight. We call these vertices extraneous because the triangle inequality ensures that a path without an extraneous vertex is guaranteed to be no longer than a path with an extraneous vertex. Consider Figures 2.13(a) and 2.13(b), in which the dashed red path is an edge-constrained path, the circled vertices are extraneous vertices on the edge-constrained path, the dotted green path is the post-smoothed path and the solid blue path is the true shortest path. In both cases, post-smoothing makes the edge-constrained path shorter and more realistic looking.

  In some cases, post-smoothing is able to "smooth" the edge-constrained path into the shortest any-angle path. For example, applying post-smoothing to the dashed red paths depicted in Figures 1.3 and 1.4 results in shortest any-angle paths, which are depicted by the solid blue paths. However, in other cases, post-smoothing is not able to smooth the edge-constrained path into the shortest any-angle path. For example, applying post-smoothing to

the dashed red path depicted in Figure 1.6 does not make the path shorter or more realistic looking. The reason for this is that post-smoothing, like all post-processing techniques, is applied to a path that is found by a find-path algorithm which only considers edge-constrained paths during the search and thus cannot make informed decisions about other paths during the search. (Nash et al., 2007; Ferguson & Stentz, 2006; Patel, 2000). In Chapter 4, we show that, in some cases, traditional edge-constrained find-path algorithms find paths that are difficult to smooth.

- **Konolige's Gradient Method** (Konolige, 2000) is a post-processing technique that uses interpolation to find paths that are shorter and more realistic looking than edge-constrained paths. Unlike post-smoothing, Konolige's Gradient Method operates on the g-values that were computed during the search and not on the path retrieved after the search terminates. When a traditional edge-constrained find-path algorithm terminates, each vertex in the search tree has a g-value. Konolige's Gradient Method interpolates between these g-values to extract paths from the search tree that are not constrained to graph edges. Like post-smoothing, Konolige's Gradient Method can allow for paths that are shorter and more realistic looking than edge-constrained paths. However, interpolation causes approximation errors which can result in paths that are longer and less realistic looking than edge-constrained paths. Finally, as is the case with all post-processing techniques, Konolige's gradient method is applied to a path that is found by a find-path algorithm that only considers edge-constrained paths during the search.

The runtime of post-processing techniques is typically much shorter than the runtimes of traditional edge-constrained find-path algorithms. This is because the number of computations performed by post-processing techniques is typically $\approx n$ where n is the number of vertices on the path found by a traditional edge-constrained find-path algorithm. For post-smoothing this means that $\approx n$ line-of-sight checks are performed (that is, each vertex on the path found by the traditional edge-constrained find-path algorithm is tested to determine whether or not it is an extraneous vertex). Therefore, as long as the runtime required to determine whether or not two vertices have line-of-sight remains relatively short, as is the case on regular grids and NavMeshes, traditional edge-constrained find-path algorithms with post-smoothing can provide a better tradeoff with respect to the runtime of the search and the length of the resulting path than traditional edge-constrained find-path algorithms without post-smoothing (Thorpe, 1984; Botea et al., 2004; Ian Millington, 2009). For Konolige's Gradient Method, this means that $\approx n$ interpolation computations are performed. Therefore, as long as the runtime required to perform linear interpolation remains relatively short, traditional edge-constrained find-path algorithms with Konolige's Gradient Method can provide a better tradeoff with respect to the runtime of the search and the length of the resulting path than traditional edge-constrained find-path algorithms without Konolige's Gradient Method (Konolige, 2000).[6]

We have argued that, in general, while post-processing techniques have a short runtime they are not a reliable way to find short and realistic looking paths. Therefore, in order to reliably find shorter and more realistic looking paths, a new class of more sophisticated find-path algorithms are needed. These two post-processing techniques provide key ideas that these more sophisticated

[6]Optimizations can be used to reduce the runtime of post-processing techniques. For example, some post-smoothing techniques first check to see whether or not the start vertex and the goal vertex have line-of-sight (Sturtevant, 2010).

find-path algorithms might use to find shorter and more realistic looking paths. All of the new any-angle find-path algorithms introduced in this dissertation make use of the triangle inequality, and the only existing any-angle find-path algorithm, namely Field D*, makes use of interpolation.

2.2.4 Existing Any-Angle Find-Path Algorithms

In this section, we examine the current state-of-the-art find-path algorithm, namely Field D*. Any-angle find-path algorithms are a new class of find-path algorithms introduced in this dissertation that have the desirable properties of traditional edge-constrained find-path algorithms, but find shorter and more realistic looking paths. Any-angle find-path algorithms accomplish this by propagating information along graph edges (to achieve a short runtime) without constraining paths to be formed by graph edges (to find any-angle paths). While the idea of creating a new class of any-angle find-path algorithms that act as desirable alternatives to traditional edge-constrained find-path algorithms is unique to this dissertation, the idea of propagating information along graph edges without constraining paths to be formed by graph edges is not. Field D* (Ferguson & Stentz, 2006) was developed by roboticists to do exactly that when solving a very specific path-planning problem, namely path planning in unknown 2D environments that have been discretized into 8-neighbor square grid graphs. While Field D* has proven to be useful for this particular problem (Ferguson, 2006; Ferguson & Stentz, 2006; Carsten et al., 2009), it was not designed to be a more sophisticated approach for solving the general find-path problem.

- **Field D*: An Incremental Any-Angle Find-Path Algorithm:** Field D* (Ferguson & Stentz, 2006) is an incremental any-angle find-path algorithm that combines the ideas from interpolation (Konolige, 2000; Kimmel & Sethian, 2001; Philippsen & Siegwart, 2005; Philippsen, Kolski, Macek, & Siegwart, 2007) with the ideas from lazy informed incremental find-path algorithms, in order to find paths that are not formed by grid graph edges in unknown 2D environments. Field D* is identical to D* Lite except that it computes g-values differently during the expansion process. The expansion equation used by Field D* is a closed form linear interpolation equation which computes g-values that represent the lengths of paths that are not formed by grid graph edges.

  The expansion equation computed by Field D* is more difficult to implement and understand that the expansion equation computed by D* Lite and thus Field D* is more difficult to implement and understand than D* Lite. Field D* is not guaranteed to find paths that are shorter than shortest grid paths because of linear interpolation error (Ferguson & Stentz, 2006). However, in the work of Ferguson and Stentz (2006) it was shown that Field D* typically finds shorter paths than D* Lite with only a slightly longer runtime (Ferguson & Stentz, 2006). The closed form linear interpolation equation that Field D* uses to compute the expansion equation requires that the search be performed on an 8-neighbor square grid graph constructed from a square grid composed of (potentially varying sized) squares.[7] Therefore, Field D* cannot be used to search any Euclidean graph.

[7] Recently, a more general closed form linear interpolation equation has been introduced which can be used on searches performed on graphs constructed from triangular meshes. However, this equation requires many more trigonometric and floating point computations than the one used in Field D* and thus has a longer runtime (Sapronov & Lacaze, 2010).

Finally, due to linear interpolation, extracting the path from the search tree is more difficult for Field D* than it is for Dijkstra's algorithm, A*, Differential A* and D* Lite, all of which just follow the parents in the search tree. Furthermore, if path extraction is not implemented with optimizations, such as a one step lookahead, then Field D* frequently finds paths that are longer and less realistic looking than shortest grid paths (Ferguson & Stentz, 2006). The difficulty of path extraction is another reason why Field D* is more difficult to implement and understand than D* Lite.

3D Field D* (Carsten et al., 2006) is a variant of Field D* that can be used to find paths in unknown 3D environments. 3D Field D* is identical to Field D*, however it computes a different expansion equation during each iteration of the expansion process. The expansion equation used by 3D Field D* is a closed form linear interpolation equation which computes g-values that represent the lengths of paths that are not formed by grid graph edges. This closed form linear interpolation equation requires that the 3D Field D* search is performed on a 26-neighbor cubic grid graph constructed from a cubic grid composed of (potentially varying sized) cubes. Both Field D* and 3D Field D* use linear interpolation, but linear interpolation is more difficult in 3D than it is in 2D. In 2D, Field D* uses linear interpolation to approximate the lengths of paths that pass through arbitrary coordinates on the side of a square grid cell. In 3D, 3D Field D* uses linear interpolation to approximate the lengths of paths that pass through arbitrary coordinates on the face of a cubic grid cell. Linear interpolation cannot be solved optimally with a closed form linear interpolation equation in the latter case and using numerical methods to repeatedly compute optimal solutions is slow. Therefore, 3D Field D* uses a closed form linear interpolation equation that makes additional approximations. As a result, 3D Field D* is both more difficult to implement and understand than Field D* and more susceptible to linear interpolation error than Field D*. 3D Field D* cannot be used to search any Euclidean graph.

In this section, we showed that, while Field D* and its variant 3D Field D*, find paths that are not constrained to be formed by graph edges, it has several short-comings. We provide a more thorough discussion of Field D* in Section 4.3.3 and compare it with our new any-angle find-path algorithms.

A summary of our classification of find-path algorithms is shown in Figure 2.14. It is annotated in a similar manner to Figure 2.10: Rounded rectangles represent different classes of find-path algorithms, ovals represent find-path algorithms and if an oval is within a rounded rectangle then it is a member of that class. Figure 2.14 also contains additional notation: If an oval is transparent then the find-path algorithm can be used to search any Euclidean graph. If an oval is opaque then the find-path algorithm cannot be used to search any Euclidean graph. For example, the A* oval is within 3 rounded rectangles, and thus A* is a **1** single-shot, **2** informed, **3** find-path algorithm. Furthermore, because the A* oval is opaque, it can be used to search any Euclidean graph (Generality Property). Finally, the dashed rectangles show how the contributions made in this dissertation relate to the different classes of find-path algorithms.

The rounded rectangles and ovals in Figure 2.14 were introduced in this section as follows: First, we separated find-path algorithms into two classes, namely single-shot find-path algorithms

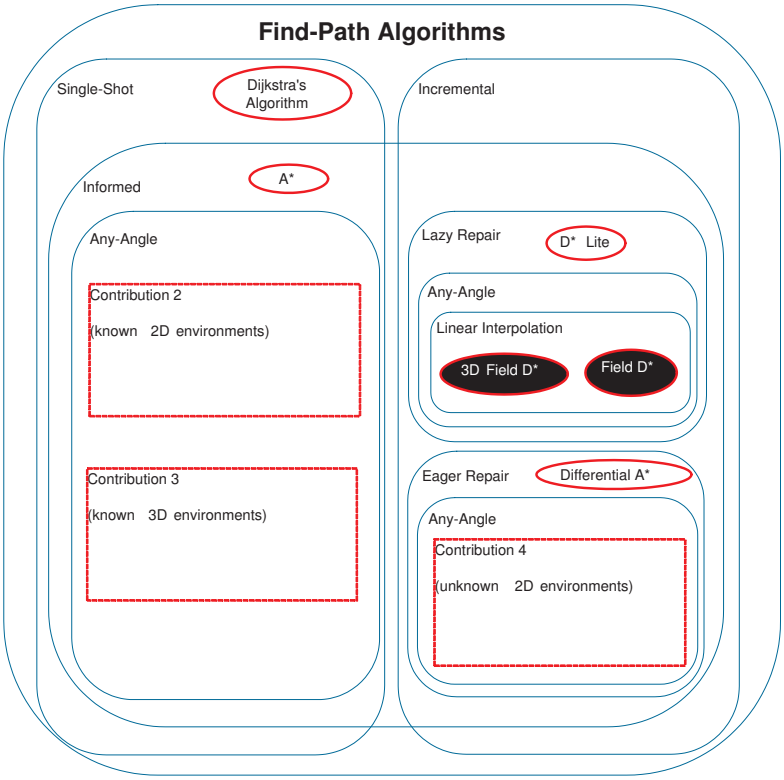


Figure 2.14: Classification of Find-Path Algorithms

(Section 2.2.2.1) and incremental find-path algorithms (Section 2.2.2.2), each of which is represented by a rounded rectangle in Figure 2.14. We then introduced several different subclasses of both single-shot find-path algorithms and incremental find-path algorithms (for example, *informed* and *any-angle*), each of which is represented by a rounded rectangle in Figure 2.14. While the subclasses of single-shot find-path algorithms are similar to the subclasses of incremental find-path algorithms, they are not identical (for example, both classes contain an *informed* subclass but both classes do not contain a *lazy repair* subclass). This is because the two classes are designed for finding paths in different types of environments and thus their members must operate differently as described above. Throughout Sections 2.2.2.1 and 2.2.2.2 we introduced several different find-path algorithms, each of which is represented by an oval in Figure 2.14.

2.3 Conclusions

In Section 2.1, we showed that path planning is a key part of navigation, which is one of the most important parts of robotics and video game applications. In Section 2.2, we showed that path planning is often separated into the generate-graph problem and the find-path problem. In Section 2.2.1, we introduced many techniques for solving the generate-graph problem, each of which performs well in certain types of continuous environments, but none of which is a panacea. Furthermore, we highlighted many of the desirable properties of regular grids since they are the focus of this dissertation. For example, we showed both that they are rigorously defined and

thus we can determine an upper bound comparing the lengths of shortest grid paths with the lengths of true shortest paths and that they work well in known 2D environments, known 3D environments and unknown 3D environments. In Section 2.2.2, we introduced several traditional edge-constrained find-path algorithms and showed that they have the following three desirable properties: Simplicity Property: They are simple to implement and understand. Efficiency Property: They provide a good tradeoff with respect to the runtime of the search and the length of the resulting path. Generality Property: They can be used to search any Euclidean graph. However, these desirable properties come with a penalty because these traditional edge-constrained find-path algorithms artificially constrain paths to be formed by graph edges and edge-constrained paths are often longer and less realistic looking than true shortest paths. In Section 2.2.3, we showed that post-processing the paths found by traditional edge-constrained find-path algorithms is often ineffective because the find-path algorithms have no knowledge of these shorter post-processed paths. Therefore, there is a need for more sophisticated any-angle find-path algorithms. In Section 2.2.2, we examined Field D* and 3D Field D*, the current state-of-the-art any-angle find-path algorithms. We showed that Field D* is an any-angle find-path algorithm designed for a very specific path-planning problem and has the following substantial shortcomings: **(1)** it can only be used to search 8-neighbor square grid graphs, **(2)** it is susceptible to linear interpolation errors which can lead to paths that are longer than shortest grid paths and **(3)** it needs to use methods, that are difficult to understand and implement, to extract a path from the search tree in order to mitigate linear interpolation error. We showed that 3D Field D* is similar to Field D* and suffers from the same shortcomings.

Chapter 3

Path Length Analysis on Regular Grids (Contribution 1)

Figure 3.1: Hexagonal Grid: Screen Shot from Civilization V (Firaxis Games)

This chapter introduces the first major contribution of this dissertation, namely Contribution 1. Specifically, this chapter introduces a set of eight new analytical worst case bounds which compare the lengths of the paths found by traditional edge-constrained find-path algorithms on grid graphs constructed from both 2D and 3D regular grids with the lengths of the true shortest paths. Therefore, these results validate Hypothesis 1.

This chapter is organized as follows: In Section 3.1, we reiterate the motivation behind Hypothesis 1. In Section 3.2, we compare the lengths of the paths found by traditional edge-constrained find-path algorithms on grid graphs constructed from 2D regular grids with the lengths of the true shortest paths. In Section 3.3, we compare the lengths of the paths found by traditional edge-constrained find-path algorithms on grid graphs constructed from 3D regular grids with the lengths of the true shortest paths. Finally, in Section 3.4 we summarize our results.

3.1 Introduction

In Section 2.2, we showed that path planning is typically composed of two parts: the generate-graph problem, which is solved by discretizing a continuous environment into a graph and the find-path problem, which is solved by searching this graph for a path from a given start vertex to a given goal vertex.

Roboticists and video game developers often solve the generate-graph problem by discretizing a continuous 2D or 3D environment into a regular grid (Patel, 2000; Deloura, 2000; Murphy, 2000; Tozour, 2004; Lengyel et al., 1990; Cohen et al., 2011; Ferguson & Stentz, 2006; Carsten et al., 2006; Chrpa & Komenda, 2011; Chrpa, 2011). A continuous 2D or 3D environment can be discretized into a regular grid by laying the regular grid over the continuous 2D or 3D environment and labeling grid cells as blocked iff they contain part of an obstacle. A grid graph is then constructed by placing vertices in either the corners or centers of grid cells and adding edges between a vertex and up to n of its adjacent vertices (resulting in an n-neighbor grid graph).

Roboticists and video game developers typically solve the find-path problem using a traditional edge-constrained find-path algorithm because they are simple, efficient, general and can find shortest grid paths (that is, shortest paths formed by grid graph edges). However, shortest grid paths can be longer than true shortest paths (that is, shortest paths in the continuous environment) because shortest grid paths are artificially constrained to be formed by grid graph edges. The paths found by traditional edge-constrained find-path algorithms can be improved with post-processing techniques, but these techniques are not guaranteed to improve paths (Section 2.2.3). While it is well known that shortest grid paths can be longer than true shortest paths, the question, namely how much longer shortest grid paths can be than true shortest paths, remains unanswered. To the best of our knowledge, all existing path length analysis results are incomplete because they ignore the potential effects of blocked grid cells and are specific to a particular tessellation. For example, both the well known result which shows that shortest grid paths on 8-neighbor square grid graphs can be $\approx 8\%$ longer than true shortest paths and the result which shows that shortest grid paths on 6-neighbor triangular grid graphs can be $\approx 15\%$ longer than true shortest paths (Nagy, 2003) do not account for the potential effects of blocked grid cells on path lengths and are specific to a particular type of grid graph. In this chapter, we introduce a simple, unified proof structure which is used to show that shortest grid paths formed by the edges of grid graphs constructed from both 2D and 3D regular grids can be longer than true shortest paths as follows: (*2D*) 100% for 3-neighbor triangular grid graphs (tri graphs), $\approx 41\%$ for 4-neighbor square grid graphs (quad graphs), $\approx 15\%$ for 6-neighbor hexagonal grid graphs (hex graphs), $\approx 15\%$ for 6-neighbor triangular grid graphs (double tri graphs), $\approx 8\%$ for 8-neighbor square grid graphs (octile graphs) and $\approx 4\%$ for 12-neighbor hexagonal grid graphs (double hex graphs), (*3D*) $\approx 73\%$ for 6-neighbor cubic grid graphs (cubic graphs) and $\approx 13\%$ for 26-neighbor cubic grid graphs (triple cubic graphs) (Nash et al., 2010).

In Section 3.2, we examine shortest grid paths formed by the edges of grid graphs constructed from regular grids that discretize a continuous 2D environment and in Section 3.3 we examine shortest grid paths formed by the edges of grid graphs constructed from regular grids that discretize continuous 3D environments.

3.2 2D Regular Grids

In 2D, roboticists and video game developers often solve the generate-graph problem by discretizing continuous 2D environments into regular grids composed of triangles (triangular grids), squares (square grids), or hexagons (hexagonal grids) (Tozour, 2004; Rabin, 2000b; Chrpa & Komenda, 2011; Björnsson et al., 2003). These are the only types of regular grids used because triangles, squares and hexagons are the only regular polygons (that is, equilateral and equiangular polygons) that can be used to tessellate a plane (that is, their interior angles divide evenly into

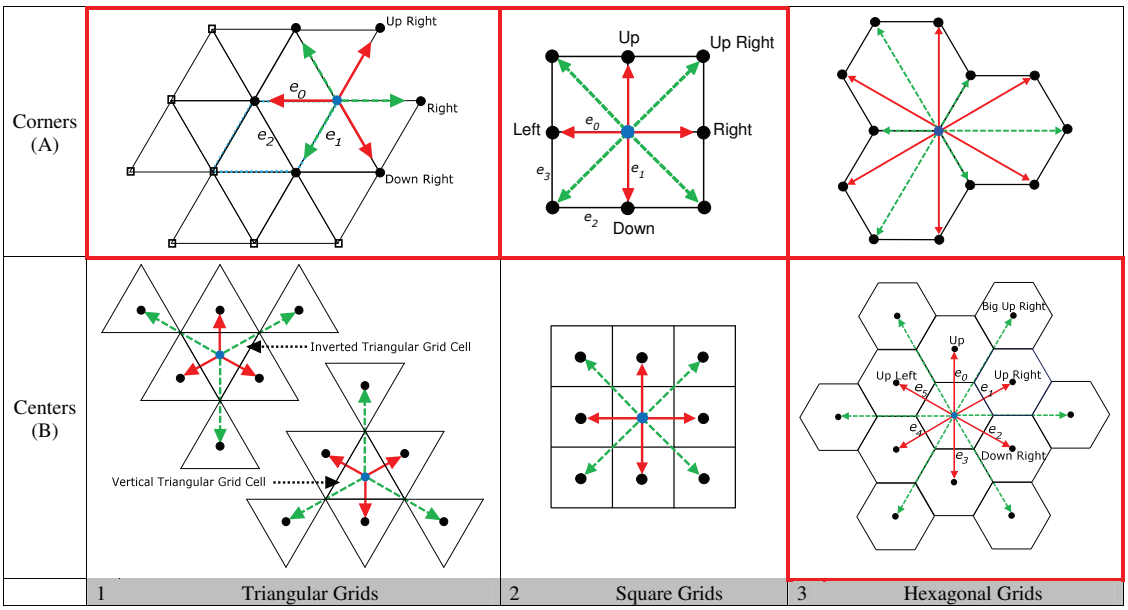


Table 3.1: Grid Graphs Constructed from 2D Regular Grids

2π). First, we show that, while regular grids are simple to understand, how one constructs a grid graph from a regular grid can have a significant impact on many aspects of a path-planning system. Second, we compare the lengths of shortest grid paths formed by the edges of grid graphs constructed from regular grids that discretize a continuous 2D environment with the lengths of true shortest paths.

3.2.1 Existing Work: 2D Regular Grids

Regular grids are widely used by roboticists and video game developers. For example, hexagonal grids are used in the video game "Civilization V" (Figure 3.1) and square grids are used in both the video game "Company of Heroes" and on the Mars rovers Spirit and Opportunity (Carsten et al., 2009). Triangular, square and hexagonal grids are used by roboticists and video game developers because each type of regular grid has desirable properties. Square grids are the simplest types of regular grids to implement because each square grid cell is exactly one unit from the grid cell that is immediately above or below it and exactly one unit from the grid cell that is immediately to the left or right of it (where a unit is defined as the length of a side of a grid cell). Thus, it is easy to determine the grid cell corresponding to any coordinate in the continuous environment. Hexagonal grids also have desirable properties. Björnsson et al. (2003) showed analytically that searches on grid graphs constructed from hexagonal grids have smaller search complexities than searches on grid graphs constructed from square grids (Björnsson et al., 2003). García and Garrido (2007) showed empirically that searches on grid graphs constructed from hexagonal grids can be faster than searches on grid graphs constructed from square grids if the grids are large or the search algorithm is not very efficient (García & Garrido, 2007). Triangular grids are not as widely used as square or hexagonal grids, but they also have desirable properties. Since 6

adjacent triangular grid cells can be grouped together to form a hexagonal grid cell, a hexagonal grid can be overlaid on top of a triangular grid, which is useful in video games. For example, in a 2D strategy game, one might have squads of 6 infantry men, 1 infantry man per triangle. The user issues both move and attack orders for the squad as a whole. When a squad receives a move order, it ignores the double tri graph and finds a path on the hex graph. When a squad receives an attack order, each infantry man in the squad ignores the hex graph and finds a path on the double tri graph (Tozour, 2010).

3.2.2 Notation and Definitions

In this section, we explain how a grid graph $G' = (V, E')$ is constructed from a continuous 2D environment. A continuous 2D environment is discretized into a triangular, square or hexagonal grid by laying the regular grid over the continuous 2D environment and labeling grid cells as blocked iff they contain part of an obstacle. We assume that the length of every side of every grid cell is one. We examine six types of grid graphs constructed from triangular, square and hexagonal grids, namely tri graphs, double tri graphs, quad graphs, octile graphs, hex graphs and double hex graphs. Vertices $u \in V$ are placed in either the centers of grid cells or upper right corners of grid cells (with precedence toward upper). Edges $(u, v) \in E'$ connect vertex u to each of its n adjacent vertices, thus defining an n-neighbor grid graph. The value of n determines the branching factor of the grid graph. For all six types of grid graphs the n adjacent vertices must be selected such that the angles of the resulting n edges, with a common endpoint vertex u, are evenly distributed between $(0, 2\pi]$. An edge type is a class of edges that have the same angle. The outgoing edges of a vertex u are the set of edges that have vertex u as one endpoint. Edges that connect vertices that do not have line-of-sight are removed from E' (line-of-sight is defined in Section 1.1.2). The length of an edge $(u, v) \in E'$ is $c(u, v)$, which is the Euclidean distance between vertices u and v. Examples of each type of grid graph can be found in Table 3.1. The blue vertex represents a vertex $u \in V$ and the arrows depict the outgoing edges that connect vertex u to its n adjacent vertices. In columns 1, 2 and 3 of Table 3.1, the solid red arrows show the outgoing edges for tri graphs, quad graphs and hex graphs, respectively and the union of the solid red arrows and dashed green arrows show the outgoing edges for double tri graphs, octile graphs and double hex graphs, respectively. Cells A1, A2 and B3 of Table 3.1 contain additional notation, namely the sides of the regular polygon that the blue vertex maps to and the different edge types. For example, in cell B3 of Table 3.1, the six sides of the hexagon that the blue vertex maps to are labeled e_0, e_1, e_2, e_3, e_4 and e_5. Similarly, some of the outgoing edges of the blue vertex are labeled with their edge types, namely Up Left, Up, Big Up Right, Up Right and Down Right.

	Corners			Centers		
	Realism	**Speed**	**Data Structure**	**Realism**	**Speed**	**Data Structure**
Tri Graphs	Yes	Yes (1)	No	No	Yes $(\frac{\sqrt{3}}{3})$	Yes
Double Tri Graphs	Yes	Yes (1)	No	Yes	No $(\frac{\sqrt{3}}{3}, \frac{2\sqrt{3}}{3})$	Yes
Quad Graphs	Yes	Yes (1)	Yes	Yes	Yes (1)	Yes
Octile Graphs	Yes	No $(1, \sqrt{2})$	Yes	Yes	No $(1, \sqrt{2})$	Yes
Hex Graphs	Yes	Yes $(\sqrt{3})$	Yes	Yes	Yes $(\sqrt{3})$	Yes
Double Hex Graphs	Yes	No $(1, \sqrt{3}, 2)$	No	Yes	No $(\sqrt{3}, 3)$	Yes

Table 3.2: Properties of Grid Graphs Constructed from 2D Regular Grids

3.2.3 Grid Graph Properties

In this section, we show that, how one constructs a grid graph from a regular grid can have a significant impact on many aspects of a path-planning system. We examine three properties: **Realism Property:** It is important that agents follow paths that are realistic looking. For example, in robotics and video games, straight paths are typically preferred over paths that have unnecessary heading changes. **Speed Property:** It is important that identical agents move at identical speeds. For example, 2D strategy games often allow agents to make a single move per time step. If the lengths of different moves are not the same then the speed of an agent varies depending on its move choices. **Data Structure Property:** It is important that a regular grid can be used for both path planning and as a simple data structure. For example, it is common to associate information with the region of the continuous environment that corresponds to a grid cell, such as the amount of gold hidden in the region or a rendering of the region when displaying the terrain.

The extent to which these properties are satisfied can be evaluated by examining properties of a grid graph as follows: If the n edge types of the n outgoing edges of every vertex are the same then grid paths contain fewer unnecessary heading changes and thus the Realism Property is satisfied. If each edge in the grid graph has the same length then the Speed Property is satisfied. If each vertex can be mapped uniquely to a grid cell (and vice versa) then the Data Structure Property is satisfied because each vertex (and thus each grid cell) can be assigned an index into a data structure which contains information associated with that grid cell.

We consider twelve different types of grid graphs, namely two types of grid graphs for each type of regular grid when placing vertices in either the corners or centers of grid cells. We evaluate each type of grid graph with respect to the Speed, Realism and Data Structure Properties. A summary of these results is given in Table 3.2 where the **Speed**, **Realism** and **Data Structure** columns indicate whether or not the Speed, Realism and Data Structure Properties are satisfied, respectively. The Speed column contains the lengths of the edges in the grid graph. The Speed Property is satisfied if all edges have the same length.

First, we examine tri graphs, quad graphs and hex graphs with vertices placed in the centers of grid cells (the solid red arrows and the vertices they point to in cells B1, B2 and B3 of Table 3.1, respectively). The Realism Property is not satisfied for tri graphs, but is satisfied for quad graphs and hex graphs. Triangular grids, unlike square and hexagonal grids, are composed of two types of grid cells, namely vertical and inverted triangular grid cells (cell B1 of Table 3.1). As a result, unlike quad graphs and hex graphs in which the n edges types of the n outgoing edges of every vertex are the same, the n edge types of the n outgoing edges of a vertex that maps to a vertical triangular grid cell are offset π radians from the n edge types of the n outgoing edges of a vertex that maps to an inverted triangular grid cell. The Speed Property is satisfied for each type of grid graph because every edge connects the centers of grid cells (and thus regular polygons) that share a side and thus every edge has the same length. The Data Structure Property is satisfied for each type of grid graph because each vertex is in the center of the grid cell that it maps to.

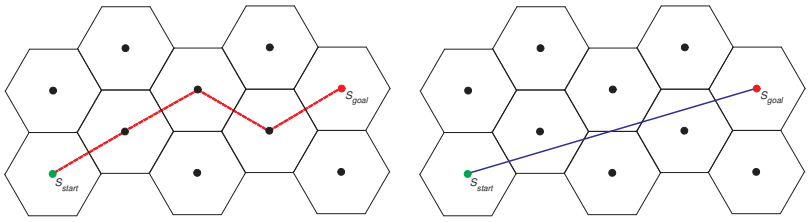
Second, we examine tri graphs, quad graphs and hex graphs with vertices placed in the corners of grid cells (the solid red arrows and the vertices they point to in cells A1, A2 and A3 of Table 3.1, respectively). The Realism Property is satisfied for each type of grid graph because the n edge types of the n outgoing edges of every vertex are the same. The Speed Property is satisfied for each type of grid graph. It is satisfied for tri graphs and quad graphs because each edge coincides with the side of a grid cell (and thus a regular polygon). It is satisfied for hex graphs because each

edge traverses the interior of a hexagonal grid cell with the same geometry, that is, each edge and two sides of a hexagonal grid cell form an isosceles triangle whose interior angles are $\frac{\pi}{6}$, $\frac{\pi}{6}$ and $\frac{2\pi}{3}$ (for example, the solid red arrows in cell A3 of Table 3.1 all have the same length). The Data Structure Property is not satisfied for tri graphs, but is satisfied for quad graphs and hex graphs. For tri graphs, vertices can map to either vertical triangular grid cells or inverted triangular grid cells because there are twice as many triangles as there are vertices. For example, in cell A1 of Table 3.1 there are 7 vertices and 14 triangles and each vertex is in the upper right corner of two adjacent triangular grid cells (that is, one vertical and one inverted triangular grid cell that share side e_2 of the inverted triangular grid cell). Notice that if the small transparent squares in cell A1 of Table 3.1 were replaced with vertices, there would still be twice as many triangles as there are vertices because each transparent square is in the upper right corner of two adjacent triangular grid cells.[1] This results in a limitation for video game developers because no vertices map to half of the grid cells (namely, either the vertical or inverted triangular grid cells). Thus, the grid graph does not contain information about the regions of the continuous environment that lie within half of the grid cells. For example, a video game developer cannot specify that there is a different amount of gold hidden within the region of the continuous environment that corresponds to a particular vertical triangular grid cell than there is within its adjacent inverted triangular grid cell. For simplicity, we address this by assigning information to two adjacent triangular grid cells. We place vertices in the upper right corners of inverted triangular grid cells and each vertex is in the upper right corner of the rhombus that it maps to, where the rhombus is defined by two adjacent triangular grid cells. For example, in cell A1 of Table 3.1, the blue vertex maps to the region within the dotted blue line which is defined by two adjacent triangular grid cells. For quad graphs we place vertices in the upper right corners of grid cells and thus each vertex is in the upper right corner of the grid cell that it maps to and for hex graphs we place vertices in the upper right corners of grid cells (with precedence toward upper) and thus each vertex is in the upper right corner of the grid cell that it maps to.

Third, we examine double tri graphs, octile graphs and double hex graphs with vertices placed in the centers of grid cells (the union of the solid red and dashed green arrows and the vertices they point to in cells B1, B2 and B3 of Table 3.1, respectively). The Realism Property is satisfied for each type of grid graph because the n edge types of the n outgoing edges of every vertex are the same. The Speed Property is not satisfied for any type of grid graph. Each vertex has twice as many outgoing edges as there are sides of a grid cell. Therefore, half of the outgoing edges connect the centers of grid cells that share a side and the other half of the outgoing edges connect the centers of grid cells that are further apart. The Data Structure Property is satisfied for each type of grid graph because each vertex is in the center of the grid cell that it maps to.

Finally, we examine double tri graphs, octile graphs and double hex graphs with vertices placed in the corners of grid cells (the union of the solid red and dashed green arrows and the vertices they point to in cells A1, A2 and A3 of Table 3.1, respectively). The Realism Property is satisfied for each type of grid graph because the n edge types of the n outgoing edges of every vertex are the same. The Speed Property is satisfied for double tri graphs because each edge coincides with the side of a grid cell, but is not satisfied for octile graphs and double hex graphs because the outgoing edges of a vertex either coincide with the sides of grid cells or traverse the interiors of grid cells. The Data Structure Property is not satisfied for double tri graphs, is satisfied for octile graphs and is not satisfied for double hex graphs. For double tri graphs, we

[1] These triangles are not shown in cell A1 of Table 3.1.



(a) Shortest Grid Path (b) Shortest Any-Angle Path

Figure 3.2: Paths on Hex Graphs with Vertices Placed in Grid Cell Centers

place vertices in the upper right corners of inverted triangular grid cells and each vertex is in the upper right corner of the rhombus, defined by two adjacent triangular grid cells, that it maps to. For octile graphs we place vertices in the upper right corners of grid cells and thus each vertex is in the upper right corner of the grid cell that it maps to. For double hex graphs we place vertices in the upper left and upper right corners of grid cells and only the latter vertices are in the upper right corners (with precedence toward upper) of the hexagonal grid cells that they map to. Unlike tri graphs and double tri graphs, which have two grid cells per vertex, double hex graphs have two vertices per grid cell.

Our path length analysis examines only six of the twelve types of grid graphs, namely tri graphs and double tri graphs with vertices placed in the upper right *corners* of two adjacent triangular grid cells, quad graphs and octile graphs with vertices placed in the upper right *corners* of grid cells and hex graphs and double hex graphs with vertices placed in the *centers* of grid cells. In Table 3.1, these types of grid graphs are highlighted in red. Only six of the twelve types of grid graphs were chosen because this set provides a good tradeoff with respect to a thorough analysis and an elegant analysis. The set is thorough, in the sense that it includes a tri graph, a double tri graph, a quad graph, an octile graph, a hex graph and a double hex graph and it is elegant, in the sense that our path length analysis for all of these types of grid graphs uses a simple, unified proof structure. Furthermore, Table 3.2 shows that if one were to use the Speed, Realism and Data Structure Properties as evaluation criteria then the tri graphs, double tri graphs, quad graphs, octile graphs, hex graphs and double hex graphs that we have chosen each has the minimal number of "No" entries (although there are ties).

3.2.4 Path Length Analysis

We now determine how much longer shortest grid paths formed by the edges of grid graphs constructed from regular grids that discretize continuous 2D environments can be than true shortest paths. To prove this result, we differentiate between any-angle paths (that is, paths formed by line segments whose endpoints are vertices in the graph) and true shortest paths. The relationship between the lengths of true shortest paths and the lengths of shortest any-angle paths is simple since true shortest paths are at most as long as shortest any-angle paths.[2] The relationship between the lengths of shortest any-angle paths and the lengths of shortest grid paths is simple as well since shortest any-angle paths are at most as long as shortest grid paths. For example, in

[2]In many cases, the difference between the lengths of shortest any-angle paths and the lengths of true shortest paths decreases as the size of grid cells decreases. This follows from the fact that smaller grid cells often mitigate the effects of digitization bias.

	R	F	θ	α	*Type*
Tri Graphs	2	$= 2$	$\frac{2\pi}{3}$	$\frac{\pi}{3}$	*Tight*
Double Tri Graphs	$\frac{2\sqrt{3}}{3}$	≈ 1.15	$\frac{\pi}{3}$	$\frac{\pi}{6}$	*Tight*
Quad Graphs	$\sqrt{2}$	≈ 1.41	$\frac{\pi}{2}$	$\frac{\pi}{4}$	*Tight*
Octile Graphs	$\frac{2}{\sqrt{2+\sqrt{2}}}$	≈ 1.08	$\frac{\pi}{4}$	$\frac{\pi}{8}$	*Asymptotically Tight*
Hex Graphs	$\frac{2\sqrt{3}}{3}$	≈ 1.15	$\frac{\pi}{3}$	$\frac{\pi}{6}$	*Tight*
Double Hex Graphs	$\frac{2}{\sqrt{2+\sqrt{3}}}$	≈ 1.04	$\frac{\pi}{6}$	$\frac{\pi}{12}$	*Asymptotically Tight*

Table 3.3: Path Length Analysis Results for Grid Graphs Constructed from 2D Regular Grids

hex graphs, shortest grid paths can be longer than shortest any-angle paths, which can be seen in Figure 3.2. We now prove that shortest grid paths formed by the edges of tri graphs, quad graphs, hex graphs, double tri graphs, octile graphs and double hex graphs can be up to $\approx 100\%$, $\approx 41\%$, $\approx 15\%$, $\approx 15\%$, $\approx 8\%$ and $\approx 4\%$ longer than shortest any-angle paths, respectively and thus can be at least $\approx 100\%$, $\approx 41\%$, $\approx 15\%$, $\approx 15\%$, $\approx 8\%$ and $\approx 4\%$ longer than true shortest paths, respectively.

Consider an arbitrary regular grid composed of blocked and unblocked grid cells, whose corners or centers form the set of all vertices V. Let $s_{start} \in V$ and $s_{goal} \in V$ represent the first and last vertex on a given path, respectively. We construct two graphs. Graph $G = (V, E)$ contains an edge (u, v) iff vertex u and vertex v have line-of-sight. Graph $G' = (V, E')$ contains the same vertex set, but only contains an edge $(u, v) \in E' \subseteq E$ iff vertex u and vertex v are adjacent and have line-of-sight.

Let P be a shortest path from the start vertex to the goal vertex on G (that is, a shortest any-angle path). Let $d(s_{start}, s_{goal})$ be the length of this path. We construct a path P' from P which is a shortest path from the start vertex to the goal vertex on G' (that is, a shortest grid path). Let $d'(s_{start}, s_{goal})$ be the length of this path. Both P and P' are sequences of line segments.

Theorem 1. *If there exists a path P from s_{start} to s_{goal} in G, then $d'(s_{\text{start}}, s_{\text{goal}}) \leq R \cdot d(s_{\text{start}}, s_{\text{goal}})$, where R is a constant determined by the type of grid graph G'.*

Proof. We apply Lemma 1 (see below) to the sequence of line segments that compose P to yield a sequence of piecewise linear paths in G', each of length at most R times the length of the corresponding line segment. Hence, $d'(s_{start}, s_{goal}) \leq R \cdot d(s_{start}, s_{goal})$. Table 3.3 contains the values of R for each type of grid graph. □

Lemma 1 (see below) is a special case of Theorem 1 where P is a single edge and thus a straight line. Its proof contains three parts. **Part 1:** We show that P' can be formed by only edges that traverse the sides or interiors of grid cells which are known to be unblocked because they are traversed by P. This allows us to determine the length $d'(u, v)$ of P' for a given length $d(u, v)$ of P independent of which grid cells are blocked. **Part 2:** We show that P' is formed by at most two edge types. **Part 3:** We maximize the ratio of the lengths of P' and P using a technique which applies to any path formed by at most two edge types.

Parts 2 and 3 of this proof apply to all of the types of grid graphs that this proof considers. We provide examples of Part 2 and 3 by applying them to hex graphs and double hex graphs in *italic* text. Small parts of Part 1 of the proof must use logic that is specific to a particular type of grid graph. For clarity, we explain these parts of the proof for hex graphs and double hex graphs in *italic* text. The proof for the other types of grid graphs can be obtained by replacing the *italic* text in Part 1 with the *italic* text in Appendix C.1 corresponding to that other type of grid graph.

3.2.4.1 Notation and Definitions

Each of the n outgoing edges of a vertex on an n-neighbor grid graph has one of n edge types. Let the set of n edge types be $M = \{m_0, m_1, \ldots, m_{n-1}\}$ and their respective angles be $A = \{a_0, a_1, \ldots, a_{n-1}\}$. Therefore, edge types m_i and m_{i+1} (likewise m_{n-1} and m_0) are angularly adjacent. For example, on quad graphs $M = \{Up, Right, Down, Left\}$ and $A = \{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\}$ (cell A2 of Table 3.1).[3] If an angle α of a line segment L is such that $a_i < \alpha < a_{i+1}$ for angularly adjacent edge types m_i and m_{i+1} then we say that L is in the sector S defined by m_i and m_{i+1}.

Lemma 1. *If $(u, v) \in E$, then $d'(u, v) \leq R \cdot d(u, v)$, where R is a constant determined by the type of grid graph G'.*

Proof.

3.2.4.2 Part 1: Unblocked Path

We begin with Part 1 of the proof. Without loss of generality, assume that vertex u is at the origin, that is, vertex u is at coordinates $(0, 0)$. If the edge $(u, v) \in E$ can be formed by edges in E', then the theorem trivially holds. Otherwise, consider the prefix L of the straight line from vertex u to vertex v that ends at the first reached vertex, which we call vertex t, at coordinates (r', u'). It may very well be that vertex t is equal to vertex v. We construct a shortest grid path L' from vertex u to vertex t on G' formed by only edges that traverse the sides or interiors of grid cells which are known to be unblocked because they are traversed by L. If vertex t is not equal to vertex v then we divide the straight line from vertex u to vertex v into a piecewise linear sequence of straight-line segments whose endpoints are vertices and iteratively consider each straight-line segment.

Without loss of generality, we consider only the case in which L is in the sector S defined by a single pair of angularly adjacent edges types m_i and m_{i+1}. This can be done because the n edge types divide the tessellated space into n symmetric sectors. A line in one sector can be reflected into any other sector with a series of one or more reflections. To illustrate this consider three angularly adjacent edge types m_i, m_{i+1} and m_{i+2}. A line in the sector defined by m_i and m_{i+1} can be reflected into the sector defined by m_{i+1} and m_{i+2} by reflecting about m_{i+1}. If the angle of L is equal to either a_i or a_{i+1} then the theorem trivially holds. For simplicity, we choose a sector such that L travels up and to the right (that is, r' is greater than or equal to 0 and u' is greater than or equal to 0).

[3] We assigned angles to the members in A with respect to a vertical line, however any reference point can be used.

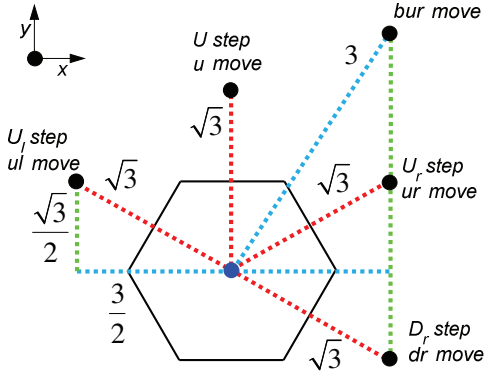
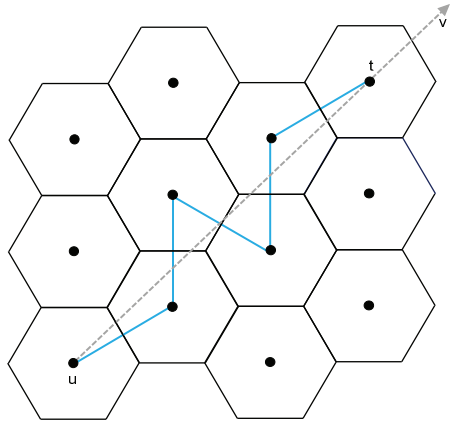
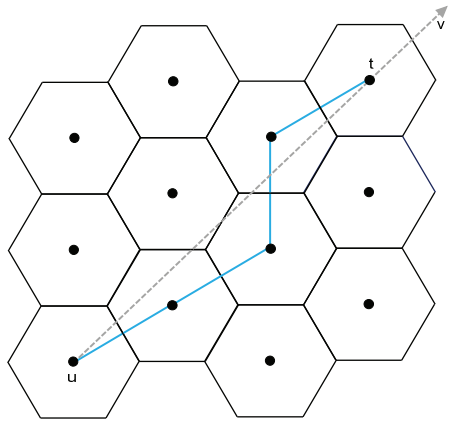
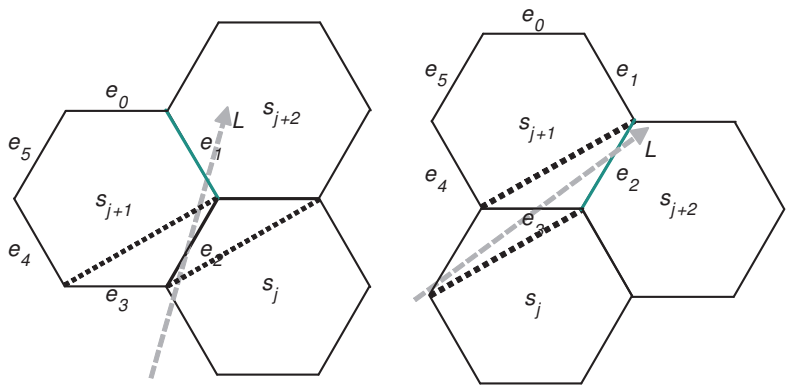
Hexagonal Grids (A)			
	1 Move/Step Lengths	2 Step Sequence	3 Move Sequence

Table 3.4: Hexagonal Grid Step Sequence and Move Sequence for a Hex Graph



(a) ul Move (b) dr Move

Figure 3.3: Replacing Moves in the Move Sequence on Hex Graphs

Hexagonal Grid Step Sequence: *Without loss of generality assume that L is in the sector S defined by m_i = an Up edge and m_{i+1} = an Up Right edge. We convert L into a step sequence $\widehat{L}$ of up left (U_l) steps, up (U) steps, up right (U_r) steps and down right (D_r) steps. U_l steps decrease the x-coordinate by $\frac{3}{2}$ and increase the y-coordinate by $\frac{\sqrt{3}}{2}$, U steps increase the y-coordinate by $\sqrt{3}$, U_r steps increase the x-coordinate by $\frac{3}{2}$ and the y-coordinate by $\frac{\sqrt{3}}{2}$ and D_r steps increase the x-coordinate by $\frac{3}{2}$ and decrease the y-coordinate by $\frac{\sqrt{3}}{2}$ (cell A1 of Table 3.4). We assign coordinates to grid cells, namely the coordinates of their centers. We start with an $\widehat{L}$ initialized to the empty sequence and traverse L from vertex u to vertex t. Whenever L leaves the interior of one grid cell c and enters the interior of another grid cell c', we append one step to $\widehat{L}$ depending on the 4 possible differences between the coordinates of c' and c. We append U_l for $(-\frac{3}{2}, \frac{\sqrt{3}}{2})$, U for $(0, \sqrt{3})$, U_r for $(\frac{3}{2}, \frac{\sqrt{3}}{2})$ and D_r for $(\frac{3}{2}, -\frac{\sqrt{3}}{2})$. The resulting step sequence $\widehat{L}$ moves from vertex u at coordinates $(0, 0)$ to vertex t at coordinates (r', u') and the steps in $\widehat{L}$ traverse the interiors of grid cells that are unblocked because L traverses their interiors.*[4]

Hex Graph Move Sequence Algorithm: *We convert $\widehat{L}$ into a move sequence L' of up left (ul), up (u), up right (ur) and down right (dr) moves (ul and dr moves exist only temporarily). ul moves decrease the x-coordinate by $\frac{3}{2}$, increase the y-coordinate by $\frac{\sqrt{3}}{2}$ and have length $\sqrt{3}$, u moves increase the y-coordinate by $\sqrt{3}$ and have length $\sqrt{3}$, ur moves increase the x-coordinate by $\frac{3}{2}$, increase the y-coordinate by $\frac{\sqrt{3}}{2}$ and have length $\sqrt{3}$, and dr moves increase the x-coordinate by $\frac{3}{2}$, decrease the y-coordinate by $\frac{\sqrt{3}}{2}$ and have length $\sqrt{3}$ (cell A1 of Table 3.4). We start with L' initialized to the empty sequence and then append moves to L' by iterating through the steps in $\widehat{L}$ as follows: If the next step in $\widehat{L}$ is a U_l, U, U_r or D_r then append a ul, u, ur or dr move, respectively. Next, we replace each ul move and the following ur move with a u move and we replace each dr move and the preceding u move with a ur move. The algorithm must ensure that each ul move is followed by a ur move and that the u move that replaces them*

[4] L can pass between two grid cells that share a side and thus L does not traverse either one, but this is a special case that we address later.

traverses unblocked grid cells and that each dr move is preceded by a u move and that the ur move that replaces them traverses unblocked grid cells:

- *We begin by showing that each ul move is followed by a ur move and the u move that replaces them traverses unblocked grid cells. Consider three consecutive grid cells s_j, s_{j+1} and s_{j+2} that are traversed by L. A ul move means that L exits s_j by intersecting e_5 of s_j and exits s_{j+1} by intersecting e_1 of s_{j+1} (after which L enters s_{j+2}). This can be seen in Figure 3.3(a) where the dotted lines have the same angle as an Up Right edge. Consider all 6 sides of s_{j+1} and coordinates on e_2 of s_{j+1}. Only coordinates on e_2 of s_{j+1} (and thus coordinates on e_5 of s_j) can be connected with coordinates on e_1 of s_{j+1} with a line in sector S. Therefore, every ul move is followed by a ur move. Furthermore, the u move traverses the interiors of s_j and s_{j+2} which are known to be unblocked because L traverses them.*

- *We now show that each dr move is preceded by a u move and that the ur move that replaces them traverses unblocked grid cells. Consider three consecutive grid cells s_j, s_{j+1} and s_{j+2} that are traversed by L. A dr move means that L exits s_j by intersecting e_0 of s_j and exits s_{j+1} by intersecting e_2 of s_{j+1} (after which L enters s_{j+2}). This can be seen in Figure 3.3(b) where the dotted lines have the same angle as an Up Right edge. Consider all 6 sides of s_{j+1} and coordinates on e_3 of s_{j+1}. Only coordinates on e_3 of s_{j+1} (and thus coordinates on e_0 of s_j) can be connected with coordinates on e_2 of s_{j+1} with a line in sector S. Therefore, every dr move is preceded by a u move. Furthermore, the ur move traverses the interiors of s_j and s_{j+2} which are known to be unblocked because L traverses them.*

Therefore, the resulting move sequence L' is formed by edges on G' since all moves traverse the interiors of grid cells that are known to be unblocked because they are traversed by L.

An example of Part 1 of the proof for hex graphs can be seen in Table 3.4. Table 3.4 depicts the step and move sequence constructed from L. Cell A2 of Table 3.4 depicts the step sequence $\widehat{L} = (U_R U D_R U U_R)$ and cell A3 of Table 3.4 depicts the move sequence $L' = (ur, ur, u, ur)$ that results from applying the hex graph move sequence algorithm to $\widehat{L}$.

In a hexagonal grid, L can pass between two grid cells that share a side and thus L does not traverse either one. For example, in cell B3 of Table 3.1, a Big Up Right edge passes between two grid cells that share a side. However, we know that, in this case, at least one of these two grid cells must be unblocked because L passes between them and thus L' can be formed by exactly one u move followed by one ur move (or vice vera).

Double Hex Graph Move Sequence Algorithm: *Without loss of generality assume that L is in the sector S defined by m_i = a Big Up Right edge and m_{i+1} = an Up Right edge. We construct a move sequence L' of big up right (bur) and up right (ur) moves. bur moves increase the x-coordinate by $\frac{3}{2}$, increase the y-coordinate by $\frac{3\sqrt{3}}{2}$ and have length 3 and ur moves increase the x-coordinate by $\frac{3}{2}$, increase the y-coordinate by $\frac{\sqrt{3}}{2}$ and have length $\sqrt{3}$ (cell A1 of Table 3.4). We start with an L' constructed using the technique defined above for hex graphs (which contains only u and ur moves) and then replace the moves in L' as follows: each u move and the preceding ur move is replaced with a bur move. The algorithm must ensure that each u move is preceded by a ur move and that the bur move that replaces them traverses unblocked grid cells.*

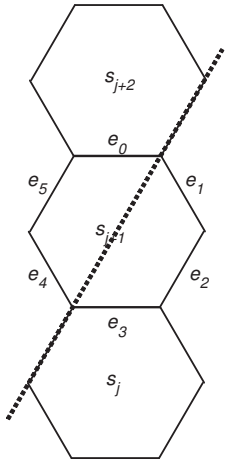


Figure 3.4: Replacing Moves in the Move Sequence on Double Hex Graphs

Each u move is preceded by a ur move due to the following two properties: **(1)** *The first move in an L' constructed using the technique defined for hex graphs cannot be a u move because L is in the sector S defined by m_i = a Big Up Right edge and m_{i+1} = an Up Right edge and thus the first move in L' cannot be a u move.* **(2)** *An L' constructed using the technique defined for hex graphs cannot contain two consecutive u moves. Consider a stack of three hexagonal grid cells such that the center grid cell shares sides e_0 and e_3 with the other two. L cannot traverse all three grid cells and thus L' cannot contain two consecutive u moves. This follows from the fact that coordinates in the interior of the bottom grid cell cannot be connected with coordinates in the interior of the top grid cell with a line segment in sector S, as illustrated in Figure 3.4 where the dotted line has the same angle as a Big Up Right edge. It follows that every u move in L' must be preceded by a ur move. Furthermore, the bur move traverses the sides or interiors of the same three grid cells that the u move and the ur move traverses and those grid cells are known to be unblocked because L traverses them.*

Therefore, the resulting L' is formed by edges on G' since all moves traverse either the sides or interiors of grid cells that are known to be unblocked because they are traversed by L.

3.2.4.3 Part 2: Two Edge Types

Now we move on to Part 2 of the proof. L' is always formed by only two angularly adjacent edge types. For any line segment L such that the angle of L is between a_i and a_{i+1}, an L' formed by only edge types m_i and m_{i+1} must be a shortest grid path from vertex u to vertex t on G'. This follows directly from the fact that the vector components of m_i and m_{i+1} in the direction of L must be greater, per unit distance, than the vector components of any other edge types in the direction of L. Therefore, if L' can be formed by only edge types m_i and m_{i+1} then it must be a shortest grid path from vertex u to vertex t on G'.

Hex Graphs (Part 2): *In Part 1, we constructed an unblocked path L' from vertex u to vertex t on G' that contained only Up edges (u moves) and Up Right edges (ur moves). These two edge types are angularly adjacent. Furthermore, L is in the sector S defined by an Up edge and an Up Right edge and thus L' must be a shortest grid path from vertex u to vertex t on G'.*

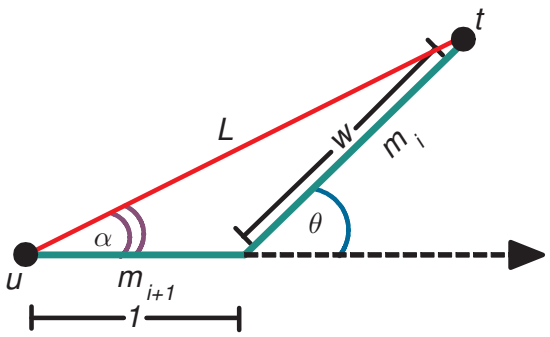


Figure 3.5: Scaling Method

Double Hex Graphs (Part 2): *In Part 1, we constructed an unblocked path L' from vertex u to vertex t on G' that contained only Big Up Right edges (bur moves) and Up Right edges (ur moves). These two edge types are angularly adjacent. Furthermore, L is in the sector S defined by a Big Up Right edge and an Up Right edge and thus L' must be a shortest grid path from vertex u to vertex t on G'.*

3.2.4.4 Part 3: Scaling Method

Now we move on to Part 3 of the proof. Our Scaling Method is a general technique for calculating a bound on the worst case ratio between a path L' consisting of at most two edge types and a line segment L. We can rearrange the edges in L' such that all the edges of one type occur before all the edges of the other type without affecting the length of L'. Therefore, all the edges of one type form one line segment as do the edges of the other type. Without loss of generality, assume that one line segment is horizontal and the second is at an angle θ to the horizontal (Figure 3.5). Furthermore, for a given vertex t, without loss of generality, by scaling, we make the following assumption: the horizontal segment is of unit length and it is followed by a line segment of some length w at an angle θ from the horizontal (Figure 3.5). The ratio R that we want to maximize is:

$$R = \frac{d'(u,t)}{d(u,t)} = \frac{1+w}{\sqrt{1+w^2+2w\cos(\theta)}}. \tag{3.1}$$

This ratio is maximized when the derivative with respect to w equals 0: $0 = \frac{1+w\cos(\theta)-w-\cos(\theta)}{(1+w^2+2\,w\cos(\theta))^{3/2}}$. Solving this equation for w we obtain $w = 1$. Substituting this back into Equation 3.1 yields:

$$R = \frac{2}{\sqrt{2+2\,\cos(\theta)}}. \tag{3.2}$$

We determine the value of R by substituting $\theta = a_{i+1} - a_i$ into Equation 3.2.

Hex Graphs (Part 3): *For hex graphs $\theta = \frac{\pi}{3}$ and Equation 3.2 yields $R = \frac{2\sqrt{3}}{3}$.*

Double Hex Graphs (Part 3): *For double hex graphs* $\theta = \frac{\pi}{6}$ *and Equation 3.2 yields* $R = \frac{2}{\sqrt{2+\sqrt{3}}}$.

We now determine whether the bound for a particular type of grid graph is tight or asymptotically tight. Let the length of the horizontal moves be h instead of 1. Let X be the set of the absolute values of the differences between the x-coordinate of a vertex and the x-coordinates of its n adjacent vertices. Let Y be the set of the absolute values of the differences between the y-coordinate of a vertex and the y-coordinates of its n adjacent vertices. Let x' and y' be the smallest members in X and Y, respectively. The x-coordinate and y-coordinate of any vertex on the grid graph is a multiple of x' and y', respectively, however vertices do not reside at coordinates $(a \cdot x', b \cdot y')$ for all integer values of a and b. For simplicity, we assume that vertices do reside at coordinates $(a \cdot x', b \cdot y')$ for all integer values of a and b. This does not affect asymptotically tight bounds because if we can demonstrate that a bound is asymptotically tight on a grid graph with a given set of vertices then that bound is also asymptotically tight on a grid graph with fewer vertices. This does not affect our tight bounds because we provide examples of (valid) paths. Consider a grid graph that was constructed from a regular grid with no blocked grid cells where the start vertex is at coordinates $(0, 0)$ and the goal vertex is at coordinates $(\lceil \frac{h+h\cos(\theta)}{x'} \rceil x', \lceil \frac{h\sin(\theta)}{y'} \rceil y')$. As $\lim h \to \infty \frac{d'(u,t)}{d(u,t)}$ either reduces to R or approaches, but never reaches R. For tri graphs, double tri graphs, quad graphs and hex graphs as $\lim h \to \infty$, $\frac{d'(u,t)}{d(u,t)}$ reduces to R. On these four types of grid graphs if L' is formed by one edge of type m_i and one edge of type m_{i+1} (or vice versa) then $\frac{d'(u,t)}{d(u,t)} = R$. For octile graphs and double hex graphs as $\lim h \to \infty$, $\frac{d'(u,t)}{d(u,t)}$ approaches, but never reaches R.

Hex Graphs: *From cell A1 of Table 3.4 we can see that* $x' = \frac{3}{2}$, $y' = \frac{\sqrt{3}}{2}$, $\theta = \frac{\pi}{3}$ *and therefore* $\frac{d'(u,t)}{d(u,t)} = \frac{2h}{\sqrt{\frac{3}{4}\lceil h\sqrt{3}\rceil^2 + \frac{9}{4}\lceil \frac{h\sqrt{3}}{3}\rceil^2}}$. *For* h *equal to* $\sqrt{3}$*, which is the length of both an Up edge and an Up Right edge,* $\frac{d'(u,t)}{d(u,t)} = R$*. Thus, the bound is tight for an* L' *formed by an Up edge followed by an Up Right edge (or vice versa).*

Double Hex Graphs: *From cell A1 of Table 3.4 we can see that* $x' = \frac{3}{2}$, $y' = \frac{\sqrt{3}}{2}$, $\theta = \frac{\pi}{6}$ *and therefore* $\frac{d'(u,t)}{d(u,t)} = \frac{2}{\sqrt{\frac{\lceil \frac{h}{2} \rceil^2}{h^2} + \frac{\lceil \frac{1}{2}h\sqrt{3}+h \rceil^2}{h^2}}}$. *As* $\lim h \to \infty \frac{d'(u,t)}{d(u,t)}$ *approaches* R*, however, there is no value of* h *that causes* $\frac{d'(u,t)}{d(u,t)}$ *to evaluate to* $\frac{2}{\sqrt{2+\sqrt{3}}}$.

□

Table 3.3 contains the values of θ that maximize R, the floating point approximation of R and the type of bound for each type of grid graph. There is a strong correlation between the branching factor of a grid graph and the value of R. As the branching factor increases the value of R decreases, as expected.

3.2.5 Geometric Relationships: 2D Regular Grids

In this section, we analyze the geometric relationship of L and L'. The worst case ratio occurs when the length of the line segment corresponding to the edges of type m_i, namely w, is equal to the length of the line segment corresponding to the edges of type m_{i+1} (which was arbitrarily set to 1). Consider two angularly adjacent edge types m_i and m_{i+1}, and their respective angles

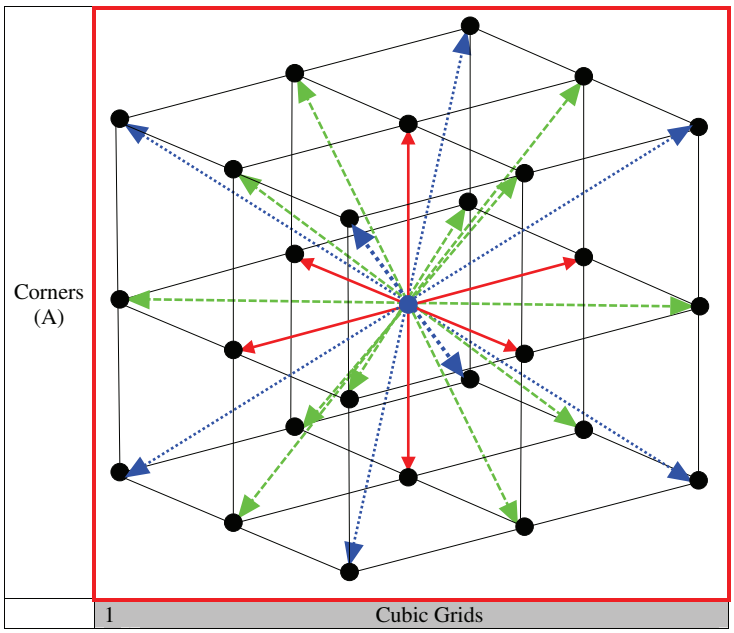


Table 3.5: Grid Graphs Constructed from 3D Regular Grids

a_i and a_{i+1}. Let m_{i+1} be horizontal edges and m_i be edges at some angle θ from m_{i+1}. Finally, let α be the smaller angle between L and the horizontal line segment (Figure 3.5). If w equals 0, then Equation 3.1 reduces to 1 and L' can trivially be formed by a sequence of edges of type m_{i+1}. If w is greater than 0, then L' must contain one or more edges of type m_i. As w increases toward 1, both α and R increase monotonically as more edges of type m_i are included in L'. The sum of the three internal angles of any triangle is π and the angles opposite the sides of an isosceles triangle with the same lengths must be equal, thus, when $w = 1$, $2 \cdot \alpha + (\pi - \theta) = \pi$ and therefore $\alpha = \frac{\theta}{2} = \frac{a_{i+1} - a_i}{2}$.

Thus, the angle α of L is precisely halfway between the angles of the two angularly adjacent edge types m_i and m_{i+1} that compose L'. For every type of grid graph, the ratio of the lengths of L' and L is maximized when α is halfway between a_i and a_{i+1}. Table 3.3 contains the values of α that maximize R for each type of grid graph. We say that for this value of α, L is diverging from L' as much as possible.

3.2.5.1 Double Tri Graphs and Hex Graphs

The value of R for hex graphs with vertices placed in the centers of grid cells is the same as the value of R for double tri graphs with vertices placed in the corners of grid cells. There is a simple geometric reason for this. The edges of a hex graph with vertices placed in the centers of grid cells define a double tri graph with vertices placed in the corners of grid cells. The angles of the 6 edge types on double tri graphs are the same as the angles of the 6 edge types on hex graphs and thus the value of R must also be the same.

	Corners			Centers		
	Realism	**Speed**	**Data Structure**	**Realism**	**Speed**	**Data Structure**
Cubic Graphs	Yes	Yes (1)	Yes	Yes	Yes (1)	Yes
Double Cubic Graphs	Yes	No $(1, \sqrt{2})$	Yes	Yes	No $(1, \sqrt{2})$	Yes
Triple Cubic Graphs	Yes	No $(1, \sqrt{2}, \sqrt{3})$	Yes	Yes	No $(1, \sqrt{2}, \sqrt{3})$	Yes

Table 3.6: Properties of Grid Graphs Constructed from 3D Regular Grids

	R	***F***	***Type***
Cubic Graphs	$\sqrt{3}$	≈ 1.73	*Tight*
Triple Cubic Graphs	$\sqrt{9 - 2\sqrt{2} - 2\sqrt{2\sqrt{3}}}$	≈ 1.13	*Asymptotically Tight*

Table 3.7: Path Length Analysis Results for Grid Graphs Constructed from 3D Regular Grids

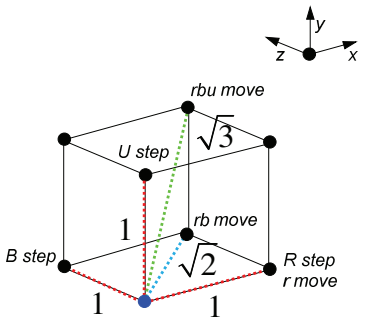
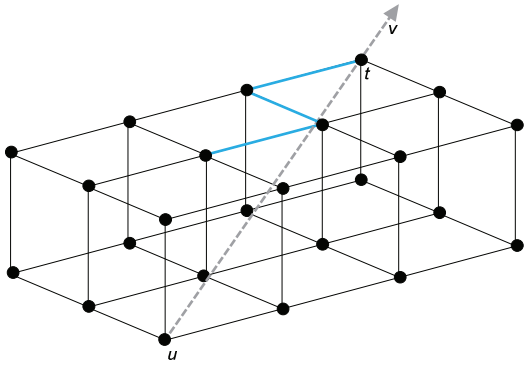
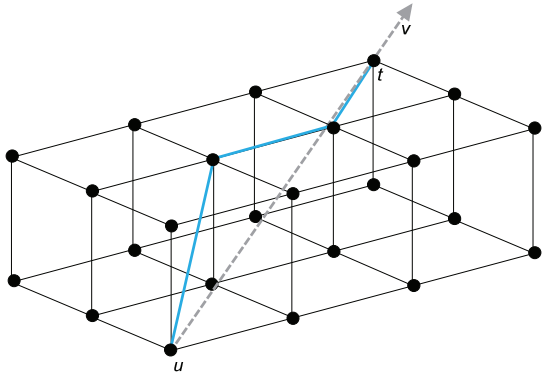
Cubic Grids (A)			
	1 Move/Step Lengths	2 Step Sequence	3 Move Sequence

Table 3.8: Cubic Grid Step Sequence and Move Sequence for a Triple Cubic Graph

3.3 3D Regular Grids

In 3D, roboticists and video game developers often solve the generate-graph problem by discretizing continuous 3D environments into regular grids composed of cubes (cubic grid) (Carsten et al., 2006; Cohen et al., 2011). This is the only type of regular polyhedron that can be used to tessellate continuous 3D environments (Gosset, 1900). Hexagons and triangles with a height parameter, that is, prismatic hexagons and triangles, can be used to tessellate continuous 3D environments as well, but they are not regular polyhedrons. Therefore, we only examine cubic grids.

3.3.1 Notation and Definitions

In this section, we explain how a grid graph $G' = (V, E')$ is constructed from a continuous 3D environment. A continuous 3D environment is discretized into a cubic grid by laying the cubic grid over the continuous 3D environment and labeling grid cells as blocked iff they contain part of an obstacle. We assume that the length of every side of every grid cell is one. We examine three types of grid graphs constructed from cubic grids, namely cubic graphs, double cubic graphs, and triple cubic graphs. Vertices $u \in V$ are placed in either the centers of grid cells or the upper right back corners of grid cells. Edges $(u, v) \in E'$ connect vertex u to each of its n adjacent vertices, thus defining an n-neighbor grid graph. The value of n determines the branching factor of the grid graph. Let x_e, y_e and z_e be the differences between the x-coordinate, y-coordinate and z-coordinate, respectively, of the two endpoints of an edge e. For grid graphs constructed from cubic grids an edge type is a class of edges such that every edge e in that class has the same values for x_e, y_e and z_e. The outgoing edges of a vertex u are the set of edges that have vertex u as one endpoint. For **cubic graphs** the n outgoing edges of a vertex all have length 1. There are six such edges and thus a cubic graph is a 6-neighbor cubic grid graph. For **double cubic graphs** the n outgoing edges of a vertex have either length 1 or $\sqrt{2}$. There are 18 such outgoing edges and thus a double cubic graph is an 18-neighbor cubic grid graph. For **triple cubic graphs** the n outgoing edges of a vertex have either length 1, $\sqrt{2}$ or $\sqrt{3}$. There are 26 such edges and thus a triple cubic graph is a 26-neighbor cubic grid graph. For all three types of grid graphs, edges that connect vertices that do not have line-of-sight are removed from E' (line-of-sight is defined in Section 1.1.2). The length of an edge $(u, v) \in E'$ is $c(u, v)$, which is the Euclidean distance between vertices u and v. Examples of each type of grid graph when vertices are placed in the upper right back corners of grid cells can be found in Table 3.5. The blue vertex represents a vertex $u \in V$ and the arrows depict the outgoing edges that connect vertex u to its n adjacent vertices. In cell A1 of Table 3.5, the solid red arrows show the 6 outgoing edges that connect a vertex u to its 6 adjacent vertices in a cubic graph, the union of the solid red and dashed green arrows show the 18 outgoing edges that connect u to its 18 adjacent vertices in a double cubic graph and the union of the solid red, dashed green and dotted blue arrows show the 26 outgoing edges that connect vertex u to its 26 adjacent vertices in a triple cubic graph. We label the six faces of a grid cell c. Let x_c, y_c and z_c be the largest x-coordinate, y-coordinate and z-coordinate among all of the corners of c, respectively and x'_c, y'_c and z'_c be the smallest x-coordinate, y-coordinate and z-coordinate among all of the corners of c, respectively. The right face is defined by the four corners of c whose x-coordinate is equal to x_c; the left face is defined by the four corners of c whose x-coordinate is equal to x'_c; the top face is defined by the four corners of c whose y-coordinate is equal to y_c; the

bottom face is defined by the four corners of c whose y-coordinate is equal to y'_c; the back face is defined by the four corners of c whose z-coordinate is equal to z_c and the front face is defined by the four corners of c whose z-coordinate is equal to z'_c.

3.3.2 Grid Graph Properties

In this section, we show that, how one constructs a grid graph from a regular grid does *not* have a significant impact on the aspects of a path-planning system discussed in Section 3.2.3.

We consider 6 different types of grid graphs, namely cubic graphs, double cubic graphs and triple cubic graphs when placing vertices in either the centers or upper right back corners of grid cells. We evaluate each type of grid graph with respect to the Realism, Speed and Data Structure Properties, which are defined in Section 3.2.3. A summary of these results is given in Table 3.6, which is annotated in the same manner as Table 3.2.

In Section 3.2.3, we demonstrated that grid graphs constructed from square grids have the exact same evaluation with respect to the Realism, Speed and Data Structure Properties whether vertices are placed in the corners or centers of grid cells. Grid graphs constructed from cubic grids are similar and thus the Realism, Speed and Data Structure Properties are not affected by whether vertices are placed in the centers or corners of grid cells (be it upper right back corners or any other corners of grid cells). Therefore, we only examine cubic graphs, double cubic graphs and triple cubic graphs with vertices placed in the corners of grid cells (the analysis of these three grid graphs with vertices placed in the centers of grid cells is similar). The Realism Property is satisfied for each type of grid graph because the n edge types of the n outgoing edges of every vertex are the same. The Speed Property is satisfied for cubic graphs because each edge coincides with the sides of a grid cell (and thus a regular polyhedron). The Speed Property is not satisfied for double cubic graphs and triple cubic graphs because the outgoing edges of a vertex either coincide with the sides of grid cells or traverse the interiors of grid cells. The Data Structure Property is satisfied for each type of grid graph because we place vertices in the upper right back corner of each grid cell and thus each vertex is in the upper right back corner of the grid cell that it maps to.

Our path length analysis examines only two of the six types of grid graphs, namely cubic graphs and triple cubic graphs with vertices placed in the upper right back *corners* of grid cells. In Table 3.5, these types of grid graphs are highlighted in red. In Section 3.2.4, we examined two different branching factors for triangular grids, square grids and hexagonal grids and thus we examine two different branching factors for cubic grids as well. We examine cubic graphs because they satisfy the Realism, Speed and Data Structure Properties. We examine triple cubic graphs because they allow us to demonstrate that our path length analysis technique can be applied to grid paths containing edges of several different edge types. We begin with triple cubic graphs and discuss cubic graphs in Section 3.3.5.

3.3.3 Path Length Analysis

We now determine how much longer shortest grid paths formed by the edges of grid graphs constructed from regular grids that discretize continuous 3D environments can be than true shortest paths. To prove this result, as we did in continuous 2D environments, we examine any-angle paths. In continuous 3D environments, the relationship between the lengths of true shortest paths

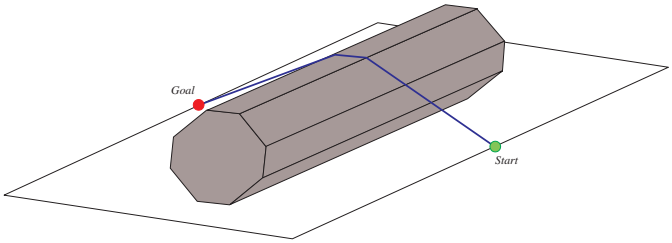
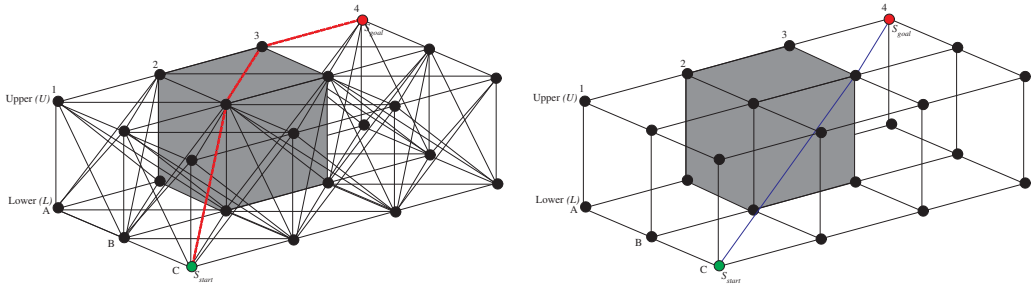


Figure 3.6: True Shortest Path in a Continuous 3D Environment



(a) Shortest Grid Path

(b) Shortest Any-Angle Path

Figure 3.7: Paths on Triple Cubic Graphs with Vertices Placed in Grid Cell Corners

and the lengths of shortest any-angle paths is similar to what it was in continuous 2D environments, in that true shortest paths are at most as long as shortest any-angle paths. However, unlike true shortest paths in continuous 2D environments, which only contain heading changes at the corners of blocked polygons, true shortest paths in continuous 3D environments can contain heading changes at both the corners and sides of blocked polyhedrons. This can be seen in Figure 3.6, in which a true shortest path between the two spheres does not contain any vertices of the polyhedral obstacle. In continuous 3D environments, the relationship between the lengths of shortest any-angle paths and the lengths of shortest grid paths is the same as it was in continuous 2D environments in that shortest any-angle paths are at most as long as shortest grid paths. For example, in triple cubic graphs, shortest grid paths can be longer than shortest any-angle paths, which can be seen in Figure 3.7. We now prove that shortest grid paths formed by the edges of cubic graphs and triple cubic graphs can be up to $\approx 73\%$ and $\approx 13\%$ longer than shortest any-angle paths, respectively and thus can be at least $\approx 73\%$ and $\approx 13\%$ longer than true shortest paths, respectively.

Grid paths formed by the edges of cubic graphs and triple cubic grid graphs are piecewise linear and thus Theorem 1 can be used to bound the ratio between shortest grid paths formed by the edges of a cubic graph or a triple cubic graph and shortest any-angle paths by updating Lemma 1.[5] As was the case for Lemma 1, Lemma 2 is a special case of Theorem 1 where P is a single edge and thus a straight line. However, the proof of Lemma 2 differs from the proof of Lemma 1 because shortest grid paths formed by the edges of cubic graphs or triple cubic graphs can contain more than 2 different edge types. Thus, Part 2 of Lemma 1 does not apply to Lemma

[5]The statements of Lemmata 1 and 2 are identical.

2 and we must use a different technique from the one that we used in Part 3 of Lemma 1. The proof of Lemma 2 contains two parts: **Part 1:** We show that P' can be formed by only edges that traverse the sides, faces or interiors of grid cells that are known to be unblocked because they are traversed by P. This allows us to determine the length $d'(u, v)$ of P' for a given length $d(u, v)$ of P independent of which grid cells are blocked (analogous to Part 1 of Lemma 1). **Part 2:** We maximize the ratio of the lengths of P' and P (analogous to Part 3 of Lemma 1). Table 3.7 contains the values of R, the floating point approximation of R and the type of bound for cubic graphs and triple cubic graphs.

Lemma 2. *If $(u, v) \in E$, then $d'(u, v) \leq R \cdot d(u, v)$, where R is a constant determined by the type of grid graph G'.*

Proof.

3.3.3.1 Part 1: Unblocked Path

We begin with Part 1 of the proof. Assume, without loss of generality, that vertex u is at the origin, that is, vertex u is at coordinates $(0, 0, 0)$. If the edge $(u, v) \in E$ can be formed by edges in E', then the theorem trivially holds. Otherwise, consider the prefix L of the straight line from vertex u to vertex v that ends at the first reached vertex (that is, corner of a grid cell), which we call vertex t at coordinates (r', u', b'). It may very well be that vertex t equals vertex v. If vertex t does not equal vertex v then we divide the straight line from vertex u to vertex v into a piecewise linear sequence of straight-line segments whose endpoints are vertices and iteratively consider each straight-line segment.

Cubic Grid Step Sequence: Without loss of generality one can assume $r' \geq b' \geq u'$. The case in which u' equals 0 is the 2D case that we have already analyzed and the case in which r' equals b' is trivial. Therefore, we assume that $r' > b' \geq u' \geq 1$.[6] We convert L into a step sequence $\widehat{L}$ of right (R), back (B) and up (U) steps. R steps increase the x-coordinate by 1, B steps increase the z-coordinate by 1 and U steps increase the y-coordinate by 1 (cell A1 of Table 3.8). We assign coordinates to grid cells, namely the coordinates of their upper right back corners (that is, the corners that are furthest from the origin). We start with $\widehat{L}$ initialized to the empty sequence and traverse L from vertex u to vertex t. Whenever L leaves the interior of one grid cell c and enters the interior of another grid cell c', we append one or two steps to $\widehat{L}$ depending on the 6 possible differences between the coordinates of c' and c. We append $\underline{\text{R}}$ for $(1, 0, 0)$, $\underline{\text{B}}$ for $(0, 0, 1)$, $\underline{\text{U}}$ for $(0, 1, 0)$, B$\underline{\text{R}}$ for $(1, 0, 1)$, R$\underline{\text{U}}$ for $(1, 1, 0)$ and B$\underline{\text{U}}$ for $(0, 1, 1)$. The resulting $\widehat{L}$ moves from coordinates $(1, 1, 1)$ to vertex t at coordinates (r', u', b'). The grid cell with coordinates $(1, 1, 1)$ and all grid cells with coordinates that are reached after each underlined step are unblocked because L traverses their interiors. The arguments in the remainder of this section are generic to both underlined steps and non-underlined steps unless otherwise stated. Thus, for simplicity we refer to steps generically without underlines. $\widehat{L}$ always has at least one R between two Bs (or the start and a B) and two Us (or the start and a U), and at least one B between two Us (or the start and a U), and starts and ends with an R (**Step Sequence Property**). All of the relationships defined in the Step Sequence Property can be shown using the parametric equation of a line:

[6] If $r' = b' \geq u' \geq 1$ then L has a slope of 1 in the x-z plane. Therefore, the difference between the coordinates of any pair of consecutive vertices on a shortest grid path from vertex u to vertex t on G' is either $(1, 0, 1)$ or $(1, 1, 1)$. It follows that a shortest grid path from vertex u to vertex t on G' is composed of only rb moves and rbu moves, which we define later in this section.

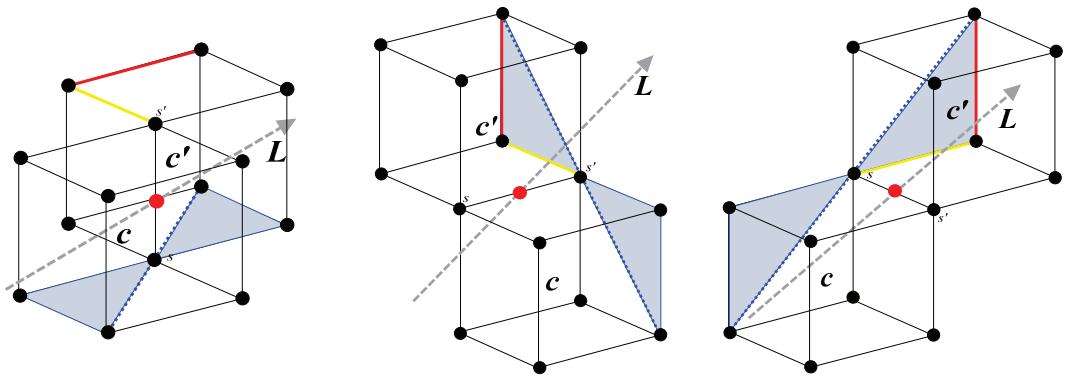


Figure 3.8: BR (left), BU (middle) and RU (right) Steps

$$x = x_0 + m \cdot r' \tag{3.3}$$

$$y = y_0 + m \cdot u' \tag{3.4}$$

$$z = z_0 + m \cdot b' \tag{3.5}$$

In Equations 3.3, 3.4 and 3.5 $x_0 = y_0 = z_0 = 0$ because vertex u is at coordinates $(0, 0, 0)$ and $0 \leq m \leq 1$ is an independent variable and therefore:

$$x = m \cdot r' \tag{3.6}$$

$$y = m \cdot u' \tag{3.7}$$

$$z = m \cdot b' \tag{3.8}$$

By combining Equations 3.7 and 3.8, we obtain $z = y\frac{b'}{u'}$ and, since $\frac{b'}{u'} \geq 1$, there must be at least one B between two Us. Clearly, if this relationship holds between Us and Bs it also holds between Bs and Rs and Us and Rs because $r' > b' \geq u' \geq 1$. $\widehat{L}$ must end in an R because vertex t is at the corner furthest from the origin of the last grid cell traversed by L and only coordinates on the right face of the second to last grid cell traversed by L can be connected with vertex t such that $r' > b' \geq u' \geq 1$. The same argument can be used to show that $\widehat{L}$ must begin with an R. There must be at least one R between the start and a U and the start and a B because $\widehat{L}$ begins in an R. There must be at least one B between the start and a U because $\frac{b'}{u'} \geq 1$.

Additional arguments are necessary to ensure that the Step Sequence Property holds when we append a BR, BU or RU to $\widehat{L}$. This is because it is not clear that the Step Sequence Property holds even though we never append an RB, UB or UR to $\widehat{L}$ in place of a BR, BU or RU, respectively. To demonstrate this we show the following about $\widehat{L}$: **(1)** A B cannot immediately precede a BR, that is, there must be an R between a B and a BR. **(2)** A U cannot immediately follow a BU, that is, there must be a B between a BU and a U. **(3)** A U cannot immediately follow an RU, that is, there must be an R between an RU and a U. When demonstrating that the Step Sequence Property holds when BRs, BUs, and RUs are appended, we ignore Us, Rs, and Bs, respectively. For BRs,

the number of Bs and Rs between two Us (or the start and a U) is the same whether a BR or an RB is appended to $\widehat{L}$. For BUs, the number of Bs and Us between two Rs (or the start and an R) is the same whether a BU or a UB is appended to $\widehat{L}$. For RUs, the number of Rs and Us between two Bs (or the start and a B) is the same whether an RU or a UR is appended to $\widehat{L}$. Therefore, in order to show that the Step Sequence Property holds when we append BR, BU or RU to $\widehat{L}$ we can examine the projection of L onto the x-z plane (ignore Us), y-z plane (ignore Rs) and x-y plane (ignore Bs), respectively. In Figure 3.8 left, middle and right the x-z plane is the bottom (or top) face of c and c', the y-z plane is the right (or left) face of c and c' and the x-y plane is the back (or front) face of c and c', respectively. Furthermore, we know that when two steps are appended to $\widehat{L}$ it implies that L intersects the side of a grid cell that is shared by c and c'. If a BR, BU or RU is appended to $\widehat{L}$ then L must intersect a side that is parallel to the y, x and z axis, respectively. In Figure 3.8, the red dots denote a potential coordinate where L intersected a side, defined by two coordinates s and s', which is shared by c and c'. Finally, because $r' > b' \geq u' \geq 1$ we know that for a BR, BU and RU step the projection of L onto the x-z plane, y-z plane and x-y plane (respectively) must lie within the shaded region in Figure 3.8 left, middle and right (respectively) where the dotted blue line denotes $r' = b'$, $b' = u'$, and $r' = u'$ (respectively). Therefore, we know the following: **(1)** A B cannot immediately precede a BR. $r' > b'$ and thus the shaded region touches the left face of c, but does not touch the front face of c (thus there must be an R between a B and a BR). This is depicted in Figure 3.8 (left). **(2)** A U cannot immediately follow a BU. $b' \geq u'$ and thus the shaded region touches the back face of c' and one point on the top face of c' (thus there must be a B between a BU and a U, even if the next B is the first step in a BU step). This is depicted in Figure 3.8 (middle). **(3)** A U cannot immediately follow an RU. $r' > u'$ and thus the shaded region touches the right face of c' and does not touch the top face of c' (thus, there must be an R between an RU and a U). This is depicted in Figure 3.8 (right). It follows that the Step Sequence Property holds when a BR, BU or RU step is appended to $\widehat{L}$.

Triple Cubic Graph Move Sequence Algorithm: We convert $\widehat{L}$ into a move sequence L' of right (r), right back (rb) and right back up (rbu) moves. r moves increase the x-coordinate by one and have length 1, rb moves increase the x-coordinate by 1, increase the z-coordinate by 1 and have length $\sqrt{2}$ and rbu moves increase the x-coordinate by 1, increase the y-coordinate by 1, increase the z-coordinate by 1 and have length $\sqrt{3}$ (cell A1 of Table 3.8). We start with an L' that contains a single rbu move and use the move sequence algorithm (below) to append moves to L' (the underlined statements are redundant and could be removed):

1. Set $storeR := 0$ and set $storeB := 0$
2. IF the next step in $\widehat{L}$ is R THEN (IF $storeR = 1$ THEN append r and set $storeR := 1$ ELSE set $storeR := 1$)
3. ELSE (IF the next step in $\widehat{L}$ is B THEN (IF $storeR = 1$ and $storeB = 1$ THEN append rb and set $storeR := 0$ and set $storeB := 1$ ELSE set $storeB := 1$))
4. ELSE (IF the next step in $\widehat{L}$ is U THEN (IF $storeR = 1$ and $storeB = 1$ THEN append rbu and set $storeR := 0$ and set $storeB := 0$ ELSE append rbu and set $storeR := 0$ and set $storeB := 0$ and delete the step after U (an R) from $\widehat{L}$))
5. Delete the step just processed from $\widehat{L}$

6. IF there are steps remaining in $\widehat{L}$ THEN go to 2 ELSE (IF $storeR = 1$ and $storeB = 1$ THEN append rb and set $storeR := 0$ and set $storeB := 0$ ELSE (IF $storeR = 1$ THEN append r and set $storeR := 0$))

The algorithm does not cover some cases because they are impossible. First, if the next step in $\widehat{L}$ is B, then it cannot be that $storeR = 0$ and $storeB = 1$. Only a B without any Rs or Us afterwards can result in $storeR = 0$ and $storeB = 1$ (because Rs result in $storeR = 1$, Us result in $storeB = 0$ and only Bs result in $storeB = 1$). But, the next step in $\widehat{L}$ cannot be B due to the Step Sequence Property. Second, if the next step in $\widehat{L}$ is U, then it cannot be that $storeB = 0$. Only a U (or the start of $\widehat{L}$) followed by zero or more Rs can result in $storeB = 0$ (because Bs result in $storeB = 1$). But, the next step in $\widehat{L}$ cannot be U due to the Step Sequence Property. Third, if there are no steps remaining in $\widehat{L}$, then it cannot be that $storeR = 0$ and $storeB = 1$, because $\widehat{L}$ ends in R and thus either $storeR = 1$ (if the algorithm processes R on Line 2) or $storeR = 0$ and $storeB = 0$ (if the algorithm processes R on Line 4).

The algorithm has one requirement (as stated in the algorithm). If the next step in $\widehat{L}$ is U and it is not the case that $storeR = 1$ and $storeB = 1$, then the step after U in $\widehat{L}$ must be R. This follows from the fact that if the next step in $\widehat{L}$ is U, then we have already shown that $storeB = 1$, which implies $storeR = 0$. Thus, the step before U in $\widehat{L}$ must have been B (because Rs result in $storeR = 1$, Us result in $storeB = 0$ and only Bs result in $storeB = 1$). Thus, the step after U in $\widehat{L}$ must be R since it cannot be B or U due to the Step Sequence Property.

By design, the algorithm obeys the **Position Property** at the end of each iteration: Consider the grid cell with the coordinates reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far. Decrease its x-coordinate by one iff $storeR = 1$. Decrease its z-coordinate by one iff $storeB = 1$. The move sequence L' created so far moves from coordinates $(0, 0, 0)$ to exactly these coordinates. In particular, at the end of the last iteration L' moves from vertex u at coordinates $(0, 0, 0)$ to vertex t at coordinates (r', u', b'). It is a shortest grid path from vertex u to vertex t on G' due to the Unblocked Path Property and the Shortest Path Property. **Unblocked Path Property:** L' is formed by edges on G' since all moves either traverse the sides, the faces or the interior of the grid cell with coordinates $(1, 1, 1)$ or grid cells with coordinates that are reached after each underlined step, which are known to be unblocked. We prove the Position Property and Unblocked Path Property in Appendix C.2. **Shortest Path Property:** L connects vertex u at coordinates $(0, 0, 0)$ and vertex t at coordinates (r', u', b') and thus L must travel at least r' units in the x direction, u' units in the y direction and b' units in the z direction. Therefore, the move sequence algorithm that constructed the move sequence L' that connects vertex u to vertex t must have processed at least r' Rs, b' Bs and u' Us. This is precisely the number of each type of step in $\widehat{L}$ and, thus, if the moves in L' combine steps into moves in the most efficient manner possible then L' must represent a shortest grid path from vertex u to vertex t on G'. This is precisely what the move sequence algorithm does. Every U in $\widehat{L}$ corresponds to an rbu move in L'. When the algorithm processes the U, it creates the rbu move. Every B in $\widehat{L}$ that does not correspond to an rbu move corresponds to an rb move in L'. When the algorithm processes the B, it either creates the rb move or sets $storeB := 1$ and later creates the rb move. Every R in $\widehat{L}$ that does not correspond to an rbu move or an rb move corresponds to an r move in L'. When the algorithm processes the R, it either creates the r move or sets $storeR := 1$ and later creates the r move. Thus, the sum of the lengths of the moves in L' is $\sqrt{3}u' + \sqrt{2}(b' - u') + (r' - b')$, and no path from vertex u to vertex t on G' can be shorter. Using this equation for the length of a shortest grid

path from vertex u to vertex t on G', we can verify that it is composed of the minimum number of each type of step: $u' + (b' - u') + (r' - b') = r'$ Rs, $u' = u'$ Us, $u' + (b' - u') = b'$ Bs.

An example of Part 1 of the proof for triple cubic graphs can be seen in Table 3.8. Table 3.8 depicts the step sequence and move sequence constructed from L. Cell A2 of Table 3.8 depicts the step sequence $\widehat{L} = (RBR)$ and cell A3 of Table 3.8 depicts the move sequence $L' = (rbu, r, rb)$ that results from applying the move sequence algorithm for triple cubic graphs to $\widehat{L}$.

3.3.3.2 Part 2: Lagrange Method

Now we move on to Part 2 of the proof. Let x_3 be the number of rbu moves (whose length is $\sqrt{3}$), x_2 be the number of rb moves (whose length is $\sqrt{2}$) and x_1 be the number of r moves (whose length is 1). We now maximize the ratio of the lengths of L' and L, that is $\frac{d'(u,t)}{d(u,t)}$, by using Lagrange Multipliers. Thus, we want to minimize $f(x_1, x_2, x_3)$ subject to $g(x_1, x_2, x_3) = c$ for some constant c, where $f(x_1, x_2, x_3) = \sqrt{(x_1 + x_2 + x_3)^2 + (x_2 + x_3)^2 + x_3^2} = d(u, t)$ and $g(x_1, x_2, x_3) = x_1 + \sqrt{2}x_2 + \sqrt{3}x_3 = d'(u, t)$, resulting in:

$$L(x_1, x_2, x_3, \lambda) = f(x_1, x_2, x_3) + \lambda(g(x_1, x_2, x_3) - c). \tag{3.9}$$

We remove the square root from the minimization because it is a monotonic function. $\nabla L = 0$ then implies the system of equations:

$$\frac{\partial L}{\partial x_1} = 0: \quad 2x_1 + 2x_2 + 2x_3 = -\lambda \tag{3.10}$$

$$\frac{\partial L}{\partial x_2} = 0: \quad 2x_1 + 4x_2 + 4x_3 = -\lambda\sqrt{2} \tag{3.11}$$

$$\frac{\partial L}{\partial x_3} = 0: \quad 2x_1 + 4x_2 + 6x_3 = -\lambda\sqrt{3}. \tag{3.12}$$

From Equations 3.10, 3.11 and 3.12 we get $x_3 = \frac{1}{2}(\sqrt{3} - \sqrt{2})(-\lambda)$, $x_2 = \frac{1}{2}(2\sqrt{2} - \sqrt{3} - 1)(-\lambda)$ and $x_1 = \frac{1}{2}(2 - \sqrt{2})(-\lambda)$. The worst case ratio of the lengths of L' and L is:

$$\begin{aligned} \frac{d'(u,t)}{d(u,t)} &= \frac{x_1 + \sqrt{2}x_2 + \sqrt{3}x_3}{\sqrt{(x_1 + x_2 + x_3)^2 + (x_2 + x_3)^2 + x_3^2}} \\ &= \frac{2 - \sqrt{2} + \sqrt{2}(2\sqrt{2} - \sqrt{3} - 1) + \sqrt{3}(\sqrt{3} - \sqrt{2})}{\sqrt{1 + (\sqrt{2} - 1)^2 + (\sqrt{3} - \sqrt{2})^2}} \\ &= \sqrt{9 - 2\sqrt{2} - 2\sqrt{2}\sqrt{3}} \\ &\approx 1.13. \end{aligned} \tag{3.13}$$

This bound is asymptotically tight. Consider a triple cubic graph without blocked grid cells where the start vertex is at coordinates $(0, 0, 0)$ and the goal vertex is at coordinates $(\lceil x_1 \rceil + \lceil x_2 \rceil + \lceil x_3 \rceil, \lceil x_3 \rceil, \lceil x_2 \rceil + \lceil x_3 \rceil)$. As $\lambda \to -\infty$, $\frac{d'(u,t)}{d(u,t)}$ approaches, but never reaches, ≈ 1.13. □

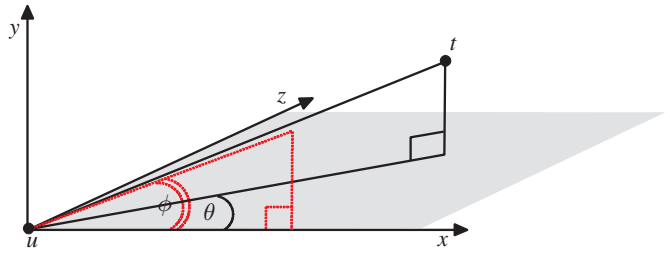


Figure 3.9: Projection of L onto the x-z and x-y Planes

3.3.4 Geometric Relationships: 3D Regular Grids

In this section we analyze the geometric relationship of L and L'. Figure 3.9 shows the spherical coordinates θ and ϕ, which are the angles of the projection of L onto the x-z and x-y planes, respectively. It holds that $\tan(\theta) = (x_2 + x_3)/(x_1 + x_2 + x_3) = \sqrt{2} - 1 = \tan(\pi/8)$ and $\tan(\phi) = x_3/(x_1 + x_2 + x_3) = \sqrt{3} - \sqrt{2}$ for a suitable coordinate system, which implies that $\theta = \pi/8$ and that $\phi = \frac{1}{2} \cdot \arctan(1/\sqrt{2})$ since $\tan(2\phi) = 2\tan(\phi)/(1 - \tan^2(\phi)) = 1/\sqrt{2}$. Thus, θ is exactly half of the smaller angle between $\vec{v_{xz}} = (1, 0, 1)$ and $\vec{v_x} = (1, 0, 0)$ (that is, $\pi/4$) and ϕ is exactly half of the smaller angle between $\vec{v_{xz}} = (1, 0, 1)$ and $\vec{v_{xyz}} = (1, 1, 1)$ (that is, $\arctan(1/\sqrt{2})$). Thus, L diverges from L' as much as possible.

3.3.5 Cubic Graphs

Part 1 and Part 2 of the proof of Lemma 2 can easily be updated so that they apply to cubic graphs. In order to update Part 1 of the proof of Lemma 2 we must update the move sequence algorithm. We begin with the step sequence $\widehat{L}$ that we constructed for triple cubic graphs. We convert $\widehat{L}$ into a move sequence L' of right (r), back (b) and up (u) moves. r moves increase the x-coordinate by 1 and have length 1, b moves increase the z-coordinate by 1 and have length 1 and u moves increase the y-coordinate by 1 and have length 1. We start with an L' that contains a single r move followed by a single b move followed by a single u move and then we append moves to L' by iterating through the steps in $\widehat{L}$ as follows: If the next step in $\widehat{L}$ is an R, B or U then we append an r, b or u move, respectively to L'. L' is formed by edges on G' since all moves traverse the sides of grid cells that are known to unblocked because they are traversed by L. L' must be a shortest grid path from vertex u to vertex t on G'. L connects vertex u at coordinates $(0, 0, 0)$ and vertex t at coordinates (r', u', b') and thus L must travel at least r' units in the x direction, u' units in the y direction and b' units in the z direction. Therefore, the move sequence algorithm that constructed the move sequence L' that connects vertex u to vertex t must have processed at least r' Rs, b' Bs and u' Us. This is precisely the number of each type of step in $\widehat{L}$ and, thus, if the moves in L' combine steps into moves in the most efficient manner possible then L' must represent a shortest grid path from vertex u to vertex t on G'. This is precisely what the move sequence algorithm does. Every U in $\widehat{L}$ corresponds to an u move in L'. When the algorithm processes the U, it creates a u move. Every B in $\widehat{L}$ corresponds to a b move in L'. When the algorithm processes the B, it creates a b move. Every R in $\widehat{L}$ corresponds to an r move in L'.

When the algorithm processes the R, it creates an r move. Thus, the sum of the lengths of the moves in L' is $u' + b' + r'$, and no path from vertex u to vertex t on G' can be shorter.

In order to update Part 2 of the proof of Lemma 2 to cubic graphs we must update the equations used in the Lagrange Method described in Section 3.3.3.2. Let x_1 be the number of r moves (whose length is 1), x_2 be the number of b moves (whose length is 1) and x_3 be the number of u moves (whose length is 1). We maximize the ratio of the lengths of L' and L, that is $\frac{d'(u,t)}{d(u,t)}$, by using Lagrange Multipliers. We want to minimize $f(x_1, x_2, x_3)$ subject to $g(x_1, x_2, x_3) = c$ for some constant c, where $f(x_1, x_2, x_3) = \sqrt{{x_1}^2 + {x_2}^2 + {x_3}^2} = d(u, t)$ and $g(x_1, x_2, x_3) = x_1 + x_2 + x_3 = d'(u, t)$ resulting in:

$$L(x_1, x_2, x_3\lambda) = f(x_1, x_2, x_3) + \lambda(g(x_1, x_2, x_3) - c). \tag{3.14}$$

We remove the square root from the minimization because it is a monotonic function. $\nabla L = 0$ then implies the system of equations:

$$\frac{\partial L}{\partial x_1} = 0 : \quad 2x = -\lambda \tag{3.15}$$

$$\frac{\partial L}{\partial x_3} = 0 : \quad 2y = -\lambda \tag{3.16}$$

$$\frac{\partial L}{\partial x_2} = 0 : \quad 2z = -\lambda. \tag{3.17}$$

From Equations 3.15, 3.16 and 3.17 we get $x_1 = x_2 = x_3 = \frac{-\lambda}{2}$. The worst case ratio of the lengths of L' and L, is:

$$\begin{aligned} \frac{d'(u,t)}{d(u,t)} &= \sqrt{3} \\ &\approx 1.73. \end{aligned} \tag{3.18}$$

This bound is tight for an L' formed by a single r move followed by a singe b move followed by a single u move.

The fact that the Lagrange Method applies to cubic graphs and triple cubic graphs suggests that the Scaling Method used in Part 3 of Lemma 1 could be replaced with the Lagrange Method used in Part 2 of Lemma 2.

3.3.6 Lagrange Method: 2D Regular Grids

In this section, we show that the Scaling Method used in Part 3 of Lemma 1 can be replaced with the Lagrange Method used in Part 2 of Lemma 2. The Lagrange Method applies to paths consisting of any number of different edge types and thus it can be used in place of the Scaling Method. However, the Scaling Method requires simpler math and is more elegant for grid graphs constructed from 2D regular grids. The Lagrange Method can be used in place of the Scaling method for all types of grid graphs that our proof considers, but we demonstrate it on hex graphs. It is quite simple to apply this same technique to the other types of grid graphs discussed in Section 3.2.4.

We show how the Lagrange Method can be applied to a line segment in sector S defined by an m_i = Up edge and an m_{i+1} = Up Right edge.

Let x_1 be the number of u moves (whose length is $\sqrt{3}$) and x_2 be the number of ur moves (whose length is $\sqrt{3}$). We maximize the ratio of the lengths of L' and L, that is $\frac{d'(u,t)}{d(u,t)}$, by using Lagrange Multipliers. We want to minimize $f(x_1, x_2)$ subject to $g(x_1, x_2) = c$ for some constant c, where $f(x_1, x_2) = \sqrt{(\sqrt{3}x_1 + \frac{\sqrt{3}x_2}{2})^2 + \frac{3x_2}{2}^2} = d(u,t)$ and $g(x_1, x_2) = \sqrt{3}x_1 + \sqrt{3}x_2 = d'(u,t)$ resulting in:

$$L(x_1, x_2, \lambda) = f(x_1, x_2) + \lambda(g(x_1, x_2) - c). \tag{3.19}$$

We remove the square root from the minimization because it is a monotonic function. $\nabla L = 0$ then implies the system of equations:

$$\frac{\partial L}{\partial x_1} = 0: \quad 6x_1 + 3x_2 = -\lambda\sqrt{3} \tag{3.20}$$

$$\frac{\partial L}{\partial x_2} = 0: \quad 3x_1 + 6x_2 = -\lambda\sqrt{3}. \tag{3.21}$$

From Equations 3.20 and 3.21 we get $x_1 = x_2 = -\frac{\sqrt{3}\lambda}{9}$. The worst case ratio of the lengths of L' and L, is:

$$\begin{aligned} \frac{d'(u,t)}{d(u,t)} &= \frac{\sqrt{3}x_1 + \sqrt{3}x_2}{\sqrt{(\sqrt{3}x_1 + \frac{\sqrt{3}x_2}{2})^2 + \frac{3x_2}{2}^2}} \\ &= \frac{2\sqrt{3}}{3} \\ &\approx 1.15. \end{aligned} \tag{3.22}$$

3.4 Conclusions

In this chapter, we compared the lengths of the paths found by traditional edge-constrained find-path algorithms with the lengths of the true shortest paths. We examined the paths found by traditional edge-constrained find-paths algorithms on grid graphs constructed from 2D and 3D regular grids. Since traditional edge-constrained find-paths algorithms can find shortest grid paths we compared the lengths of shortest grid paths formed by the edges of grid graphs constructed from 2D and 3D regular grids with the lengths of the true shortest paths. In Section 3.2, we examined regular grids that discretize continuous *2D* environments. We performed a comprehensive path length analysis on the lengths of shortest grid paths formed by the edges of grid graphs constructed from all three types of regular grids that can be used to tessellate continuous 2D environments, namely triangular, square and hexagonal grids. We showed that shortest grid paths can be longer than true shortest paths as follows: 100% for 3-neighbor triangular grid graphs, $\approx 41\%$ for 4-neighbor square grid graphs, $\approx 15\%$ for 6-neighbor hexagonal grid graphs, $\approx 15\%$ for 6-neighbor triangular grid graphs, $\approx 8\%$ for 8-neighbor square grid graphs and $\approx 4\%$ for

12-neighbor hexagonal grid graphs. In Section 3.3, we examined regular grids that discretize continuous *3D* environments. We performed a comprehensive path length analysis on the lengths of shortest grid paths formed by the edges of grid graphs constructed from the only regular polyhedron that can be used to tessellate continuous 3D environments, namely a cube. We showed that shortest grid paths can be longer than true shortest paths as follows: $\approx 73\%$ for 6-neighbor cubic graphs and $\approx 13\%$ for 26-neighbor cubic grid graphs. A summary of these bounds is shown in Table 1.1. These results validate Hypothesis 1, namely that analytical bounds can be introduced that compare the lengths of the paths found by traditional edge-constrained find-path algorithms with the lengths of the true shortest paths on certain types of graphs.

Chapter 4

Basic Theta*: Any-Angle Path Planning in Known 2D Environments (Contribution 2)

Figure 4.1: Known 2D Environments: Screen Shot from Company of Heroes (Relic Entertainment)

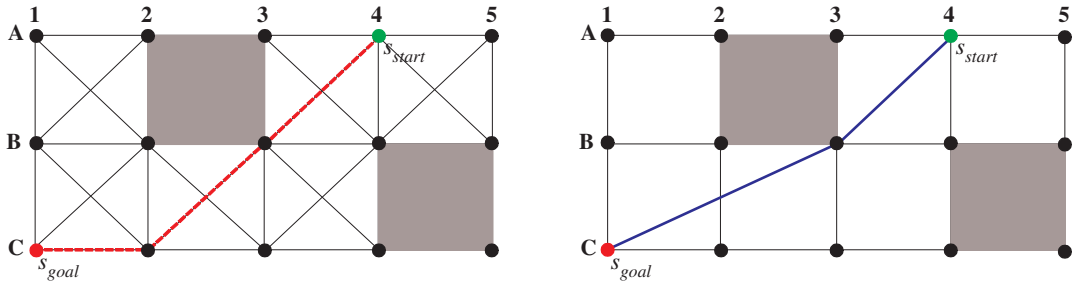
This chapter introduces the second major contribution of this dissertation, namely Contribution 2. Specifically, this chapter introduces Basic Theta* (Nash et al., 2007, 2010), a new any-angle find-path algorithm which we evaluate in known 2D environments using the Simplicity, Efficiency and Generality Properties. Our evaluation shows that Basic Theta* is simple to implement and understand, that it provides a good tradeoff with respect to the runtime of the search and the length of the resulting path, that it provides a dominating tradeoff relative to existing any-angle find-path algorithms with respect to the runtime of the search and the length of the resulting path, that it finds paths that are $\approx 4-5\%$ shorter than the paths found by traditional edge-constrained find-path algorithms on average (and $\approx 4-5\%$ shorter than shortest grid paths on average), but with a similar runtime and that it can be used to search any Euclidean graph. Therefore, these results validate Hypothesis 2 in known 2D environments. This chapter also introduces Angle-Propagation Theta*. Angle-Propagation Theta* is similar to Basic Theta* and

has a better worst-case complexity, but is more difficult to implement and understand, is not as fast, finds slightly longer paths and cannot be used to search any Euclidean graph.

This chapter is organized as follows: In Section 4.1, we reiterate the motivation behind Hypothesis 2 and provide a detailed summary of what to expect in this (lengthy) chapter. In Section 4.2, we introduce notation and definitions that are used throughout this chapter. In Section 4.3, we examine existing find-path algorithms in much more detail than we did in Section 2.2.2 and highlight their shortcomings. In Sections 4.4 and 4.5, we introduce Basic Theta* and Angle-Propagation Theta*, respectively, two new any-angle find-path algorithms that are designed for path planning in known 2D environments. In Section 4.6, we present our experimental results. In Sections 4.7-4.9, we introduce several variants and extensions of Basic Theta*. All of the subsequent chapters in this dissertation build upon Basic Theta* and thus this chapter contains an extensive analysis of Basic Theta* that subsequent chapters build on. Finally, in Section 4.10, we summarize our results.

4.1 Introduction

In this chapter, we examine path planning in the simplest type of environment in which path planning is performed, namely known 2D environments (Figure 4.1). Specifically, this chapter focuses on known 2D environments that have been discretized into 8-neighbor square grid graphs (octile graphs) with vertices placed in the corners of grid cells (Section 3.2.2). Octile graphs are widely used by roboticists and video game developers because they have many desirable properties (Sections 2.2.1.2 and 3.2.3). Our objective is to find a short unblocked path from a given start vertex to a given goal vertex (Section 1.1.2). A* finds grid paths (that is, paths formed by octile graph edges) quickly, but grid paths are often not true shortest paths (that is, shortest paths in the continuous environment) since their potential headings are artificially constrained to multiples of 45 degrees, as shown in Figure 4.2(a) (Yap, 2002). This shortcoming led to the introduction of any-angle path planning (Nash et al., 2007; Ferguson & Stentz, 2006). Any-angle find-path algorithms find paths without constraining the headings of the paths, as shown in Figure 4.2(b). We present two new any-angle find-path algorithms. Basic Theta* and Angle-Propagation Theta* are both variants of A* that propagate information along octile graph edges (to achieve a short runtime) without constraining paths to be formed by octile graph edges (to find any-angle paths). Unlike A* on visibility graphs, they are not guaranteed to find true shortest paths. Thus, the asterisk in their names (as well as the asterisks in the names of all of new any-angle find-path algorithms introduced in this dissertation) does not denote their optimality, but rather their similarity to A*. Basic Theta* is simple to implement and understand, fast, finds short paths and can be used to search any Euclidean graph. Angle-Propagation Theta* achieves a worst-case complexity per vertex expansion that is constant rather than linear in the number of vertices (like that of Basic Theta*) by propagating angle ranges when it expands vertices, but is more difficult to implement and understand, is not as fast, finds slightly longer paths and cannot be used to search any Euclidean graph. Basic Theta* and Angle-Propagation Theta* have unique properties, which we analyze in detail. We show experimentally that they find shorter paths than both A* with post-smoothed paths and Field D* (the only other variant of A* we know of that propagates information along octile graph edges without constraining paths to be formed by octile graph edges) with a runtime comparable to that of A* on octile graphs. Finally, we extend Basic Theta* to square grids that contain unblocked grid cells with non-uniform traversal costs and



(a) Grid Path (b) True Shortest Path

Figure 4.2: Grid Path Versus True Shortest Path

introduce variants of Basic Theta* which provide different tradeoffs with respect to the runtime of the search and the length of the resulting path.

4.2 Notation and Definitions

In this section, we describe the find-path problem that we study in this chapter, namely finding paths on octile graphs $G' = (V, E')$ (with vertices placed in the corners of grid cells) constructed from square grids that discretize known 2D environments (as described in Section 3.2.2). In this chapter, we assume that obstacles in the known 2D environment completely block grid cells, that is, there is no digitization bias. This means that shortest any-angle paths are equivalent to true shortest paths. The find-path problem is to find an unblocked path from a given start vertex $s_{start} \in V$ to a given goal vertex $s_{goal} \in V$. $nghbr_{vis}(s) \subseteq V$ is the set of (up to 8) visible neighbors of vertex s, that is those vertices that are adjacent to vertex s and have line-of-sight to vertex s. Figure 4.2 shows an example where the visible neighbors of vertex B4 are vertices A3, A4, A5, B3, B5, C3 and C4. Unblocked path and line-of-sight are defined in Section 1.1.2. Pseudocode for implementing the line-of-sight function on square grids is given in Appendix A.

4.3 Existing Find-Path Algorithms in Known 2D Environments

In this section, we describe some existing find-path algorithms, all of which are variants of A* (Hart et al., 1968). Therefore, we begin with a detailed description of A* (as opposed to the general description of A* provided in Section 2.2.2.1). Algorithm 1 shows the pseudocode for the implementation of A* discussed in this chapter. Line 13 is to be ignored. A* maintains three values for every vertex s:

- The g-value $g(s)$ is the length of a shortest path from the start vertex to vertex s found so far and thus is an estimate of the start distance of vertex s.
- The user-provided h-value $h(s)$ is an estimate of the goal distance of vertex s. A* uses the h-value to calculate an f-value to focus the A* search. The f-value $f(s) = g(s) + h(s)$ is an estimate of the length of a shortest path from the start vertex via vertex s to the goal vertex. In our experiments, each find-path algorithm uses consistent h-values and thus Algorithm 1 is complete, correct and optimal even though vertices in the closed list are not re-examined.

```
1  Main()
2      g(s_start) := 0;
3      parent(s_start) := s_start;
4      open := ∅;
5      open.Insert(s_start), g(s_start) + h(s_start));
6      closed := ∅;
7      while open ≠ ∅ do
8          s := open.Pop();
9          if s = s_goal then
10             return "path found";
11         closed := closed ∪ {s};
12         /* The following line is executed only by AP Theta*.  */;
13         [UpdateBounds(s)];
14         foreach s' ∈ nghbr_vis(s) do
15             if s' ∉ closed then
16                 if s' ∉ open then
17                     parent(s') := NULL;
18                     g(s') := ∞;
19                 UpdateVertex(s, s');
20     return "no path found";
21 UpdateVertex(s,s')
22     g_old := g(s');
23     ComputeCost(s, s');
24     if g(s') < g_old then
25         if s' ∈ open then
26             open.Remove(s');
27         open.Insert(s', g(s') + h(s'));
28 ComputeCost(s,s')
29     /* Path 1 */
30     if g(s) + c(s, s') < g(s') then
31         parent(s') := s;
32         g(s') := g(s) + c(s, s');
```

Algorithm 1: A*

- The parent $parent(s)$ is used to extract a path from the start vertex to the goal vertex after A* terminates.

A* also maintains two global data structures:

- The open list is a priority queue that contains the vertices that A* considers for expansion. In the pseudocode, $open.Insert(s, x)$ inserts vertex s with key x into the priority queue *open*, $open.Remove(s)$ removes vertex s from the priority queue *open*, $open.Pop()$ removes a vertex with the smallest key from the priority queue *open* and returns it and $open.TopKey()$ returns the smallest key of all vertices in the priority queue *open* unless *open* is empty in which case it returns infinity.
- The closed list is a set that contains the vertices that A* has already expanded. It ensures that A* expands every vertex at most once.

A* sets the g-value of every vertex to infinity and the parent of every vertex to NULL when it encounters the vertex for the first time [Lines 18-17]. It sets the g-value of the start vertex to zero

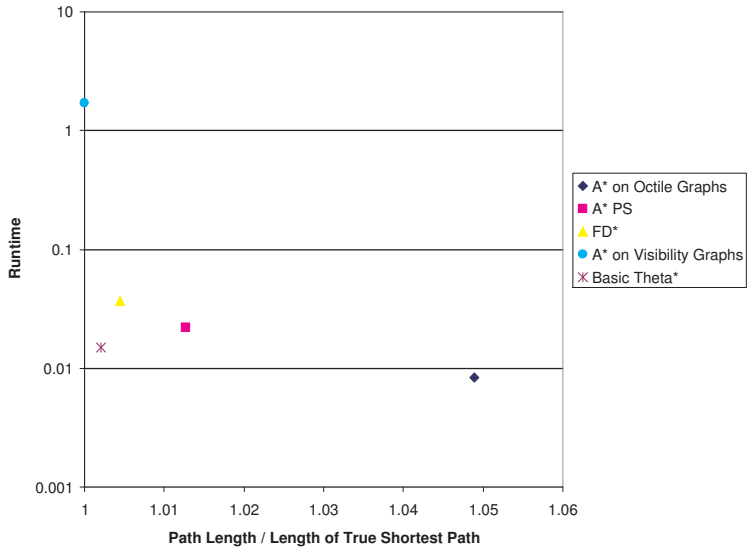


Figure 4.3: Runtime Versus Path Length (relative to the length of a true shortest path) on Random 100×100 Grids with 20 Percent Blocked Grid Cells

and the parent of the start vertex to the start vertex itself [Lines 2-3]. It sets the open and closed lists to the empty list and then inserts the start vertex into the open list with the f-value as its key [4-6]. A* then repeatedly executes the expansion process: If the open list is empty, then it reports that there is no path [Line 20]. Otherwise, it identifies a vertex s with the smallest f-value in the open list [Line 8]. If this vertex is the goal vertex, then A* reports that it has found a path [Line 10]. Path extraction [not shown in the pseudocode] follows the parents from the goal vertex to the start vertex to retrieve a path from the start vertex to the goal vertex in reverse. Otherwise, A* removes the vertex from the open list [Line 8] and expands it by inserting the vertex into the closed list [Line 11] and then generating each of its unexpanded visible neighbors, as follows: A* checks whether the g-value of vertex s plus the length of the straight line from vertex s to vertex s' is smaller than the g-value of vertex s' [Line 30]. If so, then it sets the g-value of vertex s' to the g-value of vertex s plus the length of the straight line from vertex s to vertex s' [Line 32], sets the parent of vertex s' to vertex s [Line 31] and finally inserts vertex s' into the open list with the f-value as its key or, if it was already in the open list, sets its key to the f-value [Lines 27]. It then repeats this procedure.

To summarize, when A* updates the g-value and parent of an unexpanded visible neighbor s' of vertex s in procedure ComputeCost, it considers the path from the start vertex to vertex s [= $g(s)$] and from vertex s to vertex s' in a straight line [= $c(s, s')$], resulting in a length of $g(s) + c(s, s')$ [Line 30]. A* updates the g-value and parent of vertex s' if the considered path is shorter than the shortest path from the start vertex to vertex s' found so far [= $g(s')$].

We now describe several existing find-path algorithms that are variants of A* and how they tradeoff with respect to two conflicting criteria, namely the runtime of the search and the length

```
33 PostSmoothPath([s_0, ..., s_n])
34     k := 0;
35     t_k := s_0;
36     foreach i := 1 ... n − 1 do
37         if NOT LineOfSight(t_k, s_{i+1}) then
38             k := k + 1;
39             t_k := s_i;
40     k := k + 1;
41     t_k := s_n;
42     return [t_0, ..., t_k];
```

Algorithm 2: Post-Smoothing

of the resulting path, as shown in Figure 4.3.[1] We introduce them in order of decreasing path lengths.

4.3.1 A* on Octile Graphs

One can run A* on octile graphs. The resulting paths are artificially constrained to be formed by the edges of the octile graph, which can be seen in Figure 4.2(a). As a result, the paths found by A* on octile graphs are not equivalent to the true shortest paths and are unrealistic looking since they either deviate substantially from the true shortest paths or have many more heading changes, which provides the motivation for smoothing them. We use the octile distances, which can be computed using Algorithm 5, as h-values in the experiments.

4.3.2 A* with Post-Smoothing (A* PS)

One can run A* with post-smoothing (A* PS) (Thorpe, 1984). A* PS runs A* on octile graphs and then smoothes the resulting path in a post-processing step, which often shortens it at an increase in runtime. Algorithm 2 shows the pseudocode of the simple smoothing algorithm that A* PS uses in our experiments. This smoothing algorithm is widely used by video game developers because it provides a good tradeoff with respect to the runtime of the search and the length of the resulting path (Botea et al., 2004; Ian Millington, 2009). Assume that A* on octile graphs finds the path $[s_0, s_1, \ldots, s_n]$ with $s_0 = s_{start}$ and $s_n = s_{goal}$. A* PS uses the first vertex on the path as the current vertex. It then checks whether the current vertex s_0 has line-of-sight to the successor s_2 of its successor on the path. If so, A* PS removes the extraneous vertex s_1 from the path, thus shortening it. A* PS then repeats this procedure by checking again whether the current vertex s_0 has line-of-sight to the successor s_3 of its successor on the path, and so on. As soon as the current vertex does not have line-of-sight to the successor of its successor on the path, A* PS advances the current vertex and repeats this procedure until it reaches the end of the path. We use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in the experiments.

A* PS typically finds shorter paths than A* on octile graphs, but is not guaranteed to find true shortest paths. Figure 4.4 shows an example. Assume that A* PS finds the dotted blue path, which is one of many shortest grid paths. It then smoothes this path to the solid blue path, which is not a true shortest path. The dashed red path, which moves above (rather than below) blocked grid cell B2-B3-C3-C2 is a true shortest path. A* PS is not guaranteed to find true shortest

[1] More detailed experimental results are provided in Section 4.6 and Appendix E.3

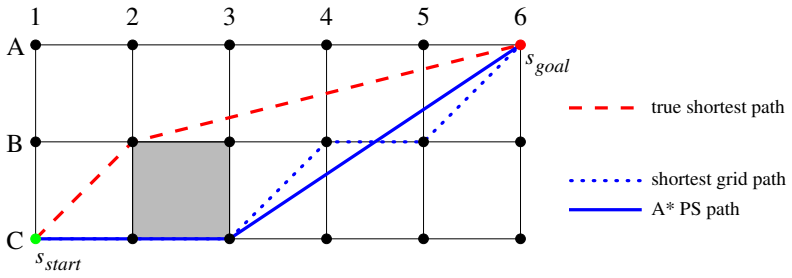


Figure 4.4: A* PS Path Versus True Shortest Path

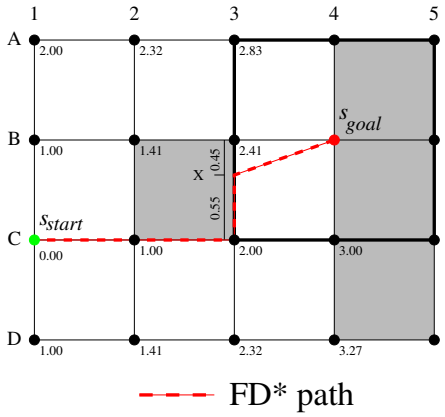


Figure 4.5: FD* Path

paths because it only considers grid paths during the A* search and thus cannot make informed decisions regarding other paths during the A* search, which motivates interleaving searching and smoothing. In fact, Basic Theta* and Angle-Propagation Theta* are similar to A* PS except that they interleave searching and smoothing.

4.3.3 Field D* (FD*)

One can run Field D* (Ferguson & Stentz, 2006) (FD*). FD* propagates information along octile graph edges without constraining paths to be formed by octile graph edges. FD* was designed to use D* Lite (Koenig & Likhachev, 2002a) for path planning in unknown 2D environments (by reusing information from previous searches to speed up the next one) and searches from the goal vertex to the start vertex. Our variant of FD* uses A* and searches from the start vertex to the goal vertex, like all other find-path algorithms in this chapter, which allows us to compare them fairly in known 2D environments.

When FD* updates the g-value and parent of an unexpanded visible neighbor s' of vertex s, it considers all paths from the start vertex to any coordinate X (not necessarily a vertex) on the perimeter of vertex s' [= $g(X)$] that has line-of-sight to vertex s', where the perimeter is formed by connecting all the neighbors of vertex s', and from coordinate X to vertex s' in a straight line [= $c(X, s')$], resulting in a length of $g(X) + c(X, s')$. FD* updates the g-value and parent of

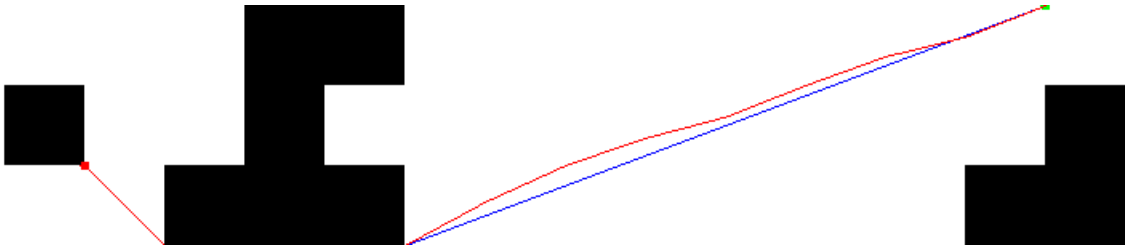


Figure 4.6: Screen Shot of FD* Path Versus True Shortest Path

vertex s' if the considered path is shorter than the shortest path from the start vertex to vertex s' found so far [= $g(s')$]. Unlike the other existing find-path algorithms introduced in this section, FD* cannot be used to search any Euclidean graph because the closed form linear interpolation equation it uses to calculate coordinate X requires that the search be performed on an octile graph (Section 2.2.4). We use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in the experiments.

Figure 4.5 shows an example. The perimeter of vertex s' =B4 is formed by connecting all of the neighbors of vertex B4, as shown in bold. Consider coordinate X on the perimeter. FD* does not know the g-value of coordinate X since it only stores g-values for vertices. It calculates the g-value using linear interpolation between the g-values of the two vertices on the perimeter that are adjacent to the coordinate X. Thus, it linearly interpolates between $g(B3) = 2.41$ and $g(C3) = 2.00$, resulting in $g(X) = 0.55 \cdot 2.41 + 0.45 \cdot 2.00 = 2.23$ since 0.55 and 0.45 are the distances from coordinate X to vertices B3 and C3, respectively. The calculated g-value of coordinate X is different from its true start distance [= 2.55] even though the g-values of vertices B3 and C3 are both equal to their true start distances. The reason for this mistake is simple. There exist true shortest paths from the start vertex through either vertex C3 or vertex B3 to the goal vertex. Thus, the linear interpolation assumption predicts that there must also exist a short path from the start vertex through any coordinate along the edge that connects vertices B3 and C3 to the goal vertex. However, this is not the case since these paths need to circumnavigate blocked grid cell B2-B3-C3-C2, which makes them longer than expected. As a result of miscalculating the g-value of coordinate X, FD* sets the parent of vertex B4 to coordinate X, resulting in a path that has an unnecessary heading change at coordinate X and is longer than even a shortest grid path.

The authors of FD* recognize that the paths found by FD* frequently have unnecessary heading changes and suggest the use of a one-step look-ahead algorithm during path extraction (Ferguson & Stentz, 2006), which FD* uses in our experiments. This one-step look-ahead algorithm allows FD* to avoid some of the unnecessary heading changes, like the one in Figure 4.5, but does not eliminate all of them. Figure 4.6 shows an example of an FD* path in red and the corresponding true shortest path in blue. The FD* path still has many unnecessary heading changes.

4.3.4 A* on Visibility Graphs

One can run A* on visibility graphs. The visibility graph constructed from a square grid with blocked and unblocked grid cells contains the start vertex, the goal vertex and the corners of all blocked grid cells (Lozano-Pérez & Wesley, 1979). We use the straight-line distances $h(s) =$

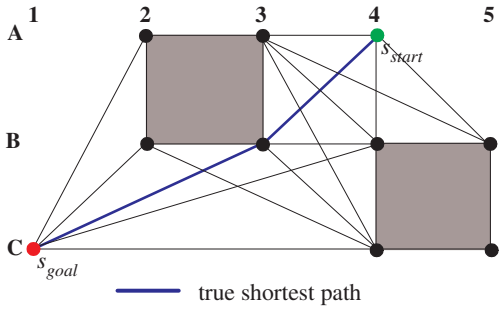


Figure 4.7: Visibility Graph Constructed From a Square Grid

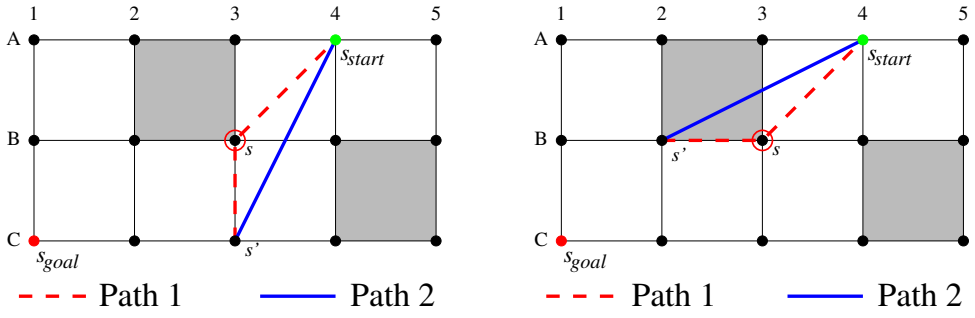
```
43 ComputeCost(s,s')
44     if LineOfSight(parent(s), s') then
45         /* Path 2 */
46         if g(parent(s)) + c(parent(s), s') < g(s') then
47             parent(s') := parent(s);
48             g(s') := g(parent(s)) + c(parent(s), s');
49     else
50         /* Path 1 */
51         if g(s) + c(s, s') < g(s') then
52             parent(s') := s;
53             g(s') := g(s) + c(s, s');
```

Algorithm 3: Basic Theta*

$c(s, s_{goal})$ as h-values in the experiments. A* on visibility graphs finds true shortest paths, as shown in Figure 4.7. True shortest paths have heading changes only at the corners of blocked grid cells, while the paths found by A* on octile graphs, A* PS and FD* can have unnecessary heading changes. On the other hand, A* on visibility graphs can be slow. It propagates information along visibility graph edges, whose number can grow quadratically in the number of vertices, while A* on octile graphs, A* PS and FD* propagate information along octile graph edges, whose number grows only linearly in the number of vertices. Furthermore, constructing the entire visibility graph before the A* search, requires that a large number of line-sight-checks are performed. In our experiments we reduced the number of line-of-sight checks performed by constructing the visibility graphs during the A* search. When it expands a vertex, it performs line-of-sight checks between the expanded vertex and the corners of all blocked grid cells (and the goal vertex).

4.4 Basic Theta*

In this section, we introduce Basic Theta* (Nash et al., 2007), our simple variant of A* for any-angle path planning that propagates information along octile graph edges without constraining paths to be formed by octile graph edges. It combines the ideas behind A* on visibility graphs (where heading changes occur only at the corners of blocked grid cells) and A* on octile graphs (where the number of edges grows only linearly in the number of vertices). Its paths are only slightly longer than true shortest paths (as found by A* on visibility graphs), yet is only slightly slower than A* on octile graphs, as shown in Figure 4.3. The key difference between Basic



(a) Path 2 is Unblocked

(b) Path 2 is Blocked

Figure 4.8: Paths 1 and 2 Considered by Basic Theta*

Theta* and A* on octile graphs is that the parent of a vertex can be any vertex when using Basic Theta*, while the parent of a vertex has to be a neighbor of the vertex when using A*.

Algorithm 3 shows the pseudocode of Basic Theta*. Procedures Main and UpdateVertex are identical to that of A* in Algorithm 1 and thus are not shown. Line 13 is to be ignored. We use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in the experiments.

4.4.1 Operation of Basic Theta*

Basic Theta* is simple. It is identical to A* except that, when it updates the g-value and parent of an unexpanded visible neighbor s' of vertex s in procedure ComputeCost, it considers two paths instead of only the one path considered by A*. Figure 4.8(a) shows an example. Basic Theta* is expanding vertex B3 with parent A4 and needs to update the g-value and parent of unexpanded visible neighbor C3. Basic Theta* considers two paths:

- **Path 1:** Basic Theta* considers the path from the start vertex to vertex s [= $g(s)$] and from vertex s to vertex s' in a straight line [= $c(s, s')$], resulting in a length of $g(s) + c(s, s')$ [Line 51]. Path 1 is the path considered by A*. It corresponds to the dashed red path [A4, B3, C3] in Figure 4.8(a)).
- **Path 2:** Basic Theta* also considers the path from the start vertex to the parent of vertex s [= $g(parent(s))$] and from the parent of vertex s to vertex s' in a straight line [= $c(parent(s), s')$], resulting in a length of $g(parent(s)) + c(parent(s), s')$ [Line 46]. Path 2 is not considered by A* and allows Basic Theta* to form any-angle paths. It corresponds to the solid blue path [A4, C3] in Figure 4.8(a).

Path 2 is no longer than Path 1 due to the triangle inequality. The triangle inequality states that the length of any side of a triangle is no longer than the sum of the lengths of the other two sides. It applies here since Path 1 consists of the path from the start vertex to the parent of vertex s, the straight line from the parent of vertex s to vertex s (Line A) and the straight line from vertex s to vertex s' (Line B), Path 2 consists of the same path from the start vertex to the parent of vertex s and the straight line from the parent of vertex s to vertex s' (Line C) and Lines A, B and C form a triangle. Path 1 is guaranteed to be unblocked but Path 2 is not. Thus, Basic Theta*

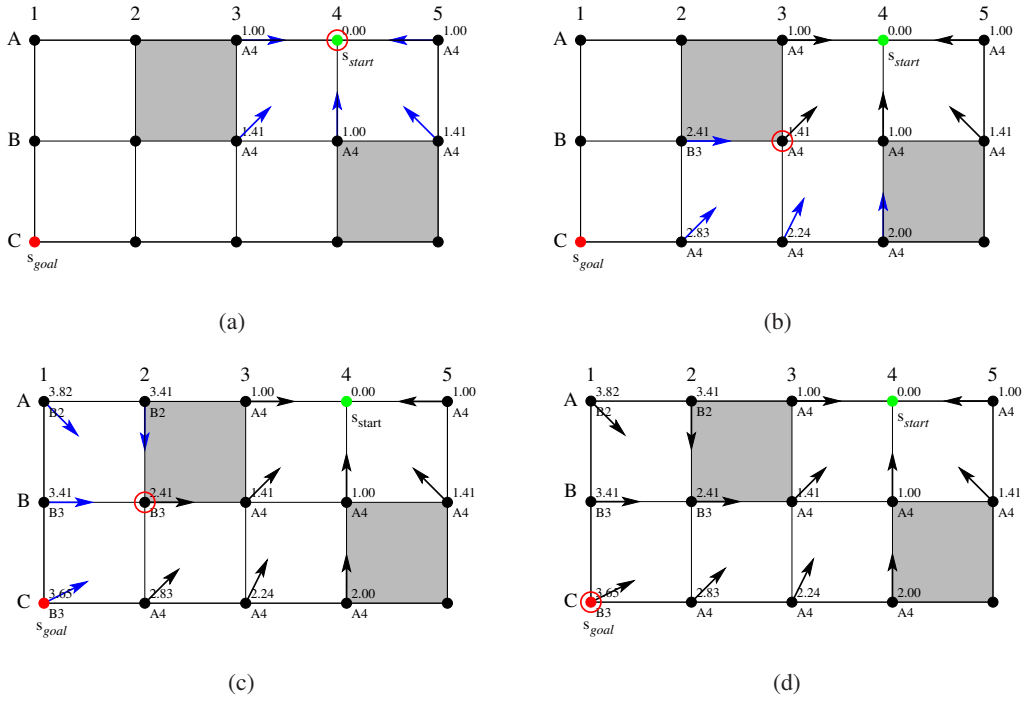


Figure 4.9: Example Trace of Basic Theta*

chooses Path 2 over Path 1 iff vertex s' has line-of-sight to the parent of vertex s and Path 2 is thus unblocked. Figure 4.8(a) shows an example. Otherwise, Basic Theta* chooses Path 1 over Path 2. Figure 4.8(b) shows an example. Basic Theta* updates the g-value and parent of vertex s' if the chosen path is shorter than the shortest path from the start vertex to vertex s' found so far [= $g(s')$]. We use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in the experiments.

4.4.2 Example Trace of Basic Theta*

Figure 4.9 shows an example trace of Basic Theta*. The vertices are labeled with their g-values and parents. The arrows point to their parents. Hollow red circles indicate vertices that are being expanded, and blue arrows indicate vertices that are generated during the current expansion. First, Basic Theta* expands start vertex A4 with parent A4, as shown in Figure 4.9(a). It sets the parent of the unexpanded visible neighbors of vertex A4 to vertex A4, just like A* would do. Second, Basic Theta* expands vertex B3 with parent A4, as shown in Figure 4.9(b). Vertex B2 is an unexpanded visible neighbor of vertex B3 that does not have line-of-sight to vertex A4. Basic Theta* thus updates it according to Path 1 and sets its parent to vertex B3. On the other hand, vertices C2, C3 and C4 are unexpanded visible neighbors of vertex B3 that have line-of-sight to vertex A4. Basic Theta* thus updates them according to Path 2 and sets their parents to vertex A4. (The g-values and parents of the other unexpanded visible neighbors of vertex B3 are not updated.) Third, Basic Theta* expands vertex B2 with parent B3, as shown in Figure 4.9(c). Vertices A1 and A2 are unexpanded visible neighbors of vertex B2 that do not have line-of-sight to vertex B3. Basic Theta* thus updates them according to Path 1 and sets their parents to vertex

B2. On the other hand, vertices B1 and C1 are unexpanded visible neighbors of vertex B2 that do have line-of-sight to vertex B3. Basic Theta* thus updates them according to Path 2 and sets their parents to vertex B3. Fourth, Basic Theta* expands goal vertex C1 with parent B3 and terminates, as shown in Figure 4.9(d). Path extraction then follows the parents from goal vertex C1 to start vertex A4 to retrieve a true shortest path [A4, B3, C1] from the start vertex to the goal vertex in reverse.

4.4.3 Properties of Basic Theta*

We now discuss the properties of Basic Theta*.

4.4.3.1 Simplicity Property

The pseudocode for Basic Theta* suggests that it is simple to implement and understand due to its similarity to A*. In order to demonstrate this property the authors of Basic Theta* helped to create a project for Introductory Artificial Intelligence classes (Koenig, Daniel, & Nash, 2008; Neller, DeNero, Klein, Koenig, Yeoh, Zheng, Daniel, Nash, Dodds, Carenini, Poole, & Brooks, 2010) that requires students to implement Basic Theta* and compare it with A*. This project has been successfully administered at both the University of Southern California (Koenig, 2009) and the University of Nevada, Reno (Bekris, 2010). In addition, the authors created a simple online tutorial of Basic Theta* for the AI Game Development website http://aigamedev.com/ (Nash, 2010). It is one of the largest AI community websites with 5000 daily RSS subscribers and almost 100,000 monthly page views (Champandard, 2010). The websites premium area is subscribed to by a variety of studios around the world (mainly in the US and EU) (Champandard, 2010). Therefore, we have demonstrated that Basic Theta* satisfies the Simplicity Property.

4.4.3.2 Generality Property

Basic Theta* can be used to search any Euclidean graph. The triangle inequality is a property of Euclidean geometry and thus Path 2 is guaranteed to be no longer than Path 1 on any Euclidean graph. As a result, Basic Theta* can be used to search any Euclidean graph. Therefore, we have demonstrated that Basic Theta* satisfies the Generality Property.

4.4.3.3 Correctness and Completeness

Basic Theta* is correct (that is, finds only unblocked paths from the start vertex to the goal vertex) and complete (that is, finds a path from the start vertex to the goal vertex if one exists). We use the following lemmata in the proof.

Lemma 3. *If there exists an unblocked path between two vertices then there also exists an unblocked grid path between the same two vertices.*

Proof. An unblocked path between two vertices exists iff an unblocked any-angle path $[s_0, \ldots, s_n]$ exists between the same two vertices. Consider any path segment $\overline{s_k, s_{k+1}}$ of this any-angle path. If the path segment is horizontal or vertical, then consider the unblocked grid path from vertex s_k to vertex s_{k+1} that coincides with the path segment. Otherwise, consider the sequence $(b_0, \ldots, b_m)$ of unblocked grid cells whose interior the path segment passes through.

Any two consecutive grid cells b_j and b_{j+1} share at least one vertex s'_{j+1} since the grid cells either share an edge or are diagonally touching. (If they share more than one vertex, pick one arbitrarily.) Consider the grid path $[s'_0 = s_k, s'_1, \ldots, s'_m, s'_{m+1} = s_{k+1}]$. This grid path from vertex s_k to vertex s_{k+1} is unblocked since any two consecutive vertices on it are corners of the same unblocked grid cell and are thus visible neighbors. Repeat this procedure for every path segment of the any-angle path and concatenate the resulting grid paths to an unblocked grid path from vertex s_0 to vertex s_n. (If several consecutive vertices on the grid path are identical, then all of them but one can be removed.) □

Lemma 4. *(At any point during the execution of Basic Theta*:) Following the parents from any vertex in the open or closed lists to the start vertex retrieves an unblocked path from the start vertex to this vertex in reverse.*

Proof. We prove by induction that the lemma holds and that the parent of any vertex in the union of the open or closed lists itself is in the union of the open or closed lists. This statement holds initially because the start vertex is the only vertex in the union of the open or closed lists and it is its own parent. We now show that the statement continues to hold whenever a vertex changes either its parent or its membership in the union of the open or closed lists. Once a vertex is a member of the union of the open or closed lists, it continues to be a member. A vertex can become a member in the union of the open or closed lists only when Basic Theta* expands some vertex s and updates the g-value and parent of an unexpanded visible neighbor s' of vertex s in procedure UpdateVertex. Vertex s is thus in the closed list, and its parent is in the union of the open or closed lists according to the induction assumption. Thus, following the parents from vertex s (or its parent) to the start vertex retrieves an unblocked path from the start vertex to vertex s (or its parent, respectively) in reverse according to the induction assumption. If Basic Theta* updates vertex s' according to Path 1, then the statement continues to hold since vertices s and s' are visible neighbors and the path segment from vertex s to vertex s' is thus unblocked. If Basic Theta* updates vertex s' according to Path 2, then the statement continues to hold since Basic Theta* explicitly checks that the path segment from the parent of vertex s to vertex s' is unblocked. There are no other ways in which the parent of a vertex can change. □

Theorem 2. *Basic Theta* terminates and path extraction retrieves an unblocked path from the start vertex to the goal vertex if such a path exists. Otherwise, Basic Theta* terminates and reports that no unblocked path exists.*

Proof. The following properties together prove the theorem. Their proofs utilize the fact that Basic Theta* terminates iff the open list is empty or it expands the goal vertex. The start vertex is initially in the open list. Any other vertex is initially neither in the open nor closed lists. A vertex neither in the open nor closed lists can be inserted into the open list. A vertex in the open list can be removed from the open list and be inserted into the closed list. A vertex in the closed list remains in the closed list.

- Property A: Basic Theta* terminates. It expands one vertex in the open list during each iteration. In the process, it removes the vertex from the open list and can then never insert it into the open list again. Since the number of vertices is finite, the open list eventually becomes empty and Basic Theta* has to terminate if it has not terminated earlier already.

- Property B: If Basic Theta* terminates because its open list is empty, then there does not exist an unblocked path from the start vertex to the goal vertex. We prove the contrapositive. Assume that there exists an unblocked path from the start vertex to the goal vertex. We prove by contradiction that Basic Theta* then does not terminate because its open list is empty. Thus, assume also that Basic Theta* terminates because its open list is empty. Then, there exists an unblocked grid path $[s_0 = s_{start}, \ldots, s_n = s_{goal}]$ from the start vertex to the goal vertex according to Lemma 3. Choose vertex s_i to be the first vertex on the grid path that is not in the closed list when Basic Theta* terminates. The goal vertex is not in the closed list when Basic Theta* terminates since Basic Theta* would otherwise have terminated when it expanded the goal vertex. Thus, vertex s_i exists. Vertex s_i is not the start vertex since the start vertex would otherwise be in the open list and Basic Theta* could not have terminated because its open list is empty. Thus, vertex s_i has a predecessor on the grid path. This predecessor is in the closed list when Basic Theta* terminates since vertex s_i is the first vertex on the grid path that is not in the closed list when Basic Theta* terminates. When Basic Theta* expanded the predecessor, it added vertex s_i to the open list. Thus, vertex s_i is still in the open list when Basic Theta* terminates. But then Basic Theta* could not have terminated because its open list is empty, which is a contradiction.

- Property C: If Basic Theta* terminates because it expands the goal vertex, then path extraction retrieves an unblocked path from the start vertex to the goal vertex because following the parents from the goal vertex to the start vertex retrieves an unblocked path from the start vertex to the goal vertex in reverse according to Lemma 4.

□

4.4.3.4 Optimality

Basic Theta* is not optimal (that is, it is not guaranteed to find true shortest paths) because the parent of a vertex has to be either a visible neighbor of the vertex or the parent of a visible neighbor, which is not always the case for true shortest paths. Figure 4.10(a) shows an example where the dashed red path [E1, B9] is a true shortest path from start vertex E1 to vertex B9 since vertex E1 has line-of-sight to vertex B9. However, vertex E1 is neither a visible neighbor nor the parent of a visible neighbor of vertex B9 since vertex E1 does not have line-of-sight to these vertices (highlighted in red). Thus, Basic Theta* cannot set the parent of vertex B9 to vertex E1 and does not find a true shortest path from vertex E1 to vertex B9. Similarly, Figure 4.10(b) shows an example where the dashed red path [E1, D8, C10] is a true shortest path from vertex E1 to vertex C10. However, vertex D8 is neither a visible neighbor nor the parent of a visible neighbor of vertex C10 since start vertex E1 either has line-of-sight to them or Basic Theta* found paths from vertex E1 to them that do not contain vertex D8. In fact, the truly shortest paths from vertex E1 to all visible neighbors of vertex C10 that vertex E1 does not have line-of-sight to move above (rather than below) blocked grid cell C7-C8-D8-D7. Thus, Basic Theta* cannot set the parent of vertex C10 to vertex D8 and thus does not find a true shortest path from vertex E1 to vertex C10. The solid blue path from vertex E1 to vertex B9 in Figure 4.10(a) and the solid blue path from vertex E1 to vertex C10 in Figure 4.10(b) are less than a factor of 1.002 longer than the true shortest paths.

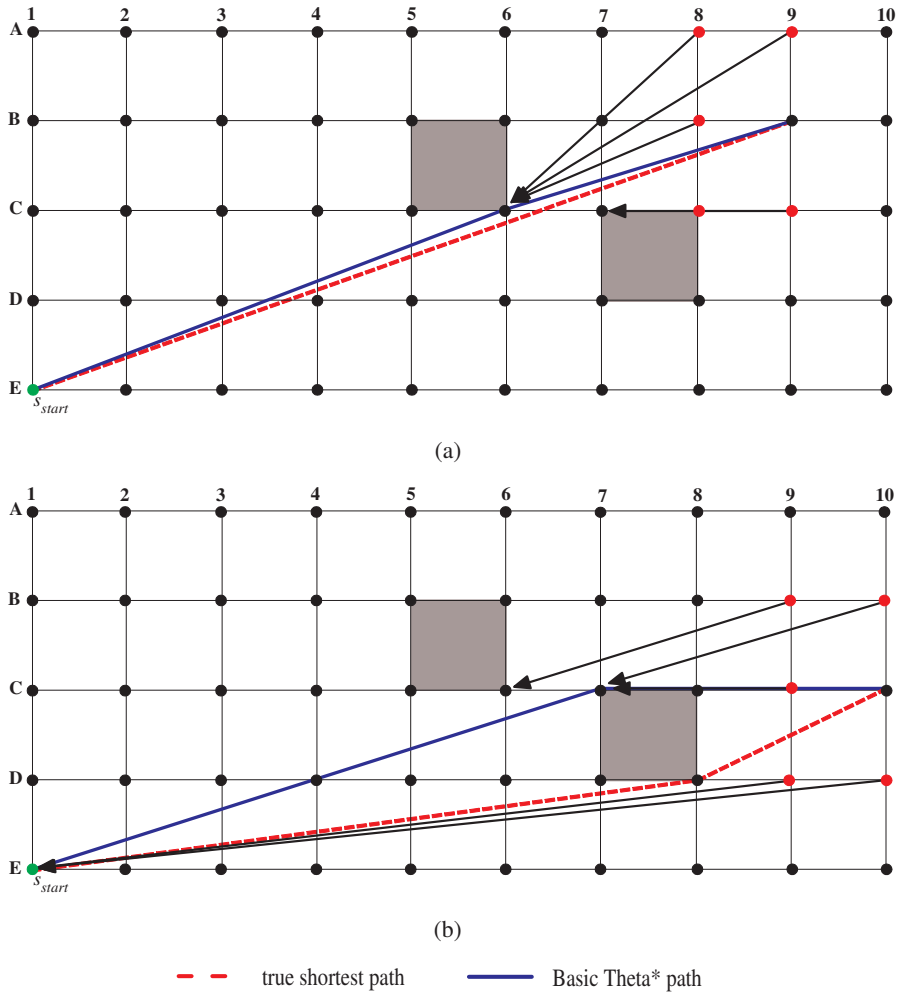


Figure 4.10: Basic Theta* Paths Versus True Shortest Paths

4.4.3.5 Heading Changes

Basic Theta* takes advantage of the fact that true shortest paths have heading changes only at the corners of blocked grid cells. However, the paths found by Basic Theta* can occasionally have unnecessary heading changes. Figure 4.11 shows an example where Basic Theta* finds the solid blue path [A1, D5, D6] from vertex A1 to vertex D6. The reason for this mistake is simple. Assume that the open list contains both vertices C5 and D5 as would be the case in an actual trace of Basic Theta*. The f-value of vertex C5 is $f(C5) = g(C5) + h(C5) = 4.61 + 1.41 = 6.02$ and its parent is vertex C4. The f-value of vertex D5 is $f(D5) = 5.00 + 1.00 = 6.00$ and its parent is vertex A1. Thus Basic Theta* expands vertex D5 before vertex C5 (since its f-value is smaller). When Basic Theta* expands vertex D5 with parent A1, it generates vertex D6. Vertex D6 is an unexpanded visible neighbor of vertex D5 that does not have line-of-sight to vertex A1. Basic Theta* thus updates it according to Path 1, sets its f-value to $f(D6) = 6.00 + 0.00 = 6.00$, sets its parent to vertex D5 and inserts it into the open list. Thus, Basic Theta* expands goal vertex

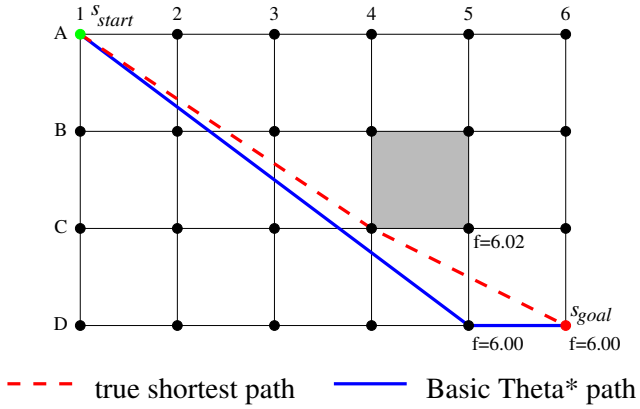


Figure 4.11: Heading Changes of Basic Theta*

D6 before vertex C5 (since its f-value is smaller) and terminates. Path extraction then follows the parents from goal vertex D6 to start vertex A1 to retrieve the solid blue path [A1, D5, D6]. Thus, Basic Theta* never expands vertex C5, which would have resulted in it setting the parent of vertex D6 to vertex C4 according to Path 2 and path extraction retrieving the dashed red path [A1, C4, D6], which is a true shortest path. The solid blue path from vertex A1 to vertex D6 in Figure 4.11 is less than a factor of 1.027 longer than true shortest path.

Figure 4.12 provides an updated summary of our classification of different find-path algorithms. It is an updated version of Figure 2.14 which accounts for the contributions made so far in this chapter. Figures 4.12 and 2.14 are annotated in the same way. However, Figure 4.12 contains additional annotation: If the border of an oval is dashed then the find-path algorithm is a new algorithm introduced in this dissertation. If the border of an oval is solid then the find-path algorithm is an existing find-path algorithm that was not introduced in this dissertation. The Basic Theta* oval is within 5 rounded rectangles, and thus Basic Theta* is a **1** single-shot, **2** informed, **3** any-angle **4** find-path algorithm that uses **5** line-of-sight checks. The Basic Theta* oval is transparent because it can be used to search any Euclidean Graph.

4.5 Angle-Propagation Theta* (AP Theta*)

The runtime of Basic Theta* per vertex expansion (that is, the runtime consumed during the generation of the unexpanded visible neighbors when expanding a vertex) can be linear in the number of vertices since the runtime of each line-of-sight check can be linear in the number of vertices. In this section, we introduce Angle-Propagation Theta* (AP Theta*), which reduces the runtime of Basic Theta* per vertex expansion from linear to constant.[2] The key difference between AP Theta* and Basic Theta* is that AP Theta* propagates angle ranges and uses them to determine whether or not two vertices have line-of-sight.

If there is a light source at a vertex and light cannot pass through blocked grid cells, then grid cells in the shadows do not have line-of-sight to the vertex while all other grid cells have

[2]While AP Theta* provides a significant improvement in the worst-case complexity over Basic Theta*, our experimental results in Section 4.6 show that it is slower and finds slightly longer paths than Basic Theta*.

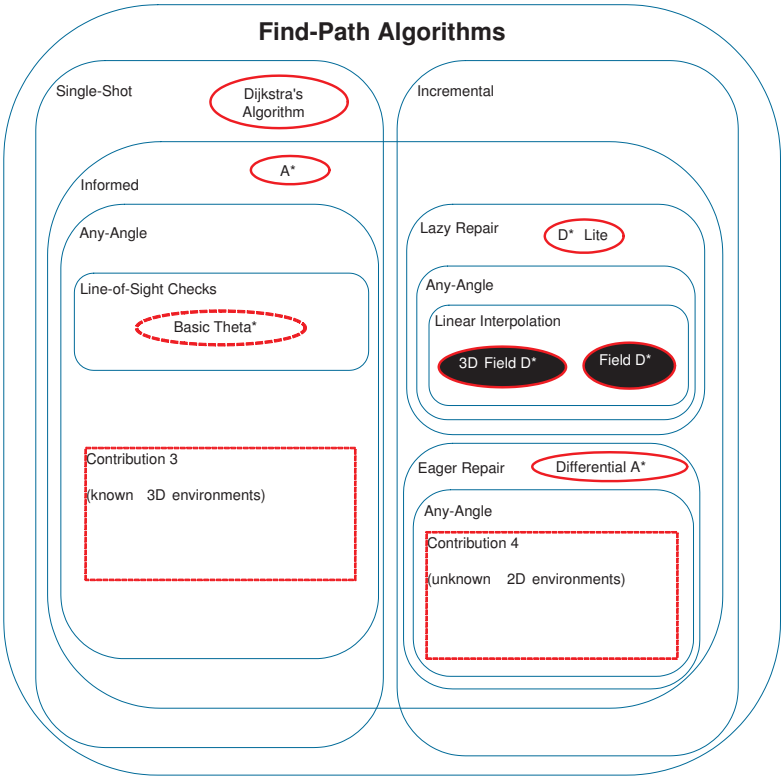


Figure 4.12: Classification of Find-Path Algorithms

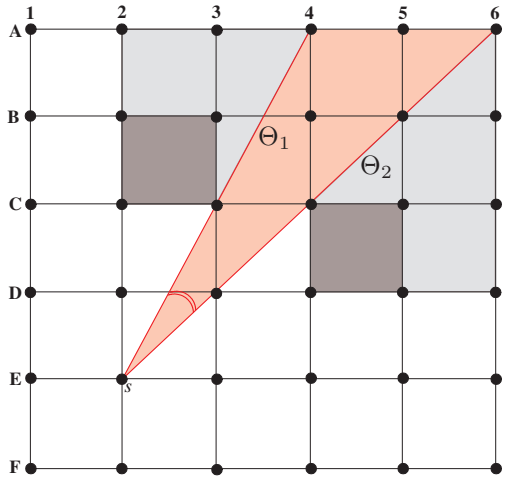


Figure 4.13: Region of Coordinates with Line-of-Sight to Vertex s

line-of-sight to the vertex. Each contiguous region of coordinates that have line-of-sight to the vertex can be characterized by two rays emanating from the vertex and thus by an angle range defined by two angle bounds. Figure 4.13 shows an example where all coordinates within the red angle range defined by the two angle bounds θ_1 and θ_2 have line-of-sight to vertex s. AP Theta*

```
54 ComputeCost(s,s')
55     if s ≠ s_start AND Θ(s, parent(s), s') ∈ [lb(s), ub(s)] then
56         /* Path 2 */
57         if g(parent(s)) + c(parent(s), s') < g(s') then
58             parent(s') := parent(s);
59             g(s') := g(parent(s)) + c(parent(s), s');
60     else
61         /* Path 1 */
62         if g(s) + c(s, s') < g(s') then
63             parent(s') := s;
64             g(s') := g(s) + c(s, s');

65 UpdateBounds(s)
66     lb(s) := −∞; ub(s) := ∞;
67     if s ≠ s_start then
68         foreach blocked grid cell b adjacent to s do
69             if ∀s' ∈ corners(b) : parent(s) = s' OR Θ(s, parent(s), s') < 0 OR
70             (Θ(s, parent(s), s') = 0 AND c(parent(s), s') ≤ c(parent(s), s)) then
71                 lb(s) = 0;
72             if ∀s' ∈ corners(b) : parent(s) = s' OR Θ(s, parent(s), s') > 0 OR
73             (Θ(s, parent(s), s') = 0 AND c(parent(s), s') ≤ c(parent(s), s)) then
74                 ub(s) = 0;
75         foreach s' ∈ nghbr_vis(s) do
76             if s' ∈ closed AND parent(s) = parent(s') AND s' ≠ s_start then
77                 if lb(s') + Θ(s, parent(s), s') ≤ 0 then
78                     lb(s) := max(lb(s), lb(s') + Θ(s, parent(s), s'));
79                 if ub(s') + Θ(s, parent(s), s') ≥ 0 then
80                     ub(s) := min(ub(s), ub(s') + Θ(s, parent(s), s'));
81             if c(parent(s), s') < c(parent(s), s) AND parent(s) ≠ s' AND (s' ∉ closed OR
               parent(s) ≠ parent(s')) then
82                 if Θ(s, parent(s), s') < 0 then
83                     lb(s) := max(lb(s), Θ(s, parent(s), s'));
84                 if Θ(s, parent(s), s') > 0 then
85                     ub(s) := min(ub(s), Θ(s, parent(s), s'));
```

Algorithm 4: AP Theta*

calculates the angle range of a vertex when it expands the vertex and then propagates it along octile graph edges, resulting in a constant runtime per vertex expansion since the angle ranges can be propagated in constant time and the line-of-sight checks can be performed in constant time as well.

Algorithm 4 shows the pseudocode of AP Theta*.[†] Procedures Main and UpdateVertex are identical to that of A* in Algorithm 1 and thus are not shown. Line 13 is to be executed. We use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in the experiments.

4.5.1 Definition of Angle Ranges

We now discuss the key concept of an angle range. AP Theta* maintains two additional values for every vertex s, namely a lower angle bound $lb(s)$ of vertex s and an upper angle bound

[†]We would like to thank Kenny Daniel for his contributions during the development of AP Theta*. AP Theta* could not have been developed without his help.

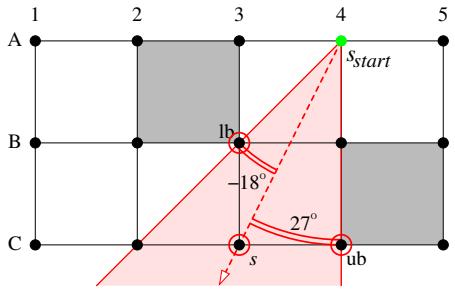


Figure 4.14: Angle Range of AP Theta*

$ub(s)$ of vertex s, that together form the angle range $[lb(s), ub(s)]$ of vertex s. The angle bounds correspond to headings of rays (measured in degrees) that originate at the parent of vertex s. The heading of the ray from the parent of vertex s to vertex s is zero degrees. A visible neighbor of vertex s is guaranteed to have line-of-sight to the parent of vertex s if (but not necessarily only if) the heading of the ray from the parent of vertex s to the visible neighbor of vertex s is contained in the angle range of vertex s. Figure 4.14 shows an example where vertex C3 with parent A4 has angle range $[-18, 27]$. Thus, all visible neighbors of vertex C3 in the red region are guaranteed to have line-of-sight to the parent of vertex C3. For example, vertex C4 is guaranteed to have line-of-sight to the parent of vertex C3 but vertex B2 is not. AP Theta* therefore assumes that vertex B2 does not have line-of-sight to the parent of vertex C3.

We now define the concept of an angle range more formally. $\Theta(s, p, s') \in [-90, 90]$, which gives AP Theta* its name, is the angle (measured in degrees) between the ray from vertex p to vertex s and the ray from vertex p to vertex s'. It is positive if the ray from vertex p to vertex s is clockwise from the ray from vertex p to vertex s', zero if the ray from vertex p to vertex s has the same heading as the ray from vertex p to vertex s', and negative if the ray from vertex p to vertex s is counterclockwise from the ray from vertex p to vertex s'. Figure 4.14 shows an example where $\Theta(C3, A4, C4) = 27$ degrees and $\Theta(C3, A4, B3) = -18$ degrees. A visible neighbor s' of vertex s is guaranteed to have line-of-sight to the parent of vertex s if (but not necessarily only if) $\Theta(s, parent(s), s') \in [lb(s), ub(s)]$ **(Visibility Property)**.

4.5.2 Updating Angle Ranges

We now discuss how AP Theta* calculates the angle range of a vertex when it expands the vertex. This calculation is complicated by the fact that AP Theta* is not guaranteed to have sufficient information to determine the angle range exactly since the order of vertex expansions depends on a variety of factors, such as the h-values. In this case, AP Theta* can constrain the angle range more than necessary to guarantee that the Visibility Property holds and that it finds unblocked paths.

When AP Theta* expands vertex s, it sets the angle range of vertex s initially to $[-\infty, \infty]$, meaning that all visible neighbors of the vertex are guaranteed to have line-of-sight to the parent of the vertex. It then constrains the angle range more and more if vertex s is not the start vertex.

AP Theta* constrains the angle range of vertex s based on each blocked grid cell b that is adjacent to vertex s (that is, that vertex s is a corner of b, written as $s \in corners(b)$) provided that at least one of two conditions is satisfied:

- **Case 1:** If every corner s' of blocked grid cell b satisfies at least one of the following conditions:
  - $parent(s) = s'$ or
  - $\Theta(s, \textit{parent}(s), s') < 0$ or
  - $\Theta(s, \textit{parent}(s), s') = 0$ and $c(\textit{parent}(s), s') \leq c(\textit{parent}(s), s)$,

  then AP Theta* assumes that a vertex s'' does not have line-of-sight to the parent of vertex s if the ray from the parent of vertex s to vertex s is counterclockwise from the ray from the parent of vertex s to vertex s'', that is, if $\Theta(s, \textit{parent}(s), s'') < 0$. AP Theta* therefore sets the lower angle bound of vertex s to $\Theta(s, \textit{parent}(s), s) = 0$ [Line 71].

- **Case 2:** If every corner s' of blocked grid cell b satisfies at least one of the following conditions:
  - $parent(s) = s'$ or
  - $\Theta(s, \textit{parent}(s), s') > 0$ or
  - $\Theta(s, \textit{parent}(s), s') = 0$ and $c(\textit{parent}(s), s') \leq c(\textit{parent}(s), s)$,

  then AP Theta* assumes that a vertex s'' does not have line-of-sight to the parent of vertex s if the ray from the parent of vertex s to vertex s is clockwise from the ray from the parent of vertex s to vertex s'', that is, if $\Theta(s, \textit{parent}(s), s'') > 0$. AP Theta* therefore sets the upper angle bound of vertex s to $\Theta(s, \textit{parent}(s), s) = 0$ [Line 74].

AP Theta* also constrains the angle range of vertex s based on each visible neighbor s' of vertex s provided that at least one of two conditions is satisfied:

- **Case 3:** If vertex s' satisfies all of the following conditions:
  - $s' \in \textit{closed}$ and
  - $\textit{parent}(s) = \textit{parent}(s')$ and
  - $s' \neq s_{\textit{start}}$,

  then AP Theta* constrains the angle range of vertex s by intersecting it with the angle range of vertex s' [Lines 78 and 80]. To do that, it first shifts the angle range of vertex s' by $\Theta(s, \textit{parent}(s), s')$ degrees to take into account that the angle range of vertex s' is calibrated so that the heading of the ray from the joint parent of vertices s and s' to vertex s' is zero degrees, while the angle range of vertex s is calibrated so that the heading of the ray from the joint parent of vertices s and s' to vertex s is zero degrees. Lines 77 and 79 ensure that the lower angle bound always remains non-positive and the upper angle bound always remains non-negative, respectively. The fact that lower angle bounds should be non-positive (and upper angle bounds non-negative) is intuitive in that if a vertex s is assigned parent vertex p then the angle of the ray from vertex p to vertex s should be included in the angle range of vertex s.

- **Case 4:** If vertex s' satisfies all of the following conditions:
  - $c(\mathit{parent}(s), s') < c(\mathit{parent}(s), s)$ and
  - $\mathit{parent}(s) \neq s'$ and
  - $s' \notin \mathit{closed}$ or $\mathit{parent}(s) \neq \mathit{parent}(s')$,

  then AP Theta* has insufficient information about vertex s'. AP Theta* therefore cannot determine the angle range of vertex s exactly and makes the conservative assumption that vertex s' barely has line-of-sight to the parent of vertex s [Lines 83 and 85].

The Visibility Property holds after AP Theta* has updated the angle range of vertex s in procedure UpdateBounds. Thus, when AP Theta* checks whether or not a visible neighbor s' of vertex s has line-of-sight to the parent of vertex s, it now checks whether or not $\Theta(s, \mathit{parent}(s), s') \in [\mathit{lb}(s), \mathit{ub}(s)]$ [Line 55] is true instead of whether or not $\mathit{LineOfSight}(\mathit{parent}(s), s')$ [Line 44] is true. These are the only differences between AP Theta* and Basic Theta*.

Figure 4.15(a) shows an example where AP Theta* calculates the angle range of vertex A4. It sets the angle range to $[-\infty, \infty]$. Figure 4.15(b) shows an example where AP Theta* calculates the angle range of vertex B3. It sets the angle range initially to $[-\infty, \infty]$. It then sets the lower angle bound to 0 degrees according to Case 1 based on the blocked grid cell A2-A3-B3-B2 [Line 71]. It sets the upper angle bound to 45 degrees according to Case 4 based on vertex B4, which is unexpanded and thus not in the closed list [Line 85]. Figure 4.15(c) shows an example where AP Theta* calculates the angle range of vertex B2. It sets the angle range initially to $[-\infty, \infty]$. It then sets the lower angle bound to 0 degrees according to Case 1 based on the blocked grid cell A2-A3-B3-B2 [Line 71]. Assume that vertex C1 is not the goal vertex. Figure 4.15(d) then shows an example where AP Theta* calculates the angle range of vertex C1. It sets the angle range initially to $[-\infty, \infty]$. It then sets the lower angle bound to -27 degrees according to Case 3 based on vertex B2 [Line 78] and the upper angle bound to 18 degrees according to Case 4 based on vertex C2, which is unexpanded and thus not in the closed list [Line 85].

4.5.3 Example Trace of AP Theta*

Figure 4.15 shows an example trace of AP Theta* using the find-path problem from Figure 4.9. The labels of the vertices now include the angle ranges.

4.5.4 Properties of AP Theta*

We now discuss the properties of AP Theta*. AP Theta* operates in the same way as Basic Theta* and thus has similar properties as Basic Theta*. For example, AP Theta* is correct and complete. It is not guaranteed to find true shortest paths, and its paths can occasionally have unnecessary heading changes.

AP Theta* sometimes constrains the angle ranges more than necessary to guarantee that it finds unblocked paths, which means that its line-of-sight checks sometimes fail incorrectly in which case it has to update vertices according to Path 1 rather than Path 2. AP Theta* is still complete since it finds an unblocked grid path if all line-of-sight checks fail, and there always exists an unblocked grid path if there exists an unblocked any-angle path. However, the paths

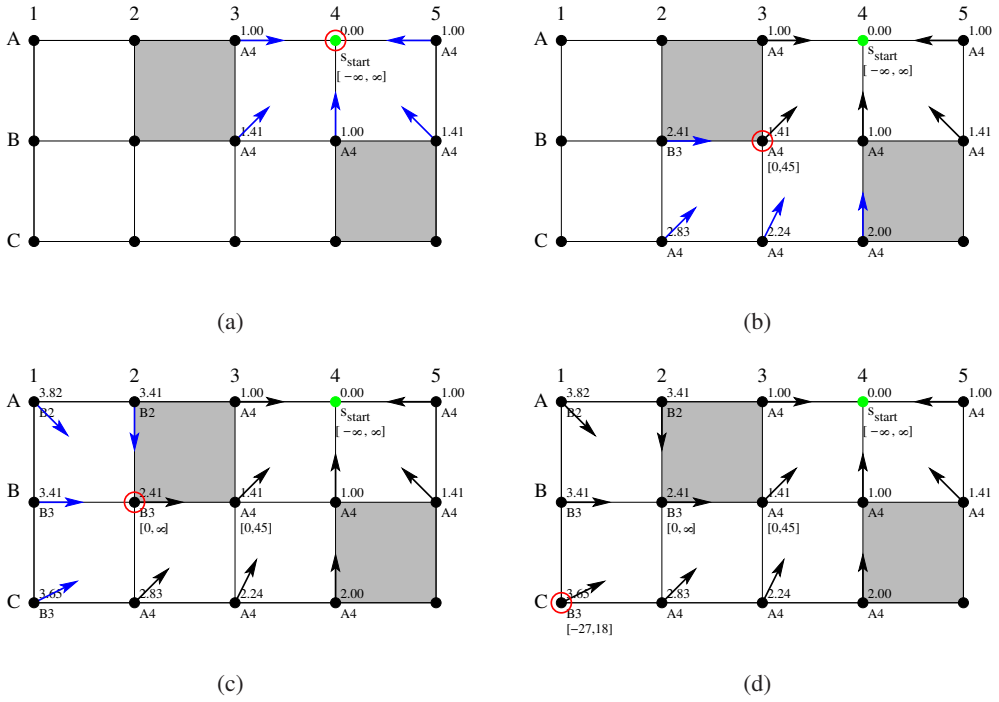


Figure 4.15: Example Trace of AP Theta*

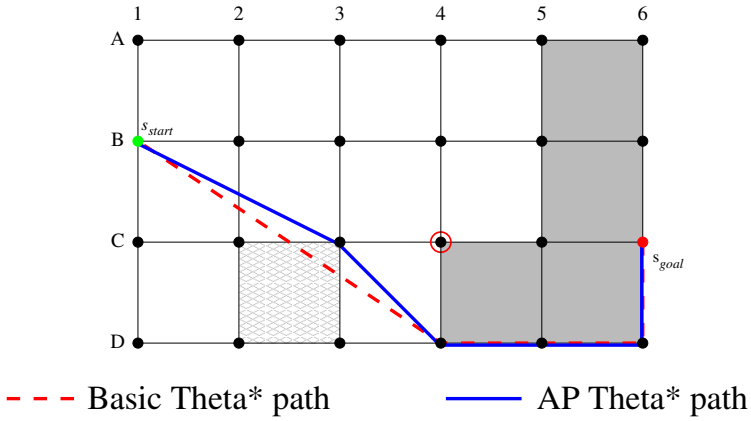


Figure 4.16: Basic Theta* Path Versus AP Theta* Path

found by AP Theta* can be longer than those found by Basic Theta*. Figure 4.16 shows an example. When AP Theta* expands vertex C4 with parent B1 and calculates the angle range of vertex C4, vertex C3 is unexpanded and thus not in the closed list. This means that AP Theta* has insufficient information about vertex C3 because, for example, it does not know whether or not grid cell C2-C3-D3-D2 is unblocked. AP Theta* therefore cannot determine the angle range of vertex C4 exactly and makes the conservative assumption that vertex C3 barely has line-of-sight to vertex B1 and sets the lower angle bound of vertex C4 according to Case 4 based on vertex

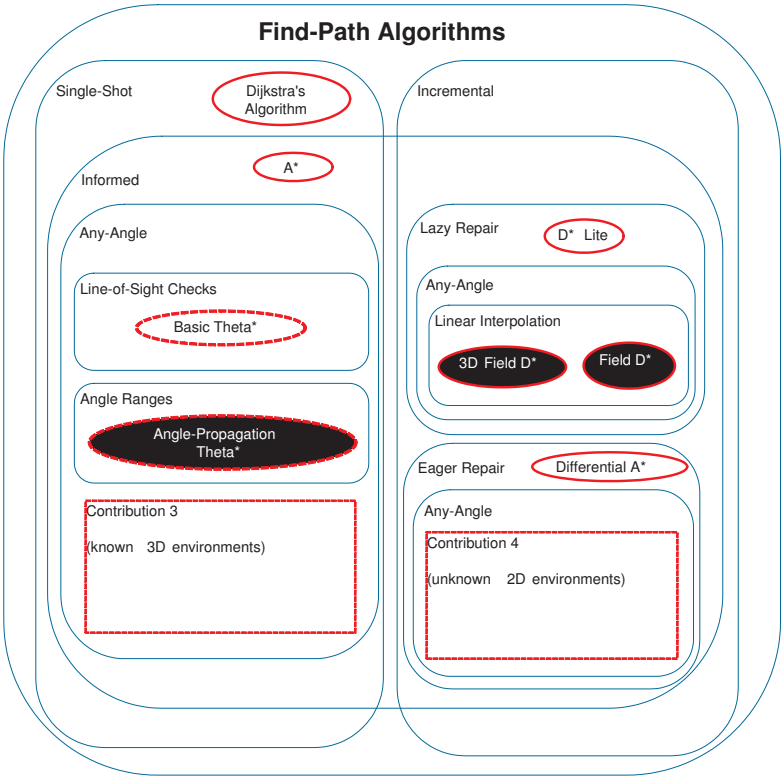


Figure 4.17: Classification of Find-Path Algorithms

C3. It then uses the resulting angle range to determine that the unexpanded visible neighbor D4 of vertex C4 is not guaranteed to have line-of-sight to vertex B1. However, vertex D4 does have line-of-sight to vertex B1 if grid cell C2-C3-D3-D2 is unblocked. AP Theta* eventually finds the solid blue path [B1, C3, D4] from start vertex B1 to vertex D4, while Basic Theta* finds the dashed red path [B1, D4], which is a true shortest path.

While AP Theta* and Basic Theta* are similar, AP Theta* performs its line-of-sight checks differently than Basic Theta*. This difference effects the properties of AP Theta*. It makes AP Theta* more difficult to implement and understand than Basic Theta*, because AP Theta*, unlike Basic Theta*, must maintain the Visibility Property (Simplicity Property). It makes AP Theta* less general than Basic Theta*, because AP Theta*, unlike Basic Theta*, cannot be used to search any Euclidean graph (Generality Property). The angle ranges maintained by AP Theta* are computed by taking advantage of properties that are unique to octile graphs. Finally, it makes the proof AP Theta*'s correctness and completeness different.

Theorem 3. *AP Theta* terminates and path extraction retrieves an unblocked path from the start vertex to the goal vertex if such a path exists. Otherwise, AP Theta* terminates and reports that no unblocked path exists.*

Proof. The proof is similar to the proof of Theorem 2 since AP Theta* uses the angle ranges only to determine whether or not Path 2 is blocked but not to determine whether or not Path 1 is blocked. The only property that needs to be proved differently is that two vertices indeed have

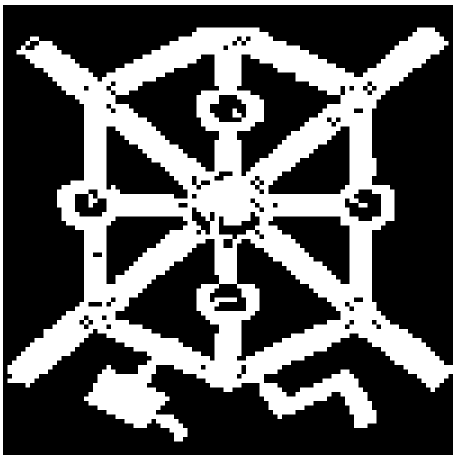


Figure 4.18: Game Map from Baldur's Gate II (BioWare)

line-of-sight if (but not necessarily only if) the line-of-sight check of AP Theta* succeeds, see Appendix B. □

Figure 4.17 provides an updated summary of our classification of different find-path algorithms. It is an updated version of Figure 4.12 which accounts for the contributions made so far in this chapter. Figures 4.12 and 4.17 are annotated in the same way. The AP Theta* oval is within 5 rounded rectangles, and thus AP Theta* is a **1** single-shot, **2** informed, **3** any-angle **4** find-path algorithm that uses **5** angle ranges. AP Theta* and Basic Theta* fall into different classes of any-angle find-path algorithms, the former finds any-angle paths by propagating angle ranges and the latter finds any-angle paths by performing line-of-sight checks.

		FD*	Basic Theta*	AP Theta*	A* on Visibility Graphs (true shortest paths)	A* on Octile Graphs (shortest grid paths)	A* PS
100×100	Game Maps	40.04	39.98	40.05	39.96	41.77	40.02
	Random Grids 0%	114.49	114.33	114.33	114.33	120.31	114.33
	Random Grids 5%	114.15	113.94	113.94	113.83	119.76	114.71
	Random Grids 10%	114.74	114.51	114.51	114.32	119.99	115.46
	Random Grids 20%	115.20	114.93	114.95	114.69	120.31	116.16
	Random Grids 30%	115.45	115.22	115.25	114.96	120.41	116.69
500×500	Game Maps	223.64	223.30	224.40	N/A	233.66	223.70
	Random Grids 0%	576.19	575.41	575.41	N/A	604.80	575.41
	Random Grids 5%	568.63	567.30	567.34	N/A	596.45	573.46
	Random Grids 10%	576.23	574.57	574.63	N/A	603.51	581.03
	Random Grids 20%	580.19	578.41	578.51	N/A	604.93	585.62
	Random Grids 30%	581.73	580.18	580.35	N/A	606.38	588.98

Table 4.1: Path Lengths

		FD*	Basic Theta*	AP Theta*	A* on Visibility Graphs (true shortest paths)	A* on Octile Graphs (shortest grid paths)	A* PS
100×100	Game Maps	0.0111	0.0060	0.0084	0.4792	0.0048	0.0052
	Random Grids 0%	0.0229	0.0073	0.0068	0.0061	0.0053	0.0208
	Random Grids 5%	0.0275	0.0090	0.0111	0.0766	0.0040	0.0206
	Random Grids 10%	0.0305	0.0111	0.0145	0.3427	0.0048	0.0204
	Random Grids 20%	0.0367	0.0150	0.0208	1.7136	0.0084	0.0222
	Random Grids 30%	0.0429	0.0183	0.0263	3.7622	0.0119	0.0240
500×500	Game Maps	0.1925	0.1166	0.1628	N/A	0.0767	0.1252
	Random Grids 0%	0.3628	0.1000	0.0234	N/A	0.0122	0.6270
	Random Grids 5%	0.4514	0.1680	0.1962	N/A	0.0176	0.6394
	Random Grids 10%	0.5608	0.2669	0.3334	N/A	0.0573	0.6717
	Random Grids 20%	0.6992	0.3724	0.5350	N/A	0.1543	0.6852
	Random Grids 30%	0.8562	0.5079	0.7291	N/A	0.3238	0.7355

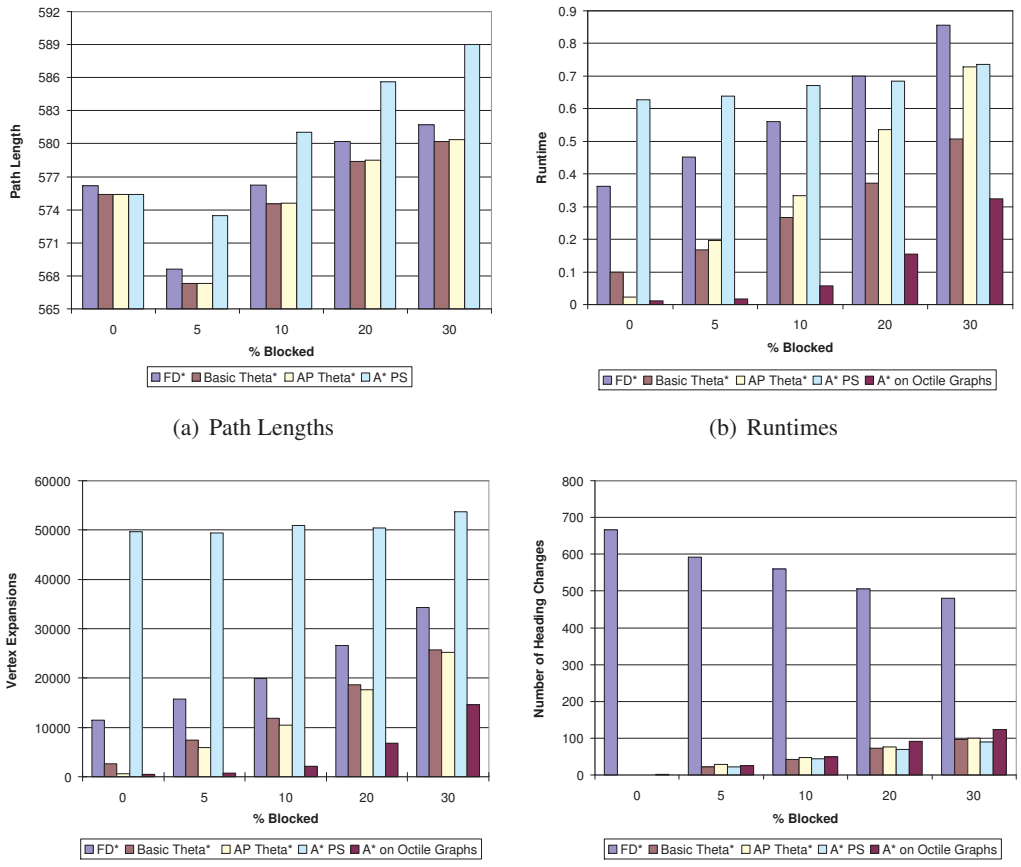
Table 4.2: Runtimes

		FD*	Basic Theta*	AP Theta*	A* on Visibility Graphs (true shortest paths)	A* on Octile Graphs (shortest grid paths)	A* PS
100×100	Game Maps	247.07	228.45	226.42	68.23	197.19	315.08
	Random Grids 0%	592.74	240.42	139.53	1.00	99.00	1997.29
	Random Grids 5%	760.17	430.06	361.17	35.35	111.96	1974.27
	Random Grids 10%	880.21	591.31	520.91	106.23	169.98	1936.56
	Random Grids 20%	1175.42	851.79	813.14	357.33	386.41	2040.10
	Random Grids 30%	1443.44	1113.40	1089.96	659.36	620.18	2153.28
500×500	Game Maps	6846.62	6176.37	6220.58	N/A	5580.32	9673.88
	Random Grids 0%	11468.11	2603.40	663.34	N/A	499.00	49686.47
	Random Grids 5%	15804.81	7450.85	5917.25	N/A	755.66	49355.41
	Random Grids 10%	19874.62	11886.95	10405.34	N/A	2203.83	50924.01
	Random Grids 20%	26640.83	18621.61	17698.75	N/A	6777.15	50358.66
	Random Grids 30%	34313.28	25744.57	25224.92	N/A	14641.36	53732.82

Table 4.3: Number of Vertex Expansions

		FD*	Basic Theta*	AP Theta*	A* on Visibility Graphs (true shortest paths)	A* on Octile Graphs (shortest grid paths)	A* PS
100×100	Game Maps	34.25	3.08	3.64	2.92	5.21	2.83
	Random Grids 0%	123.40	0.00	0.00	0.00	0.99	0.00
	Random Grids 5%	113.14	5.14	6.03	5.06	6.00	4.53
	Random Grids 10%	106.66	8.96	9.87	8.84	10.85	8.48
	Random Grids 20%	98.76	15.21	15.96	14.74	19.42	14.45
	Random Grids 30%	96.27	19.96	20.62	19.44	26.06	18.35
500×500	Game Maps	219.70	4.18	7.58	N/A	10.19	3.84
	Random Grids 0%	667.00	0.00	0.00	N/A	1.00	0.00
	Random Grids 5%	592.65	21.91	27.99	N/A	24.68	22.27
	Random Grids 10%	559.69	41.60	47.40	N/A	49.73	43.16
	Random Grids 20%	506.10	72.49	76.79	N/A	91.40	69.44
	Random Grids 30%	481.16	97.21	100.31	N/A	123.81	89.43

Table 4.4: Number of Heading Changes



(a) Path Lengths

(b) Runtimes

(c) Number of Vertex Expansions

(d) Number of Heading Changes

Figure 4.19: Find-Path Algorithms on Random 500×500 Grids

4.6 Experimental Results

In this section, we compare Basic Theta*, AP Theta*, A* on octile graphs, A* PS, FD* and A* on visibility graphs with respect to their path length, number of vertex expansions, runtime (measured in seconds) and number of heading changes.

We compare these find-path algorithms on 100×100 and 500×500 octile graphs with different percentages of randomly blocked grid cells (random grids) and scaled maps from the real-time strategy game Baldur's Gate II (game maps).[†] Figure 4.18 shows an example of a game map (Bulitko, Sturtevant, & Kazakevich, 2005). For random grids, the start vertex is in the lower left corner of the square grid and the goal vertex is in the corner of a grid cell randomly chosen from the right most column of grid cells in the square grid. This technique for selecting the start vertex and goal vertex in random grids was used in the work of Ferguson and Stentz (2006). Grid cells are blocked randomly but a one-unit border of grid cells around the square grid is left unblocked in order to guarantee that there is path from the start vertex to the goal vertex. An

[†]We would like to thank Vadim Bulitko from the University of Alberta for making maps from Baldurs Gate II available to us.

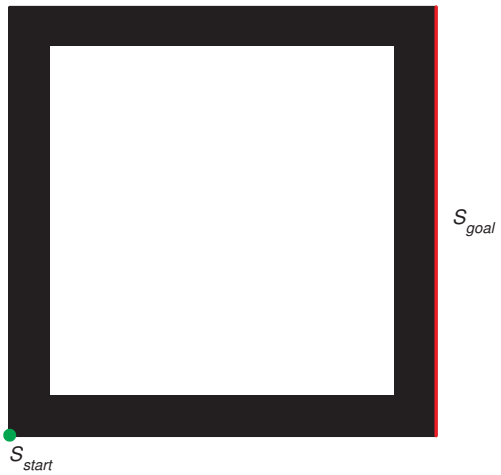


Figure 4.20: Start and Goal Vertices on Random Grids

illustration of the experimental setup for random grids can be seen in Figure 4.20, where the green circle depicts the location of the start vertex, the goal vertex is a randomly selected vertex on the red line and the black region is the one-unit border of *unblocked* grid cells around the square grid. In Appendix E.3, we provide results for random grids in which there is not a one-unit border of unblocked grid cells around the square grid and the start and goal vertices are randomly selected from the corners of unblocked grid cells. This technique for selecting the start vertex and goal vertex in random grids was used in the work of Yap et al. (2011), Šišlák et al. (2009b) and Choi and Yu (2011). Similar conclusions can be drawn about the relationships between the different find-path algorithms using either experimental setup. For game maps, the start and goal vertices are randomly chosen from the corners of unblocked grid cells. We average over 500 random 100×100 grids, 500 random 500×500 grids and 118 game maps.

All find-path algorithms are implemented in C# and executed on a 3.7 GHz Core 2 Duo with 2 GByte of RAM. Our implementations are not optimized and can possibly be improved.

A* on octile graphs, A* PS, FD* and A* on visibility graphs break ties among vertices with the same f-value in the open list in favor of vertices with larger g-values (when they decide which vertex to expand next) since this tie-breaking scheme typically results in fewer vertex expansions and thus shorter runtimes for A*. Care must thus be taken when calculating the g-values, h-values and f-values precisely. The numerical precision of these floating point numbers can be improved for A* on octile graphs by representing them in the form $m + \sqrt{2}n$ for integers m and n.[3] Basic Theta* and AP Theta* break ties in favor of vertices with smaller g-values for the reasons explained in Section 4.8.

We use all find-path algorithms with consistent h-values. Consistent h-values ensure that A* on octile graphs finds shortest grid paths. Consistent h-values satisfy the triangle inequality, that is, the h-value of the goal vertex is zero and the h-value of any potential non-goal parent of any vertex is no greater than the distance from the potential non-goal parent of the vertex to the vertex plus the h-value of the vertex (Hart et al., 1968; Pearl, 1985). Consistent h-values are lower bounds on the corresponding goal distances of vertices. Increasing consistent h-values typically

[3]This technique was not used in our experiments.

```
86 h(s)
87     Δ_x := |s.x − (s_goal).x|;
88     Δ_y := |s.y − (s_goal).y|;
89     largest := max(Δ_x, Δ_y);
90     smallest := min(Δ_x, Δ_y);
91     return √2 · smallest + (largest − smallest);
```

Algorithm 5: Calculation of Octile Distances

decreases the number of vertex expansions for A* and thus also the runtime of A*. Thus, we use all find-path algorithms with the largest consistent h-values that are easy to calculate. For Basic Theta*, AP Theta*, FD* and A* on visibility graphs, the goal distances of vertices can be equal to the true goal distances, that is, the goal distances on octile graphs if the paths are not constrained to be formed by octile graphs edges. We therefore use these find-path algorithms with the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in our experiments. The straight-line distances are the goal distances on octile graphs without blocked grid cells if the paths are not constrained to be formed by octile graphs edges. For A* on octile graphs and A* PS, the goal distances of vertices are equal to the goal distances on octile graphs if the paths are constrained to be formed by octile graph edges. We could therefore use them with the larger octile distances as h-values in our experiments. The octile distances are the goal distances on octile graphs without blocked grid cells if the paths are constrained to be formed by octile graph edges. Algorithm 5 shows how to calculate the octile distance of a given vertex s, where $s.x$ and $s.y$ are the x-coordinate and y-coordinate of vertex s, respectively. We use A* on octile graphs with the octile distances but A* PS with the straight-line distances. When using A* PS with the straight-line distances smoothing is typically able to shorten the resulting paths much more at an increase in the number of vertex expansions and thus runtime. Square grids without blocked grid cells provide an example. With the octile distances as h-values, A* on octile graphs finds paths in which all diagonal movements (whose lengths are $\sqrt{2}$) precede all horizontal or vertical movements (whose lengths are 1) due to the tie-breaking scheme used because the paths with the largest number of diagonal movements are the longest among all paths with the same number of movements. On the other hand, with the straight-line distances as h-values, A* on octile graphs finds paths that interleave the diagonal movements with the horizontal and vertical movements (which means that it is likely that there are lots of opportunities to smooth the paths even for grids with some blocked grid cells) and that are closer to the straight line between the start vertex and the goal vertex (which means that it is likely that the paths are closer to true shortest paths even for grids with some blocked grid cells), because the h-values of vertices closer to the straight line are typically smaller than the h-values of vertices farther away from the straight line.

Tables 4.1-4.4 report our experimental results. The runtime of A* on visibility graphs (which finds true shortest paths) is too long on 500×500 grids and thus is omitted. Figure 4.19 visualizes our experimental results on random 500×500 grids. The path length of A* on octile graphs is much larger than the path lengths of the other find-path algorithms and thus is omitted.

We make the following observations:

- **Path Lengths (Table 4.1):**
  - The find-path algorithms in order of increasing path lengths tend to be: A* on visibility graphs (which finds true shortest paths), Basic Theta*, AP Theta*, FD*, A* PS

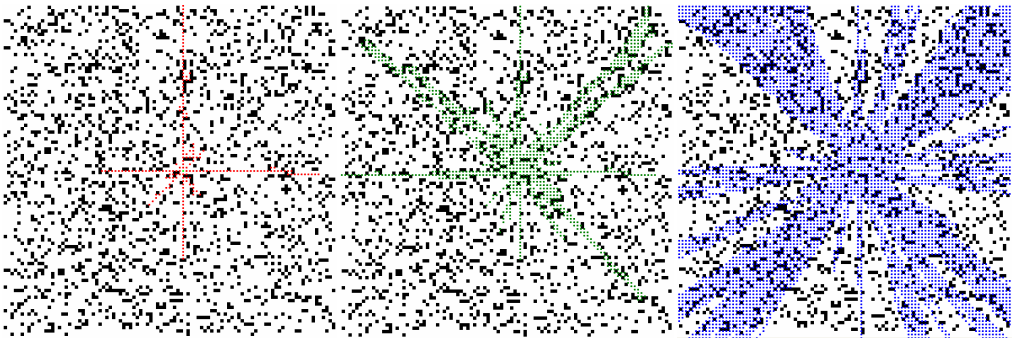


Figure 4.21: True Shortest Paths Found by FD* (left), A* PS (middle) and Basic Theta* (right)

and A* on octile graphs. On random 500×500 grids with 20 percent blocked grid cells, Basic Theta* finds shorter paths than AP Theta* 70 percent of the time, shorter paths than FD* 97 percent of the time, shorter paths than A* PS 94 percent of the time and shorter paths than A* on octile graphs 99 percent of the time.

- The paths found by Basic Theta* and AP Theta* are almost as short as the true shortest paths even though AP Theta* sometimes constrains the angle ranges more than necessary. For example, they are on average less than a factor of 1.003 longer than true shortest paths on random 100×100 grids.
- Basic Theta* finds true shortest paths more often than FD* and A* PS. Figure 4.21 shows an example where the light green vertex in the center is the start vertex and the red, green and blue vertices represent goal vertices to which FD*, A* PS and Basic Theta* find true shortest paths, respectively.

- **Runtimes (Table 4.2):** The find-path algorithms in order of increasing runtimes tend to be: A* on octile graphs, Basic Theta*, AP Theta*, A* PS, FD* and A* on visibility graphs.
- **Vertex Expansions (Table 4.3):** The find-path algorithms in order of increasing numbers of vertex expansions tend to be: A* on visibility graphs, A* on octile graphs, AP Theta*, Basic Theta*, FD* and A* PS. (The number of vertex expansions of A* on octile graphs and A* PS are different because we use them with different h-values.)
- **Heading Changes (Table 4.4):** The find-path algorithms in order of increasing numbers of heading changes tend to be: A* PS, A* on visibility graphs, Basic Theta*, AP Theta*, A* on octile graphs and FD*.

There are some exceptions to the trends reported above. We therefore perform paired t-tests. They show with confidence level $\alpha = 0.01$ that Basic Theta* indeed finds shorter paths than AP Theta*, A* PS and FD* and that Basic Theta* indeed has a shorter runtime than AP Theta*, A* PS and FD*.

To summarize, A* on visibility graphs finds true shortest paths but is slow. On the other hand, A* on octile graphs finds long paths but is fast. Any-angle find-path algorithms lie between these two extremes and thus our runtime and path length results demonstrate that Basic Theta*, and to a lesser extent AP Theta*, satisfy the Efficiency Property, that is, they provide a good tradeoff with

	FD*	Basic Theta*	AP Theta*	A* PS
Runtime	5.21	3.65	5.70	3.06
Runtime per Vertex Expansion	0.000021	0.000015	0.000023	0.000012

Table 4.5: Find-Path Algorithms without Post-Processing Steps on Random 500×500 Grids with 20 Percent Blocked Grid Cells

respect to the runtime of the search and the length of the resulting path. Basic Theta* dominates AP Theta*, A* PS and FD* in terms of the tradeoff it provides with respect to the runtime of the search and the length of the resulting path. It finds paths that are almost as short as true shortest paths and is almost as fast as A* on octile graphs. It is also simpler to implement and understand than AP Theta*. Therefore, we build on Basic Theta* for the remainder of this chapter, although we report some experimental results for AP Theta* as well. However, AP Theta* reduces the runtime of Basic Theta* per vertex expansion from linear to constant.

4.7 Extensions of Basic Theta* and AP Theta*

In this section, we extend Basic Theta* and AP Theta* to find paths from a given start vertex to all other vertices and we extend Basic Theta* to find paths on square grids that contain unblocked grid cells with non-uniform traversal costs.

4.7.1 Single Source Paths

So far, Basic Theta* has found paths from a given start vertex to a given goal vertex. We now discuss Single Source Basic Theta*, a variant of Basic Theta* that finds single source paths (that is, paths from a given start vertex to all other vertices) by terminating only when the open list is empty instead of when either the open list is empty or it expands the goal vertex.

Finding single source paths requires that all find-path algorithms expand the same number of vertices, which eliminates the influence of the h-values on the runtime and thus results in a clean comparison since the h-values are sometimes chosen to vary the tradeoff that a find-path algorithm provides with respect to the runtime of the search and the length of the resulting path. We eliminate the influence of h-values by assuming that, for every vertex $s \in V$, $h(s) = 0$ and thus $f(s) = g(s)$.

The runtimes of A* PS and FD* are affected more than those of Basic Theta* and AP Theta* when finding single source paths since they require post-smoothing or path-extraction steps for each path, and thus need to post process many paths. Table 4.5 reports the runtimes of the find-path algorithms without these post-processing steps. The runtime of Basic Theta* per vertex expansion is similar to that of A* PS and shorter than that of either AP Theta* and FD* because the latter two algorithms require more floating point computations.

Single Source Basic Theta* obeys the Closed List Property (Section 2.2.2.1), that is, when a vertex s is added to the closed list for the first time, a shortest grid path from the start vertex to s has been found. This unblocked path can be extracted by following parents from vertex s to the start vertex in reverse and the g-value of vertex s is its length. We use the following lemma in the proof.

Lemma 5. *(At any point during the execution of Single Source Basic Theta*:) Following the parents from any vertex s in the open or closed lists to the start vertex retrieves an unblocked path from the start vertex to vertex s in reverse and the g-value of vertex s is its length.*

Proof. We prove by induction that the lemma holds and that the parent of any vertex in the union of the open or closed lists itself is in the union of the open or closed lists. This statement holds initially because the start vertex is the only vertex in the union of the open or closed lists, it is its own parent and $g(s_{start}) = 0$. We now show that the statement continues to hold whenever a vertex changes either its parent and g-value (whenever the parent of a vertex is changed, its g-value is changed on the next line) or its membership in the union of the open or closed lists. Once a vertex is a member of the union of the open or closed lists, it continues to be a member. A vertex can become a member in the union of the open or closed lists only when Single Source Basic Theta* expands some vertex s and updates the g-value and parent of an unexpanded visible neighbor s' of vertex s in procedure UpdateVertex. Vertex s is in the closed list and thus, according to the induction assumption, following the parents from vertex s to the start vertex retrieves an unblocked path from the start vertex to vertex s in reverse and the g-value of vertex s is its length. The parent of vertex s is in the union of the open or closed lists because vertex s is in the closed list and thus, according to the induction assumption, following the parents from the parent of vertex s to the start vertex retrieves an unblocked path from the start vertex to the parent of vertex s in reverse and the g-value of the parent of vertex s is its length. Thus, following the parents from vertex s (or its parent) to the start vertex retrieves an unblocked path from the start vertex to vertex s (or its parent, respectively) in reverse and the g-value of vertex s is its length (or the g-value of its parent, respectively) according to the induction assumption. If Basic Theta* updates vertex s' according to Path 1, then the statement continues to hold since vertices s and s' are visible neighbors which means the path segment from vertex s to vertex s' is unblocked, the parent of vertex s' is vertex s and the g-value of vertex s' is $g(s)+c(s, s')$. If Basic Theta* updates vertex s' according to Path 2, then the statement continues to hold since Basic Theta* explicitly checks that the path segment from the parent of vertex s to vertex s' is unblocked, the parent of vertex s' is the parent of vertex s and the g-value of vertex s' is $g(\mathit{parent}(s)) + c(\mathit{parent}(s), s')$. Furthermore, since vertex s' is in the open list and the parent of vertex s' is in the closed list, no vertex can have vertex s' as its parent and thus it cannot be the case that the g-value of a vertex no longer represents the length of the path from the start vertex to that vertex because vertex s' changed its g-value and parent (this also ensures that a cycle can never be introduced). There are no other ways in which the parent and g-value of a vertex can change. □

Theorem 4. *When a vertex s is added to the closed list for the first time, a shortest grid path from the start vertex to vertex s has been found. This unblocked path can be extracted by following parents from vertex s to the start vertex in reverse and the g-value of vertex s is its length.*

Proof. We prove by induction that, every time a vertex is added to the closed list, the theorem holds. This statement holds initially because the start vertex is the only vertex in the closed list, it is its own parent and $g(s_{start}) = 0$. We now show that the statement continues to hold whenever a vertex is added to the closed list. Consider an arbitrary iteration of Single Source Basic Theta* in which a vertex s is selected to be added to the closed list. The statement holds for every vertex in the closed list according to the induction assumption. Any path $\widehat{P}$ from the start vertex to vertex s, whether it be an any-angle path or a grid path, must contain at least one path segment

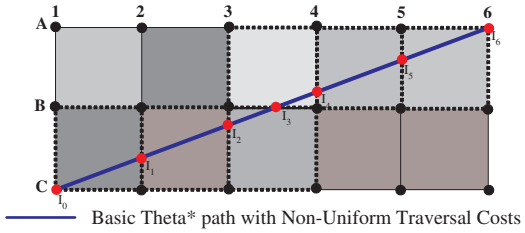


Figure 4.22: Basic Theta* on Square Grids with Grid Cells with Non-Uniform Traversal Costs of Unblocked Grid Cells

$\overline{u, v}$ where vertex u is a member of the closed list and vertex v is not a member of the closed list, because the start vertex is a member of the closed list and vertex s is not a member of the closed list. The length of $\widehat{P}$ is greater than or equal to the length of the subpath of $\widehat{P}$ from the start vertex to vertex v which is equal to the length of the subpath of $\widehat{P}$ from the start vertex to vertex $u + c(u, v) \geq g(u) + c(u, v) \geq g(v) \geq g(s)$. In the string of inequalities above, the first is true because our edge lengths are non-negative; the equality is true by definition; the second inequality is true according to Lemma 5 and the inductive hypothesis applied to vertex u which is a member of the closed list; the third is true because vertex u is a member of the closed list and must have been examined during procedure UpdateVertex during a previous iteration; the fourth is true because vertex s was chosen on Line 8 of the current iteration. If this relationship holds for any path then it must also hold for a shortest grid path. Therefore, when a vertex s is added to the closed list for the first time, a shortest grid path from the start vertex to vertex s has been found. This unblocked path can be extracted by following parents from vertex s to the start vertex in reverse and the g-value of vertex s is its length according to Lemma 5. Therefore, the theorem holds. □

4.7.2 Non-Uniform Traversal Costs

So far, Basic Theta* has found paths on octile graphs constructed from square grids that contain unblocked grid cells with uniform traversal costs. In this case, true shortest paths have heading changes only at the corners of blocked grid cells and the triangle inequality holds, which means that Path 2 is no longer than Path 1. We now discuss a variant of Basic Theta* that finds paths on octile graphs constructed from square grids that contain unblocked grid cells with non-uniform traversal costs by computing and comparing path lengths (which are now path costs) appropriately. In this case, true shortest paths can also have heading changes at the boundaries between unblocked grid cells with different traversal costs and the triangle inequality is no longer guaranteed to hold, which means that Path 2 can be more costly than Path 1. Thus, Basic Theta* no longer unconditionally chooses Path 2 over Path 1 if Path 2 is unblocked [Line 44] but chooses the path with the smaller cost. It uses the standard Cohen-Sutherland clipping algorithm from computer graphics (Foley, van Dam, Feiner, & Hughes, 1992) to calculate the cost of Path 2 during the line-of-sight check. Figure 4.22 shows an example for the path segment $\overline{C1, A6}$ from vertex C1 to vertex A6. This straight line is split into grid segments at the coordinates where it intersects grid cell boundaries. The different shades of grey denote grid cells with different traversal costs and the dashed lines denote grid cells traversed by path segment $\overline{C1, A6}$. The cost of the path

(a) Small Contiguous Regions of Uniform Traversal Costs

	A* on Octile Graphs	FD*	Basic Theta*
Path Cost	4773.59	4719.26	4730.96
Runtime	11.28	14.98	19.02

(b) Large Contiguous Regions of Uniform Traversal Costs

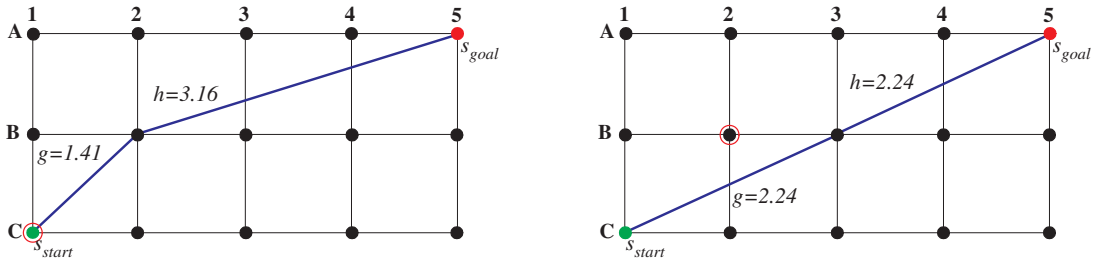
	A* on Octile Graphs	FD*	Basic Theta*
Path Cost	1251.88	1208.89	1207.06
Runtime	3.42	5.31	5.90

Table 4.6: Find-Path Algorithms on Random 1000×1000 Grids with Non-Uniform Traversal Costs

segment $\overline{C1, A6}$ is the sum of the costs of its grid segments $\overline{I_i, I_{i+1}}$, and the cost of each grid segment is the product of its length and the traversal cost of the corresponding unblocked grid cell.

We found that changing the test on Line 51 in Algorithm 3 from "strictly less than" to "less than or equal to" slightly reduces the runtime of Basic Theta*. This is a result of the fact that it is faster to compute the cost of a path segment that corresponds to Path 1 than Path 2 since it tends to consist of fewer grid segments.

We compare Basic Theta* to A* on octile graphs and FD* with respect to their path cost and runtime (measured in seconds) since A* can easily be adapted to grids that contain unblocked grid cells with non-uniform traversal costs and FD* was designed for this case. We compare these find-path algorithms on 1000×1000 grids, where each grid cell is assigned an integer traversal cost from 1 to 15 (corresponding to an unblocked grid cell) and infinity (corresponding to a blocked grid cell). If a path lies on the boundary between two grid cells with different traversal costs, then we use the smaller traversal cost of the two grid cells. The start vertex is in the lower left corner of the square grid and the goal vertex is in the corner of a grid cell randomly chosen from the right most column of grid cells in the square grid. We average over 100 random grids. This was the same experimental setup that was used in the work of Ferguson and Stentz (2006). Table 4.6 (a) reports our results if every traversal cost is chosen with uniform probability, resulting in small contiguous regions of uniform traversal costs. The path cost and runtime of FD* are both smaller than those of Basic Theta*. The path cost of A* on octile graphs is only about 1 percent larger than that of FD* although its runtime is much smaller than that of FD*. Thus, any-angle find-path algorithms do not have a large advantage over A* on octile graphs. Table 4.6(b) reports our results if traversal cost one is chosen with probability 50 percent and all other traversal costs are chosen with uniform probability, resulting in large contiguous regions of uniform traversal costs. The path cost of Basic Theta* is now smaller than that of FD* and its runtime is about the same as that of FD*. The paths found by FD* tend to have many more unnecessary heading changes in regions with the same traversal costs than those of Basic Theta* (Section 4.6), which outweighs the paths found by Basic Theta* not having necessary heading changes on the boundary between two grid cells with different traversal costs. The path cost of A* on octile graphs is more than 3 percent larger than that of Basic Theta*. Thus, any-angle find-path algorithms now has a larger advantage over A* on octile graphs. Recently, a new technique



(a) (b)

Figure 4.23: Non-Monotonicity of f-Values of Basic Theta*

was introduced by Choi and Yu (2011) which extends Basic Theta* to octile graphs constructed from square grids that contain unblocked grid cells with non-uniform traversal costs. We discuss this technique in Appendix E.2.2.

4.8 Trading Off Runtime and Path Length: Exploiting h-Values

There are strategies for trading off the runtime of the search and length of the resulting path that A* on octile graphs and Basic Theta* share. However, their behavior can be different even though the two find-path algorithms have similar pseudocode. In this section, we develop variants of Basic Theta* that might be able to find shorter paths at an increase in runtime, including variants that use weighted h-values with weights less than one, that break ties among vertices with the same f-value in the open list in favor of vertices with smaller g-values (when they decide which vertex to expand next) and that re-expand vertices whose f-values have decreased.

We use all find-path algorithms with consistent h-values. A* on octile graphs then has the following properties (Pearl, 1985): The f-value of any expanded vertex is no larger than the f-value of any of its unexpanded visible neighbors after updating them according to Path 1, which implies that the f-value of any vertex that is expanded before some other vertex is no larger than the f-value of this other vertex. Consequently, the Closed List Property holds. In other words, at any point in time during a search once a vertex has been expanded, following the parents from the expanded vertex to the start vertex retrieves a shortest grid path from the start vertex to the expanded vertex in reverse, which implies that A* cannot find shorter paths by expanding vertices more than once. Basic Theta* has different properties: The f-value of an expanded vertex can be larger than the f-value of one or more of its unexpanded visible neighbors after updating them according to Path 2, which implies that the f-value of a vertex that is expanded before some other vertex can be larger than the f-value of this other vertex. Consequently, the Closed List Property does not necessarily hold. In other words, at any point in time during a search once a vertex has been expanded, following the parents from the expanded vertex to the start vertex is not guaranteed to retrieve a shortest grid path from the start vertex to the expanded vertex in reverse, which implies that Basic Theta* might find shorter paths by expanding vertices more than once. Figure 4.23 shows an example. When Basic Theta* expands start vertex C1 with parent C1, it generates vertex B2. Vertex B2 is an unexpanded visible neighbor of vertex C1 that has line-of-sight to vertex C1. Basic Theta* thus updates it according to Path 2 (which is the same as Path 1 in this case), sets its f-value to $f(B2) = 1.41 + 3.16 = 4.57$, sets its parent to vertex

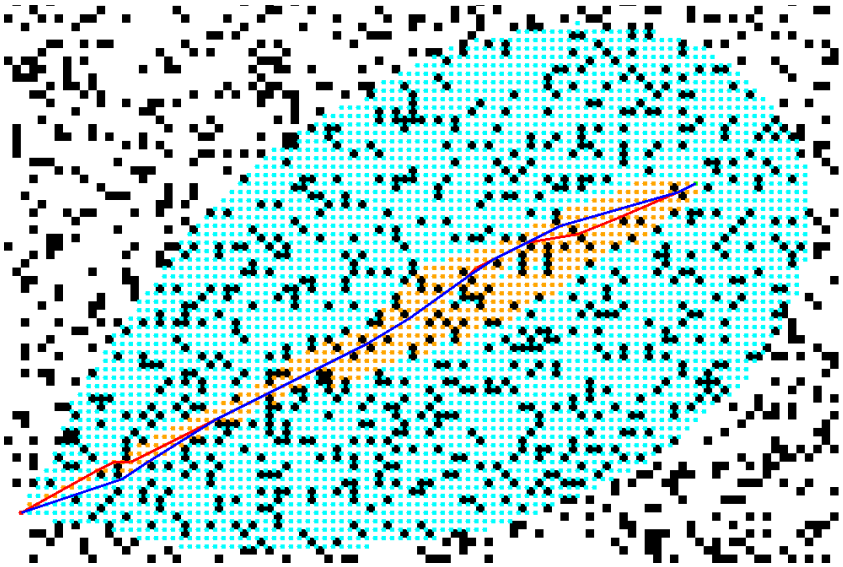
C1 and inserts it into the open list (Figure 4.23(a)). When Basic Theta* later expands vertex B2 with parent C1, it generates vertex B3. Vertex B3 is an unexpanded visible neighbor of vertex B2 that has line-of-sight to vertex C1. Basic Theta* thus updates it according to Path 2, sets its f-value to $f(B3) = 2.24 + 2.24 = 4.48$, sets its parent to vertex C1 and inserts it into the open list (Figure 4.23(b)). Thus, the f-value of expanded vertex B2 is indeed larger than the f-value of its unexpanded visible neighbor B3 after updating it according to Path 2 because the increase in g-value from vertex B2 to vertex B3 [= 0.83] is less than the decrease in h-value from vertex B2 to vertex B3 [= 0.92]. When Basic Theta* later expands vertex B3, the f-value of vertex B2 [= 4.57] that is expanded before vertex B3 is indeed larger than the f-value of vertex B3 [= 4.48].

These properties suggest that Basic Theta* might be able to find shorter paths at an increase in runtime by re-expanding vertices or expanding additional vertices (for example, by using weighted h-values with weights less than one) while A* cannot. At the same time, standard optimizations of A* that decrease its runtime might also be able to decrease the runtime of Basic Theta* (such as breaking ties among vertices with the same f-value in the open list in favor of vertices with larger g-values). In this section, we investigate these tradeoffs.

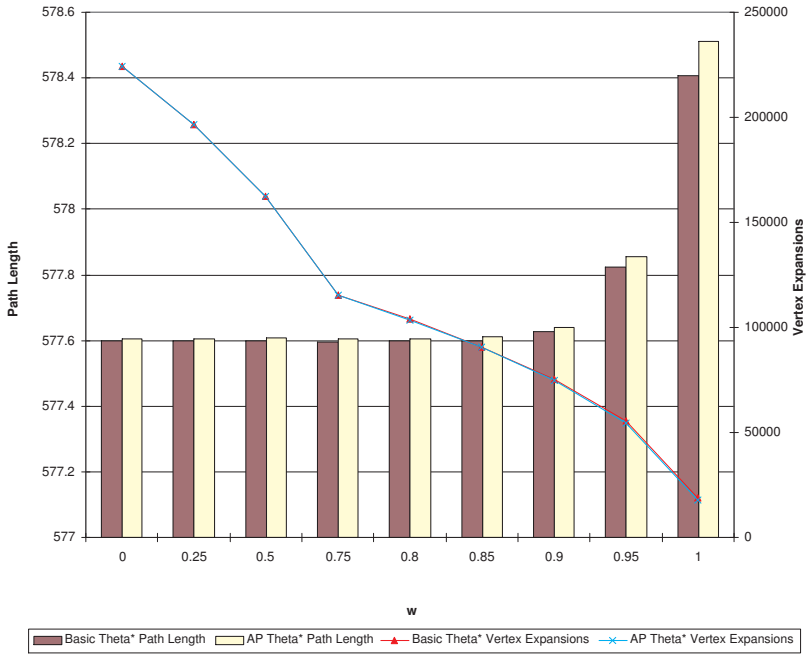
4.8.1 Weighted h-Values with $w < 1$

So far, Basic Theta* has used consistent h-values. A* with consistent h-values finds paths of the same length no matter how small or large the h-values are. Decreasing consistent h-values typically increases the number of vertex expansions for A*. We therefore discuss a variant of Basic Theta* that might be able to find shorter paths at an increase in runtime by using weighted h-values with weights less than one. This variant of Basic Theta* uses the h-values $h(s) = w \cdot c(s, s_{goal})$ for a given weight $0 \leq w < 1$ and thus is similar to Weighted A* (Pohl, 1973), except that Weighted A* typically uses weights greater than one. We discuss variants of Basic Theta* with weights greater than one in Section 5.7.1. Figure 4.24(a) shows an example of the resulting effect on the number of vertex expansions and path length. The green vertex in the upper right region of the square grid is the start vertex, and the red vertex in the lower left region of the square grid is the goal vertex. Basic Theta* with $w = 1.00$ (as used so far) expands the orange vertices and finds the red path. Basic Theta* with $w = 0.75$ expands the blue vertices and finds the blue path. Thus, Basic Theta* expands more vertices with $w = 0.75$ than with $w = 1.00$ and the resulting path is shorter since it passes through vertices that are expanded with $w = 0.75$ but not with $w = 1.00$.

Figure 4.24(b) reports the effect of different values of w on the path length and number of vertex expansions of Basic Theta* and AP Theta* on random 500×500 grids with 20 percent blocked grid cells. (The graphs of the number of vertex expansions of Basic Theta* and AP Theta* nearly coincide.) Decreasing w decreases the path length at an increase in the number of vertex expansions and thus the runtime. The path length decreases more for AP Theta* than Basic Theta* since AP Theta* can constrain the angle ranges more than necessary and thus benefits in two ways from expanding more vertices. However, neither Basic Theta* nor AP Theta* are guaranteed to find true shortest paths even if their weights are zero.



(a) Expanded Vertices by Basic Theta* with Different Values of $w < 1$



(b) Random 500×500 Grids with 20 Percent Blocked Grid Cells

Figure 4.24: Weighted h-Values with $w < 1$

4.8.2 Tie Breaking

So far, Basic Theta* has broken ties among vertices in the open list with the same f-value in favor of vertices with larger g-values (when it decides which vertex to expand next). A* with consistent h-values finds paths of the same length no matter which tie-breaking scheme it uses. Breaking

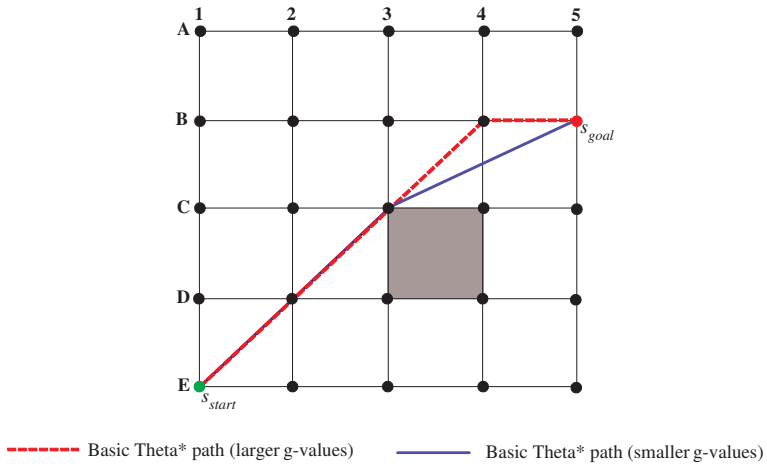


Figure 4.25: Basic Theta* Paths for Different Tie-Breaking Schemes

ties in favor of vertices with smaller g-values typically increases the number of vertex expansions and thus the runtime. We therefore discuss a variant of Basic Theta* that might be able to find shorter paths at an increase in runtime by breaking ties in favor of vertices with smaller g-values. Figure 4.25 shows an example of the resulting effect on path length. Vertices C4 and B4 have the same f-value but vertex B4 has a larger g-value since $f(C4) = 3.83 + 1.41 = 5.24$ and $f(B4) = 4.24 + 1 = 5.24$. If Basic Theta* breaks ties in favor of vertices with larger g-values, then it expands vertex B4 with parent E1 before vertex C4 with parent C3 and eventually expands the goal vertex with parent B4 and terminates. Path extraction then follows the parents from goal vertex B5 to start vertex E1 to retrieve the dashed red path [E1, B4, B5]. However, if Basic Theta* breaks ties in favor of vertices with smaller g-values, then it expands vertex C4 with parent C3 before vertex B4 with parent E1 and eventually expands the goal vertex with parent C3 and terminates. Path extraction then follows the parents from goal vertex B5 to start vertex E1 to retrieve the shorter solid blue path [E1, C3, B5].

Table 4.7 reports the effect of the tie-breaking scheme on the path length, number of vertex expansions and runtime of Basic Theta* and AP Theta* on random 500×500 grids with 20 percent blocked grid cells. Breaking ties in favor of vertices with smaller g-values neither changes the path length, number of vertex expansions nor runtime significantly. The effect of the tie-breaking scheme is small since fewer vertices have the same f-value for Basic Theta* and AP Theta* than for A* on octile graphs because the number of possible g-values and h-values is larger for any-angle find-path algorithms.

4.8.2.1 Tie Breaking and the Triangle Inequality

So far, Basic Theta* has chosen Path 2 over Path 1 if an unexpanded visible neighbor of a vertex has line-of-sight to the parent of the vertex. However, it can choose Path 1 over Path 2 if both paths are equally long, which increases the runtime due to the additional comparison. Figure 4.25 shows an example of the resulting effect on path length. Assume that Basic Theta* expands vertex B4 before vertex C4. If Basic Theta* chooses Path 2 over Path 1 then it expands vertex B4 with parent E1 and eventually expands the goal vertex B5 with parent B4 and terminates. Path

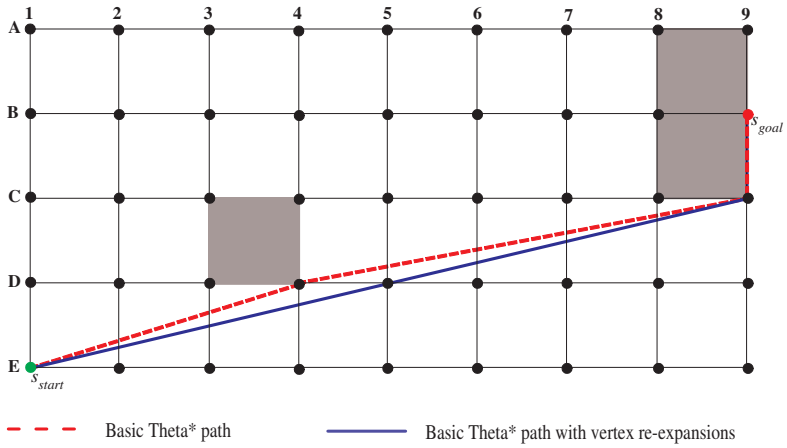


Figure 4.26: Basic Theta* Paths with and without Vertex Re-Expansions

extraction then follows the parents from goal vertex B5 to start vertex E1 to retrieve the dashed red path [E1, B4, B5]. However, if Basic Theta* chooses Path 1 over Path 2 then it expands vertex B4 with parent C3 and eventually expands goal vertex B5 with parent C3 and terminates. Path extraction then follows the parents from goal vertex B5 to start vertex E1 to retrieve the shorter solid blue path [E1, C3, B5]. The effect of this tie-breaking scheme is small since Path 1 and Path 2 are equally long very infrequently. We use this tie-breaking scheme in Section 6.4 (although we use it for different reasons).

4.8.3 Re-Expanding Vertices

So far, Basic Theta* has used a closed list to ensure that it expands each vertex at most once. A* with consistent h-values does not re-expand vertices whether or not it uses a closed list since it cannot find a shorter path from the start vertex to a vertex after expanding that vertex. On the other hand, Basic Theta* can re-expand vertices if it does not use a closed list since it can find a shorter path from the start vertex to a vertex after expanding the vertex. It then re-inserts the vertex into the open list and eventually re-expands it.[4] Figure 4.26 shows an example of the effect of vertex re-expansions on path length. Basic Theta* without vertex re-expansions eventually expands vertex C8 with parent D4. Vertex C9 is an unexpanded visible neighbor of vertex C8 that has line-of-sight to vertex D4. Basic Theta* without vertex re-expansions thus updates it according to Path 2 and sets its parent to vertex D4. After termination, path extraction follows the parents from goal vertex B9 to start vertex E1 to retrieve the dashed red path [E1, D4, C9, B9]. However, Basic Theta* with vertex re-expansions eventually expands vertex C8 with parent D4 and later re-expands vertex C8 with parent E1. Vertex C9 is a visible neighbor of vertex C8 that has line-of-sight to vertex E1. Basic Theta* with vertex re-expansions thus updates it according to Path 2 and sets its parent to vertex E1. After termination, path extraction follows the parents from goal vertex B9 to start vertex E1 to retrieve the shorter solid blue path [E1, C9, B9].

[4]Basic Theta* with vertex re-expansions could also delay the expansion of the goal vertex (for example, by increasing its f-value artificially) so that it can re-expand more vertices before it terminates, but our variant of Basic Theta* with vertex re-expansions does not do that.

	Smaller g-Values		Larger g-Values	
	Basic Theta*	**AP Theta***	**Basic Theta***	**AP Theta***
Path Length	578.41	578.51	578.44	578.55
Number of Vertex Expansions	18621.61	17698.75	18668.03	17744.94
Runtime	0.3724	0.5350	0.3829	0.5389

Table 4.7: Tie Breaking: Random 500×500 Grids with 20 Percent Blocked Grid Cells

	Basic Theta* without Vertex Re-Expansions	**Basic Theta* with Vertex Re-Expansions**
Path Length	578.41	577.60
Number of Vertex Expansions	18621.61	22836.37
Runtime	0.3724	0.5519

Table 4.8: Re-Expanding Vertices: Random 500×500 Grids with 20 Percent Blocked Grid Cells

Table 4.8 reports the effect of vertex re-expansions on the path length, number of vertex expansions and runtime of Basic Theta* on random 500×500 grids with 20 percent blocked grid cells. Vertex re-expansions decrease the path length slightly at an increase in the number of vertex expansions and thus the runtime.

Basic Theta* with vertex re-expansions is correct and complete.[5]

Theorem 5. *Basic Theta* with vertex re-expansions terminates and path extraction returns an unblocked path from the start vertex to the goal vertex if such a path exists. Otherwise, Basic Theta* with vertex re-expansions terminates and reports that no unblocked path exists.*

Proof. The proof is similar to the proof of Theorem 2. The only property that needs to be proved differently is that Basic Theta* with vertex re-expansions terminates since it is no longer true that it can never insert a vertex into the open list again once it has removed the vertex from the open list. However, since the number of vertices is finite, there are only a finite number of acyclic paths from the start vertex to each vertex. Therefore, the number of possible g-values is finite. It follows that Basic Theta* with vertex re-expansions can reduce the g-value of each vertex only a finite number of times and thus inserts each vertex into the open list a finite number of times. Thus, the open list eventually becomes empty and Basic Theta* has to terminate if it has not terminated earlier already. □

4.9 Trading Off Runtime and Path Length: Other Approaches

There are additional strategies for trading off the runtime of the search and the length of the resulting path that are specific to Basic Theta*. In this section, we develop variants of Basic Theta* that might be able to find shorter paths at an increase in runtime by examining more paths, including variants that check for line-of-sight to the parent of a parent, that use key vertices to identify promising parents and that increase the number of visible neighbors and thus the number of potential parents when updating vertices according to Path 1.

4.9.1 Three Paths

So far, Basic Theta* has considered two paths (namely Paths 1 and 2) when it updates the g-value and parent of an unexpanded visible neighbor s' of vertex s. We now discuss a variant of Basic Theta* that considers a third path, namely the path from the start vertex to the parent of the parent of vertex s [= $g(\mathit{parent}(\mathit{parent}(s)))$] and from it to vertex s' in a straight line [= $c(\mathit{parent}(\mathit{parent}(s)), s')$], resulting in a length of $g(\mathit{parent}(\mathit{parent}(s))) + c(\mathit{parent}(\mathit{parent}(s)), s')$. This variant of Basic Theta* might be able to find shorter paths at an increase in runtime since

[5] When Basic Theta* with vertex re-expansions terminates, path extraction retrieves an unblocked *acyclic* path from the start vertex to the goal vertex in reverse, but this is not trivial as it was for Basic Theta*. This is because a vertex in the closed list can change its parent. Assume for contradiction that a vertex s sets its parent to vertex s' such that a cycle is introduced. Prior to vertex s setting its parent to vertex s' there was an unblocked acyclic path from vertex s' to vertex s (following parents from vertex s' reaches vertex s). Our contradictory assumption ensures that such a path exists. Immediately after vertex s set its parent to vertex s' a cycle is introduced. In order for vertex s to set its parent to vertex s' the g-value of vertex s must have strictly decreased according to Line 46 and 51. This is impossible because the Euclidean distance between any two vertices must be non-negative and thus the g-value of vertex s' must be at least as large as the g-value of vertex s and $c(s, s')$ must be greater than or equal to 0. Thus vertex s could not set its parent to vertex s'. Contradiction.

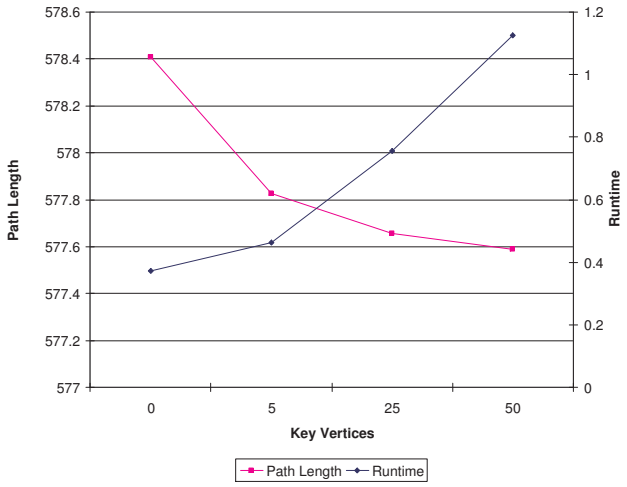


Figure 4.27: Basic Theta* with Key Vertices on Random 500×500 Grids with 20 Percent Blocked Grid Cells

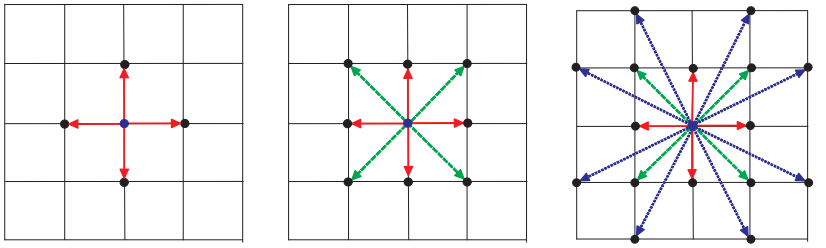
the third path is no longer than Path 2 due to the triangle inequality. However, the third path does not decrease the path length significantly because the original variant of Basic Theta* already determines that the parent of the parent of vertex s does not have line-of-sight to some vertex that shares its parent with vertex s. Thus, it is very unlikely that the parent of the parent of vertex s has line-of-sight to vertex s' and thus that the third path is unblocked.

4.9.2 Key Vertices

So far, Basic Theta* has considered two paths (namely Paths 1 and 2) when it updates the g-value and parent of an unexpanded visible neighbor s' of vertex s. The parent of a vertex then is either a visible neighbor of the vertex or the parent of a visible neighbor, which is not always the case for true shortest paths. We now discuss a variant of Basic Theta* that considers additional paths, namely the paths from the start vertex to cached key vertices and from them to vertex s' in a straight line. This variant of Basic Theta* might be able to find shorter paths at an increase in runtime due to the fact that the parent of a vertex can now also be one of the key vertices. Figure 4.27 reports the effect of key vertices on the path length and runtime of Basic Theta* on random 500×500 grids with 20 percent blocked grid cells. Larger numbers of key vertices indeed decrease the path length at an increase in runtime. Recently, a new technique was introduced by Šišlák, Volf, and Pěchouček (2009a) which uses an idea similar to key vertices to find any-angle paths. We discuss this technique in Appendix E.2.2.

4.9.3 Larger Branching Factors

So far, Basic Theta* has operated on octile graphs. We now discuss a variant of Basic Theta* that operates on square grids with different numbers of neighbors and thus different branching factors. Figure 4.28 shows the neighbors of the center blue vertex on quad graphs, octile graphs



(a) Quad Graph (b) Octile Graph (c) 16-Neighbor Square Grid Graph

Figure 4.28: Square Grids with Difference Branching Factors

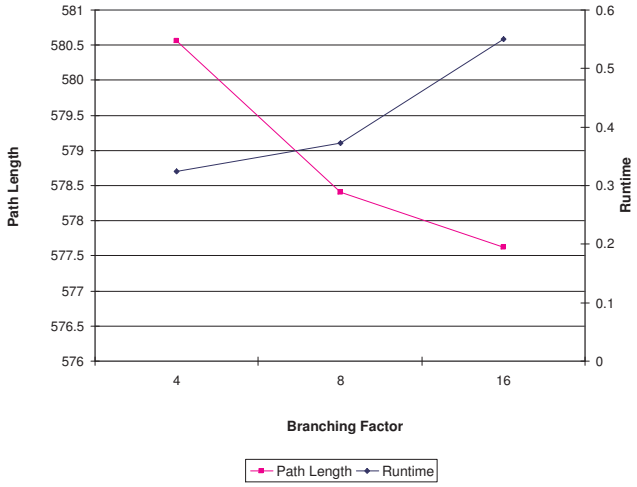


Figure 4.29: Basic Theta* with Different Branching Factors on Random 500×500 Grids with 20 Percent Blocked Grid Cells

and 16-neighbor square grid graphs, respectively. This variant of Basic Theta* might be able to find shorter paths at an increase in runtime since larger branching factors increase the number of visible neighbors of vertices and thus their number of potential parents when updating them according to Path 1. Figure 4.29 reports the effect of larger branching factors on the path length and runtime of Basic Theta* on random 500×500 grids with 20 percent blocked grid cells. Larger branching factors indeed decrease the path length at an increase in runtime.

4.10 Conclusions

In this chapter, we examined path planning in known 2D environments that have been discretized into octile graphs with vertices placed in the corners of grid cells. We presented two new correct and complete any-angle find-path algorithms. Basic Theta* is simple to implement and understand, fast and finds short paths. However, it is not guaranteed to find true shortest paths. Basic

Theta* is simple to implement because its pseudocode is very similar to the pseudocode of A*. Basic Theta* is simple to understand because the only difference between Basic Theta* and A* is Path 2 and Path 2 takes advantage of triangle inequality which is easy to understand (Simplicity Property). Furthermore, the triangle inequality is guaranteed to hold on any Euclidean graph and thus Basic Theta* can be used to search any Euclidean graph (Generality Property). AP Theta* achieves a worst-case complexity per vertex expansion that is constant (like that of A* on octile graphs) rather than linear in the number of vertices (like that of Basic Theta*) by propagating angle ranges when it expands vertices. However, AP Theta* is more difficult to implement and understand than Basic Theta*, is not as fast, finds slightly longer paths and cannot be used to search any Euclidean graph.

We proved the correctness and completeness of Basic Theta* and AP Theta* and then compared them against three existing find-path algorithms, namely, A* on octile graphs, A* with post-smoothed paths (A* PS), A* on visibility graphs and Field D* (FD*), the only other variant of A* we know of that propagates information along octile graph edges without constraining paths to be formed by octile graph edges. Basic Theta* and AP Theta* (unlike A* on octile graphs and A* PS) consider paths not constrained to be formed by octile graph edges during their search and thus can make informed decisions regarding these paths during the search. Basic Theta* and AP Theta* (unlike FD*) take advantage of the fact that true shortest paths have heading changes only at the corners of blocked grid cells.

A* on visibility graphs finds true shortest paths but is slow. On the other hand, A* on octile graphs finds long paths but is fast. Any-angle find-path algorithms lie between these two extremes. Basic Theta* dominates AP Theta*, A* PS and FD* in terms of their tradeoffs with respect to the runtime of the search and length of the resulting path. It finds paths that are almost as short as true shortest paths and is almost as fast as A* on octile graphs. In other words, we showed that Basic Theta* and AP Theta* provide a good tradeoff with respect to the runtime of the search and the length of the resulting path (Efficiency Property).

We extended Basic Theta* to find paths from a given start vertex to all other vertices and to find paths on grids that contain grid cells with non-uniform traversal costs. We showed that when Basic Theta* finds paths from a given start vertex to all other vertices it obeys the Closed List Property. The f-value of an expanded vertex of Basic Theta* (unlike A* on octile graphs) with consistent h-values can be larger than the f-value of one or more of its unexpanded visible neighbors, which means that Basic Theta* might be able to find shorter paths at an increase in runtime by re-expanding vertices or expanding additional vertices. We thus developed variants of Basic Theta* that use weighted h-values with weights less than one, that break ties among vertices with the same f-value in the open list in favor of vertices with smaller g-values (when they decide which vertex to expand next), that re-expand vertices whose f-values have decreased, that check for line-of-sight to the parent of a parent, that use key vertices to identify promising parents and that increase the branching factor.

To summarize, in this chapter we validated Hypothesis 2 in known 2D environments by introducing Basic Theta*, a new any-angle find-path algorithm that satisfies the Simplicity, Efficiency and Generality Properties when path planning in known 2D environments. Simplicity Property: Basic Theta* is simple to implement and understand (Section 4.4.3.1). Efficiency Property: Basic Theta* provides a good tradeoff with respect to the runtime of the search and the length of the resulting path (Section 4.6). Generality Property: Basic Theta* can be used to search any Euclidean graph (Section 4.4.3.2).

Chapter 5

Lazy Theta*: Any-Angle Path Planning in Known 3D Environments (Contribution 3)

Figure 5.1: Known 3D Environments: Screen Shot from James Cameron's Avatar: The Game (Ubisoft)

This chapter introduces the third major contribution of this dissertation, namely Contribution 3. Specifically, this chapter introduces Lazy Theta* (Nash et al., 2010), a new any-angle find-path algorithm which we evaluate in known 3D environments using the Simplicity, Efficiency and Generality Properties. Our evaluation shows that Lazy Theta* is simple to implement and understand, that it provides a good tradeoff with respect to the runtime of the search and the length of the resulting path, that it provides a dominating tradeoff relative to existing any-angle find-path algorithms with respect to the runtime of the search and the length of the resulting path, that it finds paths that are $\approx 7 - 8\%$ shorter than the paths found by traditional edge-constrained find-path algorithms on average (and $\approx 7 - 8\%$ shorter than shortest grid paths on average) that, when used with weighted h-values where the weights are greater than one, it finds paths faster with only a small increase in path lengths and that it can be used to search any Euclidean graph. Therefore, these results validate Hypothesis 2 in known 3D environments.

This chapter is organized as follows: In Section 5.1, we explain why path planning in known 3D environments is more difficult than path planning in known 2D environments. In Section 5.2, we introduce notation and definitions that are used throughout this chapter. In Section 5.3, we re-examine several of the find-path algorithms from the previous chapter, namely A* on triple cubic graphs, A* PS and Basic Theta*, in the context of path planning in known 3D environments.

In Section 5.4, we introduce Lazy Theta*, a new any-angle find-path algorithm that is designed for path planning in known 3D environments. In Section 5.5, we introduce two variants of Lazy Theta*. In Section 5.6, we present our experimental results. In Section 5.7, we show that, when Lazy Theta* uses weighted h-values with weights greater than one, it expands fewer vertices than Lazy Theta* with consistent h-values while finding paths that have nearly the same lengths as the paths found by Lazy Theta* with consistent h-values. Thus, when Lazy Theta* uses weighted h-values with weights greater than one it provides a nearly dominating tradeoff, relative to Lazy Theta* with consistent h-values, with respect to the runtime of the search and the length of the resulting path. Finally, in Section 5.8, we summarize our results.

5.1 Introduction

Path planning in known 3D environments is computationally more expensive than path planning in known 2D environments. True shortest paths in continuous 2D environments with polygonal obstacles can be found in polynomial time using visibility graphs. Visibility graphs can be defined for known 3D environments with polyhedral obstacles, but they do *not* necessarily contain true shortest paths (Choset et al., 2005). This can be seen in Figure 3.6, in which a true shortest path between the two spheres does not contain any vertices of the polyhedral obstacle. Path planning in known 3D environments with polyhedral obstacles is NP-Hard (Canny & Reif, 1987) and thus finding true shortest paths is not feasible. Thus, roboticists and video game developers often find approximate solutions by performing an A* search on a grid graph constructed from a cubic grid (Nash et al., 2009; Carsten et al., 2006; Cohen et al., 2011). However, we showed in Chapter 3 that shortest grid paths formed by the edges of a 26-neighbor cubic grid graph (triple cubic graph) can be $\approx 13\%$ longer than true shortest paths while shortest grid paths formed by the edges of an 8-neighbor square grid graph (octile graph) can be $\approx 8\%$ longer than true shortest paths. In other words, the two alternatives that are often considered when path planning in known 2D environments, namely grid graphs and visibility graphs, are less appealing when path planning in known 3D environments and thus the need for more sophisticated find-path algorithms is even greater.

In Chapter 4, we showed that Basic Theta* provided a good tradeoff with respect to the runtime of the search and the length of the resulting path on octile graphs (Efficiency Property). However, a Basic Theta* search performs a line-of-sight check for each unexpanded visible neighbor of each expanded vertex and there are far more unexpanded visible neighbors on a triple cubic graph than there are on an octile graph. Therefore, one would expect that the number of line-of-sight checks performed during a Basic Theta* search on a triple cubic graph would be larger than the number of line-of-sight checks performed during a Basic Theta* search on an octile graph and thus one would also expect that the runtime of a Basic Theta* search on a triple cubic graph would be longer than the runtime of a Basic Theta* search on an octile graph. Thus, the question becomes can we develop a simple (Simplicity Property) and generic (Generality Property) any-angle find-path algorithm that provides a dominating tradeoff, relative to Basic Theta*, with respect to the runtime of the search and the length of the resulting path (Efficiency Property) in known 3D environments, specifically triple cubic graphs.

In this chapter, we first demonstrate that Basic Theta* can be applied to triple cubic graphs without any changes to the pseudocode. We then introduce Lazy Theta*, a variant of Basic Theta*, that is as simple to implement and understand as Basic Theta*, but is faster because

it performs fewer line-of-sight checks. Basic Theta* and Lazy Theta* are unique, in that they are the only any-angle find-path algorithms we know of that propagate information along the edges of any Euclidean graph, without constraining paths to be formed by graph edges. We show experimentally that Lazy Theta* finds paths faster than Basic Theta* with one order of magnitude fewer line-of-sight checks and with only a small increase in path lengths. Finally, we show that using weighted h-values with weights greater than one allows Lazy Theta* and Basic Theta* to find paths faster, by reducing the number of vertices that they expand, without a significant increase in the lengths of the resulting paths.

5.2 Notation and Definitions

In this section, we describe the find-path problem that we study in this chapter, namely finding paths on triple cubic graphs $G' = (V, E')$ (with vertices placed in the corners of grid cells) constructed from cubic grids that discretize known 3D environments (as described in Section 3.3.1). In this chapter, we assume that obstacles in the known 3D environment completely block grid cells, that is, there is no digitization bias. The find-path problem is to find an unblocked path from a given start vertex $s_{start} \in V$ to a given goal vertex $s_{goal} \in V$. $nghbr_{vis}(s) \subseteq V$ is the set of (up to 26) visible neighbors of vertex s, that is, those vertices that are adjacent to vertex s and have line-of-sight to vertex s. Unblocked path and line-of-sight are defined in Section 1.1.2. In some examples, we refer to a vertex using a letter, a number and a subscript U or L. A vertex with a subscript U is directly above a vertex with a subscript L. For example, in Figure 5.3, the red sphere, vertex $A4_U$, is the goal vertex and it is directly above vertex $A4_L$.

5.3 Find-Path Algorithms in Known 3D Environments

In order to better understand the differences between path planning in known 2D and 3D environments we briefly re-introduce the find-path algorithms from the previous chapter in the context of triple cubic graphs.

All find-path algorithms discussed in this chapter build on A*. Algorithm 1 shows the pseudocode for the implementation of A* discussed in this chapter. Line 13 is to be ignored. We use the same pseudocode for A* when path planning in known 3D environments that we used when path planning in known 2D environments and therefore we refer the reader to Section 4.3 for a detailed description of how A* operates.

We now describe several find-path algorithms that are variants of A* and how they trade off with respect to two conflicting criteria, namely the runtime of the search and the length of the resulting path, as shown in Figure 5.2.[1] For each existing find-path algorithm we highlight some of the important differences between path planning in known 3D environments and path planning in known 2D environments. We introduce them in order of decreasing path lengths.

5.3.1 A* on Triple Cubic Graphs

One can run A* on triple cubic graphs. The resulting paths are artificially constrained to be formed by the edges of the triple cubic graph, which can be seen in Figure 3.7(a). As a result,

[1] More detailed experimental results are provided in Section 5.6.

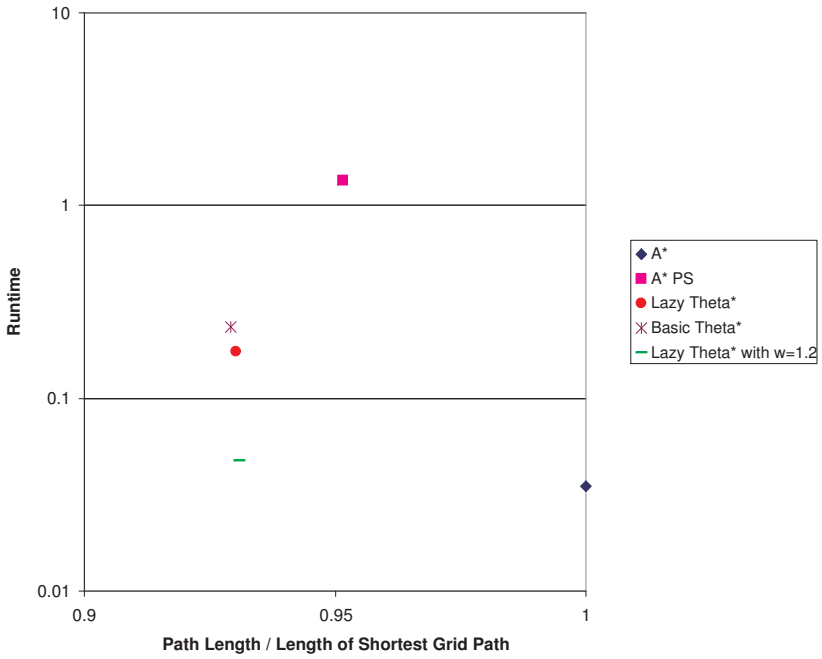


Figure 5.2: Runtime Versus Path Length (relative to the length of a shortest grid path) on Random $100 \times 100 \times 100$ Grids with 20 Percent Blocked Grid Cells

```
92 h(s)
93     Δx := |s.x − s_goal.x|;
94     Δy := |s.y − s_goal.y|;
95     Δz := |s.z − s_goal.z|;
96     largest := Max(Δx, Δy, Δz);
97     middle := Middle(Δx, Δy, Δz);
98     smallest := Min(Δx, Δy, Δz);
99     return √3 · smallest + √2 · (middle − smallest) + (largest − middle);
```

Algorithm 6: Calculation of 3D Octile Distances

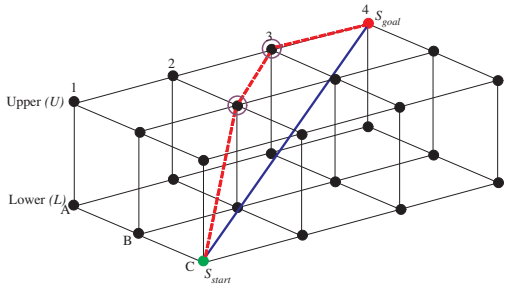


Figure 5.3: Unrealistic Looking Path on a Triple Cubic Graph

the paths found by A* on triple cubic graphs are not equivalent to the true shortest paths and

are unrealistic looking since they either deviate substantially from the true shortest paths or have many more heading changes. The paths found by A* on triple cubic graphs can be longer and less realistic looking, relative to the true shortest paths, than the paths found by A* on octile graphs. In Chapter 3, we showed that shortest grid paths formed by the edges of triple cubic graphs can be $\approx 13\%$ longer than true shortest paths and that shortest grid paths formed by the edges of octile graphs can be $\approx 8\%$ longer than true shortest paths. Shortest grid paths formed by the edges of triple cubic graphs can be less realistic looking than shortest grid paths formed by the edges of octile graphs, relative to true shortest paths. For example, consider a shortest grid path formed by the edges of a triple cubic graph constructed from a cubic grid without blocked grid cells that has the fewest number of heading changes. This path can have as many as two extraneous vertices whereas an analogous path formed by the edges of an octile graph can have only one extraneous vertex. In Figure 5.3, vertex $B2_U$ and vertex $A3_U$ are extraneous vertices on the dashed red path, which is a shortest grid path, and the solid blue path is a true shortest path. No shortest grid path can have fewer heading changes (and thus extraneous vertices). We use the 3D octile distances as h-values in the experiments because they are the largest consistent h-values that are easy to compute. The 3D octile distances are the distances on triple cubic graphs without blocked grid cells. Algorithm 6 shows how to calculate the 3D octile distance of a given vertex s, where $s.x$, $s.y$ and $s.z$ are the x-coordinate, y-coordinate and z-coordinate of vertex s, respectively.

5.3.2 A* with Post-Smoothing (A* PS)

One can run A* with post-smoothing (A* PS). A* PS runs A* on triple cubic graphs and then smoothes the resulting path in a post-processing step, which often shortens it at an increase in runtime. Algorithm 2 shows the pseudocode of the simple smoothing algorithm that A* PS uses in our experiments. We use the same pseudocode for A* PS when finding paths on triple cubic graphs that we used when finding paths on octile graphs and therefore we refer the reader to Section 4.3.2 for a detailed description of how A* PS operates. We showed in Section 4.6 that A* PS typically finds shorter paths than A* on octile graphs, but is not guaranteed to find true shortest paths because it only considers grid paths during the A* search and thus cannot make informed decisions regarding other paths during the A* search. This is also the case for A* PS on triple cubic graphs. This can be seen in Figure 3.7. Figure 3.7(a) shows a shortest grid path and Figure 3.7(b) shows a shortest any-angle path. Post-smoothing removes the extraneous vertex $A3_U$, resulting in a path that is still longer than a shortest any-angle path. Shortest grid paths formed by the edges of triple cubic graphs can be longer and less realistic looking than shortest grid paths formed by the edges of octile graphs, relative to true shortest paths. Thus, if post-smoothing is unable to shorten the shortest grid paths then the paths found by A* PS on triple cubic graphs can be longer and less realistic looking than the paths found by A* PS on octile graphs, relative to the true shortest path. We use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in the experiments for the reasons described in Section 4.6.

5.3.3 Basic Theta*

The key difference between Basic Theta* and A* when path planning in known 3D environments is the same as the difference between Basic Theta* and A* when path planning in known 2D

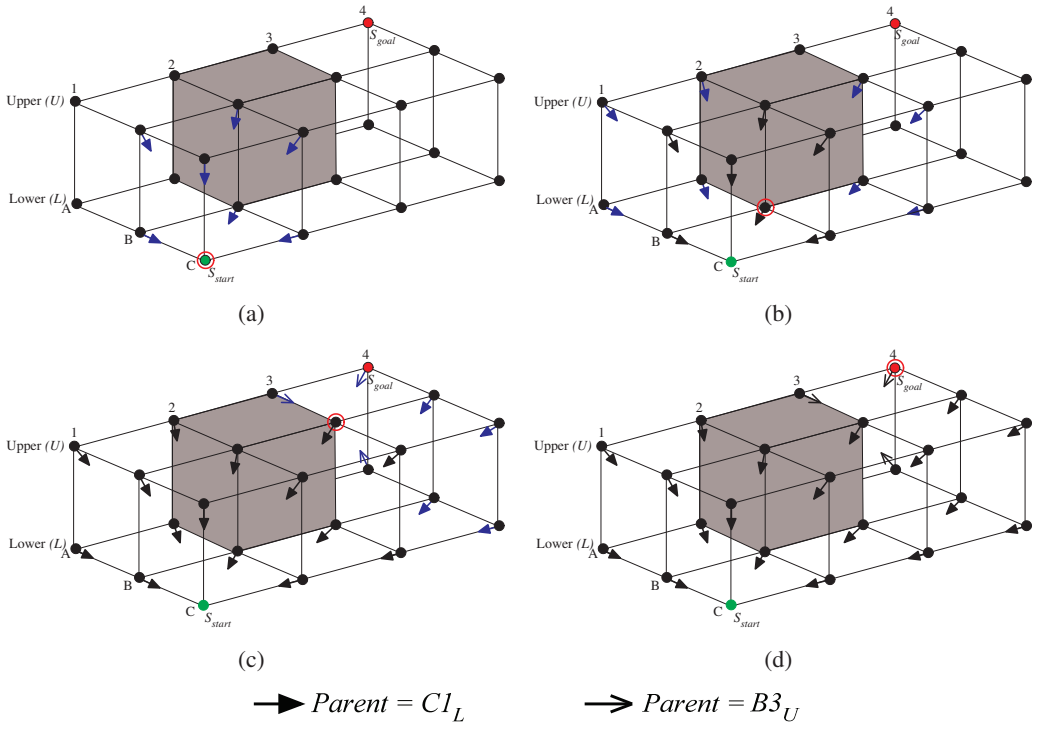


Figure 5.4: Example Trace of Basic Theta*

environments, namely that Basic Theta* allows the parent of a vertex to be any vertex, unlike A* where the parent of a vertex must be a visible neighbor.

Algorithm 3 shows the pseudocode for Basic Theta*. We use the same pseudocode for Basic Theta* when path planning in known 3D environments that we used when path planning in known 2D environments, however we re-examine the operation of Basic Theta* in order to highlight how it differs from Lazy Theta*. We use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in the experiments for Basic Theta* and all of the remaining find-path algorithms in this chapter because they are the largest consistent h-values that are easy to calculate.[2]

Basic Theta* is identical to A* except that, when it updates the g-value and parent of an unexpanded visible neighbor s' of vertex s in procedure ComputeCost, it finds any-angle paths by considering both Path 1 and Path 2 (Section 4.4). Path 1: As done by A*, Basic Theta* considers the path from the start vertex to vertex s and from vertex s to vertex s' in a straight-line [Line 51]. Path 2: To allow for any-angle paths, Basic Theta* also considers the path from the start vertex to the parent of vertex s and from the parent of vertex s to vertex s' in a straight-line if vertex s' has line-of-sight to the parent of vertex s [Line 46]. Path 2 is guaranteed to be no longer than Path 1 due to the triangle inequality if vertex s' has line-of-sight to the parent of vertex s. Basic Theta* uses Path 2 if the parent of vertex s has line-of-sight to vertex s' [Line 44] and then sets the g-value of vertex s' to the length of Path 2 and the parent of vertex s' to the parent of vertex s. Otherwise, Basic Theta* uses Path 1 and sets the g-value of vertex s' to the length of

[2]For Basic Theta* and all of the remaining find-path algorithms in this chapter the 3D-octile distances can overestimate the goal distances because the paths are not constrained to triple cubic graph edges.

Path 1 and the parent of vertex s' to vertex s. The key idea behind Basic Theta* and Path 2 is that, unlike A*, where the parent of a vertex must be a visible neighbor of that vertex, Basic Theta* allows the parent of a vertex to be any vertex. Thus, the parent of a vertex can be any vertex for Basic Theta* while it is a visible neighbor of the vertex for A*.

5.3.3.1 Example Trace of Basic Theta*

Figure 5.4 shows an example trace of Basic Theta*. The arrows point to the parent of a vertex. Hollow red circles indicate vertices that are being expanded, and blue arrows indicate vertices that are generated during the current expansion. First, Basic Theta* expands start vertex $C1_L$ with parent $C1_L$, just like A* would do, as shown in Figure 5.4(a). Second, Basic Theta* expands vertex $B2_L$ with parent $C1_L$ as shown in Figure 5.4(b). Third, Basic Theta* expands vertex $B3_U$ with parent $C1_L$, as shown in Figure 5.4(c). Vertex $A4_U$ is an unexpanded visible neighbor of vertex $B3_U$ which does not have line-of-sight to vertex $C1_L$. Basic Theta* thus updates it according to Path 1 and sets the parent of $A4_U$ to vertex $B3_U$. On the other hand, vertex $B4_U$ is an unexpanded visible neighbor of vertex $B3_U$ which does have line-of-sight to vertex $C1_L$. Basic Theta* thus updates it according to Path 2 and sets the parent of $B4_U$ to vertex $C1_L$. Fourth, Basic Theta* expands vertex $A4_U$ with parent $B3_U$ and terminates, as shown in Figure 5.4(d). Path extraction then follows the parents from goal vertex $A4_U$ to start vertex $C1_L$ to retrieve the shortest any-angle path [$A4_U$,$B3_U$,$C1_L$] from the start vertex to the goal vertex in reverse.

Basic Theta* performs a line-of-sight check for each unexpanded visible neighbor of an expanded vertex and, since each vertex can have up to 26 visible neighbors in a triple cubic graph, Basic Theta* can perform many line-of-sight checks. For example, in the trace depicted in Figure 5.4, Basic Theta* performs 37 line-of-sight checks (7 when vertex $C1_L$ is expanded, 15 when vertex $B2_L$ is expanded, and 15 when vertex $B3_U$ is expanded).

Using Basic Theta* to find paths on triple cubic graphs is important because it demonstrates that Basic Theta* can be used to search any Euclidean graph (Generality Property). We used Basic Theta* to find paths on triple cubic graphs without any changes to the pseudocode. While A* can be applied in a similar fashion, interpolation based find-path algorithms, such as Field D*, cannot. Field D* uses a closed form linear interpolation equation that requires that searches be performed on octile graphs. Therefore, when Field D* was modified so that it could be used to search triple cubic graphs, resulting in 3D Field D*, it required substantial changes to the pseudocode and additional approximations (Section 2.2.4). Basic Theta* on the other hand takes advantage of the triangle inequality which is guaranteed to hold for any Euclidean graph.

5.4 Lazy Theta*

Basic Theta* can be less efficient on triple cubic graphs than octile graphs because it performs a line-of-sight check for each unexpanded visible neighbor of each expanded vertex. Line-of-sight checks on square and cubic grids can be implemented efficiently with line drawing algorithms (Bresenham, 1965; Vykruta, 2002), but the runtime per line-of-sight check is linear in the number of vertices and there are many more line-of-sight checks per expanded vertex on triple cubic graphs than there are on octile graphs. Line-of-sight checks for other discretizations of a continuous environment, such as NavMeshes, typically cannot be implemented as efficiently (Section 2.2.1.2). We therefore reduce the number of line-of-sight checks that Basic Theta* performs. Our

```
100 Main()
101     g(s_start) := 0;
102     parent(s_start) := s_start;
103     open := ∅;
104     open.Insert(s_start, g(s_start) + h(s_start));
105     closed := ∅;
106     while open ≠ ∅ do
107         s := open.Pop();
108         SetVertex(s);
109         if s = s_goal then
110             return "path found";
111         closed := closed ∪ {s};
112         foreach s' ∈ nghbr_vis(s) do
113             if s' ∉ closed then
114                 if s' ∉ open then
115                     parent(s') := NULL;
116                     g(s') := ∞;
117                 UpdateVertex(s, s');
118     return "no path found";
119 SetVertex(s)
120     if NOT LineOfSight(parent(s), s) then
121         /* Path 1*/
122         parent(s) := argmin_{s'∈nghbr_vis(s)∩closed}(g(s') + c(s', s));
123         g(s) := min_{s'∈nghbr_vis(s)∩closed}(g(s') + c(s', s));
124 ComputeCost(s, s')
125     /* Path 2 */
126     if g(parent(s)) + c(parent(s), s') < g(s') then
127         parent(s') := parent(s);
128         g(s') := g(parent(s)) + c(parent(s), s');
```

Algorithm 7: Lazy Theta*

inspiration is provided by Probabilistic Road Maps which use lazy evaluation to reduce the number of line-of-sight checks (collision checks) by delaying them until they are necessary (Bohlin & Kavraki, 2000; Sanchez & Latombe, 2002).

In this section, we introduce Lazy Theta*, a simple variant of Basic Theta* which uses lazy evaluation to perform only one line-of-sight check per expanded vertex (but with slightly more expanded vertices), while Basic Theta* performs one line-of-sight check for each unexpanded visible neighbor of each expanded vertex.

Algorithm 7 shows the pseudocode for Lazy Theta*. Procedure UpdateVertex in Algorithm 7 is identical to procedure UpdateVertex in Algorithm 1 and thus is not shown.[3]

5.4.1 Operation of Lazy Theta*

Lazy Theta* is simple. Basic Theta* updates the g-value and parent of an unexpanded visible neighbor s' of vertex s in procedure ComputeCost by considering both Path 1 and Path 2. It considers Path 2 if vertex s' and the parent of vertex s have line-of-sight. Otherwise, it considers Path 1. Lazy Theta* optimistically assumes that vertex s' and the parent of vertex s have line-of-sight without performing a line-of-sight check. Thus, it delays the line-of-sight check and

[3] Procedure Main in Algorithm 1 is identical to procedure Main in Algorithm 7 if Lines 13 and 108 are ignored.

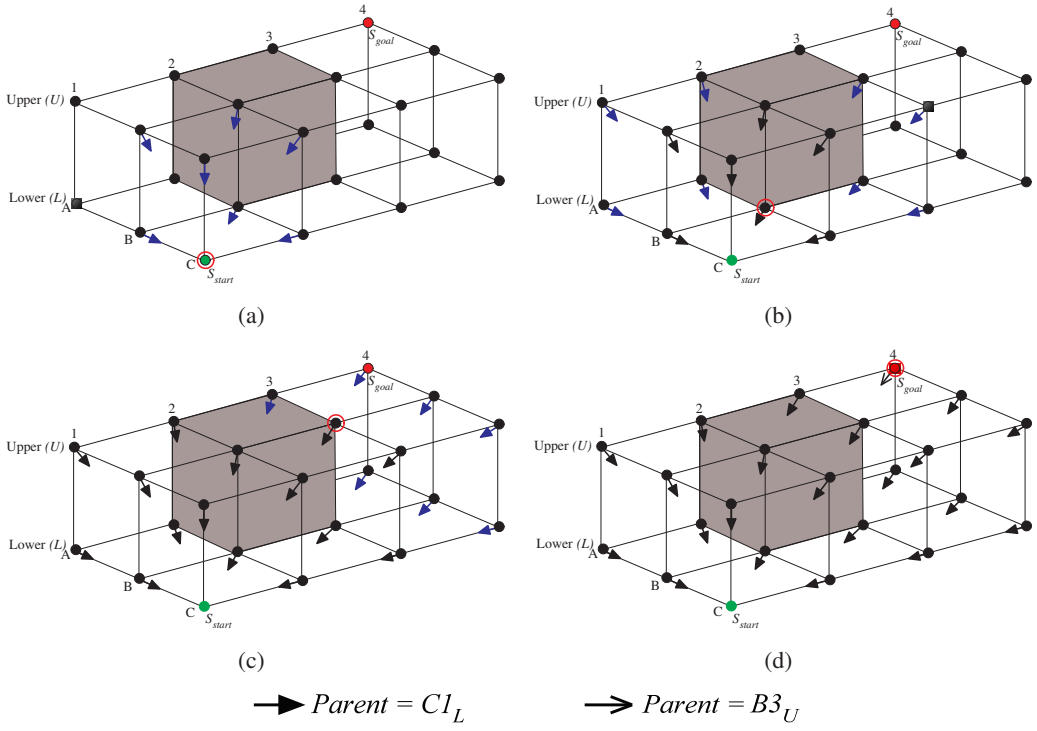


Figure 5.5: Example Trace of Lazy Theta*

considers only Path 2. This assumption may, of course, be incorrect. Therefore, Lazy Theta* performs the line-of-sight check in procedure SetVertex immediately before expanding vertex s'.[4] If vertex s' and the parent of vertex s' indeed have line-of-sight [Line 120], then the assumption was correct and Lazy Theta* does not change the g-value and parent of vertex s'. If vertex s' and the parent of vertex s' do not have line-of-sight, then Lazy Theta* updates the g-value and parent of vertex s' according to Path 1 by considering the path from the start vertex to each expanded visible neighbor s'' of vertex s' and from vertex s'' to vertex s' in a straight line and choosing the shortest such path. We know that vertex s' has at least one expanded visible neighbor because vertex s' was added to the open list when Lazy Theta* expanded such a neighbor.

5.4.2 Example Trace of Lazy Theta*

Figure 5.5 shows an example trace of Lazy Theta*, using the same find-path problem as Figure 5.4. First, Lazy Theta* expands start vertex $C1_L$ with parent $C1_L$, just like A* and Basic Theta* would do, as shown in Figure 5.5(a). Second, Lazy Theta* expands vertex $B2_L$ with parent $C1_L$, just like Basic Theta* would do, as shown in Figure 5.5(b). Third, Lazy Theta* expands vertex $B3_U$ with parent $C1_L$, as shown in Figure 5.5(c). Vertex $A4_U$ is an unexpanded visible neighbor of vertex $B3_U$. Lazy Theta* optimistically assumes that vertex $A4_U$ has line-of-sight to vertex $C1_L$. Lazy Theta* thus updates it according to Path 2 and sets the parent of vertex $A4_U$ to vertex $C1_L$. Fourth, Lazy Theta* expands vertex $A4_U$ with parent $C1_L$ as shown in Figure 5.5(d).

[4] Vertex s' corresponds to argument vertex s in procedure SetVertex.

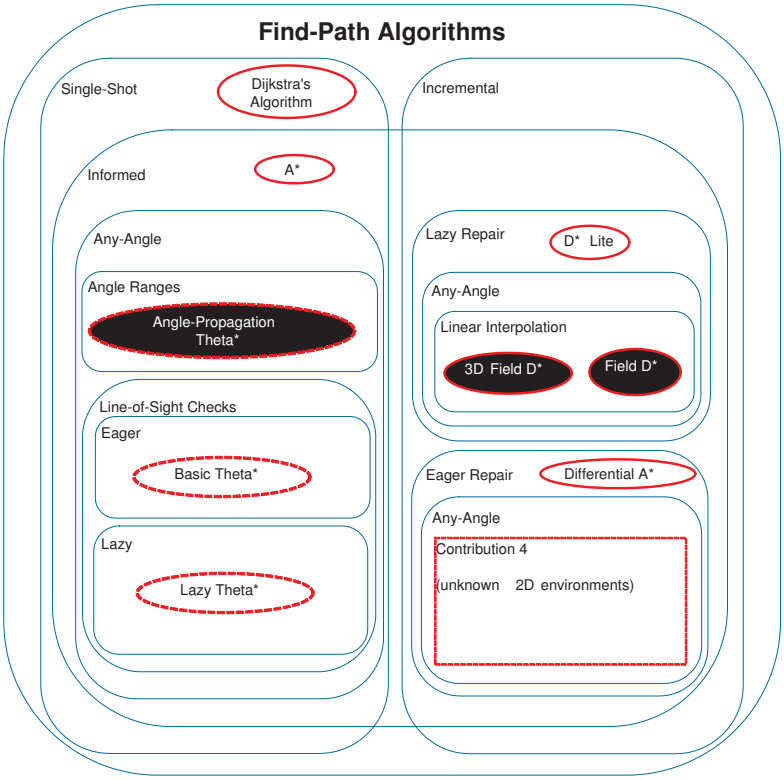


Figure 5.6: Classification of Find-Path Algorithms

Vertex $A4_U$ and vertex $C1_L$ do not have line-of-sight. Lazy Theta* thus updates it according to Path 1 by considering the paths from vertex $C1_L$ to each expanded visible neighbor s'' of vertex $A4_U$ (namely, vertex $B3_U$) and from vertex s'' to vertex $A4_U$ in a straight line. Lazy Theta* sets the parent of vertex $A4_U$ to vertex $B3_U$ since the path from start vertex $C1_L$ to vertex $B3_U$ and from vertex $B3_U$ to vertex $A4_U$ in a straight line is the shortest such path. Finally, Lazy Theta* terminates. Path extraction then follows the parents from goal vertex $A4_U$ to start vertex $C1_L$ to retrieve the shortest any-angle path [$A4_U$,$B3_U$,$C1_L$] from the start vertex to the goal vertex in reverse.

In this example, Lazy Theta* and Basic Theta* find the same path from start vertex $C1_L$ to goal vertex $A4_U$, but Lazy Theta* performs only 4 line-of-sight checks (one for each expanded vertex), while Basic Theta* performs 37 line-of-sight checks (one for each unexpanded visible neighbor of each expanded vertex).

Lazy Theta* is similar to Basic Theta* both in terms of how difficult it is to implement (the pseudocode for the two algorithms is extremely similar) and how difficult it is to understand (the two algorithms both extend A* with the same key idea, namely the triangle inequality and Path 2) and thus Lazy Theta*, like Basic Theta*, satisfies the Simplicity Property.

Figure 5.6 provides an updated summary of our classification of different find-path algorithms. It is an updated version of Figure 4.17 which accounts for the contributions made in this chapter. Figures 5.6 and 4.17 are annotated in the same way. The Lazy Theta* oval is within 6 rounded rectangles, and thus Lazy Theta* is a **1** single-shot, **2** informed, **3** any-angle **4** find-path

```
129 SetVertex(s)
130     if NOT LineOfSight(parent(s), s) then
131         /* Path 1*/
132         parent(s) := argmin_{s' ∈ nghbr_vis(s) ∩ closed}(g(s') + c(s', s));
133         g(s) := min_{s' ∈ nghbr_vis(s) ∩ closed}(g(s') + c(s', s));
134         open.Insert(s, g(s) + h(s));
135         Goto Line 107;
```

Algorithm 8: Lazy Theta*-R (Re-Insertions Variant)

```
136 SetVertex(s)
137     /* Path 2 */
138     if LineOfSight(parent(parent(s)), s) then
139         parent(s) := parent(parent(s));
140         g(s) := g(parent(s)) + c(parent(s), s);
```

Algorithm 9: Lazy Theta*-P (Pessimistic Variant)

algorithm that **5** lazily performs **6** line-of-sight checks. The Lazy Theta* oval, like the Basic Theta* oval is transparent. This is because Lazy Theta*, like Basic Theta*, only relies on the triangle inequality and thus can be used to search any Euclidean graph (Generality Property). Basic Theta* and Lazy Theta* belong to different classes of any-angle find-path algorithms that use line-of-sight checks, Basic Theta* performs line-of-sight checks eagerly and Lazy Theta* performs line-of-sight checks lazily.

5.5 Variants of Lazy Theta*

We now introduce two variants of Lazy Theta* that help us better understand the performance of Lazy Theta* with different lazy evaluation techniques.

- **Lazy Theta*-R:** Algorithm 8 shows the pseudocode for Lazy Theta*-R. All procedures other than SetVertex are identical to those of Algorithm 7 and thus are not shown. Procedure UpdateVertex can be found in Algorithm 1. Lazy Theta*-R is identical to Lazy Theta* except for procedure SetVertex. If Lazy Theta*-R considers a vertex s for expansion and updates it according to Path 1 in procedure SetVertex, it re-inserts vertex s into the open list with an updated key (which we call a **key update**) and continues on Line 107 rather than expanding vertex s. The idea behind deferring the expansion of vertex s is that it gives Lazy Theta*-R the opportunity to discover shorter paths from the start vertex to vertex s since the shortest path from the start vertex to vertex s cannot change once vertex s is expanded.

- **Lazy Theta*-P:** Algorithm 9 shows the pseudocode for Lazy Theta*-P. Lazy Theta*-P uses procedure Main from Algorithm 7 and procedures UpdateVertex and ComputeCost from Algorithm 1. In other words, Lazy Theta*-P is identical to A* except for procedure SetVertex. Like A*, Lazy Theta*-P updates the g-value and parent of an unexpanded visible neighbor s' of a vertex s in procedure ComputeCost by considering only Path 1. Unlike A*, Lazy Theta*-P updates the g-value and parent of vertex s' in procedure SetVertex by considering Path 2 immediately before expanding vertex s'. During procedure SetVertex, Lazy Theta*-P checks whether or not vertex s' has line-of-sight to the parent of the parent

of vertex s'. If they have line-of-sight, Lazy Theta*-P updates the g-value and parent of vertex s' according to Path 2, namely the path from the start vertex to the parent of the parent of vertex s' and from the parent of the parent of vertex s' to vertex s' in a straight line. Thus, in procedure ComputeCost, Lazy Theta*-P pessimistically assumes that an unexpanded visible neighbor s' of a vertex s does not have line-of-sight to the parent of vertex s, while Lazy Theta* optimistically assumes that it does. Both Lazy Theta*-P and Lazy Theta* then check their assumption in procedure SetVertex and, if necessary, correct it immediately before expanding vertex s'. The idea behind the pessimistic assumption is that it allows Lazy Theta*-P to maintain a desirable property of Basic Theta*, namely that every vertex in the open or closed list has line-of-sight to its parent. This property is required by variants of Basic Theta*, such as Incremental Phi*, which we introduce in Chapter 6.

		A* PS	Basic Theta*	Lazy Theta*	Lazy Theta*-R	Lazy Theta*-P
$50 \times 50 \times 50$	Random Grids 0%	1.0839	1.0839	1.0839	1.0839	1.0839
	Random Grids 5%	1.0675	1.0783	1.0780	1.0781	1.0758
	Random Grids 10%	1.0596	1.0761	1.0753	1.0761	1.0729
	Random Grids 20%	1.0464	1.0704	1.0688	1.0702	1.0658
	Random Grids 30%	1.0389	1.0652	1.0633	1.0651	1.0591
$100 \times 100 \times 100$	Random Grids 0%	1.0836	1.0836	1.0836	1.0836	1.0836
	Random Grids 5%	1.0656	1.0807	1.0803	1.0806	1.0788
	Random Grids 10%	1.0615	1.0816	1.0811	1.0816	1.0795
	Random Grids 20%	1.0508	1.0761	1.0749	1.0762	1.0719
	Random Grids 30%	1.0440	1.0761	1.0748	1.0762	1.0712

Table 5.1: A* on Triple Cubic Graphs Path Length / Path Length

		A* on Triple Cubic Graphs (shortest grid paths)	A* PS	Lazy Theta*	Lazy Theta*-R	Lazy Theta*-P
$50 \times 50 \times 50$	Random Grids 0%	4.82	0.03	1.00	1.00	0.44
	Random Grids 5%	7.11	0.04	0.74	0.96	0.46
	Random Grids 10%	9.40	0.06	0.74	1.04	0.47
	Random Grids 20%	13.82	0.11	0.77	1.00	0.58
	Random Grids 30%	10.55	0.16	0.80	1.01	0.61
$100 \times 100 \times 100$	Random Grids 0%	9.60	0.01	1.00	1.00	0.45
	Random Grids 5%	15.12	0.02	0.66	0.95	0.37
	Random Grids 10%	27.19	0.04	0.81	1.02	0.53
	Random Grids 20%	36.39	0.07	0.79	1.00	0.54
	Random Grids 30%	29.40	0.10	0.85	1.04	0.62

Table 5.2: Basic Theta* Vertex Expansions / Vertex Expansions

		A* PS	Lazy Theta*	Lazy Theta*-R	Lazy Theta*-P
$50 \times 50 \times 50$	Random Grids 0%	98.74	19.83	19.83	8.72
	Random Grids 5%	137.18	13.65	6.37	8.43
	Random Grids 10%	181.03	13.05	6.42	8.14
	Random Grids 20%	279.10	12.22	5.90	9.18
	Random Grids 30%	383.37	11.63	5.77	8.79
$100 \times 100 \times 100$	Random Grids 0%	177.59	18.12	18.12	8.11
	Random Grids 5%	271.21	11.55	4.87	6.54
	Random Grids 10%	446.09	13.04	5.31	8.58
	Random Grids 20%	738.23	11.53	5.08	7.88
	Random Grids 30%	1008.25	11.43	5.09	8.36

Table 5.3: Basic Theta* Line-of-Sight Checks / Line-of-Sight Checks

		A* on Triple Cubic Graphs (shortest grid paths)	A* PS	Lazy Theta*	Lazy Theta*-R	Lazy Theta*-P
$50 \times 50 \times 50$	Random Grids 0%	5.59	0.14	2.97	3.53	1.62
	Random Grids 5%	4.35	0.10	1.49	1.41	1.03
	Random Grids 10%	5.60	0.12	1.39	1.33	1.05
	Random Grids 20%	6.29	0.16	1.33	1.25	1.10
	Random Grids 30%	5.81	0.20	1.27	1.21	1.08
$100 \times 100 \times 100$	Random Grids 0%	10.38	0.15	5.72	5.51	2.33
	Random Grids 5%	6.18	0.09	1.54	1.04	0.95
	Random Grids 10%	5.78	0.13	1.66	1.03	1.07
	Random Grids 20%	6.69	0.17	1.33	0.96	0.98
	Random Grids 30%	9.89	0.22	1.31	0.92	1.08

Table 5.4: Basic Theta* Runtime / Runtime

5.6 Experimental Results

In this section, we compare A* on triple cubic graphs, A* PS, Basic Theta*, Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P with respect to their path length, number of vertex expansions, runtime and number of line-of-sight checks.

We compare these find-path algorithms on $50 \times 50 \times 50$ and $100 \times 100 \times 100$ triple cubic graphs with different percentages of randomly blocked grid cells (random grids). The start vertex is always in the lower left front corner of the cubic grid, and the goal vertex is in the corner of a grid cell randomly chosen from the right most column*s* of grid cells. Grid cells are blocked randomly but a one-unit border of grid cells around the cubic grid is left unblocked in order to guarantee that there is path from the start vertex to the goal vertex. We average over 100 random $50 \times 50 \times 50$ grids and 100 random $100 \times 100 \times 100$ grids.

All find-path algorithms use the Euclidean straight-line distances as h-values except for A* on triple cubic graphs which uses 3D octile distances (Algorithm 6). A* on triple cubic graphs and A* PS break ties among vertices with the same f-value in the open list in favor of vertices with larger g-values (when deciding which vertex to expand next) since this tie-breaking scheme typically results in fewer vertex expansions and thus shorter runtimes for A* on triple cubic graphs. Basic Theta*, Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P break ties in favor of vertices with smaller g-values for the reasons explained in Section 4.8.

Table 5.1 compares the ratio of the lengths of the paths found by *A** on triple cubic graphs and the lengths of the paths found by the other find-path algorithms. Table 5.2 compares the ratios of the number of vertices expanded by *Basic Theta** and the number of vertices expanded by the other find-path algorithms. Table 5.3 compares the ratios of the number of line-of-sight checks performed by *Basic Theta** and the number of line-of-sight checks performed by the other find-path algorithms. Table 5.4 compares the ratios of the runtimes of *Basic Theta** and the runtimes of the other find-path algorithms. We make the following observations:

- **Path Lengths (Table 5.1):** The lengths of the paths found by Basic Theta*, Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P are similar to one another and shorter than the lengths of the paths found by A* on triple cubic graphs. Shortest grid paths formed by the edges of octile graphs and thus the paths found by A* on octile graphs (with consistent h-values) can be up to $\approx 8\%$ longer than the true shortest paths. Thus, the paths found by A* on octile graphs can be at most $\approx 8\%$ longer than the paths found by Basic Theta*. Table 5.1 shows that the paths found by A* on triple cubic graphs constructed from random grids with 0 percent blocked grid cells are on average more than $\approx 8\%$ longer than the paths found by Basic Theta* and thus the paths found by A* on triple cubic graphs are at least $\approx 8\%$ longer than the true shortest paths. The relationship between the lengths of the paths found by A* on triple cubic graphs and Basic Theta* are similar for larger percentages of blocked grid cells. This is important because it shows that more sophisticated find-path algorithms are even more desirable in known 3D environments than they are in known 2D environments. This follows from the fact that shortest grid paths formed by the edges of triple cubic graphs are longer than shortest grid paths formed by the edges of octile graphs, relative to true shortest paths. The values in Table 5.1 were presented with respect to A* on triple cubic graphs to highlight this fact.

The lengths of the paths found by A* PS are shorter than the lengths of the paths found by A* on triple cubic graphs, but longer than the lengths of the paths found by Basic Theta*, Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P.

- **Vertex Expansions (Table 5.2):** Lazy Theta* (which makes optimistic assumptions) and Lazy Theta*-P (which makes pessimistic assumptions) both expand more vertices than Basic Theta* (which makes no assumptions). The assumptions made by both Lazy Theta* and Lazy Theta*-P make their g-values (and thus the keys of the vertices in the open list) less informed than those of Basic Theta*. Lazy Theta* expands more vertices than Lazy Theta*-R (except for random grids with 0 percent blocked grid cells), but Lazy Theta*-R performs about 2-3 key updates for every vertex that it expands, which increases both its runtime and the number of line-of-sight checks that it performs.

  A* PS expands far more vertices than any of the other find-path algorithms and A* on triple cubic graphs expands far fewer vertices than any of the other find-path algorithms. A* on triple cubic graphs expands very few vertices for the same reasons that A* on octile graphs expands very few vertices (Section 4.6). 3D octile distances and octile distances estimate goal distances very accurately, especially on triple cubic graphs constructed from cubic grids with small percentages of blocked grid cells and octile graphs constructed from square grids with small percentages of blocked grid cells, respectively. In fact, for these experiments, the 3D octile distances are extremely effective at keeping the total number of vertex expansions small (the average number of vertex expansions for all of these experiments is ≈ 50 on random $50 \times 50 \times 50$ grids and ≈ 100 on random $100 \times 100 \times 100$ grids and in both cases the variance was small), relative to the total number of vertices in the graph (the total number of vertices in triple cubic graphs is $125,000$ for random $50 \times 50 \times 50$ grids and $1,000,000$ for random $100 \times 100 \times 100$ grids). We show in the next section that there are other experimental setups in which the 3D octile distances do not estimate goal distances as accurately, which results in a larger number of vertices expanded by A* on triple cubic graphs.

- **Line-of-Sight Checks (Table 5.3):** Basic Theta* performs more line-of-sight checks than Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P because it does not use lazy evaluation to reduce the number of line-of-sight checks that it performs. Lazy Theta*-R performs more line-of-sight checks than Lazy Theta* and Lazy Theta*-P since it can repeatedly consider a vertex for expansion, each of which requires a line-of-sight check.[5] Lazy Theta*-P performs more line-of-sight checks than Lazy Theta* because it expands more vertices than Lazy Theta*.

  The number of line-sight-checks performed by A* PS depends only on the number of vertices on the path, which remained relatively constant at ≈ 50 and ≈ 100 on random $50 \times 50 \times 50$ cubic grids and random $100 \times 100 \times 100$ grids, respectively, for all percentages of blocked grid cells.

- **Runtime (Table 5.4):** Basic Theta* has a longer runtime than Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P. Lazy Theta* has a shorter runtime than Lazy Theta*-R, Lazy Theta*-P

[5] In random grids with 0 percent blocked grid cells, Lazy Theta* and Lazy Theta*-R perform the same number of line-of-sight checks.

and Basic Theta*, which is not surprising since the runtime is heavily influenced by the number of line-of-sight checks and Lazy Theta* performs the fewest line-of-sight checks.

A* on triple cubic graphs has a shorter runtime than all of the other find-path algorithms and A* PS has a longer runtime that all of the other find-path algorithm. This is not surprising since the runtime is heavily influenced by the number of vertex expansions and the number of vertices expanded by A* on triple cubic graphs is by far the smallest and the number of vertices expanded by A* PS is by far the largest.

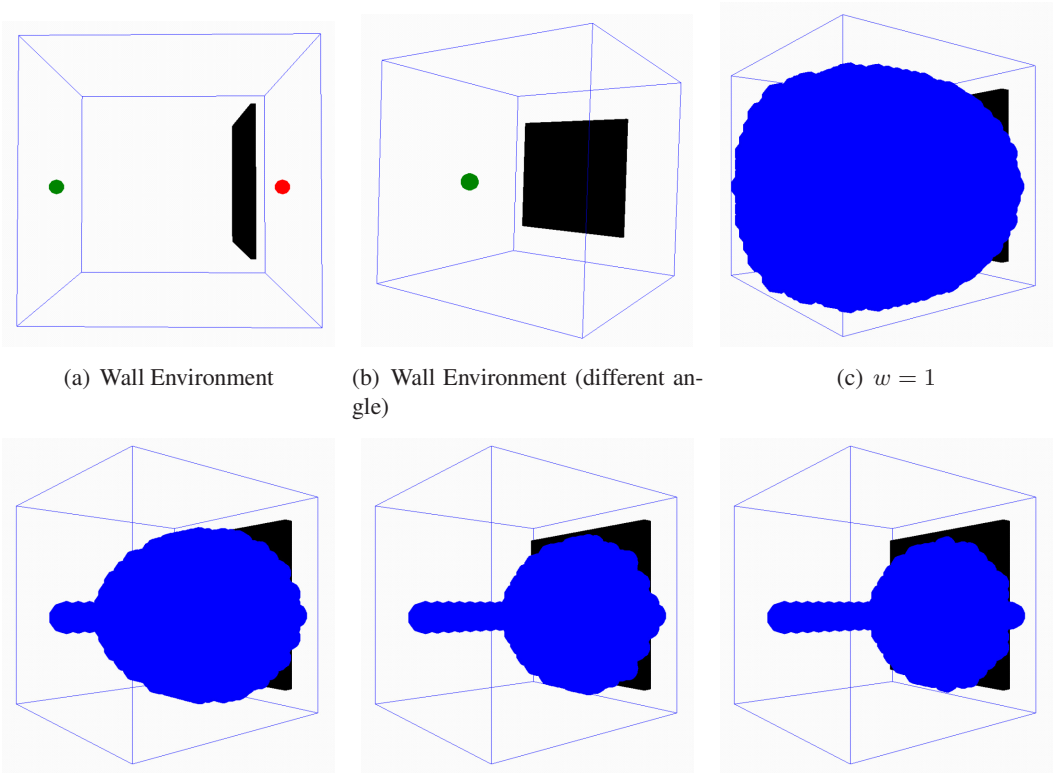
However, these runtime comparisons must be examined with care. The actual runtime differences between the various find-path algorithms varies (potentially significantly) with the runtime of a single vertex expansion and a single line-of-sight check, which can vary for different machines or with different implementations.

We performed paired t-tests to verify the statistical significance of the results above. They show with confidence level $\alpha = 0.01$ that Basic Theta* and Lazy Theta* indeed find shorter paths than A* on triple cubic graphs and A* PS, that Basic Theta* finds shorter paths than Lazy Theta* and that Lazy Theta* indeed has a shorter runtime than A* PS and Basic Theta*.

To summarize, the experimental results demonstrate that Lazy Theta* provides a good tradeoff with respect to the number of line-of-sight checks and runtime of the search on one hand and the lengths of the resulting paths on the other hand. The runtime of Lazy Theta* depends on the runtime required to perform a line-of-sight check. As the runtime of a line-of-sight check increases, so does the runtime advantage of Lazy Theta* over Basic Theta*. This is relevant because Lazy Theta* and Basic Theta* can be used to search graphs constructed from many different discretizations of a continuous environment and the runtime of a line-of-sight check varies depending on which one is used. The experimental results also demonstrate that Basic Theta*, Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P provide a dominating tradeoff, relative to A* PS, with respect to the runtime of the search and the length of the resulting path. Finally, the experimental results demonstrate that, while Basic Theta*, Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P find shorter paths than A* on triple cubic graphs (in fact, the difference between the lengths of the paths found Basic Theta* and A* on triple cubic graphs, relative to the lengths of true shortest paths, is greater for triple cubic graphs ($\approx 7 - 8\%$) than it is for octile graphs ($\approx 4-5\%$) on average), they do so with longer runtimes. This is largely due to the fact that A* on triple cubic graphs performs far fewer vertex expansions than Basic Theta*, Lazy Theta*, Lazy Theta*-R and Lazy Theta*-P. In the next section, we introduce optimizations to Basic Theta* and Lazy Theta* that allow them to perform approximately the same number of vertex expansions that A* on triple cubic graphs performs, with only a small increase in path lengths (relative to Basic Theta* and Lazy Theta* without these optimization).

5.7 Trading Off Runtime and Path Length: Exploiting h-Values

There are strategies for trading off the runtime of the search and the length of the resulting paths that A* on triple cubic graphs, Basic Theta* and Lazy Theta* share. However, their behaviors can be different even though the three find-path algorithms have similar pseudocode. In this section, we develop variants of Lazy Theta* and Basic Theta* that might be able to find longer paths at an decrease in runtime by using weighted h-values with weights greater than one.



(a) Wall Environment

(b) Wall Environment (different angle)

(c) $w = 1$

(d) $w = 2$

(e) $w = 5$

(f) $w = 10$

Figure 5.7: Expanded Vertices by Lazy Theta* with Different Values of $w > 1$ (in an environment in which find-path algorithms encounter local minima)

5.7.1 Weighted h-Values with $w > 1$

In this section, we show that, by using weighted h-values with weights greater than one, Lazy Theta* and Basic Theta* can find paths faster without a significant increase in the lengths of the resulting paths.

So far, Basic Theta* and Lazy Theta* have used consistent h-values (Section 4.8). A* with consistent h-values finds paths of the same length no matter how small or large the h-values are. A* with inconsistent h-values is able to find longer paths at a decrease in runtime by using weighted h-values with weights greater than one. We therefore now discuss variants of Basic Theta* and Lazy Theta* that might be able to find longer paths at a decrease in runtime by using weighted h-values with weights greater than one. These variants of Basic Theta* and Lazy Theta* use the h-values $h(s) = w \cdot c(s, s_{goal})$ for a given weight $w > 1$ and thus are similar to Weighted A*. Weighted A* with $w > 1$ can significantly reduce runtimes without a significant increase in path lengths (Korf, 1993; Bonet & Geffner, 2001; Zhou & Hansen, 2002) and/or allow searches to be performed on larger graphs (Likhachev, Ferguson, Gordon, Stentz, & Thrun, 2008).

A* searches with inconsistent h-values typically expand fewer vertices, but often find longer paths because every vertex omitted from a search explicitly omits a potential path. If a vertex s is not expanded during a search then a visible neighbor s' of vertex s cannot have vertex s as its

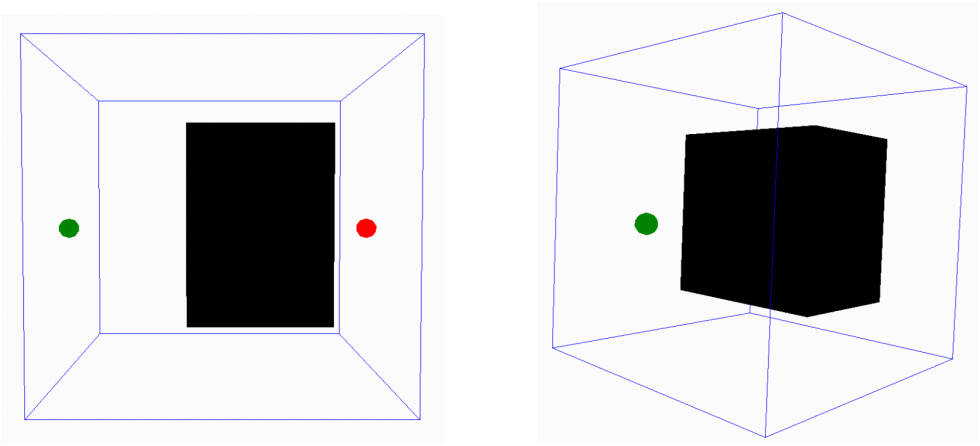
parent and thus vertex s cannot be on any path. For Basic Theta* and Lazy Theta*, the effect of not expanding a vertex during the search is different because Basic Theta* and Lazy Theta* consider both Path 1 and Path 2. If a vertex s is not expanded during a search then a visible neighbor s' of vertex s cannot have vertex s as its parent, but vertex s' can still potentially have parent *parent*(s) (that is, the parent vertex s would have had if vertex s had been expanded) due to Path 2. In fact, it is likely that several other vertices have parent *parent*(s) from which vertex s' can inherit parent *parent*(s). This property suggests that Basic Theta* and Lazy Theta* might be able to find paths faster with only a small increase in the lengths of the resulting paths by using weighted h-values with $w > 1$. An example of this can be seen in Figure 5.7, which compares four different Lazy Theta* searches on identical triple cubic graphs with the same start vertex (green sphere) and goal vertex (red sphere). Figures 5.7(a) and 5.7(b) show the same known 3D environment from two different angles. This environment has a large wall between the start vertex and the goal vertex. In Figure 5.7(c), a Lazy Theta* search was performed with $w = 1$; in Figure 5.7(d), a Lazy Theta* search was performed with $w = 2$; in Figure 5.7(e), a Lazy Theta* search was performed with $w = 5$ and in Figure 5.7(f), a Lazy Theta* search was performed with $w = 10$. The blue spheres in each figure represent vertices that were expanded during the Lazy Theta* search. While it cannot be seen in the figures, it turns out that Lazy Theta* finds the same path in all four searches. A* searches on the other hand find longer paths as w increases (Likhachev & Stentz, 2008). Figure 5.7 suggests that Basic Theta* and Lazy Theta* with $w > 1$ might expand fewer and fewer vertices as w increases with little or no increase in the lengths of the paths that they find.

In the following sections, we compare find-path algorithms with $w > 1$ on both environments in which the h-values provide accurate estimates of the goal distances and environments in which they do not provide accurate estimates of the goal distances. The number of vertices expanded reflects the ability of the h-values to focus the search. This dependency on the h-values leads to a problem when they guide the search into a local minimum. A local minimum of the h-values is a portion of the graph from which there is no way to reach vertices with smaller h-values without passing through vertices with larger h-values. Therefore, we compare find-path algorithms with $w > 1$ on both environments in which find-path algorithms do encounter local minima (Figure 5.7) and environments in which they do not encounter local minima (Figure 5.10). For A* PS, Basic Theta* and Lazy Theta* we use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values. For A* on triple cubic graphs, we use the 3D octile distances as h-values.

5.7.1.1 Environments in which Find-Path Algorithms Encounter Local Minima

In this section, we compare find-path algorithms using weighted h-values with $w > 1$ on environments in which find-path algorithms encounter local minima. This can occur, for example, when there is a large obstacle blocking the goal vertex (Figures 5.7 and 5.8). The obstacle presents a large local minimum for the h-values and thus any find-path algorithm has to expand all of the vertices surrounding the large obstacle before it finds a way around it. This remains true for any value of w (Likhachev & Stentz, 2008).

We compare A* on triple cubic graphs, A* PS, Basic Theta* and Lazy Theta* with respect to their path lengths, number of vertex expansions, and number of line-of-sight checks when finding paths on the random $50 \times 50 \times 50$ grids depicted in Figure 5.7 (Wall Environment) and Figure 5.8 (Cul-De-Sac Environment). Although the latter pair of figures appear to depict a solid box



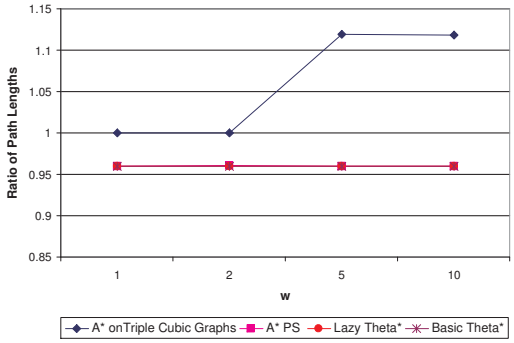
(a) Cul-De-Sac Environment (b) Cul-De-Sac Environment (different angle)

Figure 5.8: Cul-De-Sac Environment

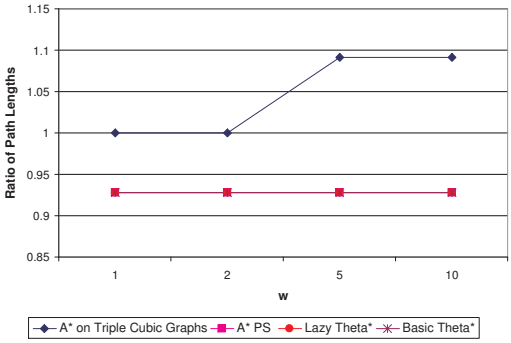
they actually depict a cul-de-sac of depth 20. For both the Wall Environment and the Cul-De-Sac Environment, the start vertex is always at coordinates $(0, 24, 24)$, and the goal vertex is always at coordinates $(49, 24, 24)$.

Searches on triple cubic graphs constructed from $n \times n \times n$ cubic grids perform at least n vertex expansions when finding a path between our start vertex and goal vertex. Therefore, we compare the number of vertex expansions performed by each find-path algorithm relative to $n = 50$. For line-of-sight checks we also compare each find-path algorithm relative to $n = 50$. Post-smoothing algorithms, such as Algorithm 2, shorten paths by removing extraneous vertices from the path found by A* on triple cubic graphs using the triangle inequality. Any path between our start vertex and goal vertex has at least n vertices all of which should be considered for removal by the post-smoothing algorithm (except for the start and goal vertex). For path length we compare the lengths of the paths found by each find-path algorithm relative to the lengths of the paths found by A* on triple cubic graphs with $w = 1$.

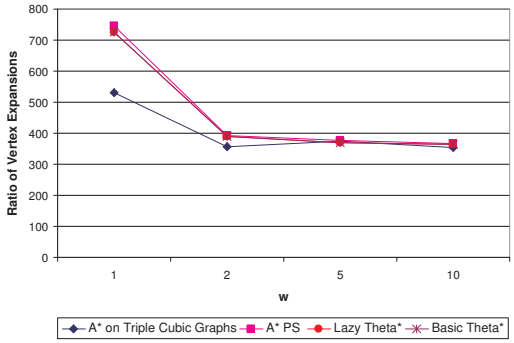
The intent of this experimental setup is to provide a more objective analysis as to the tradeoffs that A* on triple cubic graphs, A* PS, Basic Theta* and Lazy Theta* provide with respect to the runtime of the search and the length of the resulting path. Runtime is somewhat subjective in the sense that it is effected by both the specifications of the machine and various implementation decisions (for example, programming language) and thus we use the number of vertex expansions and the number of line-of-sight checks as proxies for runtime. In previous experiments, such as those in Section 4.6, vertex expansions and line-of-sight checks could not be used effectively as proxies for runtime because each find-path algorithm performed very different computations during a vertex expansion (for example, Field D* performed linear interpolation, AP Theta* performed trigonometric computations and Basic Theta* and A* PS performed line-of-sight checks) and each find-path algorithm performed a different number of vertex expansions (for example, A* on octile graphs with the octile distances expanded far fewer vertices than Basic Theta*). In these experiments the types of computations performed by each find-path algorithm during a vertex expansion are very similar (except for line-of-sight checks) and we use weighted h-values with $w > 1$ to significantly reduce the differences between the number of vertex expansions performed



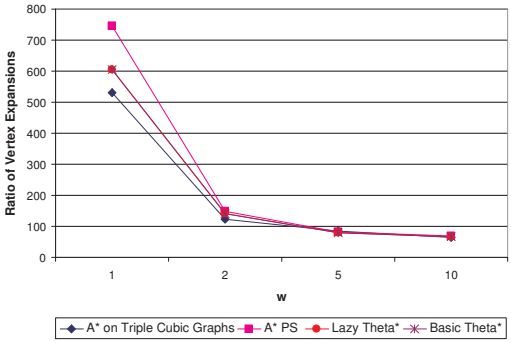
(a) Path Lengths in the Cul-De-Sac Environment



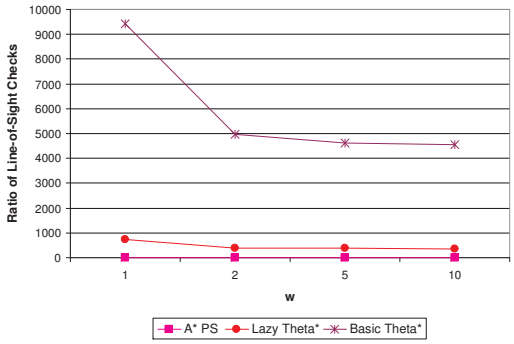
(b) Path Lengths in the Wall Environment



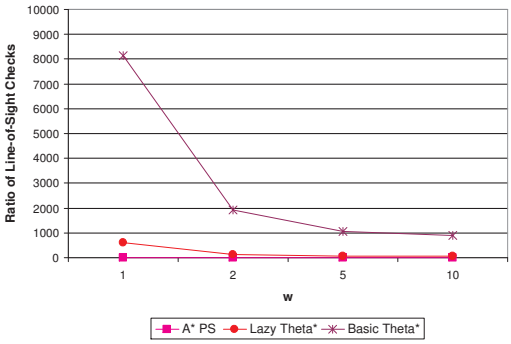
(c) Number of Vertex Expansions in the Cul-De-Sac Environment



(d) Number of Vertex Expansions in the Wall Environment



(e) Number of Line-of-Sight Checks in the Cul-De-Sac Environment



(f) Number of Line-of-Sight Checks in the Wall Environment

Figure 5.9: Weighted h-Values with $w > 1$ on $50 \times 50 \times 50$ Grids

by each find-path algorithm. For vertex expansions and paths lengths we compare each find-path algorithm relative to values that are frequently achieved by traditional edge-constrained find-path algorithms, that is, $n = 50$ vertex expansions and the lengths of the paths found by A* on triple cubic graphs with $w = 1$. For line-of-sight checks we compare each find-path algorithm relative

to values that are frequently achieved by post-smoothing algorithms, that is, $n = 50$ line-of-sight checks.

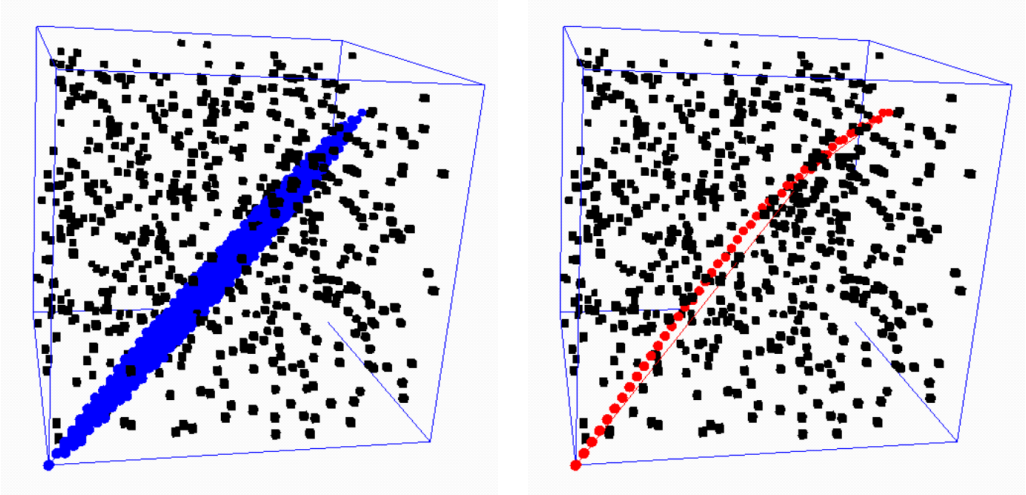
Figure 5.9 reports our experimental results. In each sub figure, the x-axis indicates the value of w.

- **Path Lengths:** In Figures 5.9(a) and 5.9(b), the y-axis indicates the ratio of the lengths of the paths found by each find-path algorithm and the lengths of the paths found by A* on triple cubic graphs with $w = 1$. Values smaller than 1 imply that the find-path algorithm found shorter paths than A* on triple cubic graphs with $w = 1$. We make the following observations: for all values of w, A* PS, Basic Theta* and Lazy Theta* find shorter paths than A* on triple cubic graphs. For all values of w, the lengths of the paths found by A* PS, Basic Theta* and Lazy Theta* remain constant. The paths found by A* on triple cubic graphs with $w > 1$ can be more than $\approx 10\%$ longer than the paths found by A* on triple cubic graphs with $w = 1$. The paths found by a A* on triple cubic graphs with $w > 1$ can be $\approx 15\%$ longer than the paths found by Basic Theta* and Lazy Theta* with $w > 1$.

- **Vertex Expansions:** In Figures 5.9(c) and 5.9(d), the y-axis indicates the ratio of the number of vertex expansions performed by each find-path algorithm and 50. Values larger than 1 imply that the find-path algorithm expanded more than the lower bound on the number of vertex expansions (that is, 50). We make the following observations: starting with $w = 2$ and continuing as w increases, all find-path algorithms expand approximately the same number of vertices. This validates our earlier statement that any find-path algorithm expands all of the vertices either adjacent to the wall for the Wall Environment or within the cul-de-sac for the Cul-De-Sac Environment before it finds a way around it. This remains true for any value of w.

- **Line-of-Sight Checks:** In Figures 5.9(e) and 5.9(f), the y-axis indicates the ratio of the number of line-of-sight checks performed by each find-path algorithm and 50. Values larger than one imply that the find-path algorithm performed more line-of-sight checks than the lower bound on the number of line-of-sight checks (that is, 50). A* on triple cubic graphs has been omitted for obvious reasons. We make the following observations: for all values of w, A* PS performs ≈ 50 line-of-sight checks. For all values of w, Lazy Theta* performs more line-of-sight checks than A* PS and Basic Theta* performs many more line-of-sight checks than both A* PS and Lazy Theta*.

In this section, we compared A* on triple cubic graphs, A* PS, Basic Theta* and Lazy Theta*, with $w > 1$ on environments in which find-path algorithms encounter local minima. We compared these algorithms in terms of the number of vertex expansions they perform, the number of line-of-sight checks they perform and the lengths of the paths they find. We showed that, much like A* on triple cubic graphs with $w > 1$, Basic Theta* and Lazy Theta* with $w > 1$ expand fewer and fewer vertices as w increases until they reach some minimum number of vertex expansions. However, *unlike* A* on triple cubic graphs with $w > 1$, Basic Theta* and Lazy Theta* with $w > 1$ find paths of the same length as w increases. As a result, the paths found by Basic Theta* and Lazy Theta* with $w > 1$ can be $\approx 15\%$ shorter than the paths found by A* on triple cubic graphs with the same $w > 1$. This is despite the fact that A* on triple cubic graphs, A* PS, Basic Theta* and Lazy Theta* expand approximately the same number of vertices. The number of line-of-sight checks performed by Basic Theta* and Lazy Theta* with $w > 1$ depends on the number



(a) $w = 1$ (b) $w = 1.3$

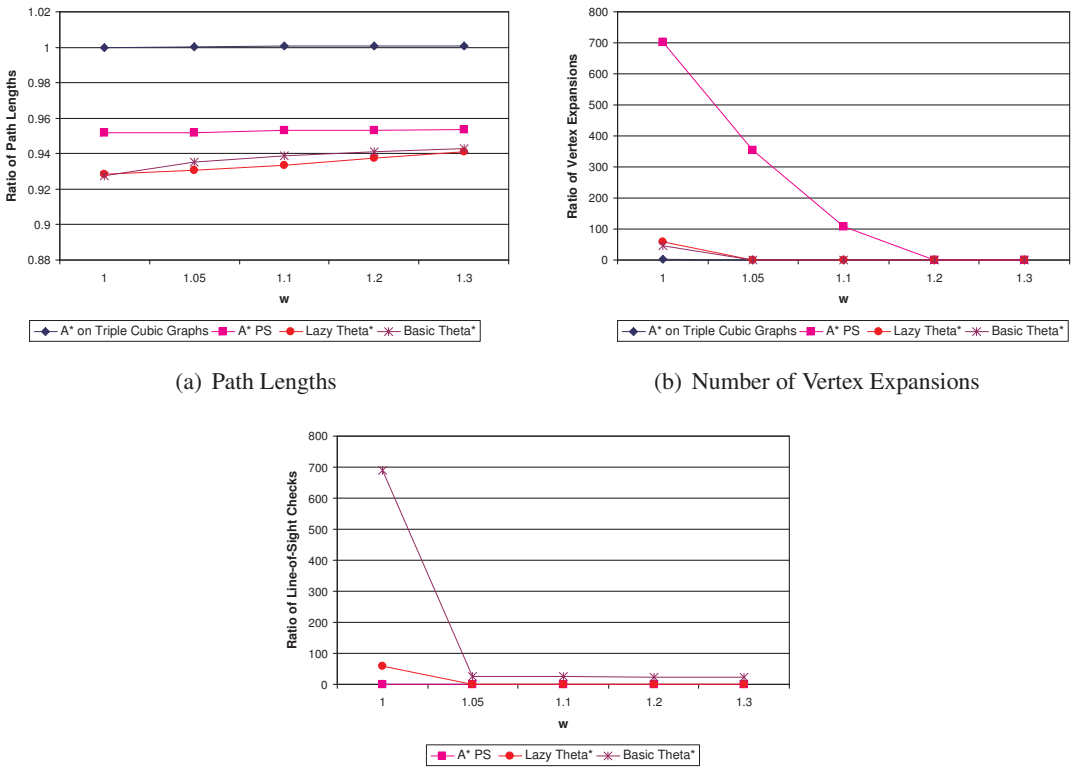
Figure 5.10: Lazy Theta* with $w > 1$ on an Environment in which Find-Path Algorithms do *not* Encounter Local Minima

vertex expansions performed by Basic Theta* and Lazy Theta* with $w > 1$. Therefore, since Basic Theta* and Lazy Theta* with $w > 1$ expand fewer and fewer vertices as w increases, until they perform some minimum number of vertex expansions, they also perform fewer and fewer line-of-sight checks as w increases, until they perform some minimum number of line-of-sight checks.

5.7.1.2 Environments in which Find-Path Algorithms do *not* Encounter Local Minima

In this section, we compare find-path algorithms with $w > 1$ on environments in which find-path algorithms do not encounter local minima. This can occur in environments with small percentages of randomly blocked grid cells because for these types of environments a weighted h-value efficiently focuses the search (Figure 5.10). An example of this can be seen in Figure 5.10, which compares two different searches on identical random grids with the same start vertex and goal vertex. In Figure 5.10(a), a Lazy Theta* search was performed with $w = 1$ and, in Figure 5.10(b), a Lazy Theta* search was performed with $w = 1.3$. The blue dots in Figure 5.10(a) and the red dots in Figure 5.10(b) represent vertices that were expanded during the Lazy Theta* search with $w = 1$ and $w = 1.3$, respectively. In the latter case, 15 times fewer vertices were expanded, but the path was only 1.003 times longer. Figure 5.10 suggests that Lazy Theta* with $w > 1$ expands fewer and fewer vertices as the value of w increases with only a small increase in path lengths.

We compare Lazy Theta*, Basic Theta*, A* on triple cubic graphs and A* PS with respect to their path lengths, number of vertex expansions, and number of line-of-sight checks when finding paths on random $100 \times 100 \times 100$ grids with 20 percent randomly blocked grid cells. The results are similar for different percentages of blocked grid cells. The start vertex is always in the lower left front corner of the cubic grid, and the goal vertex is in the corner of a grid cell randomly chosen from the right most column*s* of grid cells. Grid cells are blocked randomly but a one-unit border of grid cells around the cubic grid is left unblocked in order to guarantee that there is



(a) Path Lengths

(b) Number of Vertex Expansions

(c) Number of Line-of-Sight Checks

Figure 5.11: Weighted h-Values with $w > 1$ on Random $100 \times 100 \times 100$ Grids with 20 Percent Blocked Grid Cells

path from the start vertex to the goal vertex. We average over 100 path-planning problems. For reasons described in the previous section, we compare the number of vertex expansions performed by each find-path algorithm relative to $n = 100$, we compare the number of line-of-sight checks performed by each find-path algorithm relative to $n = 100$ and we compare the length of the path found by each find-path algorithm relative to the length of the path found by A* on triple cubic graphs with $w = 1$.

Figure 5.11 reports our experimental results. In each sub figure, the x-axis indicates the value of w.

- **Path Lengths:** In Figure 5.11(a), the y-axis indicates the ratio of the lengths of the paths found by each find-path algorithm and the lengths of the paths found by A* on triple cubic graphs using $w = 1$. Values smaller than 1 imply that the find-path algorithm found shorter paths than the paths found by A* on triple cubic graphs with $w = 1$. We make the following observations: for all values of w, Basic Theta* and Lazy Theta* find shorter paths than A* on triple cubic graphs and A* PS. For all values of w, the lengths of the paths found by both A* on triple cubic graphs and A* PS remain relatively constant. Starting with $w = 1.05$ and continuing as w increases, the lengths of the paths found by both Basic Theta* and Lazy Theta* increase slightly.

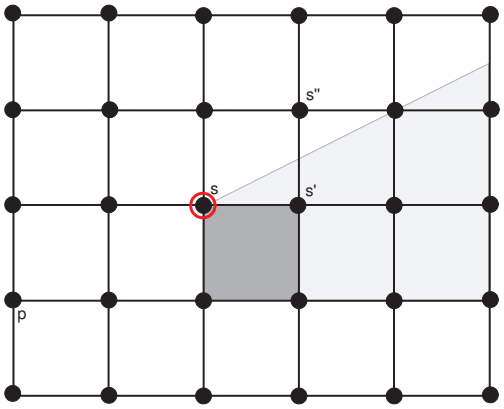


Figure 5.12: Basic Theta* Path Lengths and Lazy Theta* Path Lengths with $w > 1$

Interestingly, we also observe that, as w increases, unlike what we saw in Section 5.6, Lazy Theta* finds slightly shorter paths than Basic Theta*. We use Figure 5.12, in which all the vertices in the shaded region behind the blocked grid cell do not have line-of-sight to p and the goal vertex is somewhere far off in the upper right portion of the shaded region, to provide a potential explanation. Since exact coordinates for the goal vertex is not provided, assume that the h-values of vertices s' and s'' are equal.

- Basic Theta* with $w > 1$ is more likely to expand vertices outside the shaded region than inside the shaded region. Consider the case in which Basic Theta* expands a vertex s with parent p and updates an unexpanded visible neighbor s'' of vertex s according to Path 2 and an unexpanded visible neighbor s' of vertex s according to Path 1. Vertex s'' is likely expanded before vertex s' because $g(s'') < g(s')$, $w \cdot h(s'') = w \cdot h(s')$ and thus $f(s'') < f(s')$. Furthermore, because the h-values are weighted with $w > 1$, it is likely that vertex s' is never expanded. Therefore, Basic Theta* with $w > 1$ has the tendency to expand vertices just outside the boundary between those vertices that have line-of-sight to parent p and those vertices that do not have line-of-sight to parent p. At some point, however, the path from the start vertex to the goal vertex must cross into the shaded region and an unnecessary heading change is then introduced into the path.
- Lazy Theta* with $w > 1$ is more likely to expand vertices inside the shaded region than outside the shaded region. When Lazy Theta* expands a vertex s with parent p and updates unexpanded visible neighbors s'' and s' of vertex s it updates both of them according to Path 2 even though vertex s' does not have line-of-sight to parent p. Vertex s' is likely expanded before vertex s'' because $g(s') < g(s'')$, $w \cdot h(s'') = w \cdot h(s')$ and thus $f(s') < f(s'')$. Furthermore, because the h-values are weighted with $w > 1$, it is likely that vertex s'' is never expanded. Therefore, Lazy Theta* with $w > 1$ has the tendency to expand vertices inside the boundary between those vertices that have line-of-sight to parent p and those vertices that do not have line-of-sight to parent p. The path from the start vertex to the goal vertex found by Lazy Theta* crosses into the shaded region at vertex s, where a "necessary" heading change is

introduced. Because the path from the start vertex to the goal vertex found by Lazy Theta* does not have an unnecessary heading change it is shorter than the path from the start vertex to the goal vertex found by Basic Theta*.

- **Vertex Expansions:** In Figure 5.11(b), the y-axis indicates the ratio of the number of vertex expansions performed by each find-path algorithm and 100. Values larger than 1 imply that the find-path algorithm expanded more than the lower bound on the number of vertex expansions (that is, 100). We make the following observations: for all values of w, A* on triple cubic graphs expands ≈ 100 vertices. Starting with $w = 1.05$ and continuing as w increases, Basic Theta* and Lazy Theta* expand ≈ 100 vertices. Starting with $w = 1.2$ and continuing as w increases, A* PS (which uses the straight line distances as h-values) expands ≈ 100 vertices.

- **Line-of-Sight Checks:** In Figure 5.11(c), the y-axis indicates the ratio of the number of line-of-sight checks performed by each find-path algorithm and 100. Values larger than 1 imply that the find-path algorithm performed more line-of-sight checks than the lower bound on the number of line-of-sight checks (that is, 100). A* on triple cubic graphs has been omitted for obvious reasons. We make the following observations: for all values of w, A* PS performs ≈ 100 line-of-sight checks. Starting with $w = 1.05$ and continuing as w increases, Lazy Theta* performs ≈ 100 line-of-sight checks. Starting with $w = 1.05$ and continuing as w increases, Basic Theta* performs ≈ 26 times more line-of-sight checks than A* PS and Lazy Theta*.

In this section, we compared A* on triple cubic graphs, A* PS, Basic Theta* and Lazy Theta* using weighted h-values with $w > 1$ on environments in which find-path algorithms do *not* encounter local minima. We compared these algorithms in terms of the number of vertex expansions they perform, the number of line-of-sight checks they perform and the lengths of the paths they find. We showed that, much like A* on triple cubic graphs with $w > 1$, Basic Theta* and Lazy Theta* with $w > 1$ expand fewer and fewer vertices as w increases until they perform n vertex expansions. We showed that Basic Theta* and Lazy Theta* with $w > 1$ find paths of approximately the same length as w increases and that these paths are $\approx 6 - 8\%$ shorter than the paths found by A* on triple cubic graphs with $w \geq 1$. The number of line-of-sight checks performed by Basic Theta* and Lazy Theta* with $w > 1$ depends on the number vertex expansions performed by Basic Theta* and Lazy Theta* with $w > 1$. Therefore, since Basic Theta* and Lazy Theta* with $w > 1$ expand fewer and fewer vertices as w increases, until they perform n vertex expansions, they also perform fewer and fewer line-of-sight checks as w increases, until they perform $\approx 26 \cdot n$ and n line-of-sight checks, respectively.

Tables 5.2 and 4.3 demonstrate that A* on triple cubic graphs with 3D octile distances as h-values and A* on octile graphs with the octile distances as h-values, respectively, expand fewer vertices than Basic Theta*, but Basic Theta* is able to find shorter paths (especially on grids with small percentages of blocked grid cells). In Section 5.7.1, we showed in both $50 \times 50 \times 50$ grids in which find-path algorithms encounter local minima and $100 \times 100 \times 100$ grids in which find-path algorithms do not encounter local minima that Basic Theta* and Lazy Theta* with $w > 1$ expand approximately the same number of vertices as A* on triple cubic graphs with $w = 1$ while finding shorter paths. In the latter environments, we showed that, with a cost of only 100 line-of-sight checks, Lazy Theta* with $w > 1$ finds paths that are $\approx 6 - 8\%$ shorter than those found by A* on

triple cubic graphs with $w = 1$. For Basic Theta* with $w > 1$ the cost of these shorter paths was $\approx 26 \cdot 100$ line-of-sight checks which is one of the reasons we introduced weighted h-values with $w > 1$ in this section rather than Section 4.8. In the former environments, we showed that, due to more vertex expansions, the cost of these shorter paths in terms of the number of line-of-sight checks performed was greater for both Basic Theta* and Lazy Theta*. In both environments, Basic Theta* and Lazy Theta* with $w > 1$ expand fewer vertices and perform fewer line-of-sight checks than Basic Theta* and Lazy Theta* with $w = 1$, however the paths found by Basic Theta* and Lazy Theta* with $w > 1$ are only slightly longer than the paths found by Basic Theta* and Lazy Theta* with $w = 1$. These results demonstrate that Basic Theta* and Lazy Theta* satisfy the Efficiency Property.

5.8 Conclusions

In this chapter, we examined path planning in known 3D environments that have been discretized into triple cubic graphs constructed from cubic grids with vertices placed in the corners of grid cells. We showed that path planning in known 3D environments is more difficult than path planning in known 2D environments. We presented Lazy Theta*, a variant of Basic Theta* that was designed to find paths in known 3D environments. Lazy Theta* is simple to implement because its pseudocode is similar to the pseudocode of A* and Basic Theta*. Lazy Theta* is simple to understand because, like Basic Theta*, it takes advantage of Path 2 and the triangle inequality which has proven to be easy for computer science students and video game developers to understand (Simplicity Property). We compared Lazy Theta* and Basic Theta* against two existing find-path algorithms, namely, A* on triple graphs and A* with post-smoothed paths (A* PS). We showed that Lazy Theta* provides a nearly dominating tradeoff, relative to Basic Theta*, with respect to the runtime of the search and the length of the resulting path. It does this by performing one order of magnitude fewer line-of-sight checks than Basic Theta*. We showed that Basic Theta* and Lazy Theta* provide a good tradeoff with respect to the runtime of the search and the length of the resulting path. Lazy Theta* and Basic Theta* find shorter paths than A* on triple cubic graphs and A* PS. While Lazy Theta* and Basic Theta* provide a dominating tradeoff, relative to A* PS, with respect to the runtime of the search and the length of the resulting path, they are slower than A* on triple cubic graphs. Thus, we developed variants of Basic Theta* and Lazy Theta* that use weighted h-values with weights greater than one to reduce their runtimes. We showed that these variants of Basic Theta* and Lazy Theta* have a similar runtime to A* on triple cubic graphs, but find much shorter paths (Efficiency Property).

To summarize, in this chapter we validated Hypothesis 2 in known 3D environments by introducing Lazy Theta*, a new any-angle find-path algorithm that satisfies the Simplicity, Efficiency and Generality Properties when path planning in known 3D environments. Simplicity Property: Lazy Theta* is very similar to Basic Theta* which is simple to implement and understand (Section 4.4.3.1). Efficiency Property: Lazy Theta* provides a good tradeoff with respect to the runtime of the search and the length of the resulting path (Section 5.6 and 5.7.1). Generality Property: Lazy Theta* can be used to search any Euclidean graph.

Chapter 6

Incremental Phi*: Any-Angle Path Planning in Unknown 2D Environments (Contribution 4)

Figure 6.1: Unknown 2D Environments: Screen Shot from Warcraft II (Blizzard Entertainment)

This chapter introduces the fourth major contribution of this dissertation, namely Contribution 4. Specifically, this chapter introduces Incremental Phi* (Nash et al., 2009), a new incremental any-angle find-path algorithm, which we evaluate in unknown 2D environments using the Simplicity, Efficiency and Generality Properties. Our evaluation shows that Incremental Phi* is simple to implement and understand, that it provides a good tradeoff with respect to the runtime of the search and the length of the resulting path and that it can find paths one order of magnitude faster than repeated Basic Theta* searches while finding paths with nearly same lengths. Therefore, these results validate Hypothesis 2 in unknown 2D environments.

This chapter is organized as follows: In Section 6.1, we explain why path planning in unknown 2D environments is more difficult than path planning in known 2D environments. In Section 6.2, we introduce notation and definitions that are used throughout this chapter. In Section 6.3, we examine Basic Theta* in the context of path planning in unknown 2D environments. In

Section 6.4, we introduce Phi*, a new any-angle find-path algorithm. In Section 6.5, we introduce Incremental Phi*, which uses Phi* for path planning in unknown 2D environments. In Section 6.6, we present our experimental results. Finally, in Section 6.7, we summarize our results.

6.1 Introduction

Path planning in unknown 2D environments is more computationally expensive than path planning in known 2D environments. In this chapter, we study goal directed navigation problems in unknown 2D environments, where an agent has to move to a given goal coordinate (Figure 6.1). Moving the agent on a shortest path to a given goal coordinate requires a large conditional plan which is time consuming to compute (Koenig et al., 2003). As a result, several approaches have been developed that tradeoff the runtime of the search and the length of the resulting path. **Planning with the freespace assumption** is such an approach, which has been used successfully on mobile robots (Stentz & Hebert, 1995; Hebert et al., 1999). We discretize unknown 2D environments into square grids and construct 8-neighbor square grid graphs (octile graphs) from the square grids as is commonly done in this context (Koenig & Likhachev, 2002a, 2002b; Koenig et al., 2004; Stentz, 1995). An agent that uses planning with the freespace assumption finds a short path from its current vertex to the goal vertex given its current knowledge of the blockage status of the grid cells (Koenig et al., 2003). The agent then moves one unit along this path, observes the blockage status of grid cells within its sensor radius (for example, a robot's sensor radius or a video game character's ability to see), updates the graph that represents the traversable space in the continuous environment to reflect its revised knowledge of the blockage status of grid cells (that is, solves the generate-graph problem), finds a new short path from its current vertex to the goal vertex on the new graph (that is, solves the find-path problem) and then repeats this process. Thus, the agent must at least solve the generate-graph and find-path problems each time it observes its current path to be blocked, which can be slow on large grids. In Section 2.2.1.4, we showed how octile graphs can be updated efficiently to account for newly blocked grid cells. In this chapter, we study finding short paths repeatedly and efficiently.

An agent could repeatedly use a (single-shot) find-path algorithm (for example, A* or Dijkstra's algorithm) to re-plan each time it learns more about the traversable space in the continuous environment. However, the resulting runtimes can be quite long in large environments, which can force the agent to stop until it finishes planning (Stentz, 1994, 1995). Incremental find-path algorithms, such as Differential A* (Trovato & Dorst, 2002) and D* Lite (Koenig & Likhachev, 2002a), solve a series of similar find-path problems faster than repeated single-shot A* searches because they reuse information from previous searches to speed up the next one (Section 2.2.2.2). However, they find edge-constrained paths. In Chapters 4 and 5, we showed that any-angle find-path algorithms find shorter paths than edge-constrained paths. For example, the any-angle path in Figure 4.2(b) avoids the unnecessary heading change at vertex C2 in the grid path in Figure 4.2(a). It therefore makes sense to study combinations of incremental and any-angle find-path algorithms. Field D*, an extension of D* Lite, is one such combination, but it suffers from unnecessary heading changes, which often leads to long and unrealistic looking paths due to linear interpolation error (Section 4.3.3). Basic Theta* finds shorter paths than Field D* (Section 4.6), but it cannot easily be made incremental using the key ideas from D* Lite because it does not fit a standard assumption that holds for existing incremental find-path algorithms, namely that the parent of a vertex in the search tree must also be its neighbor. Lazy Theta* is even more difficult

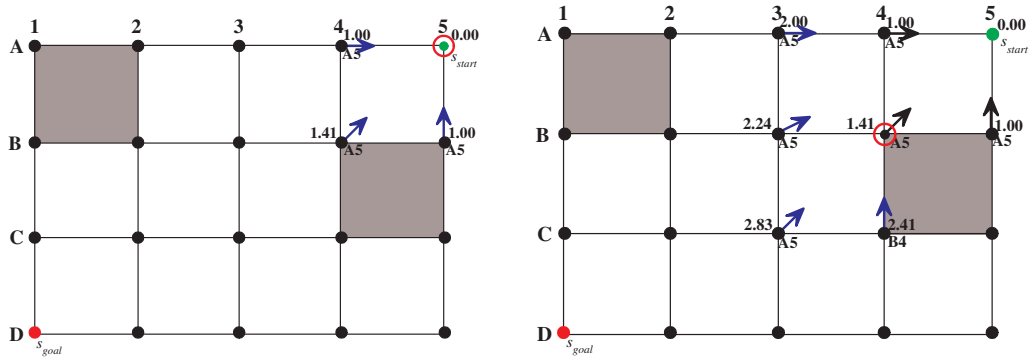


Figure 6.2: Example Trace of Basic Theta*

to make incremental because not only does it not fit this standard assumption, but it is also not guaranteed that every vertex in the search tree has line-of-sight to its parent. Therefore, we make Basic Theta* incremental. In this chapter, we present Incremental Phi*, an incremental any-angle find-path algorithm based on Basic Theta*, and show experimentally that it can speed up Basic Theta* by about one order of magnitude when path planning in unknown 2D environments with the freespace assumption.

6.2 Notation and Definitions

In this section, we describe the find-path problem that we study in this chapter, namely finding paths on octile graphs $G' = (V, E')$ (with vertices placed in the corners of grid cells) constructed from square grids that discretize unknown 2D environments (as described in Section 3.2.2). The blockage status of a grid cell never changes, but is not necessarily known a-priori to the agent, in which case the agent assumes that the grid cell is unblocked. The find-path problem is to find an unblocked path from a given start vertex $s_{start} \in V$ to a given goal vertex $s_{goal} \in V$. In this section, we use the same notation and definitions that we used in Section 4.2 with the following additions: $nghbr_8(s) \in V$ is the set of 8 neighbors of vertex s in the eight compass directions (some of which may not have line-of-sight to vertex s). $nghbr_4(s) \subseteq nghbr_8(s)$ is the set of four neighbors to the immediate north, east, south and west of vertex s. We refer to $nghbr_4(s)$ as the crossbar neighbors of vertex s since the four line segments that connect them to vertex s form a crossbar.

6.3 Find-Path Algorithms in Unknown 2D Environments

In order to better understand the differences between path planning in known 2D environments and unknown 2D environments, we briefly re-introduce Basic Theta* in the context of path planning with the free space assumption in unknown 2D environments.

```
141 Main()
142     Initialize();
143     ComputeShortestPath();
144     if g(s_goal) ≠ ∞ then
145         return "path found";
146     else
147         return "no path found";

148 Initialize()
149     open := ∅;
150     closed := ∅;
151     InitializeVertex(s_start);
152     InitializeVertex(s_goal);
153     g(s_start) := 0;
154     parent(s_start) := s_start;
155     local(s_start) := s_start;
156     open.Insert(s_start, g(s_start) + h(s_start));

157 InitializeVertex(s)
158     g(s) := ∞;
159     parent(s) := NULL;
160     local(s) := NULL;
161     [lb(s) := −∞];
162     [ub(s) := ∞];

163 ComputeShortestPath()
164     while open.TopKey() < g(s_goal) + h(s_goal) do
165         s := open.Pop();
166         closed := closed ∪ {s};
167         foreach s' ∈ nghbr_vis(s) do
168             if s' ∉ closed then
169                 if s' ∉ open then
170                     InitializeVertex(s');
171                 UpdateVertex(s, s');

172 ComputeCost(s, s')
173     if LineOfSight(parent(s), s') then
174         /* Path 2 */
175         if g(parent(s)) + c(parent(s), s') < g(s') then
176             parent(s') := parent(s);
177             g(s') := g(parent(s)) + c(parent(s), s');
178             local(s') := s;
179     else
180         /* Path 1 */
181         if g(s) + c(s, s') < g(s') then
182             parent(s') := s;
183             g(s') := g(s) + c(s, s');
184             local(s') := s;
```

Algorithm 10: Basic Theta*

6.3.1 Basic Theta*

Algorithm 10 shows the pseudocode for the implementation of Basic Theta* discussed in this chapter. Procedure UpdateVertex can be found in Algorithm 1.[1] Lines 161 and 162 are to be ignored.

[1] If Lines 178 and 184 are ignored then procedure ComputeCost in Algorithm 10 is identical to procedure ComputeCost in Algorithm 3.

This variant of Basic Theta* is different from the one we saw in Section 4.4. The reason for this is that, when planning with the freespace assumption, an agent must perform several searches (which explains the addition of a new procedure Main), some of which require that the search continues after the goal vertex has been expanded (which explains the new termination criterion on Line 164). However, for single-shot searches, Algorithms 3 and 10 expand the same vertices in the same order and thus find identical paths (provided that they break ties in the same way). Similarly, if Lines 173-179 are ignored, then Algorithm 10 and Algorithm 1 (that is, A*) expand the same vertices in the same order and thus find identical paths (provided that they break ties in the same way). We refer the reader to Section 4.3 for a detailed description of how A* operates.

As we saw earlier, Basic Theta* maintains three values for every vertex s: **(1)** The g-value $g(s)$ is the length of the shortest path from the start vertex to s found so far. **(2)** The user-provided h-value $h(s)$ is an estimate of the goal distance of vertex s. All of the find-path algorithms in this chapter use the straight-line distances $h(s) = c(s, s_{goal})$ as h-values in the experiments because they are the largest consistent h-values that are easy to compute. **(3)** The parent $parent(s)$ is used to extract a path from the start vertex to the goal vertex after the search terminates. This variant of Basic Theta* maintains a fourth value for every vertex s, which is required by Phi* and Incremental Phi*, namely its local parent $local(s)$. The local parent of a vertex s is the vertex that was expanded when the g-value and parent of vertex s were set. Notice that, if Lines 173-179 are removed, then the local parent and parent are always the same. The local parent path $G_p(s, parent(s))$ consists of the vertices encountered by repeatedly following the local parents from vertex s to its parent (inclusive of vertex s, but exclusive of its parent). As we saw earlier, Basic Theta* maintains two global data structures, namely the open list and closed list, which are defined exactly as they were in Section 4.3.

Basic Theta* is identical to A* except that, when it updates the g-value, parent and local parent of an unexpanded visible neighbor s' of vertex s in procedure ComputeCost it finds any-angle paths by considering both Path 1 and Path 2 (Section 4.4). Path 1: As done by A*, Basic Theta* considers the path from the start vertex to vertex s and from vertex s to vertex s' in a straight-line [Line 181]. Path 2: To allow for any-angle paths, Basic Theta* also considers the path from the start vertex to the parent of vertex s and from the parent of vertex s to vertex s' in a straight-line [Line 175]. Path 2 is guaranteed to be no longer than Path 1 due to the triangle inequality if vertex s' has line-of-sight to the parent of vertex s. Basic Theta* uses Path 2 if the parent of vertex s has line-of-sight to vertex s' [Line 173] and then sets the g-value of vertex s' to the length of Path 2, the parent of vertex s' to the parent of vertex s and the local parent of vertex s' to vertex s. Otherwise, Basic Theta* uses Path 1 and sets the g-value of vertex s' to the length of Path 1 and both the parent and local parent of vertex s' to vertex s. The key idea behind Basic Theta* and Path 2 is that, unlike A*, where the parent of a vertex must be a visible neighbor of that vertex, Basic Theta* allows the parent of a vertex to be any vertex. Thus, the parent of a vertex can be any vertex for Basic Theta* while it is a visible neighbor of the vertex for A*.

6.3.1.1 Example Trace of Basic Theta*

Figure 6.2 shows an example trace of Basic Theta*. The vertices are labeled with their g-values and parents. The arrows point to their parents. Hollow red circles indicate vertices that are being expanded, and blue arrows indicate vertices that are generated during the current expansion. First,

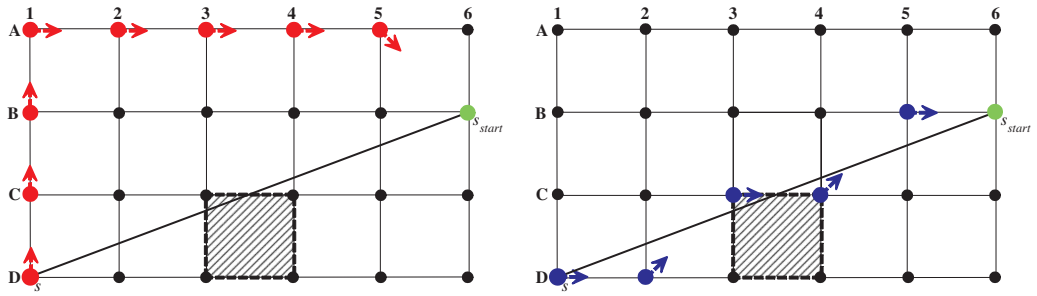


Figure 6.3: Path 2 Parent Problem

Basic Theta* expands start vertex A5 with parent A5 just like A* would do, as shown in Figure 6.2(a). Second, Basic Theta* expands vertex B4 with parent A5 as shown in Figure 6.2(b). Vertex C4 is an unexpanded visible neighbor of vertex B4 that does not have line-of-sight to vertex A5 and thus is updated according to Path 1. Basic Theta* therefore sets both its parent and local parent to vertex B4. On the other hand, vertex B3 is an unexpanded visible neighbor of vertex B4 that does have line-of-sight to vertex A5 and thus is updated according to Path 2. Basic Theta* therefore sets its parent to vertex A5 and its local parent to vertex B4.

When an agent performs planning with the freespace assumption, it must at least find a new short path each time it observes its current path to be blocked. This problem could be solved by repeatedly performing single-shot Basic Theta* searches, but this would be too slow on large grids. Alternatively, the agent could reuse information from previous searches when finding a new short path. Incremental find-path algorithms, such as Differential A* and D* Lite, extend A* to planning with the freespace assumption by efficiently reusing information from previous searches to speed up the next one. Despite the similarities between Basic Theta* and A*, these methods do not apply to Basic Theta* because they assume that the parent of a vertex in the search tree must also be its neighbor and Basic Theta* allows the parent of a vertex in the search tree to be any vertex. We refer to this as the **Path 2 Parent Problem**. Reusing information from the previous searches requires that all vertices that no longer have line-of-sight to their parent due to a newly blocked grid cell be identified. The fact that the paths found by A* are constrained to grid graph edges makes this easy because any vertex that no longer has line-of-sight to its parent must be a corner of a newly blocked grid cell. For example, assume that grid cell A3-A4-B4-B3 becomes blocked in Figure 4.2(a). Vertex B3 no longer has line-of-sight to its parent A4 which can easily be identified because vertex B3 and vertex A4 are both corners of the newly blocked grid cell. The paths found by Basic Theta* are not constrained to grid graph edges, which makes this more difficult because vertices that no longer have line-of-sight to their parents are not necessarily corners of newly blocked grid cells. For example, assume that grid cell C3-C4-D4-D3 becomes blocked in Figure 6.3(a), which is *not* an actual trace of Basic Theta*. The larger red circles and red arrows indicate the local parent path of vertex D1: G_p(D1, B6) = [D1, C1, B1, A1, A2, A3, A4, A5]. Vertex D1 no longer has line-of-sight to its parent B6, but neither it nor any vertex on its local parent path is a corner of the newly blocked grid cell.

```
185 ComputeCost(s, s')
186     if ∠(s', parent(s)) is not a multiple of 45° and LineOfSight(parent(s), s') and
        Φ(s, parent(s), s') ∈ [lb(s), ub(s)] then
187         /* Path 2 */
188         if g(parent(s)) + c(parent(s), s') < g(s') then
189             parent(s') := parent(s);
190             g(s') := g(parent(s)) + c(parent(s), s');
191             local(s') := s;
192             l := min_{s''∈nghbr4(s')} Φ(s', parent(s), s'');
193             h := max_{s''∈nghbr4(s')} Φ(s', parent(s), s'');
194             δ = Φ(s, parent(s), s');
195             lb(s') := max(l, lb(s) − δ);
196             ub(s') := min(h, ub(s) − δ);
197     else
198         /* Path 1 */
199         if g(s) + c(s, s') < g(s') then
200             parent(s') := s;
201             g(s') := g(s) + c(s, s');
202             local(s') := s;
203             lb(s') := −45°;
204             ub(s') := 45°;
```

Algorithm 11: Phi*

6.4 Phi*

Phi* is a variant of Basic Theta* that can be made incremental because it addresses the Path 2 Parent Problem by maintaining the **Local Parent Path Property**: *The local parent path of any vertex that no longer has line-of-sight to its parent after grid cells become blocked must contain some corner of a newly blocked grid cell.*[2]

Algorithm 11 shows the pseudocode for Phi*. All procedures other than procedure Compute-Cost are identical to Basic Theta* in Algorithm 10 and thus are not shown. Lines 161 and 162 are to be executed. We constructed Phi* so that we can prove the following theorem which implies the Local Parent Path Property:

Theorem 6. *The local parent path* $G_p(s, \text{parent}(s))$ *of any vertex* s *contains at least one corner of each grid cell that the path segment* $\overline{s, \text{parent}(s)}$ *traverses.*

Proof. See Appendix D. □

If the line segment from a vertex to its parent traverses a blocked grid cell, then it must touch at least one of the four grid graph edges (two horizontal and two vertical) that form the perimeter of the traversed blocked grid cell. We constructed Phi* so that the local parent path of any vertex contains at least one of the two end points of each horizontal and vertical grid graph edge that the path segment from the vertex to its parent touches. Phi* satisfies this property by maintaining an angle range for each vertex that restricts which of its unexpanded visible neighbors can be updated according to Path 2. To illustrate this property, assume that grid cell C3-C4-D4-D3 becomes blocked in Figure 6.3(b), which is *not* an actual trace of Phi*. The larger blue circles

[2] Statements like these only apply to vertices that are in the open or closed lists and only apply at particular times during the execution of Phi*, but we do not mention this in this section to improve readability. In Appendix D, these restrictions are explicitly stated and proven.

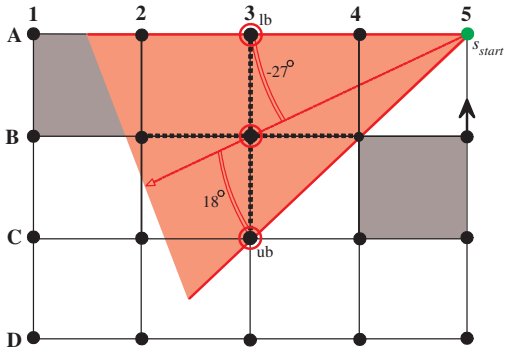


Figure 6.4: Angle Range of Phi*

and blue arrows indicate the local parent path of D1: G_p(D1, B6) = [D1, D2, C3, C4, B5]. Vertex D1 no longer has line-of-sight to its parent B6, but, unlike Figure 6.3(a), this can easily be identified because a vertex on its local parent path is a corner of the newly blocked grid cell. In fact, the local parent path of vertex D1 contains at least one of the two end points of each horizontal and vertical grid graph edge that the path segment from vertex D1 to its parent B6 touches.

6.4.1 Definition of Angle Ranges

Phi* maintains two additional values for every vertex s, namely a lower angle bound $lb(s)$ of vertex s and an upper angle bound $ub(s)$ of vertex s, that together form the angle range $[lb(s), ub(s)]$ of vertex s. We define $\Phi(s, p, s')$, which gives Phi* its name, to explain their meaning. $\Phi(s, p, s') \in [-90, 90]$, is the angle (measured in degrees) between the ray from p through vertex s' and the ray from p through vertex s. It is positive if the ray from vertex p to vertex s is clockwise from the ray from vertex p to vertex s', zero if the ray from vertex p to vertex s has the same heading as the ray from vertex p to vertex s', and negative if the ray from vertex p to vertex s is counterclockwise from the ray from vertex p to vertex s'. We compute Φ just as we computed Θ in Section 4.5.1, but we use Φ to maintain different properties which explains the different terminology. Assume that Phi* expands a vertex s and considers a Path 2 update for an unexpanded visible neighbor s' of vertex s. Phi* constrains the angle range of vertex s so that the following property holds: The local parent path of vertex s' contains at least one corner of each grid cell that the path segment $\overline{s', parent(s)}$ traverses if $\Phi(s, parent(s), s') \in [lb(s), ub(s)]$. Figure 6.4 shows an example where vertex B3 with parent A5 has angle range $[-27, 18]$. The dashed arrow within this angle range is $\Phi(B3, A5, B3) = 0$, which is the reference point of the angle range.

6.4.2 Updating Angle Ranges

When Phi* expands a vertex s, it updates the g-value, parent and local parent of each unexpanded visible neighbor s' of vertex s by considering Path 1 and Path 2. It updates the g-value, parent and local parent of vertex s' in the same way as Basic Theta*, but also updates the angle range of vertex s'.

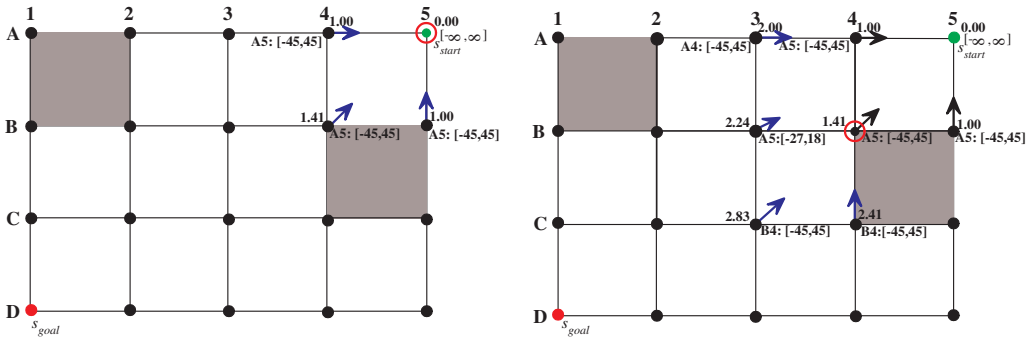
- **Path 1:** When Phi* expands vertex A5 in Figure 6.5(a), it updates its unexpanded visible neighbor B4 according to Path 1 (Line 199). Phi* sets the angle range of vertex B4 to the angle range defined by the crossbar centered at vertex B4, as follows: It computes $\Phi(s', \mathit{parent}(s), s'') = \Phi$(B4, A5, s'') for each crossbar neighbor s'' of vertex B4, namely vertices A4, B5, C4 and B3. (This is independent of the blocked grid cell B4-B5-C5-C4.) It sets the upper angle bound of vertex B4 to the maximum of the four computed angles, namely 45 degrees due to vertex B5. It sets the lower angle bound of vertex B4 to the minimum of the four computed angles, namely -45 degrees due to vertex A4. Phi* does not need to compute these angle bounds since the upper angle bound is always 45 degrees and the lower angle bound is always -45 degrees in the Path 1 case (Lines 203 and 204).

- **Path 2:** When Phi* expands vertex B4 in Figure 6.5(b), it updates its unexpanded visible neighbor B3 according to Path 2 (Line 188). Phi* then sets the angle range of vertex B3 to the intersection of the angle range defined by the crossbar centered at vertex B3 and the angle range of its local parent B4, as follows: It computes $\Phi(s', \mathit{parent}(s), s'') = \Phi$(B3, A5, s'') for each crossbar neighbor s'' of vertex B3, namely vertices A3, B4, C3 and B2 (the dotted lines in Figure 6.4 show the crossbar of vertex B3). Phi* computes the maximum of the four computed angles [Line 193], namely 18 degrees due to vertex B4 and vertex C3. It sets the upper angle bound of vertex B3 to the minimum of that angle and the upper angle bound of vertex B4 shifted so that the reference point is now vertex B3 rather than vertex B4 [Line 196], resulting in 18 degrees. Phi* also computes the minimum of the four computed angles [Line 192], namely -27 degrees due to vertex A3. It sets the lower angle bound of vertex B3 to the maximum of that angle and the lower angle bound of vertex B4 shifted so that the reference point is now vertex B3 rather than vertex B4 [Line 195], resulting in -27 degrees. Thus, it calculates the angle range of vertex s' as for Path 1, but then intersects it with the angle range of vertex s.

Assume that Phi* expands a vertex s. If it updates the unexpanded visible neighbor s' of vertex s according to Path 2, then it sets the angle range of vertex s' to the intersection of the angle range defined by the crossbar centered at vertex s' and the angle range of the local parent of vertex s'. Thus, it sets the angle range of vertex s' to the intersection of the angle ranges defined by all crossbars centered on vertices that are members of the local parent path of vertex s', including vertex s', but not the parent of vertex s'. We refer to this as the **Crossbar Property**. If Phi* updates the unexpanded visible neighbor s' of vertex s according to Path 1, then it sets the angle range of vertex s' to the angle range defined by the crossbar centered at vertex s'. Thus, the Crossbar Property holds in this case as well.

6.4.3 Using Angle Ranges

Due to the Crossbar Property any ray r within the angle range of a vertex s touches all crossbars centered on vertices that are members of the local parent path of vertex s. Therefore any unexpanded visible neighbor s' of vertex s that satisfies $\Phi(s, \mathit{parent}(s), s') \in [\mathit{lb}(s), \mathit{ub}(s)]$ [Line 186] are such that the path segment $\overline{s', \mathit{parent}(s)}$ touches all crossbars centered on vertices that are members of the local parent path of vertex s', including vertex s' (path segment $\overline{s', \mathit{parent}(s)}$ starts at vertex s', which is trivially part of the crossbar centered at vertex s'). This suggests, but



(a) (b)

Figure 6.5: Example Trace of Phi*

does not imply, that the local parent path of vertex s' contains at least one corner of each grid cell that the path segment traverses, which is how Phi* satisfies Theorem 6.

6.4.4 Tie Breaking and the Triangle Inequality

When Phi* expands a vertex s, it updates each unexpanded visible neighbor s' of vertex s by considering Path 1 and Path 2. $\angle(s, p)$ is the smaller angle formed by the ray from p through vertex s and the vertical line through p. Path 2 is no longer than Path 1 due to the triangle inequality. They are equally long if $\angle(s', \mathit{parent}(s))$ is a multiple of 45 degrees, that is, if the vertices lie on a horizontal, vertical or 45 degree line. In this case, it is better to update vertex s' according to Path 1, which explains the first condition on Line 186. The reason is that an update according to Path 1 results in fewer floating point computations (because angle ranges do not need to be computed) and can slightly reduce path lengths (as described in Section 4.8.2.1).[3]

6.4.5 Example Trace of Phi*

Figure 6.5 shows an example trace of Phi* using the find-path problem from Figure 6.2. The labels of the vertices now include the angle ranges.

Figure 6.6 provides an updated summary of our classification of different find-path algorithms. It is an updated version of Figure 5.6 which accounts for the contributions made so far in this chapter. Figures 6.6 and 5.6 are annotated the same way. The Phi* oval is within 7 rounded rectangles, and thus Phi* is a **1** single-shot, **2** informed, **3** any-angle **4** find-path algorithm that **5** uses angle ranges and **6** eagerly performs **7** line-of-sight checks. Furthermore, the Phi* oval is opaque. Like AP Theta*, the angle ranges computed by Phi* require that the search be performed on an octile graph and thus it cannot be used to search any Euclidean graph (Generality Property). Finally, notice that Phi* both uses angles ranges, like AP Theta*, and eagerly performs line-of-sight checks, like Basic Theta*.

[3]This also simplifies the proof of Theorem 6.

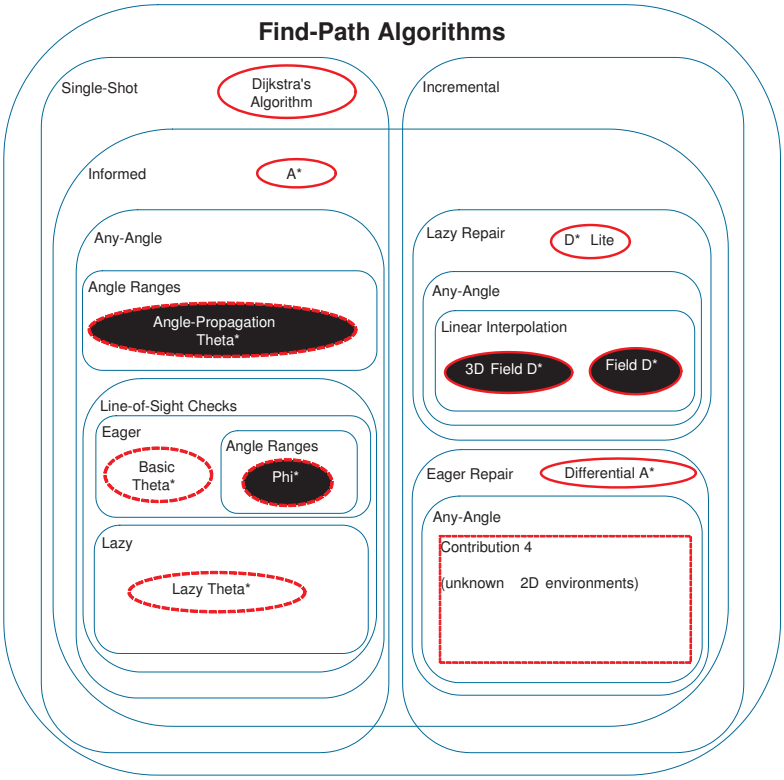


Figure 6.6: Classification of Find-Path Algorithms

6.5 Incremental Phi*

Incremental Phi* is an incremental variant of Phi* that re-computes any-angle paths faster than repeated single-shot Basic Theta* searches when grid cells become blocked as required when planning with the freespace assumption.

Algorithm 12 shows the pseudocode for Incremental Phi*. All procedures other than procedures Main and ClearSubtree are identical to Phi* in Algorithm 11 and thus are not shown. Lines 161 and 162 are to be executed. Incremental Phi* maintains two global data structures in addition to those maintained by Phi* (and Basic Theta*):

- The under list is a FIFO queue. In the pseudocode, *under*.*Enqueue*(s) inserts vertex s at the end of the FIFO queue *under* and *under*.*Dequeue* removes a vertex from the front of the FIFO queue *under*.
- The over list is a FIFO queue similar to the under list.

Incremental Phi* is complete and correct (as proved in Appendix D), but can find paths that are longer than those found by repeated single-shot Basic Theta* or Phi* searches. We constructed Phi* so that all vertices that no longer have line-of-sight to their parents due to newly blocked grid cells can be identified and removed quickly. Incremental Phi* uses a pre-processing technique that does this in a way that is similar in spirit to the pre-processing technique used by Differential A* [Lines 209-214]. We constructed this pre-processing technique so that we

```
205 Main()
206     Initialize();
207     while true do
208         ComputeShortestPath();
209         Wait for grid cells to become blocked;
210         foreach newly blocked grid cell c do
211             Update blockage status of grid cell c to blocked;
212             foreach s' ∈ corners(c) do
213                 if (s' ∈ closed or s' ∈ open) and s' ≠ s_start then
214                     ClearSubtree(s');

215 ClearSubtree(s)
216     under := over := ∅;
217     under.Enqueue(s);
218     while under ≠ ∅ do
219         u := under.Dequeue();
220         over.Enqueue(u);
221         InitializeVertex(u);
222         if u ∈ open then
223             open.Remove(u);
224         if u ∈ closed then
225             closed := closed \ {u};
226         foreach s' ∈ nghbr_8(u) do
227             if local(s') = u then
228                 under.Enqueue(s');
229     while over ≠ ∅ do
230         v := over.Dequeue();
231         foreach s' ∈ nghbr_vis(v) do
232             if s' ∈ closed then
233                 UpdateVertex(s', v);
```

Algorithm 12: Incremental Phi*

can prove the **Line-of-Sight Property**: *Any vertex $s \in V$ that is in the open or closed lists has line-of-sight to its parent each time procedure ComputeShortestPath is called on Line 208.*

If any vertex in the open or closed lists has line-of-sight to its parent immediately prior to calling procedure ComputeShortestPath then every vertex that is in the open or closed lists has line-of-sight to its parent immediately afterwards as well. If grid cells become blocked [Line 211], Incremental Phi* uses the pre-processing technique to maintain the Line-of-Sight Property prior to the next call to procedure ComputeShortestPath. The pre-processing technique uses the Local Parent Path Property to identify all vertices that no longer have line-of-sight to their parent. This can be done easily because the Local Parent Path Property ensures that the local parent path of any vertex that no longer has line-of-sight to its parent after grid cells become blocked must contain some corner of a newly blocked grid cell. The pre-processing technique thus calls procedure ClearSubtree on each corner of each newly blocked grid cell that is in the open or closed lists and not equal to the start vertex [Lines 210 and Line 212]. Each call to procedure ClearSubtree performs a breadth-first search that follows the local parents backwards. It removes all encountered vertices from both the open and closed lists and re-initializes their values as if they had never been visited by procedure ComputeShortestPath [Lines 221-225]. It then updates the open list by re-inserting all vertices with a visible neighbor in the closed list back into the

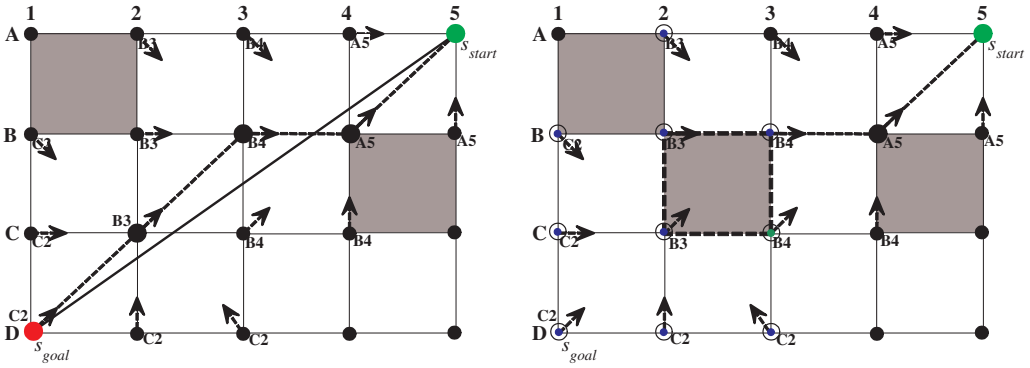


Figure 6.7: Example Trace of Incremental Phi*

open list [Lines 229-233]. The next call to procedure ComputeShortestPath is then a standard Phi* search that starts with a non-empty search tree and thus it reuses information from previous searches. The next search operates like a standard Phi* search because the Line-of-Sight Property ensures that every vertex in the search tree has line-of-sight to its parent and Lines 229-233 re-creates the fringe of the search tree (that is, all of the visible neighbors of a vertex in the closed list are in the open or closed lists). Notice that procedure ClearSubtree is fast because it uses FIFO queues.

6.5.1 Example Trace of Incremental Phi*

Figure 6.7 shows an example trace of Incremental Phi* using the find-path problem from Figure 6.2. The vertices are labeled with *only* their local parents. The arrows point to their local parents. Figure 6.7(a) shows a snap shot of Incremental Phi* at the conclusion of the first call to procedure ComputeShortestPath. The larger circles indicate vertices that are in the closed list at the conclusion of the first call to procedure ComputeShortestPath. Assume that grid cell B2-B3-C3-C2 becomes blocked in Figure 6.7(b). The pre-processing technique then iterates over the four corners of the newly blocked grid cell, calling procedure ClearSubtree on each corner that is in the open or closed lists and not equal to the start vertex [Line 213]. Assume that procedure ClearSubtree is called on vertices B3, C3, C2 and B2 in that order. When procedure ClearSubtree is called on vertex B3, the hollow blue vertices B1, C1, D1, A2, B2, C2, D2, B3 and D3 are re-initialized after which vertex B3 is re-inserted into the open list with parent A5 and local parent B4. When procedure ClearSubtree is called on vertex C3, the hollow green vertex C3 is re-initialized and re-inserted into the open list with parent and local parent B4. Procedure ClearSubtree is not called on vertices C2 and B2 as neither corner is in the open or closed lists after procedure ClearSubtree is called on vertices B3 and C3.

6.5.2 Moving Agent

Incremental Phi* needs to handle a moving agent (that is, the start vertex of the search is different from search to search) to be applicable to planning with the freespace assumption. Similar to D* (Stentz, 1995) and D* Lite, we let Incremental Phi* search from the goal vertex to the current

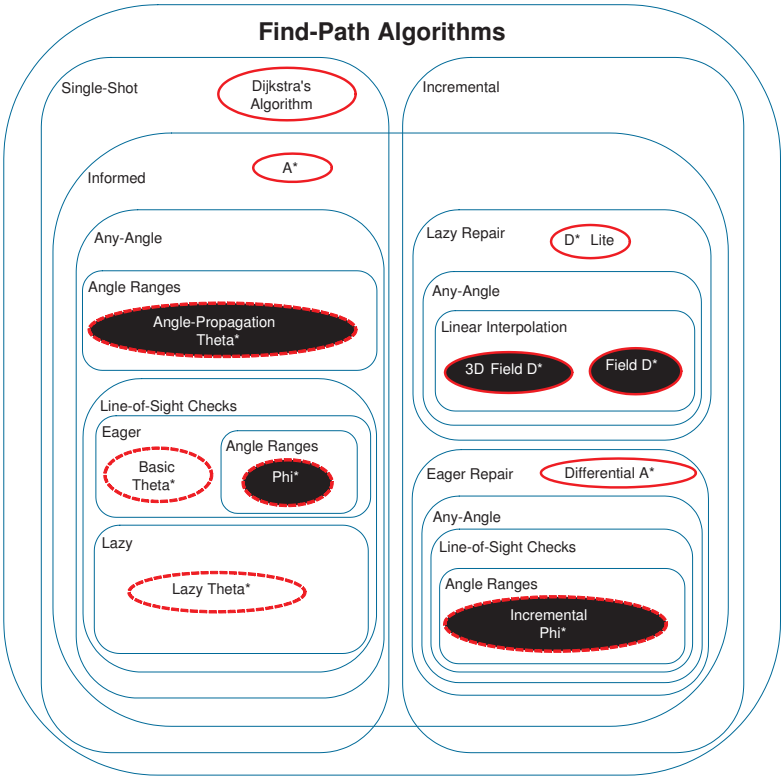


Figure 6.8: Classification of Find-Path Algorithms

vertex of the agent. The h-values then correspond to an approximation of the distance from a vertex to the current vertex of the agent. The keys of the vertices in the open list are thus no longer valid when the agent moves. Incremental Phi* uses the heap re-ordering technique of D* and D* Lite instead of repeatedly re-ordering the priority queue by recomputing all keys.

Figure 6.8 provides an updated summary of our classification of different find-path algorithms. It is an updated version of Figure 6.6 which accounts for the contributions made in this chapter. Figures 6.8 and 6.6 are annotated the same way. The Incremental Phi* oval is within 7 rounded rectangles, and thus Incremental Phi* is an **1** incremental, **2** informed, **3** any-angle **4** find-path algorithm that uses **5** angle ranges, **6** eagerly performs line-of-sight checks and **7** eagerly repairs the search tree between searches. Like AP Theta*, the angle ranges computed by Incremental Phi* require that the search be performed on an octile graph and thus it cannot be used to search any Euclidean graph, which is why its oval is opaque (Generality Property). Finally, notice that Phi* and Incremental Phi* both use angles ranges, like AP Theta* and eagerly perform line-of-sight checks, like Basic Theta*.

6.6 Experimental Results

In this section, we compare Phi*, Incremental Phi* and Basic Theta* with respect to their path length, number of vertex expansions and runtime.

		Path Lengths	Vertex Expansions	Runtimes
100×100	Random Grids 0%	1.0000	1.6138	1.2994
	Random Grids 5%	0.9999	1.0847	0.8944
	Random Grids 10%	0.9999	1.0466	0.7195
	Random Grids 20%	0.9998	1.0101	0.7296
500×500	Random Grids 0%	1.0000	3.4788	2.2412
	Random Grids 5%	1.0000	1.1923	0.9900
	Random Grids 10%	1.0002	1.1155	0.9167
	Random Grids 20%	1.0004	1.0669	0.9012

Table 6.1: Basic Theta* Versus Phi* on Random Grids

We compare Phi* and Basic Theta* on 100×100 and 500×500 octile graphs with different percentages of randomly blocked grid cells (random grids). We use planning with the freespace assumption to compare Incremental Phi* and Basic Theta* on random 100×100, 250×250 and 500×500 grids and scaled indoor and outdoor maps from robotics and the real-time strategy game Baldur's Gate II (Bulitko et al., 2005) (non-random grids).[†] Figure 4.18 shows an example of a map from Baldur's Gate II. For random grids, the start vertex is in the lower left corner of the square grid and the goal vertex is in the corner of a grid cell randomly chosen from the right most column of grid cells in the square grid. Grid cells are blocked randomly, but a one-unit border of grid cells around the square grid is left unblocked in order to guarantee that there is path from the start vertex to the goal vertex. An illustration of the experimental setup for random grids can be seen in Figure 4.20. For non-random grids, the start and goal vertices are randomly chosen from the corners of unblocked grid cells with the constraint that the distance between them is at least 250. We average over 500 path-planning problems for random grids and 650 path-planning problems for non-random grids (50 path-planning problems were run on each non-random grid of which there were 12 game maps and one robotics map).[4]

All algorithms use the Euclidean straight-line distances as h-values and break ties among vertices with the same key in favor of the vertex with the smaller g-value (when deciding which vertex to expand next) for the reasons explained in Section 4.6.

6.6.1 Phi*

Table 6.1 compares Phi* and Basic Theta* for single-shot searches on random 100×100 grids and random 500×500 grids, where both find-path algorithms search from the start vertex to the goal vertex. Table 6.1 reports ratios such that values larger than 1 imply that Basic Theta* did worse than Phi*. Phi* finds paths of essentially the same length as Basic Theta*, which is important because we showed in Section 4.6 that Basic Theta* typically finds shorter paths than A*, A* PS and, more importantly, Field D*. Phi* runs faster than Basic Theta* on octile graphs with 0 percent blocked grid cells and slightly slower on octile graphs with larger percentages of blocked grid cells, for the following reason: Phi* expands far fewer vertices than Basic Theta* on octile graphs with 0 percent blocked grid cells because it updates fewer vertices according to

[†]We would like to thank Willow Garage for making the robotics maps available to us.

[4]12 game maps that contained a significant number of obstacles were selected from the 118 game maps used in Section 4.6 to ensure that the incremental find-path algorithms were forced to find new paths frequently.

		Path Lengths	Vertex Expansions	Runtimes
100×100	Random Grids 0%	0.9902	3.2551	1.2870
	Random Grids 5%	0.9925	4.4492	1.5056
	Random Grids 10%	0.9947	5.1763	1.7062
	Random Grids 20%	0.9995	5.7573	1.9680
250×250	Random Grids 0%	0.9902	7.8718	3.3653
	Random Grids 5%	0.9922	11.5943	3.9075
	Random Grids 10%	0.9937	13.7031	4.5855
	Random Grids 20%	0.9970	15.4465	5.6277
500×500	Random Grids 0%	0.9902	15.3940	7.0040
	Random Grids 5%	0.9919	25.7134	8.0542
	Random Grids 10%	0.9944	31.1268	10.0422
	Random Grids 20%	0.9971	33.8728	11.7307

Table 6.2: Square Grid Sizes and Percentages of Blocked Grid Cells on Random Grids

	Path Lengths	Vertex Expansions	Runtimes
Sensor Radius 5	0.9954	31.9153	9.6181
Sensor Radius 10	0.9961	28.3299	8.1223
Sensor Radius 20	0.9966	19.8923	5.5846

Table 6.3: Sensor Radius on Random 500×500 Grids

Path Length	Vertex Expansions	Runtimes
1.0037	25.1068	12.0073

Table 6.4: Non-Random 500×500 Grids

Path 2 than Basic Theta* due to its additional angle range constraint. When Phi* updates a vertex according to Path 1 and Basic Theta* updates that same vertex according to Path 2, Phi* inserts it into the open list with a larger key than Basic Theta*. Thus, Phi* is more likely to finish the search without expanding that vertex than Basic Theta*. Phi* expands more of these vertices as the percentage of blocked grid cells increases.

6.6.2 Incremental Phi*

Tables 6.2-6.4 compare Incremental Phi* and Basic Theta* for planning with the freespace assumption. Tables 6.2-6.4 report ratios such that values larger than 1 imply that Basic Theta* did worse than Incremental Phi*. For these experiments we maintain two square grids: one represents the traversable space in the continuous environment (terrain grid) and the other represents the knowledge that the agent has of the traversable space in the continuous environment (knowledge grid). The agent finds a short path from its current vertex, initially the start vertex, to the goal vertex in its knowledge grid and then repeatedly moves one unit along this path. After each move, it scans all grid cells within a given sensor radius around its current coordinate and updates the scanned grid cells in its knowledge grid to match the blockage status of those same grid cells

in the terrain grid. The agent then finds a new short path from its current vertex to the goal vertex in its knowledge grid and repeats this process until it reaches the goal vertex. Unblocked grid cells in the knowledge grid can become blocked but not vice versa. For random grids, the initial knowledge grid contained a given percentage of randomly blocked grid cells and the terrain grid corresponded to the knowledge grid except that 20 percent of randomly chosen grid cells were blocked in addition to the blocked grid cells in the knowledge grid. For non-random grids, the initial knowledge grid contained only unblocked grid cells and the terrain grid corresponded to the given map. Incremental Phi* operated as described earlier. Basic Theta* always searched from the start vertex to the goal vertex if the current path in the knowledge grid became blocked. The number of vertex expansions includes all vertices that were removed from the open list and updated their unexpanded visible neighbors. The runtime includes the entire time from the start of the search until the agent reached the goal vertex. Table 6.2 reports on random grids with a sensor radius of 3 units. As the grid size and the percentage of blocked grid cells in the knowledge grid increases, Incremental Phi* expands fewer vertices than Basic Theta* and its speedup relative to Basic Theta* increases. Table 6.3 reports on random 500×500 grids with 10 percent blocked grid cells in the knowledge grid. As the sensor radius decreases, Incremental Phi* expands fewer vertices than Basic Theta* and its speedup relative to Basic Theta* increases. Table 6.4 reports on non-random 500×500 grids and a sensor radius of 3 units, where the find-path algorithms re-plan more frequently than on random grids. Across all tables, Incremental Phi* finds paths of essentially the same lengths as Basic Theta* (although both find-path algorithms search in opposite directions and thus the paths of the agent quickly diverge). However, Incremental Phi* does so up to 12 times faster than Basic Theta*. In other words, Incremental Phi* provides a nearly dominating tradeoff, relative to Basic Theta*, with respect to the runtime of the search and the length of the resulting path.

6.7 Conclusions

In this chapter, we examined path planning in unknown 2D environments. Specifically, we examined planning with the freespace assumption in unknown 2D environments that have been discretized into octile graphs. We showed that path planning in unknown 2D environments is more difficult than path planning in known 2D environments and thus it is typically solved using incremental find-path algorithms. We introduced Incremental Phi*, which is an incremental variant of Basic Theta*. In order to do that we first introduced Phi*, a variant of Basic Theta* that can be made incremental, and then extended it to Incremental Phi*. This was non trivial due to the Path 2 Parent problem. Incremental find-path algorithms are more difficult to implement and understand than single-shot find-path algorithms. However, the pseudocode of Phi* is very similar to the pseudocode of Basic Theta* and thus Phi* and Incremental Phi* are simple to implement. Phi* and Incremental Phi* take advantage of Path 2, the triangle inequality and the Line-of-Sight and Local Parent Path Properties. While the Line-of-Sight and Local Parent Path Properties make Phi* and Incremental Phi* more difficult to understand than Basic Theta* it is no more difficult to understand than the properties and invariants behind Differential A* and D* Lite (Simplicity Property). We demonstrated that Incremental Phi* finds paths of essentially the same lengths as Basic Theta* (which we have already shown to provide a good tradeoff with respect to the runtime of the search and the length of the resulting path on octile graphs) and that it can

provide a speedup of approximately one order of magnitude when planning with the freespace assumption (Efficiency Property).

To summarize, in this chapter we validated Hypothesis 2 in unknown 2D environments by introducing Incremental Phi*, a new any-angle find-path algorithm that satisfies the Simplicity and Efficiency Properties for path planning in unknown 2D environments. Simplicity Property: Incremental Phi* extends Phi*, which is similar to Basic Theta*, but slightly more difficult to implement and understand. However, it is no more difficult to implement and understand than other incremental find-path algorithms (Section 6.4). Efficiency Property: Incremental Phi* provides a good tradeoff with respect to the runtime of the search and the length of the resulting path (Section 6.6). Generality Property: Incremental Phi* cannot be used to search any Euclidean graph. While Incremental Phi* takes advantage of Path 2 like Basic Theta* and Lazy Theta*, it also relies on properties that are specific to octile graphs (Local Parent Path Property).

Chapter 7

Conclusions

Navigating an agent from a given start coordinate to a given goal coordinate through a continuous environment is one of the most important problems faced by roboticists and video game developers. A key part of navigation is path planning. Path planning is typically composed of two parts: the generate-graph problem, which is solved by discretizing a continuous environment into a graph and the find-path problem, which is solved by searching this graph for a path from a given start vertex to a given goal vertex. Traditional find-path algorithms have the following desirable properties: Simplicity Property: They are simple to implement and understand. Efficiency Property: They provide a good tradeoff with respect to the runtime of the search and the length of the resulting path. Generality Property: They can be used to search any Euclidean graph. However, these desirable properties come with a penalty because traditional edge constrained find-path algorithms find edge-constrained paths which are not equivalent to true shortest paths. While this fact is well known by roboticists and video game developers two important questions remain: Question 1: How much longer can the paths found by traditional edge-constrained find-path algorithms be than the true shortest paths? Question 2: Can more sophisticated find-path algorithms be developed that find shorter and more realistic looking paths than traditional edge-constrained find-path algorithms, while maintaining the desirable properties of traditional edge-constrained find-path algorithms?

This dissertation addressed these two questions by introducing four contributions which validate the following two hypotheses:

- *Hypothesis 1: Analytical bounds can be introduced which compare the lengths of the paths found by traditional edge-constrained find-path algorithms on certain types of graphs with the lengths of the true shortest paths.*

Contribution 1: In Chapter 3, we examined the lengths of the paths found by traditional edge-constrained find-paths algorithms on grid graphs constructed from 2D and 3D regular grids. Since traditional edge-constrained find-paths algorithms can find shortest grid paths we compared the lengths of shortest grid paths formed by the edges of grid graphs constructed from 2D and 3D regular grids with the lengths of the true shortest paths. We used a simple unified proof structure to compare the lengths of the shortest paths formed by the edges of grid graphs constructed from both 2D and 3D regular grids with the lengths of the true shortest paths. First, we examined regular grids that are used to discretize a continuous *2D* environment. We performed a comprehensive path length analysis on the lengths of the shortest grid paths formed by the edges of grid graphs

constructed from all three types of regular grids that can be used to tessellate continuous 2D environments, namely triangular, square and hexagonal grids. We showed that shortest grid paths can be longer than true shortest paths as follows: 100% for 3-neighbor triangular grid graphs, $\approx 41\%$ for 4-neighbor square grid graphs, $\approx 15\%$ for 6-neighbor hexagonal grid graphs, $\approx 15\%$ for 6-neighbor triangular grid graphs, $\approx 8\%$ for 8-neighbor square grid graphs and $\approx 4\%$ for 12-neighbor hexagonal grid graphs. Second, we examined regular grids that are used to discretize a continuous *3D* environment. We performed a comprehensive path length analysis on the lengths of shortest grid paths formed by the edges of grid graphs constructed from the only type of regular grid that can be used to tessellate continuous 3D environments, namely a cubic grid. We showed that shortest grid paths can be longer than true shortest paths as follows: $\approx 73\%$ for 6-neighbor cubic grid graphs and $\approx 13\%$ for 26-neighbor cubic grid graphs.

Therefore, these results validate Hypothesis 1. We now examine potential future work that relates to Hypothesis 1. Future work should be focused on applying the proof structure introduced in this dissertation to different types of graphs to increase the number of analytical bounds that compare the lengths of the paths found by traditional edge-constrained find path algorithms with the lengths of the true shortest paths. For example, analytical bounds should be developed that compare the lengths of true shortest paths with the lengths of shortest paths formed by the edges of grid graphs constructed from Tex grids (Yap, 2002; Björnsson et al., 2003), which combine square grids (which are easy to implement) and hexagonal grids (which can be searched more efficiently), or regular grids with different branching factors (for example, 16-neighbor square grid graphs). The proof structure introduced in Chapter 3 (that is, Parts 1-3 of Lemma 1) can likely be extended to these types of graphs. However, Parts 1-3 of these new proofs would likely not be as elegant because they would require a more case based approach. For example, the angles of the edge types of the outgoing edges of a vertex in a 16-neighbor square grid graph, such as the one depicted in Figure 4.28(c), are not evenly distributed between $(0, 2\pi]$. Therefore, Part 1 of Lemma 1, which currently uses this property and reflection to consider only a single case (that is, a line segment between a single pair of adjacent edge types), would have to analyze more cases (that is, line segments between several different pairs of adjacent edges types).

- *Hypothesis 2: A new class of any-angle find-path algorithms, that propagate information along graph edges, without constraining paths to be formed by graph edges, can be used to quickly find paths that are shorter than the paths found by traditional edge-constrained find-path algorithms, while maintaining the Simplicity and Generality Properties of traditional edge-constrained find-path algorithms.*

Robotics and video game applications require that agents find paths in many different types of continuous environments. We examined known 2D environments, known 3D environments and unknown 2D environments. All three types of continuous environments are widely used by roboticists and video game developers and thus it is important to evaluate find-path algorithms in all three types of environments. Known 2D environments are the simplest type of environment in which path planning is performed. Known 3D environments and unknown 2D environments make path planning more difficult. We introduced a new class of any-angle find-path algorithms that propagate information along graph edges (to achieve a short runtime) without constraining paths to be formed by graph edges (to find any-angle paths). We introduced new members to this class and evaluated each member in one of the three types of continuous environments.

Contribution 2: In Chapter 4, we examined path planning in the simplest type of continuous environments, namely known 2D environments. We demonstrated that an any-angle find-path algorithm can satisfy the Simplicity, Efficiency and Generality Properties when used in known 2D environments. To that end, we introduced Basic Theta*, a new any-angle find-path algorithm. Basic Theta* is an any-angle variant of A*, that is, it propagates information along graph edges (to achieve a short runtime), without constraining paths to be formed by graph edges (to find any-angle paths). Basic Theta* does this by intelligently using line-of-sight-checks to consider shortcuts that take advantage of the triangle inequality during an A* search. Basic Theta* is extremely similar to A*, and thus it is as simple to implement and understand as A* (Simplicity Property). We evaluated Basic Theta* on 8-neighbor square grid graphs which are widely used by roboticists and video game developers when path planning in known 2D environments. We compared Basic Theta* experimentally with A*, A* with a post-processing technique and Field D* (the current state-of-the-art any-angle find-path algorithm). We showed that Basic Theta* provides a dominating tradeoff over Field D* and A* with a post-processing technique with respect to the runtime of the search and the length of the resulting path. We showed that Basic Theta* provides a good tradeoff with respect to the runtime of the search and the length of the resulting path. The runtime of Basic Theta* is similar to that of A*, but Basic Theta* finds paths which are $\approx 4 - 5\%$ shorter than the paths found by A* on average (Efficiency Property).[1] The lengths of the paths found by Basic Theta* are nearly identical to the lengths of the true shortest paths. We used metrics which correlate with realism to show that the paths found by Basic Theta* are more realistic looking than those found by Field D*. The key ideas behind Basic Theta* depend only on the triangle inequality and, since the triangle inequality is guaranteed to hold in any Euclidean environment, Basic Theta* can be used to search any Euclidean graph (Generality Property). The worst-case complexity of Basic Theta* is greater than that of A* because the runtime of each line-of-sight check can be linear in the number of vertices. We introduced Angle-Propagation Theta*, which reduces the worst-case complexity of Basic Theta* so that it is the same as that of A*. Angle-Propagation Theta* does this by intelligently using angle ranges (instead of line-of-sight checks) to consider shortcuts that take advantage of the triangle inequality during an A* search. Angle-Propagation Theta* is similar to Basic Theta* and has a better worst-case complexity, but is more difficult to implement and understand, is not as fast, finds slightly longer paths and cannot be used to search any Euclidean graph.

Contribution 3: In Chapter 5, we examined path planning in known 3D environments. We demonstrated that an any-angle find-path algorithm can satisfy the Simplicity, Efficiency and Generality Properties when used in known 3D environments. Path planning in known 3D environments is more difficult than path planning in known 2D environments. Basic Theta* can be applied to known 3D environments without any changes to the pseudocode. However, Basic Theta* is less efficient in known 3D environments than it is in known 2D environments. Thus, in order for an any-angle find-path algorithm to satisfy the Efficiency Property in known 3D environments, it must be smarter about when it performs line-of-sight checks. To that end, we introduced Lazy Theta*, a new any-angle find-path algorithm that is a more efficient variant of Basic Theta* designed for known 3D environments. Lazy Theta* is similar to Basic Theta* and thus is simple to implement and understand (Simplicity Property). We evaluated Basic Theta* and Lazy Theta* on 26-neighbor cubic grid graphs which are widely used by roboticists and video

[1] In our experiments, A* is guaranteed to find shortest grid paths and thus the paths found by Basic Theta* are $\approx 4 - 5\%$ shorter than shortest grid paths on average.

game developers when path planning in known 3D environments. We compared Lazy Theta* experimentally with A*, A* with a post-processing technique and Basic Theta*. We showed that Lazy Theta* provides a nearly dominating tradeoff over Basic Theta* with respect to the runtime of the search and the length of the resulting path. Lazy Theta* is more efficient than Basic Theta* because it frequently performs more than one order of magnitude fewer line-of-sight checks than Basic Theta* while finding paths whose lengths are nearly identical to the lengths of the paths found by Basic Theta*. We showed that Lazy Theta* and Basic Theta* both find paths that are $\approx 7-8\%$ shorter than the paths found by A* on average.[2] We showed that Lazy Theta* and Basic Theta* provide a dominating tradeoff over A* with a post-processing technique with respect to the runtime of the search and the length of the resulting path. We showed that, in certain types of known 3D environments, Lazy Theta* with optimizations can provide a dominating tradeoff over A* with a post-processing technique and Basic Theta* with respect to the runtime of the search and the length of the resulting path. Finally, we showed that, in certain types of known 3D environments, Lazy Theta* with optimizations can finds paths that are $\approx 15\%$ shorter than the paths found by A* on average, but with a similar runtime (Efficiency Property). Like Basic Theta*, the key ideas behind Lazy Theta* depend only on the triangle inequality and, since the triangle inequality is guaranteed to hold in any Euclidean environment, Lazy Theta* can be used to search any Euclidean graph (Generality Property).

Contribution 4: In Chapter 6, we examined path planning in unknown 2D environments. We demonstrated that an any-angle find-path algorithm can satisfy the Simplicity and Efficiency Properties when used in unknown 2D environments. Path planning in unknown 2D environments is more difficult than path planning in known 2D environments because an agent's knowledge of the traversable space in the continuous environment can change as it navigates towards a given goal coordinate. We handled this uncertainty by using the freespace assumption, that is, an agent assumes that it is able to traverse areas that are unknown to it. Incremental find-path algorithms are a class of algorithms which are commonly used when path planning with the freespace assumption because they efficiently solve a series of similar find-path problems by reusing information from the previous find-path problems to find a solution to the next find-path problem more quickly. To that end, we introduced Incremental Phi*, a new incremental any-angle find-path algorithm. Incremental Phi* is a new incremental find-path algorithm that is a more efficient variant of Basic Theta* designed for unknown 2D environments. Incremental Phi* is similar to both Basic Theta* and existing incremental find-path algorithms and thus is simple to implement and understand (Simplicity Property). However, incremental find-path algorithms, including Incremental Phi*, are more difficult to implement and understand than traditional edge-constrained find-path algorithms because they solve a more difficult problem. We evaluated Incremental Phi* on 8-neighbor square grid graphs using the freespace assumption because this experimental setup is widely used by roboticists when path planning in unknown 2D environments. We compared Incremental Phi* experimentally with Basic Theta*. We showed that Incremental Phi* provides a nearly dominating tradeoff over repeated Basic Theta* searches with respect to the runtime of the search and the length of the resulting path. Incremental Phi* solves a series of similar find-path problems one order of magnitude faster than repeated Basic Theta* searches from scratch by reusing information from previous find-path problems to speed up the next one and it does so while finding paths with nearly identical lengths (Efficiency Property). Making a variant of

[2] In our experiments, A* is guaranteed to find shortest grid paths and thus the paths found by Basic Theta* are $\approx 7-8\%$ shorter than shortest grid paths on average.

Basic Theta* incremental was non-trivial in that any-angle find-path algorithms do not conform to the standard assumption of incremental find-path algorithms, namely that the parent of a vertex in the search tree must also be its neighbor. Due to the lack of this property, Incremental Phi* requires that graphs have additional properties other than just being Euclidean. In other words, Incremental Phi* does not satisfy the Generality Property.

Therefore, Contributions 2-4 validate Hypothesis 2. We now examine potential future work that relates to Hypothesis 2. Future work should be focused on developing new any-angle find-path algorithms that expand the number of different robotics and video game applications that can use any-angle find-path algorithms. For example, there are robotics and video game applications that perform path planning in *unknown 3D environments*. In order for any-angle find-path algorithms to be able to find paths on graphs constructed from different discretizations of a continuous 3D environment, an incremental any-angle find-path algorithm must be able to search any Euclidean graph. This is non-trivial because the Path 2 Parent problem (Section 6.3.1.1) must be solved in a manner that does not take advantage of properties that are unique to a particular generate-graph technique. Both existing techniques for solving this problem use either linear interpolation (Field D*) or angle ranges (Incremental Phi*), which require that the search be performed on a particular type of Euclidean graph.

In general, Figure 6.8 can be used to help guide which find-path algorithms should be developed in the future because it provides a summary of the find-path algorithms that were introduced in this dissertation and how they relate to existing find-path algorithms. This can be done by examining the rounded rectangles, each of which represents a useful class of find-path algorithms, and the types of ovals that do or do not reside in them. For example:

- There are no *transparent* ovals that are within both an incremental rounded rectangle and an any-angle rounded rectangle. This means that no incremental any-angle find-path algorithms exist that can be used to search any Euclidean graph.

- There are no ovals with *dashed borders* that are within an incremental rounded rectangle, an any-angle rounded rectangle and a lazy repair rounded rectangle. No such algorithm exists because developing an incremental any-angle find-path variant of Basic Theta* that can lazily repair the search tree is non-trivial for two reasons: the Path 2 Parent problem, and the fact that the f-values in variants of Basic Theta* are not necessarily monotonically non-decreasing. The latter problem adds additional difficulty because algorithms, such as D* Lite, which lazily repair the search tree, use this property to determine which parts of the search tree need to be repaired. Such an algorithm would allow any-angle find-path algorithms to be used in a broader range of robotics and video game applications because lazy incremental find-path algorithms can be more efficient than eager incremental find-path algorithms (Koenig, Likhachev, & Sun, 2007).

Furthermore, there is likely room for new rounded rectangles in Figure 6.8. For example, more efficient variants of Basic Theta* that perform constant time line-of-sight checks would be useful. Angle-Propagation Theta* uses angle ranges to perform constant time line-of-sight checks. However, the trigonometric and floating point computations required to maintain these angle ranges are slow. Thus, a new technique for performing constant time line-of-sight checks would likely require a fundamentally different technique and thus a new class of any-angle find-path algorithms.

Prior to this dissertation, roboticists and video game developers typically solved the find-path problem using traditional edge-constrained find-path algorithms. These algorithms satisfy the Simplicity, Efficiency and Generality Properties, but they find paths that are long and unrealistic looking because they are artificially constrained to be formed by graph edges. In this dissertation, we developed an elegant new technique for determining how much longer these paths can be than true shortest paths and a new class of any-angle find-path algorithms that find shorter and more realistic looking paths. Any-angle find-path algorithms are identical to traditional edge-constrained find-path algorithms except for a small snippet of pseudocode. This snippet of pseudocode is simple to implement and understand and yet it allows any-angle find-path algorithms to quickly find paths on any Euclidean graph that are nearly the same lengths as true shortest paths. Therefore, this dissertation provides several useful tools that allow roboticists and video game developers to develop better path-planning systems. In Appendix E, we examine how the contributions made in this dissertation have already made an impact on path-planning research.

Bibliography

Aurenhammer, F. (1991). Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys*, *23*(3), 345–405.

Bekris, K. (2010) Personal communication.

Björnsson, Y., Enzenberger, M., Holte, R., Schaeffer, J., & Yap, P. (2003). Comparison of different grid abstractions for pathfinding on maps. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Board, B., & Ducker, M. (2002). Area navigation: Expanding the path finding paradigm. In Tregua, D. (Ed.), *Game Programming Gems 3*, pp. 240–256. Charles River Media.

Bohlin, R., & Kavraki, L. (2000). Path planning using lazy PRM. In *Proceedings of the IEEE Transactions on Robotics and Automation*.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Journal of Artificial Intelligence*, *129*, 5–33.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development*, *1*, 1–22.

Bresenham, J. (1965). Algorithm for computer control of a digital plotter. *IBM Systems Journal*, *4*, 25–30.

Bulitko, V., Sturtevant, N., & Kazakevich, M. (2005). Speeding up learning in real-time search via automatic state abstraction. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Bulitko, V., Luštrek, M., Schaeffer, J., Björnsson, Y., & Sigmundarson, S. (2008). Dynamic control in real-time heuristic search. *Journal of Artificial Intelligence Research*, *32*, 419–452.

Canny, J. (1988). *The complexity of robot motion planning*. MIT Press.

Canny, J., & Reif, J. (1987). New lower bound techniques for robot motion planning problems. In *Proceedings of the Symposium on the Foundations of Computer Science*.

Carsten, J., Ferguson, D., & Stentz, A. (2006). 3D Field D*: Improved path planning and replanning in three dimensions. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*.

Carsten, J., Rankin, A., Ferguson, D., & Stentz, A. (2009). Global planning on the Mars exploration rovers: Software integration and surface testing. *Journal of Field Robotics*, *26*, 337–357.

Champandard, A. (2010) Personal communication.

Chazelle, B., & Dobkin, D. (1979). Decomposing a polygon into its convex parts. In *Proceedings of the ACM Symposium on Theory of Computing*.

Choi, J.-W., Curry, R., & Elkaim, G. (2010). Curvature-continuous trajectory generation with corridor constraint for autonomous ground vehicles. In *Proceedings of the IEEE Conference on Decision and Control*.

Choi, S., Lee, J.-Y., & Yu, W. (2011). Fast any-angle path planning on grid maps with non-collision pruning. In *Proceedings of the IEEE International Conference on Robotics and Biomimetics*.

Choi, S., & Yu, W. (2011). Any-angle path planning on non-uniform costmaps. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Choset, H., Lynch, K., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L., & Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press.

Chrpa, L. (2011). Trajectory planning on grids: Considering speed limit constraints. In *Proceedings of the Scandinavian Conference on Artificial Intelligence*.

Chrpa, L., & Komenda, A. (2011). Smoothed hex-grid trajectory planning using helicopter dynamics. In *Proceedings of the International Conference on Agents and Artificial Intelligence*.

Cohen, B., Subramanian, G., Chitta, S., & Likhachev, M. (2011). Planning for manipulation with adaptive motion primitives. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Cormen, T., Stein, C., Rivest, R., & Leiserson, C. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education.

Crous, C. (2009). Autonomous robot path planning. Master's thesis, University of Stellenbosch.

Dechter, R., & Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery*, *3*, 505–536.

Deloura, M. (2000). *Game Programming Gems*. Charles River Media.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Dolgov, D., & Thrun, S. (2009). Autonomous driving in semi-structured environments: mapping and planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Dolgov, D., Thrun, S., Montemerlo, M., & Diebel, J. (2008). Practical search techniques in path planning for autonomous driving. In *Proceedings of the International Symposium on Search Techniques in Artificial Intelligence and Robotics*.

Dolgov, D., Thrun, S., Montemerlo, M., & Diebel, J. (2010). Path planning for autonomous vehicles in unknown semi-structured environments. In *The International Journal of Robotics Research*.

Ferguson, D. (2006). *Single Agent and Multi Agent Path Planning in Unknown and Dynamic Environments*. Ph.D. thesis, Carnegie Mellon University.

Ferguson, D., & Stentz, A. (2006). Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics*, *23*, 79–101.

Fernandez-Perdomo, E., Cabrera-Gamez, J., Hernandez-Sosa, D., Isern-Gonzalez, J., Dominguez-Brito, A., Redondo, A., Coca, J., Ramos, A., Fanjul, E., & Garcia, M. (2010). Path planning for gliders using regional ocean models: Application of pinzn path planner with the eseoat model and the ru27 trans-atlantic flight data. In *Proceedings of the IEEE OCEANS Conference*.

Foley, J., van Dam, A., Feiner, S., & Hughes, J. (1992). *Computer Graphics: Principles and Practice*. Addison-Wesley.

García, H., & Garrido, L. (2007). Towards exploration of unknown dynamic worlds using multiple robots. In *Proceedings of the Mexican International Conference on Artificial Intelligence*.

Gonzalez, J. (2008). *Planning with Uncertainty in Position Using High-Resolution Maps*. Ph.D. thesis, Carnegie Mellon University.

Gosset, T. (1900). On the regular and semi-regular figures in space of n dimensions. *Messenger of Mathematics*, *29*, 43–48.

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*, 100–107.

Hebert, M., MacLachlan, R., & Chang, P. (1999). Experiments with driving modes for urban robots. In *Proceedings of the SPIE*.

Higgins, D. (2002). Generic A* path finding. In Rabin, S. (Ed.), *AI Game Programming Wisdom*, pp. 114–121. Charles River Media.

Howard, T., & Kelly, A. (2005). Trajectory generation on rough terrain considering actuator dynamics. In *Proceedings of the International Conference on Field and Service Robotics*.

Howard, T., Knepper, R., & Kelly, A. (2006). Constrained optimization path following of wheeled robots in natural terrain. In *Proceedings of the International Symposium on Experimental Robotics*.

Hsu, D., Latombe, J., & Motwani, R. (1997). Path planning in expansive configuration spaces. *International Journal of Computational Geometry and Applications*, *3*, 2719–2726.

Ian Millington, J. F. (2009). *Artificial Intelligence for Games, Second Edition*. Morgan Kaufman.

Kavraki, L., Svestka, P., Latombe, J., & Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *Proceedings of the IEEE Transactions on Robotics and Automation*.

Kimmel, R., & Sethian, J. (2001). Optimal algorithm for shape from shading and path planning. *Journal of Mathematical Imaging and Vision*, *14*, 237–244.

Kleinberg, J., & Tardos, E. (2005). *Algorithm Design*. Addison Wesley.

Koenig, S., Daniel, K., & Nash, A. (2008). A project on any-angle path planning for computer games for 'introduction to artificial intelligence' classes. Tech. rep., Department of Computer Science, University of Southern California, Los Angeles, California.

Koenig, S., & Likhachev, M. (2002a). D* Lite. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Koenig, S., & Likhachev, M. (2002b). Incremental A*. In *Advances in Neural Information Processing Systems*.

Koenig, S., Likhachev, M., Liu, Y., & Furcy, D. (2004). Incremental heuristic search in artificial intelligence. *Artificial Intelligence Magazine*, *25*, 99–112.

Koenig, S., Likhachev, M., & Sun, X. (2007). Speeding up moving-target search. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*.

Koenig, S., Smirnov, Y., & Tovey, C. (2003). Performance bounds for planning in unknown terrain. *Artificial Intelligence Journal*, *147*, 253–279.

Koenig, S. (2009) Personal communication.

Konolige, K. (2000). A gradient method for realtime robot control. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*.

Korf, R. (1993). Linear-space best-first search. *Artificial Intelligence*, *62*, 41–78.

Kümmerle, R., Hähnel, D., Dolgov, D., Thrun, S., & Burgard, W. (2009). Autonomous driving in a multi-level parking structure. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Latombe, J. (1991). *Robot Motion Planning*. Kluwer Academic Publishers.

LaValle, S., Branicky, M., & Lindemann, S. (2004). On the relationship between classical grid search and probabilistic roadmaps. *International Journal of Robotics Research*, *23*, 673–692.

LaValle, S., & Kuffner, J. (2001). Rapidly-exploring random trees: Progress and prospects. In Donald, B. R., Lynch, K. M., & Rus, D. (Eds.), *Algorithmic and Computational Robotics: New Directions*, pp. 293–308. A K Peters.

LaValle, S. (2006). *Planning Algorithms*. Cambridge University Press.

Lee, D. (1978). *Proximity and reachability in the plane*. Ph.D. thesis, University of Illinois at Urbana-Champaign.

Lengyel, J., Reichert, M., Donald, B., & Greenberg, D. (1990). Real-time robot motion planning using rasterizing computer graphics hardware. In *Proceedings of the Special Interest Group on Graphics and Interactive Techniques*.

Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., & Thrun, S. (2008). Anytime search in dynamic graphs. *Artificial Intelligence Journal*, *172*, 1613–1643.

Likhachev, M., & Stentz, A. (2008). R* search. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Liu, Y., & Arimoto, S. (1992). Path planning using a tangent graph for mobile robots among polygonal and curved obstacles. *International Journal of Robotics Research*, *11*, 376–382.

Lozano-Pérez, T., & Wesley, M. (1979). An algorithm for planning collision-free paths among polyhedral obstacles. *Communication of the ACM*, *22*, 560–570.

Lumelsky, V., & Stepanov, A. (1990). *Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape*. Springer Verlag.

Matthews, J. (2002). Basic A* made simple. In Rabin, S. (Ed.), *AI Game Programming Wisdom*, pp. 105–113. Charles River Media.

Mazer, E., Ahuactzin, J., & Bessire, P. (1996). The Ariadne's clew algorithm. *Journal of Artificial Intelligence Research*, *9*, 9–295.

Mills-Tettey, G., Stentz, A., & Dias, M. (2008). Continuous-field path planning with constrained path-dependent state variables. In *Proceedings of the IEEE International Conference on Robotics and Automation Workshop on Path Planning on Costmaps*.

Murphy, R. (2000). *Introduction to AI Robotics*. MIT Press.

Nagy, B. (2003). Shortest paths in triangular grids with neighbourhood sequences. *Journal of Computing and Information Technology*, *11*(2), 111122.

Nash, A. (2010). *Theta*: Any-Angle Path Planning for Smoother Trajectories in Continuous Environments*.

Nash, A., Daniel, K., Koenig, S., & Felner, A. (2007). Theta*: Any-angle path planning on grids. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Nash, A., Daniel, K., Koenig, S., & Felner, A. (2010). Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, *39*, 533–579.

Nash, A., Koenig, S., & Likhachev, M. (2009). Incremental Phi*: Incremental any-angle path planning on grids. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Nash, A., Koenig, S., & Tovey, C. (2010). Lazy Theta*: Any-angle path planning and path length analysis in 3D. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Neller, T., DeNero, J., Klein, D., Koenig, S., Yeoh, W., Zheng, X., Daniel, K., Nash, A., Dodds, Z., Carenini, G., Poole, D., & Brooks, C. (2010). Model AI assignments. In *Proceedings of the Symposium on Educational Advances in Artificial Intelligence*.

OŃeil, J. (2004). Efficient navigation mesh implementation. *Journal of Game Development*, *1*, 71–90.

Patel, A. (2000). *Amit's Game Programming Information*.

Pearl, J. (1985). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Philippsen, R., Kolski, S., Macek, K., & Siegwart, R. (2007). Path planning, replanning and execution for autonomous driving in urban and offroad environments. In *Proceedings of the IEEE International Conference on Robotics and Automation Workshop on Planning, Perception and Navigation of Intelligent Vehicles*.

Philippsen, R., & Siegwart, R. (2005). An interpolated dynamic navigation function. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Rabin, S. (2000a). A* aesthetic optimizations. In Deloura, M. (Ed.), *Game Programming Gems*, pp. 264–271. Charles River Media.

Rabin, S. (2000b). A* speed optimizations. In Deloura, M. (Ed.), *Game Programming Gems*, pp. 272–287. Charles River Media.

Rabin, S. (2002). *AI Game Programming Wisdom*. Charles River Media.

Rabin, S. (2004). *AI Game Programming Wisdom 2*. Charles River Media.

Rabin, S. (2006). *AI Game Programming Wisdom 3*. Charles River Media.

Rabin, S. (2008). *AI Game Programming Wisdom 4*. Charles River Media.

Reif, J. (1979). Complexity of the mover's problem and generalizations. In *Proceedings of the Symposium on the Foundations of Computer Science*.

Rufli, M., Ferguson, D., & Siegwart, R. (2009). Smooth path planning in constrained environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Russel, S., & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall.

Samet, H. (1982). Neighbor finding techniques for images represented by quadtrees. In *Computer Graphics and Image Processing*.

Samet, H. (1988). The quadtree and related hierarchical data structures. *ACM Computing Surveys*, *16*(2), 187–260.

Sanchez, G., & Latombe, J. (2002). On delaying collision checking in PRM planning: Application to multi-robot coordination. *The International Journal of Robotics Research*, *21*, 5–26.

Sapronov, L., & Lacaze, A. (2010). Path planning for robotic vehicles using generalized Field D*. *Proceedings of the SPIE*, *6962*, 69621C–69621C–12.

Snook, G. (2000). Simplified 3D movement and pathfinding using navigation meshes. In Rabin, S. (Ed.), *Game Programming Gems*, pp. 288–304. Charles River Media.

Stentz, A. (1994). Optimal and efficient path planning for partially known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Stentz, A., & Hebert, M. (1995). A complete navigation system for goal acquisition in unknown environments. In *Proceedings of the IEEE International Conference On Intelligent Robotic Systems*.

Stout, B. (2000). The basics of A* for path planning. In Rabin, S. (Ed.), *Game Programming Gems*, pp. 254–262. Charles River Media.

Sturtevant, N. (2009). Optimizing motion-constrained pathfinding. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*.

Sturtevant, N. (2010) Personal communication.

Thorpe, C. (1984). Path relaxation: Path planning for a mobile robot. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Tozour, P. (2002). Building a near-optimal navigation mesh. In Rabin, S. (Ed.), *AI Game Programming Wisdom*, pp. 171–185. Charles River Media.

Tozour, P. (2004). Search space representations. In Rabin, S. (Ed.), *AI Game Programming Wisdom 2*, pp. 85–102. Charles River Media.

Tozour, P. (2008). *Fixing Pathfinding Once and For All*.

Tozour, P. (2010) Personal communication.

Trovato, K., & Dorst, L. (2002). Differential A*. *Proceedings of the IEEE Transaction on Knowledge and Data Engineering*, *14*(6), 1218–1229.

Šišlák, D., Volf, P., & Pěchouček, M. (2009a). Accelerated A* path planning. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*.

Šišlák, D., Volf, P., & Pěchouček, M. (2009b). Accelerated A* trajectory planning: Grid-based path planning comparison. In *Proceedings of the International Conference on Automated Planning and Scheduling Workshop on Planning and Plan Execution for Real-World Systems*.

Vykruta, T. (2002). Simple and efficient line-of-sight for 3D landscapes. In Rabin, S. (Ed.), *AI Game Programming Wisdom*, pp. 83–89. Charles River Media.

Walsh, K., & Banerjee, B. (2008). Variable resolution A*. In *Proceedings of the EUROSIS GAMEON-NA Conference*.

Wooden, D. (2006). *Graph-based Path Planning for Mobile Robots*. Ph.D. thesis, Georgia Institute of Technology.

Wu, P., Campbell, D., & Merz, T. (2009). On-board multi-objective mission planning for unmanned aerial vehicles. In *Proceedings of the IEEE Aerospace Conference*.

Yap, P. (2002). Grid-based path-finding. In *Proceedings of the Canadian Conference on Artificial Intelligence*.

Yap, P. (2011) Personal communication.

Yap, P., Burch, N., Holte, R., & Schaeffer, J. (2011). Block A*: Database-driven search with applications in any-angle path-planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Zhou, R., & Hansen, E. (2002). Multiple sequence alignment using A*. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Appendix A

Checking Line-of-Sight

In this appendix, we explain a technique for performing line-of-sight checks fast. For simplicity, we allow straight lines to pass between diagonally touching blocked grid cells. Performing a line-of-sight check is similar to determining which points to plot on a raster display when drawing a straight line between two points. The plotted points correspond to the grid cells that the straight line passes through. Thus, two vertices have line-of-sight iff none of the plotted points correspond to blocked grid cells. This allows Basic Theta* to perform its line-of-sight checks with the standard Bresenham line-drawing algorithm from computer graphics (Bresenham, 1965), that uses only fast logical and integer computations rather than floating-point computations. Algorithm 13 shows the resulting line-of-sight algorithm, where $s.x$ and $s.y$ are the x-coordinate and y-coordinates of vertex s, respectively, $grid$ represents the square grid and $grid(x, y)$ is true iff the corresponding grid cell is blocked. Recently, a new technique was introduced to reduce the runtime of the line-of-sight checks performed by Basic Theta*. We discuss this technique in Appendix E.2.1.

```
LineOfSight(s, s')
    x_0 := s.x;
    y_0 := s.y;
    x_1 := s'.x;
    y_1 := s'.y;
    d_y := y_1 − y_0;
    d_x := x_1 − x_0;
    f := 0;
    if d_y < 0 then
        d_y := −d_y;
        s_y := −1;
    else
        s_y := 1;
    if d_x < 0 then
        d_x := −d_x;
        s_x := −1;
    else
        s_x := 1;
    if d_x ≥ d_y then
        while x_0 ≠ x_1 do
            f := f + d_y;
            if f ≥ d_x then
                if grid(x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)) then
                    return false;
                y_0 := y_0 + s_y;
                f := f − d_x;
            if f ≠ 0 AND grid(x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)) then
                return false;
            if d_y = 0 AND grid(x_0 + ((s_x − 1)/2), y_0) AND grid(x_0 + ((s_x − 1)/2), y_0 − 1) then
                return false;
            x_0 := x_0 + s_x;
    else
        while y_0 ≠ y_1 do
            f := f + d_x;
            if f ≥ d_y then
                if grid(x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)) then
                    return false;
                x_0 := x_0 + s_x;
                f := f − d_y;
            if f ≠ 0 AND grid(x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)) then
                return false;
            if d_x = 0 AND grid(x_0, y_0 + ((s_y − 1)/2)) AND grid(x_0 − 1, y_0 + ((s_y − 1)/2)) then
                return false;
            y_0 := y_0 + s_y;
    return true;
```

Algorithm 13: Line-of-Sight Algorithm

Appendix B

AP Theta* Proofs

In this appendix, we prove that AP Theta* never returns a blocked path.[†]

Theorem 7. *AP Theta* never returns a blocked path.*

Proof. We define a path to be blocked iff at least one vertex on the path does not have line-of-sight to its successor on the path. Thus, a path is blocked iff at least one of its path segments passes through the interior of a blocked grid cell or passes between two blocked grid cells that share a side.

We first prove that AP Theta* never returns a path with a path segment that passes through the interior of a blocked grid cell. We prove by contradiction that AP Theta* cannot assign some parent p to some vertex s such that the path segment from parent p to vertex s passes through the interior of some blocked cell b. Assume otherwise. To simplify the proof, we translate and rotate the grid such that blocked grid cell b is immediately south-west of the origin b_0 of the grid and parent p is in quadrant II, as shown in Figure B.1. We define the quadrant of a vertex s as follows, where $s.x$ and $s.y$ are the x-coordinate and y-coordinate of vertex s, respectively:

- Quadrant I is the north-east quadrant (excluding the x-axis) given by $s.x \geq 0$ and $s.y > 0$.
- Quadrant II is the north-west quadrant (excluding the y-axis) given by $s.x < 0$ and $s.y \geq 0$.
- Quadrant III is the south-west quadrant (excluding the x-axis) given by $s.x \leq 0$ and $s.y < 0$.
- Quadrant IV is the south-east quadrant (excluding the y-axis but including the origin b_0) given by $s.x > 0$ and $s.y \leq 0$ or $s.x = 0$ and $s.y = 0$.

We refer to the neighbors of vertex s as $east(s)$, $northeast(s)$, $north(s)$, $northwest(s)$, $west(s)$, $southwest(s)$, $south(s)$, $southeast(s)$, as shown in Figure B.2.

Assume that there is a light source at vertex p and that light cannot pass through blocked grid cell b, which creates a shadow. A vertex s is in the shadow iff the straight line from parent p to vertex s passes through the interior of blocked grid cell b. We distinguish two parts of the perimeter of this shadow, namely the upper and lower boundary, as shown in Figure B.1. We define a boundary vertex to be any vertex not in the shadow that has at least one neighbor

[†]We would like to thank Kenny Daniel for his contributions to this proof. It could not have been completed without his help.

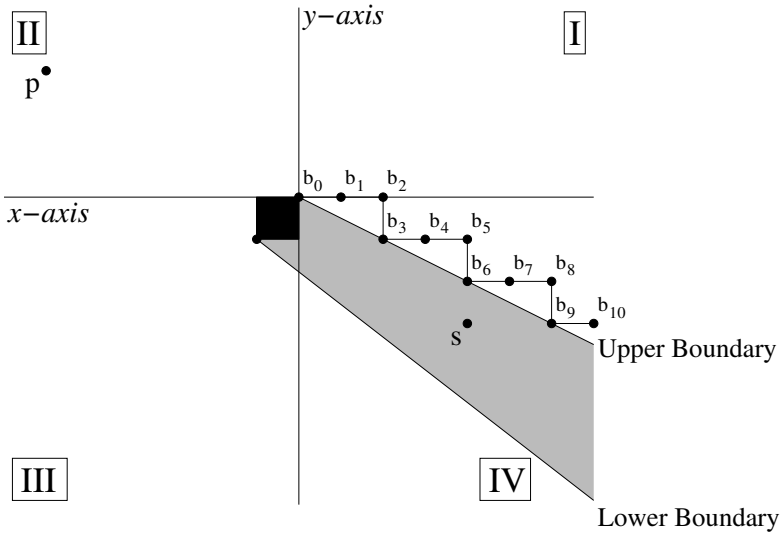


Figure B.1: Parent, Blocked Grid Cell and Boundary Vertices

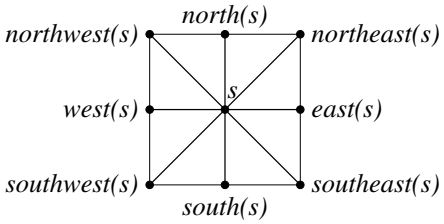


Figure B.2: Neighbors of Vertex s

(although not necessarily a visible neighbor) in the shadow. The origin b_0 is not in the shadow but its neighbor $south(b_0)$ is in the shadow. Thus, the origin b_0 is a boundary vertex. We consider only the upper boundary without loss of generality. Then, a boundary vertex (to be precise: an upper boundary vertex) is any vertex s with $\Theta(s, p, b_0) \leq 0$ (that is, on or above the upper boundary and thus outside of the shadow) that has at least one neighbor s' with $\Theta(s', p, b_0) > 0$ (that is, below the upper boundary and thus inside of the shadow). It is easy to see that all boundary vertices are in quadrant IV and form an infinite boundary path $[b_0, b_1, \ldots]$ that starts at the origin b_0 and repeatedly moves either south or east, that is, $b_{i+1} = south(b_i)$ or $b_{i+1} = east(b_i)$.

We define a vertex s to be sufficiently constrained iff $\Theta(s, p, b_0) \leq lb(s)$ for its parent p. Once vertex s is sufficiently constrained, it remains sufficiently constrained since no operation of AP Theta* can decrease its lower angle bound $lb(s)$. We prove in the following that every boundary vertex is sufficiently constrained at the time it is expanded if it is expanded with parent p. Consider any vertex s below the upper boundary (that is, $\Theta(s, p, b_0) > 0$ and thus $\Theta(b_0, p, s) < 0$) that is a visible neighbor of some boundary vertex b_i. Vertex s cannot have been updated according to Path 1 and been assigned parent p at the time its parent p was expanded since the straight line from parent p to vertex s passes through the interior of a blocked grid cell and they are therefore not visible neighbors. It cannot have been updated according to Path 2 and been

assigned parent p at the time boundary vertex b_i was expanded with parent p because boundary vertex b_i is sufficiently constrained at that time and thus $\Theta(b_i, p, b_0) \leq lb(b_i)$, which implies that $\Theta(b_i, p, s) = \Theta(b_i, p, b_0) + \Theta(b_0, p, s) < \Theta(b_i, p, b_0) \leq lb(b_i)$ and the condition on Line 55 remains unsatisfied. Consequently, no vertex in the shadow can have parent p.

We now prove by induction on the order of the vertex expansions that every boundary vertex is sufficiently constrained at the time it is expanded if it is expanded with parent p. Assume that boundary vertex b_0 is expanded with parent p. Then, the condition on Line 69 is satisfied and Line 71 is executed for blocked grid cell b at the time boundary vertex b_0 is expanded with parent p. Boundary vertex b_0 is sufficiently constrained afterwards since its lower angle bound is set to zero. Now assume that boundary vertex b_i with $i > 0$ is expanded with parent p. Then, boundary vertex b_i cannot be identical to parent p (since they are in different quadrants) nor to the start vertex (since the start vertex does not have parent p). Boundary vertex b_i cannot have been updated according to Path 1 and been assigned parent p at the time its parent p was expanded since $p.x < 0$ and $(b_i).x > 0$ and they are thus not neighbors. Consequently, boundary vertex b_i must have been updated according to Path 2 and been assigned parent p at the time one of its visible neighbors x was expanded with parent p. Vertex x must be on or above the upper boundary (that is, $\Theta(x, p, b_0) \leq 0$) and cannot be identical to parent p (since they are in different quadrants). We distinguish two cases:

- Assume that vertex x is a boundary vertex. It is sufficiently constrained at the time it is expanded with parent p according to the induction assumption (that is, $\Theta(x, p, b_0) \leq lb(x)$) since it is expanded before boundary vertex b_i. Boundary vertex b_i was updated according to Path 2 at the time vertex x was expanded with parent p. Thus, the condition on Line 55 is satisfied at that time (that is, $lb(x) \leq \Theta(x, p, b_i)$) and thus $lb(x) + \Theta(b_i, p, x) = lb(x) - \Theta(x, p, b_i) \leq 0$. Then, the conditions on Lines 76 and 77 are satisfied and Line 78 is executed with $s' = x$ at the time boundary vertex b_i is expanded with parent p. Boundary vertex b_i is sufficiently constrained afterwards since its lower angle bound is set to $\max(lb(b_i), lb(x) + \Theta(b_i, p, x))$ and $\Theta(b_i, p, b_0) = \Theta(b_i, p, x) + \Theta(x, p, b_0) \leq lb(x) + \Theta(b_i, p, x) \leq \max(lb(b_i), lb(x) + \Theta(b_i, p, x))$.

- Assume that vertex x is not a boundary vertex.

  Lemma 6. *Assume that a vertex s and a boundary vertex b_i are visible neighbors, $c(p, b_i) < c(p, s)$ and $\Theta(s, p, b_i) < 0$. Assume that boundary vertex b_i is sufficiently constrained at the time vertex s is expanded with parent p if boundary vertex b_i has been expanded with parent p at that time. Then, vertex s is sufficiently constrained at the time it is expanded if it is expanded with parent p.*

  Proof. Assume that vertex s is expanded with parent p. Then, $\Theta(s, p, b_0) = \Theta(s, p, b_i) + \Theta(b_i, p, b_0) < 0$ since $\Theta(s, p, b_i) < 0$ and $\Theta(b_i, p, b_0) \leq 0$. We distinguish two cases:

  - Assume that boundary vertex b_i is not expanded before vertex s or is expanded with a parent other than parent p. Then, the conditions on Lines 81 and 82 are satisfied and Line 83 is executed with $s' = b_i$ at the time vertex s is expanded with parent p. Vertex s is sufficiently constrained afterwards since its lower angle bound is set to $\max(lb(s), \Theta(s, p, b_i))$ and $\Theta(s, p, b_0) = \Theta(s, p, b_i) + \Theta(b_i, p, b_0) \leq \Theta(s, p, b_i) \leq \max(lb(s), \Theta(s, p, b_i))$.

– Assume that boundary vertex b_i is expanded with parent p before vertex s is expanded with parent p. Boundary vertex b_i is sufficiently constrained at the time vertex s is expanded with parent p according to the premise (that is, $\Theta(b_i, p, b_0) \leq lb(b_i)$). Furthermore, $lb(b_i) \leq 0$ (since no operation of AP Theta* can make the lower angle bound positive) and thus $lb(b_i) + \Theta(s, p, b_i) \leq 0$. Then, the conditions on Lines 76 and 77 are satisfied and Line 78 is executed with $s' = b_i$ at the time vertex s is expanded with parent p. Vertex s is sufficiently constrained afterwards since its lower angle bound is set to $\max(lb(s), lb(b_i) + \Theta(s, p, b_i))$ and $\Theta(s, p, b_0) = \Theta(s, p, b_i) + \Theta(b_i, p, b_0) \leq lb(b_i) + \Theta(s, p, b_i) \leq \max(lb(s), lb(b_i) + \Theta(s, p, b_i))$.

□

Boundary vertex b_i is either immediately south or east of boundary vertex b_{i-1} since the boundary path moves only south or east. We distinguish three subcases:

– Assume that parent p is on the x-axis in quadrant II. Then, the boundary path is along the x-axis. Vertices $west(b_i)$ and $east(b_i)$ are boundary vertices, and vertices $southwest(b_i)$, $south(b_i)$, and $southeast(b_i)$ are below the upper boundary. Thus, vertex x is identical to one of vertices $northwest(b_i)$, $north(b_i)$ or $northeast(b_i)$. In all cases, there is a boundary vertex b_j immediately south of vertex x. If vertices x and b_j were not visible neighbors, then there would be blocked grid cells immediately south-west and south-east of vertex x and vertices x and b_i could thus not be visible neighbors. Thus, vertices x and b_j are visible neighbors. Furthermore, boundary vertex b_j is immediately south of vertex x and thus $c(p, b_j) < c(p, x)$ and $\Theta(x, p, b_j) < 0$. Finally, boundary vertex b_j is sufficiently constrained according to the induction assumption at the time boundary vertex b_i is expanded with parent p if boundary vertex b_j has been expanded with parent p at that time. Thus, vertex x is sufficiently constrained at the time it is expanded with parent p according to Lemma 6 (that is, $\Theta(x, p, b_0) \leq lb(x)$). Consequently, the conditions on Lines 76 and 77 are satisfied (for the reason given before) and Line 78 is executed with $s' = x$ at the time boundary vertex b_i is expanded with parent p. Boundary vertex b_i is sufficiently constrained afterwards since its lower angle bound is set to $\max(lb(b_i), lb(x) + \Theta(b_i, p, x))$ and $\Theta(b_i, p, b_0) = \Theta(b_i, p, x) + \Theta(x, p, b_0) \leq lb(x) + \Theta(b_i, p, x) \leq \max(lb(b_i), lb(x) + \Theta(b_i, p, x))$.
– Assume that parent p is not on the x-axis in quadrant II and that boundary vertex b_i is immediately east of boundary vertex b_{i-1} and thus $c(p, b_{i-1}) < c(p, b_i)$ and $\Theta(b_i, p, b_{i-1}) < 0$. Furthermore, boundary vertex b_{i-1} is sufficiently constrained according to the induction assumption at the time boundary vertex b_i is expanded with parent p if boundary vertex b_{i-1} has been expanded with parent p at that time. If boundary vertices b_{i-1} and b_i are visible neighbors, then boundary vertex b_i is sufficiently constrained at the time it is expanded with parent p according to Lemma 6. If boundary vertices b_{i-1} and b_i are not visible neighbors, then there must be blocked grid cells immediately north-west and south-west of boundary vertex b_i. Then, Line 69 is satisfied and Line 71 is executed for the blocked grid cell immediately south-west of boundary vertex b_i at the time boundary vertex b_i is expanded with parent p.

Boundary vertex b_i is sufficiently constrained afterwards since its lower angle bound is set to zero.

– Assume that parent p is not on the x-axis in quadrant II and that boundary vertex b_i is immediately south of boundary vertex b_{i-1}.

Lemma 7. *Assume that a vertex s in quadrant IV is on or above the upper boundary. Then, vertex s is a boundary vertex iff the vertex immediately south-west of vertex s is below the upper boundary.*

Proof. If the vertex s' immediately south-west of vertex s is below the upper boundary, then vertex s is a boundary vertex by definition. On the other hand, if vertex s' is on or above the upper boundary (that is, $\Theta(s', p, b_0) \leq 0$), then vertex s is not a boundary vertex because every neighbor of it is on or above the upper boundary. The neighbors of vertex s are

$$east(s), northeast(s), north(s), northwest(s),$$
$$west(s), southwest(s), south(s) \text{ and } southeast(s).$$

or, equivalently,

$$east(east(north(s'))), east(east(north(north(s')))), east(north(north(s'))),$$
$$north(north(s')), north(s'), s', east(s') \text{ and } east(east(s')).$$

Thus, every neighbor s'' of vertex s can be reached from vertex s' by repeatedly moving either north or east and thus $\Theta(s'', p, s') \leq 0$. Consequently, $\Theta(s'', p, b_0) = \Theta(s'', p, s') + \Theta(s', p, b_0) \leq 0$ and thus every neighbor s'' of vertex s is on or above the upper boundary. □

We distinguish two subcases:

* Assume that boundary vertex b_{i+1} is immediately east of boundary vertex b_i. Vertices $north(b_i)$ and $east(b_i)$ are boundary vertices. Vertices $west(b_i)$, $southwest(b_i)$ and $south(b_i)$ are south-west of boundary vertices b_{i-1}, b_i and b_{i+1}, respectively, and thus below the upper boundary according to Lemma 7. Vertices $northwest(b_i)$ and $southeast(b_i)$ are either boundary vertices or south-west of boundary vertices b_{i-2} and b_{i+2}, respectively, and then below the upper boundary according to Lemma 7. Thus, vertex x is identical to vertex $northwest(b_i)$.
* Assume that boundary vertex b_{i+1} is immediately south of boundary vertex b_i. Vertices $north(b_i)$ and $south(b_i)$ are boundary vertices. Vertices $west(b_i)$ and $southwest(b_i)$ are south-west of boundary vertices b_{i-1} and b_i, respectively, and thus below the upper boundary according to Lemma 7. Vertex $northwest(b_i)$ is either a boundary vertex or south-west of boundary vertex b_{i-2} and then below the upper boundary according to Lemma 7. Thus, vertex x is identical to one of vertices $northeast(b_i)$, $east(b_i)$ or $southeast(b_i)$.

In all cases, vertex x is immediately east of some boundary vertex b_j and thus $c(p, b_j) < c(p, x)$ and $\Theta(x, p, b_j) < 0$. If vertices x and b_j were not visible neighbors, then there would be blocked grid cells immediately north-west and south-west

of vertex x and vertices x and b_i could not be visible neighbors. Thus, vertices x and b_j are visible neighbors. Furthermore, boundary vertex b_j is sufficiently constrained according to the induction assumption at the time boundary vertex b_i is expanded with parent p if boundary vertex b_j has been expanded with parent p at that time. Thus, vertex x is sufficiently constrained at the time it is expanded with parent p according to Lemma 6 (that is, $\Theta(x, p, b_0) \leq lb(x)$). Consequently, the conditions on Lines 76 and 77 are satisfied (for the reason given before) and Line 78 is executed with $s' = x$ at the time boundary vertex b_i is expanded with parent p. Boundary vertex b_i is sufficiently constrained afterwards since its lower angle bound is set to $\max(lb(b_i), lb(x) + \Theta(b_i, p, x))$ and $\Theta(b_i, p, b_0) = \Theta(b_i, p, x) + \Theta(x, p, b_0) \leq lb(x) + \Theta(b_i, p, x) \leq \max(lb(b_i), lb(x) + \Theta(b_i, p, x))$.

This concludes the proof that every boundary vertex is sufficiently constrained at the time it is expanded if it is expanded with parent p and thus also the proof that AP Theta* never returns a path with a path segment that passes through the interior of a blocked grid cell.

We now prove that AP Theta* never returns a path with a path segment that passes between two blocked grid cells that share an edge. We prove by contradiction that AP Theta* cannot assign some parent p to some vertex s such that the path segment from parent p to vertex s passes between two blocked grid cells that share an edge. Assume otherwise and consider the first time AP Theta* assigns some parent p to some vertex s such that the path segment from parent p to vertex s passes between two blocked grid cells that share an edge. The path segment must be either horizontal or vertical. Vertex s cannot have been updated according to Path 1 and been assigned parent p at the time its parent p was expanded since then the straight line from parent p to vertex s passes through the interior of a blocked grid cell and they are therefore not visible neighbors. It cannot have been updated according to Path 2 and been assigned parent p at the time some visible neighbor s' was expanded with parent p since then either a) neighbor s' would not be collinear with vertices p and s and the straight line from parent p to vertex s' would thus pass through the interior of a blocked grid cell or b) neighbor s' would be collinear with vertices p and s and the straight line from parent p to vertex s' would pass between two blocked grid cells that share an edge, which is a contradiction of the assumption. This concludes the proof that AP Theta* never returns a path with a path segment that passes between two blocked grid cells that share an edge.

Thus, AP Theta* never returns a blocked path. □

Appendix C

Regular Grid Proofs

In this appendix we examine the parts of the proofs of Lemma 1 and 2 that were omitted from Chapter 3. In Section C.1, we examine Part 1 of Lemma 1 for tri graphs, double tri graphs, quad graphs and octile graphs and in Section 3.3 we examine Part 1 of Lemma 2 for triple cubic graphs.

C.1 Part 1 of Lemma 1 for 2D Regular Grids

In this section, we prove Part 1 of Lemma 1 for tri graphs, double tri graphs, quad graphs and octile graphs.

- **Square Grid Step Sequence:** *Without loss of generality assume that L is in the sector S defined by m_i = an Up edge and m_{i+1} = a Right edge. We convert L into a step sequence $\widehat{L}$ of right (R) and up (U) steps. R steps increase the x-coordinate by 1 and U steps increase the y-coordinate by 1 (cell B1 of Table C.1). We assign coordinates to grid cells, namely the coordinates of their upper right corners (that is, the corners that are furthest from the origin). We start with an $\widehat{L}$ initialized to the empty sequence and traverse L from vertex u to vertex t. Whenever L leaves the interior of one grid cell c and enters the interior of another grid cell c', we append one step to $\widehat{L}$ depending on the 2 possible differences between the coordinates of c' and c. We append R for $(1,0)$ and U for $(0,1)$. The resulting $\widehat{L}$ moves from coordinates $(1,1)$ to vertex t at coordinates (r',u') and every step in $\widehat{L}$ traverses the side of grid cells that are known to be unblocked because L traverses their interiors.*

  - **Quad Graph Move Sequence Algorithm:** *We convert $\widehat{L}$ into a move sequence L' of right (r) and up (u) moves. r moves increase the x-coordinate by 1 and have length 1 and u moves increase the y-coordinate by 1 and have length 1 (cell B1 of Table C.1). We start with an L' that contains a single r move followed by a single u move and then we append moves to L' by iterating through the steps in $\widehat{L}$ as follows: If the next step in $\widehat{L}$ is an R or U then append an r or u move, respectively, to L'. L' is formed by edges on G' since all moves traverse the sides of grid cells that are known to unblocked because they are traversed by L. Furthermore, L' contains only Right edges (r moves) and Up edges (u moves) and L is in the sector S defined by a Right edge and an Up edge and thus L' must be a shortest grid path from vertex u to vertex t on G'.*
  - **Octile Graph Move Sequence Algorithm:** *Without loss of generality assume that L is in the sector S defined by m_i = an Up Right edge and m_{i+1} = a Right edge. We*

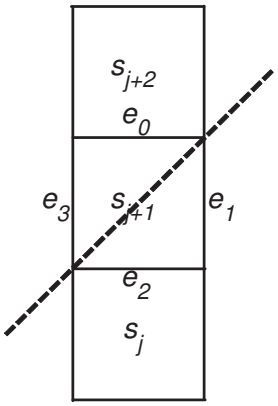


Figure C.1: Replacing Moves in the Move Sequence on Octile Graphs

construct a move sequence L' of right (r) and up right (ur) moves. r moves increase the x-coordinate by 1 *and have length* 1 *and ur moves increase the x-coordinate by* 1*, increase the y-coordinate by* 1 *and have length* $\sqrt{2}$ *(cell B1 of Table C.1). We start with an L' constructed using the technique defined above for quad graphs (which contains only u and r moves) and then replace moves in L' as follows: each u move and the preceding r move are replaced by a ur move. The algorithm must ensure that each u move is preceded by an r move and that the ur move that replaces them traverses an unblocked grid cell.*

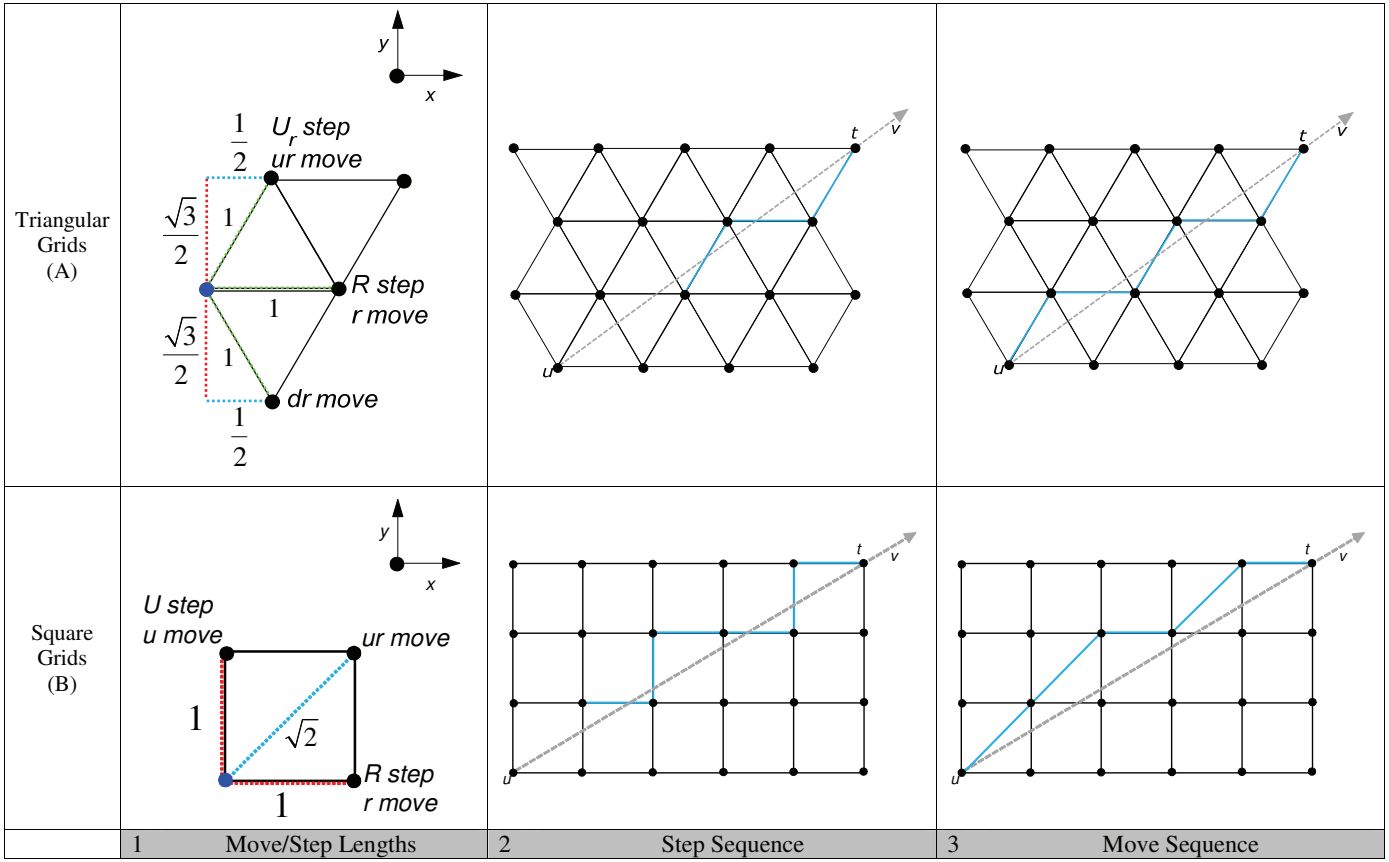


Table C.1: Triangular and Square Grid Step Sequence and Move Sequence for a Double Tri Graph and an Octile Graph

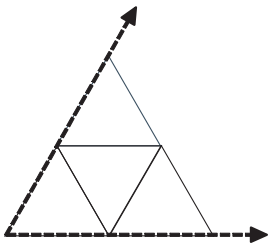


Figure C.2: Blocked Triangular Grid Cells

Each u move is preceded by an r move due to the following 2 properties: **(1)** *The first move in an L' constructed using the technique for quad graphs cannot be a u move because L is in the sector S defined by m_i = an Up Right edge and m_{i+1} = a Right edge and thus the first move cannot be a u move.* **(2)** *An L' constructed using the technique defined for quad graphs cannot contain two consecutive u moves. Consider a stack of three square grid cells such that the center grid cell shares sides e_0 and e_2 with the other two. L cannot traverse all three grid cells and thus L' cannot contain two consecutive u moves. This follows from the fact that coordinates in the interior of the bottom grid cell cannot be connected with coordinates in the interior of the top grid cell with a line segment in the sector S, as illustrated in Figure C.1 where the dashed line has the same angle as an Up Right edge. Finally, a u move preceded by an r move implies that L traverses the grid cell that the ur move traverses. Therefore, L' is formed by edges on G' since all moves traverse either the sides or interiors of grid cells that are known to be unblocked because they are traversed by L. Furthermore, L' contains only Right edges (r moves) and Up Right edges (ur moves) and L is in the sector S defined by an Up Right edge and a Right edge and thus L' must be a shortest grid path from vertex u to vertex t on G'.*

An example of Part 1 of the proof for octile graphs can be seen in row B of Table C.1. Row B of Table C.1 depicts the step and move sequences constructed from L. Cell B2 of Table C.1 depicts the step sequence $\widehat{L} = (RURRUR)$ and cell B3 of Table C.1 depicts the move sequence $L' = (ur, ur, r, ur, r)$ that results from applying the octile graph move sequence algorithm to $\widehat{L}$.

- **Triangular Grid Step Sequence:** *Without loss of generality assume that L is in the sector S defined by m_i = an Up Right edge and m_{i+1} = a Right edge. We convert L into a step sequence $\widehat{L}$ of right (R), and up right (U_r) steps. R steps increase the x-coordinate by 1 and U_r steps increase the x-coordinate by $\frac{1}{2}$ and the y-coordinate by $\frac{\sqrt{3}}{2}$ (cell A1 of Table C.1). We assign coordinates to rhombi defined by adjacent vertical and inverted triangular grid cells that share side e_2 of the inverted triangular grid cell, namely the coordinates of their upper right corners (that is, the corners that are furthest from the origin).*[1] *Because each vertex maps to a rhombus we assume that an entire rhombus must be either blocked or*

[1] As described in Section 3.2.3, the vertices in tri graphs and double tri graphs cannot map to the corners of triangular grid cells.

unblocked.[2] *We start with an $\widehat{L}$ initialized to the empty sequence and traverse L from vertex u to vertex t. Every time L leaves the interior of one rhombus c and enter the interior of another rhombus c', we append a step to $\widehat{L}$ depending on the 2 possible differences between the coordinates of c' and c. We append U_r for $(\frac{1}{2}, \frac{\sqrt{3}}{2})$ and R for $(1, 0)$. The resulting $\widehat{L}$ moves from coordinates $(\frac{3}{2}, \frac{\sqrt{3}}{2})$ to vertex t at coordinates (r', u') and every step in $\widehat{L}$ traverses the side of a rhombus (and thus a grid cell) that is known to be unblocked because L traverses its interior.*

- **Double Tri Graph Move Sequence Algorithm:** *We convert $\widehat{L}$ into a move sequence L' of right (r) and up-right (ur) moves. r moves increase the x-coordinate by 1 and have length 1 and ur moves increase the x-coordinate by $\frac{1}{2}$, increase the y-coordinate by $\frac{\sqrt{3}}{2}$ and have length 1 (cell A1 of Table C.1). We start with an L' that contains a single ur move followed by a single r move and then we append moves to L' by iterating through the steps in $\widehat{L}$ as follows: If the next step in $\widehat{L}$ is an R or U_r then we append an r or ur move, respectively, to L'. L' is formed by edges on G' since all moves traverse the sides of grid cells that are known to unblocked because they are traversed by L. Furthermore, L' contains only Right edges (r moves) and Up Right edges (ur moves) and L is in the sector S defined by a Right edge and an Up Right edge and thus L' must be a shortest grid path from vertex u to vertex t on G'.*

  An example of Part 1 of the proof for double tri graphs can be seen in row A of Table C.1. Row A of Table C.1 depicts the step and move sequences constructed from L. Cell A2 of Table C.1 depicts the step sequence $\widehat{L} = (U_R R U_R)$ and cell A3 of Table C.1 depicts the move sequence $L' = (ur, r, ur, r, ur)$ that results from applying the double tri graph move sequence algorithm to $\widehat{L}$.

- **Tri Graph Move Sequence Algorithm:** *Without loss of generality assume that L is in the sector S defined by m_i = an Up Right edge and m_{i+1} = a Down Right edge. We construct a move sequence L' of up right (ur), and down right (dr) moves. ur moves increase the x-coordinate by $\frac{1}{2}$, increase the y-coordinate by $\frac{\sqrt{3}}{2}$ and have length 1 and dr moves increase the x-coordinate by $\frac{1}{2}$, decrease the y-coordinate by $\frac{\sqrt{3}}{2}$ and have length 1 (cell A1 of Table C.1). We start with an L' constructed using the technique defined above for double tri graphs and then replace moves in L' as follows: each r move, which does not exist in the tri graph, is replaced by a dr move followed by a ur move (or vice versa). The algorithm must ensure that the dr and ur moves connect the same vertices as an r move and that they traverse unblocked grid cells. Assume that an r move connects vertices s and s'. A dr move followed by a ur move (or vice versa) connect vertices s and s'. An r move implies that either the rhombus whose top side coincides with the r move or the rhombus whose bottom side coincides with the r move must be unblocked. This follows from the fact that L must traverse at least one of the aforementioned two rhombi and thus at least one of the*

[2] If L is in sector S, allowing either vertical or inverted triangular grid cells (but not both) to be either blocked or unblocked is very similar to allowing entire rhombi to be either blocked or unblocked. Consider Figure C.2 in which the dashed arrows have the same angle as a_i and a_{i+1}. No coordinate in the interior of a vertical triangular grid cell can be connected with a coordinate in the interior of either of the other two vertical triangular grid cells with a straight line segment in sector S without traversing the interior of the inverted triangular grid cell. Therefore, if the inverted triangular grid cell is blocked it implies that the adjacent vertical triangular grid cell will not be traversed.

two rhombi must be unblocked. Thus a dr move followed by a ur move (or vice versa) connects s and s' and traverses the sides or interiors of rhombi that are known to be unblocked because one of the two rhombi is traversed by L. Therefore, L' is formed by edges on G' since all moves traverse the sides or interiors of grid cells that are known to be unblocked because they are traversed by L. Furthermore, L' contains only Up Right edges (ur moves) and Down Right edges (dr moves) and L is in the sector S defined by an Up Right edge and a Down Right edge and thus L' must be a shortest grid path from vertex u to vertex t on G'.

C.2 Part 1 of Lemma 2 for 3D Regular Grids

In this section, we prove the correctness of the triple cubic graph move sequence algorithm (Part 1 of Lemma 2 in Section 3.3), that is, that the move sequence algorithm finds a shortest grid path from vertex u to vertex t on G' .

The move sequence algorithm converts the step sequence $\widehat{L}$ (as defined in Section 3.3) into a grid path from vertex u to vertex t on G'. During the course of execution, the move sequence algorithm obeys the Position Property and the Unblocked Path Property. When the move sequence algorithm terminates, the resulting move sequence L' is a shortest grid path from vertex u to vertex t on G' due to the Shortest Path Property and the Unblocked Path Property. We showed in Section 3.3 that the move sequence algorithm obeys the Shortest Path Property. In this section we prove that the Unblocked Path Property and the Position Property hold after each iteration of the move sequence algorithm.

$\widehat{L}$ is constructed as follows: We start with $\widehat{L}$ being initialized to the empty sequence and traverse L from vertex u to vertex t. Whenever L leaves the interior of one grid cell c and enters the interior of another grid cell c', we append one or two steps to $\widehat{L}$ depending on the 6 possible differences between the coordinates of c' and c. We append $\underline{\text{R}}$ for $(1,0,0)$, $\underline{\text{B}}$ for $(0,0,1)$, $\underline{\text{U}}$ for $(0,1,0)$, B$\underline{\text{R}}$ for $(1,0,1)$, R$\underline{\text{U}}$ for $(1,1,0)$ and B$\underline{\text{U}}$ for $(0,1,1)$. We divide these 6 possible differences into two categories: **Category 1:** the hamming distance between the coordinates of c' and c is one, namely $\underline{\text{R}}$, $\underline{\text{B}}$ and $\underline{\text{U}}$. **Category 2:** the hamming distance between the coordinates of c' and c is two, namely B$\underline{\text{R}}$, R$\underline{\text{U}}$ and B$\underline{\text{U}}$.

We use induction and a case-based analysis to prove that the Position Property and the Unblocked Path Property hold after each iteration of the move sequence algorithm. First, we analyze the 3 cases in Category 1 and then we analyze the 3 cases in Category 2. In the former case, there is a one-to-one correspondence between a step in $\widehat{L}$ and a grid cell that is traversed by L. In the latter case, there are two steps in $\widehat{L}$ for a grid cell that is traversed by L and thus we must ensure that the moves in L' only traverse the sides, faces or interiors of grid cells that correspond to underlined steps.[3]

[3]The cases in Category 2 imply that L intersects one of the 12 defining sides of a grid cell (4 in the top face, 4 in the bottom face and 4 connecting the top and bottom face) . There are up to 4 grid cells in a cubic grid that share a side. Because L traverses the interior of 2 of those grid cells at least 3 of the 4 grid cells must be unblocked. Therefore, while L' does not need to be constructed of moves that traverse the sides, faces or interiors of the 2 grid cells (namely c and c') that L traverses the interior of, we show that the move sequence algorithm does in fact construct such an L'.

C.2.1 Notation and Definitions

The moves in L' connect the coordinates of grid cells with straight lines. The coordinates of grid cells coincide with the coordinates of vertices and thus we define the **current vertex** as the coordinates of the last grid cell (that is, vertex) in L' at the start of the current iteration of the move sequence algorithm. We concatenate two sequences of one or more steps by writing the two sequences next to each other. For example, $\underline{\text{RR}}$ represents consecutive $\underline{\text{R}}$s in $\widehat{L}$. We define the algorithm state to be the values of storeR and storeB. <start> indicates the start of $\widehat{L}$ and <end> indicates the end of $\widehat{L}$. Thus, $\widehat{L}$ begins with <start>$\underline{\text{R}}$ and ends with $\underline{\text{R}}$<end> due to the Step Sequence Property. We say that a move traverses a grid cell c iff the move either traverses a side, a face or the interior of c. Finally, we say that a move sequence L' is unblocked iff every move in the move sequence traverses a grid cell that is known to be unblocked because it is traversed by L.

We now prove that the Position Property and the Unblocked Path Property are guaranteed to hold.

Lemma 8. *The move sequence algorithm obeys the following two invariants at the end of each iteration:* **Position Property:** *Consider the grid cell with the coordinates reached from the coordinates of the first grid cell, namely coordinates* $(1, 1, 1)$ *after the execution of all steps deleted so far. Decrease its* x*-coordinate by one iff storeR = 1. Decrease its* z*-coordinate by one iff storeB = 1. The move sequence* L' *created so far moves from coordinates* $(0, 0, 0)$ *to exactly these coordinates.* **Unblocked Path Property:** L' *is formed of edges on* G' *since all moves traverse the grid cell at coordinates* $(1, 1, 1)$ *or grid cells with coordinates that are reached after each underlined step, which are known to be unblocked.*

Proof. We use induction to show that the lemma holds each time the move sequence algorithm finishes processing an underlined step in $\widehat{L}$. We do this by examining the 3 cases in Category 1, in which our induction step examines one underlined step in $\widehat{L}$, and the 3 cases in Category 2, in which our induction step examines one non-underlined step followed by one underlined step in $\widehat{L}$. When examining the 3 cases in Category 1 it is possible that, while one step is processed by the move sequence algorithm, two steps are deleted due to Line 4 in the move sequence algorithm which deletes an $\underline{\text{R}}$ if storeR = 0 when a $\underline{\text{U}}$ is being processed. When examining the 3 cases in Category 2 it is possible that, while two steps are processed by the move sequence algorithm, three steps are deleted due to Line 4 in the move sequence algorithm which deletes an $\underline{\text{R}}$ if storeR = 0 when a $\underline{\text{U}}$ is being processed.

Let $\widehat{Q}$ be a sequence of one or two steps in $\widehat{L}$. If we are examining one of the three cases in Category 1 then $\widehat{Q}$ is either $\underline{\text{R}}$, $\underline{\text{B}}$ or $\underline{\text{U}}$. If we are examining one of the three cases in Category 2 then $\widehat{Q}$ is either B$\underline{\text{R}}$, R$\underline{\text{U}}$ or B$\underline{\text{U}}$. Thus, in all 6 cases our induction step begins immediately after an underlined step $\underline{\text{P}}$ has been processed (or <start>). Our induction step enumerates every possible combination of $\underline{\text{P}}\widehat{Q}$ and every possible algorithm state (that is, values of storeR and storeB) after the move sequence algorithm has processed $\underline{\text{P}}$. Any $\widehat{L}$ that obeys the Step Sequence Property can be generated by combinations of the cases that we consider and in each of these cases we show that Lemma 8 holds.

We start with the base case. Let c and c' be the first two unblocked grid cells traversed by L. In Figure C.3, grid cell c' has a top face defined by 7, 11, 10, 8 (that is, its assigned coordinates correspond to vertex 11) and grid cell c has a top face defined by 2, 7, 8, 3 (that is, its assigned coordinates correspond to vertex 7). Initially, L' contains a single rbu move from coordinates

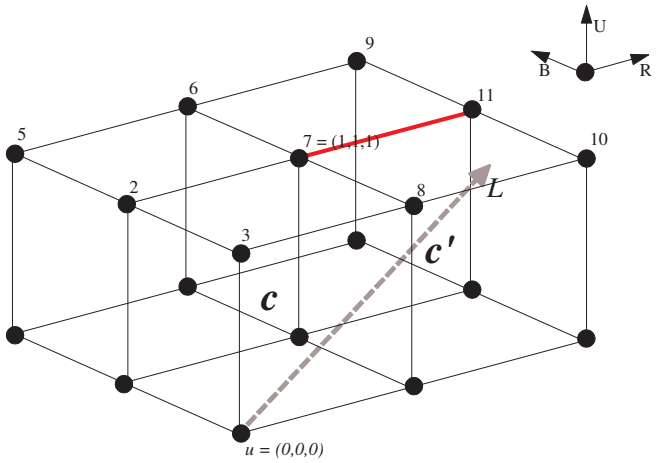


Figure C.3: <start> Step

$(0, 0, 0)$ to coordinates $(1, 1, 1)$, which traverses unblocked grid cell c, storeR = 0, storeB = 0 and the current vertex is 7 with coordinates $(1, 1, 1)$. The first step in $\widehat{L}$ must be an R due to the Step Sequence Property. Thus, we analyze P$\widehat{Q}$: <start>R.

- **P$\widehat{Q}$ =<start>R:** We know that the current vertex w at the start of the algorithm is 7 and that L' moves from coordinates $(0, 0, 0)$ to the current vertex $w = 7$ and is unblocked.
  - **$w = 7$:** The algorithm has just begun, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 7. The algorithm processes the R, sets storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 7. Due to the current vertex and the algorithm state, the lemma holds (vertex 11 is reached from $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 0 and the current vertex is 7).

We now move to the inductive step. Consider the move sequence algorithm immediately after processing the last underlined step P in an arbitrarily selected $\widehat{Q}$ in $\widehat{L}$.

C.2.2 Category 1

In this section, we consider the 3 steps in Category 1, namely $\widehat{Q} =$ R, $\widehat{Q} =$ B and $\widehat{Q} =$ U. L traverses consecutive unblocked grid cells c and c' and the coordinates of c and c' are connected by a solid red line in Figures C.4, C.5 and C.6, which depict R, B and U steps, respectively.

C.2.2.1 Right Step ($\widehat{Q} =$ R)

The step before $\widehat{Q} =$ R can only be P = R, P = B, or P = U due to the Step Sequence Property. Thus, we analyze the following three values of P$\widehat{Q}$: RR, BR and UR. (<start>R is possible, but was already addressed with the base case.) Let c' be the grid cell with a top face defined by 7, 11, 10, 8 (that is, its assigned coordinates correspond to vertex 11) and c be the grid cell with a top face defined by 2, 7, 8, 3 (that is, its assigned coordinates correspond to vertex 7) in Figure C.4.

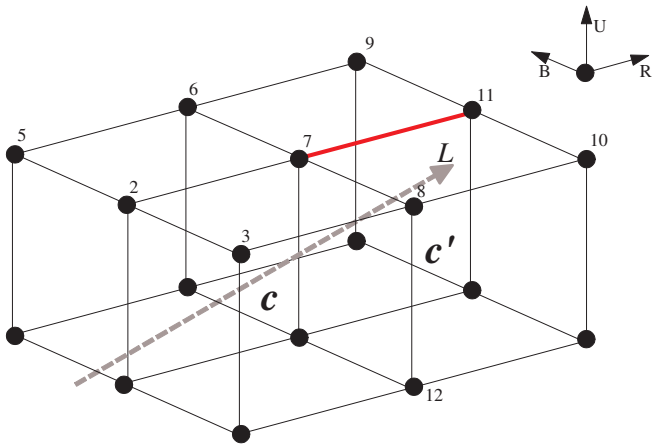


Figure C.4: Right Step ($\widehat{Q} = \underline{\text{R}}$)

- $\underline{\text{P}}\widehat{Q} = \underline{\text{RR}}$**:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the first $\underline{\text{R}}$ in the $\underline{\text{RR}}$ sequence must be either 2, 3 or 7. The first $\underline{\text{R}}$ was either processed by the move sequence algorithm, in which case storeR = 1 and $w = 2$ or $w = 3$, or it was deleted when the preceding $\underline{\text{U}}$ was processed, in which case storeR = 0, storeB = 0 and $w = 7$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 2$**:** The algorithm processed the first $\underline{\text{R}}$, set storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 2. The algorithm processes the second $\underline{\text{R}}$, appends an r move which traverses grid cell c, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 7. Due to the current vertex, the algorithm state and the fact that the r move traversed unblocked grid cell c, the lemma holds (vertex 11 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 0 and the current vertex is 7).
  - $w = 3$**:** The algorithm processed the first $\underline{\text{R}}$, set storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 3. The algorithm processes the second $\underline{\text{R}}$, appends an r move which traverses grid cell c, the state of the algorithm is (storeR = 1, storeB = 1) and the current vertex is 8. Due to the current vertex, the algorithm state and the fact that the r move traversed unblocked grid cell c, the lemma holds (vertex 11 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 1 and the current vertex is 8).
  - $w = 7$ **(The first $\underline{\text{R}}$ was deleted as a result of the $\underline{\text{U}}$ that immediately preceded it):** The algorithm deleted the first $\underline{\text{R}}$, set storeR := 0, storeB := 0, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 7. The algorithm processes the second $\underline{\text{R}}$, sets storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 7. Due to the current vertex and the algorithm state, the lemma holds (vertex 11 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 0 and the current vertex is 7).

- $\underline{\mathbf{P}}\widehat{Q} = \underline{\mathbf{BR}}$**:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the $\underline{\text{B}}$ in the $\underline{\text{BR}}$ sequence must be either 3 or 8. When the move sequence algorithm processes a $\underline{\text{B}}$, storeB = 1 and thus $w = 3$ or $w = 8$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 3$**:** The algorithm processed the $\underline{\text{B}}$, set storeB := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 3. The algorithm processes the $\underline{\text{R}}$, appends an r move which traverses c, the state of the algorithm is (storeR = 1, storeB = 1) and the current vertex is 8. Due to the current vertex, the algorithm state and the fact that the r move traversed unblocked grid cell c, the lemma holds (vertex 11 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 1 and the current vertex is 8).
  - $w = 8$**:** The algorithm processed the $\underline{\text{B}}$, set storeB := 1, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 8. The algorithm processes the $\underline{\text{R}}$, sets storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 8. Due to the current vertex and the algorithm state, the lemma holds (vertex 11 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 1 and the current vertex is 8).

- $\underline{\mathbf{P}}\widehat{Q} = \underline{\mathbf{UR}}$**:** Due to the Position Property we know that the current vertex w of grid cell c after the algorithm has finished processing the $\underline{\text{U}}$ in the $\underline{\text{UR}}$ sequence must be either 7 or 11. When the move sequence algorithm processes a $\underline{\text{U}}$, storeR = 0 and storeB = 0 and thus $w = 7$ or $w = 11$. In the latter case the $\underline{\text{R}}$ is deleted due to the $\underline{\text{U}}$ that immediately precedes it. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 7$**:** The algorithm processed the $\underline{\text{U}}$, set storeR := 0, storeB := 0, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 7. The algorithm processes the $\underline{\text{R}}$, sets storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 7. Due to the current vertex and the algorithm state, the lemma holds (vertex 11 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 0 and the current vertex is 7).
  - $w = 11$ **(The $\underline{\text{R}}$ in the $\underline{\text{UR}}$ sequence was deleted as a result of the $\underline{\text{U}}$ that immediately preceded it):** When the algorithm processed the $\underline{\text{U}}$ the algorithm state was (storeR = 0, storeB = 1) (Section 3.3.3.1). Thus, the step before the $\underline{\text{U}}$ must have been a $\underline{\text{B}}$ due to the Step Sequence property because ($\underline{\text{UU}}$ is impossible and $\underline{\text{RU}}$ would have set storeR := 1). Therefore the current vertex prior to $\underline{\text{U}}$ was 12, when the $\underline{\text{U}}$ was processed an rbu move was inserted that traversed c', the $\underline{\text{R}}$ was then deleted, the algorithm state was (storeR = 0, storeB = 0) and the current vertex was 11. Due to the current vertex, the algorithm state and the fact that the rbu move traversed unblocked grid cell c', the lemma holds (vertex 11 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 11). This is identical to the $\underline{\text{BUR}}$ sequence which is handled as one of the cases for the $\underline{\text{BU}}$ sequence in Section C.2.2.3.

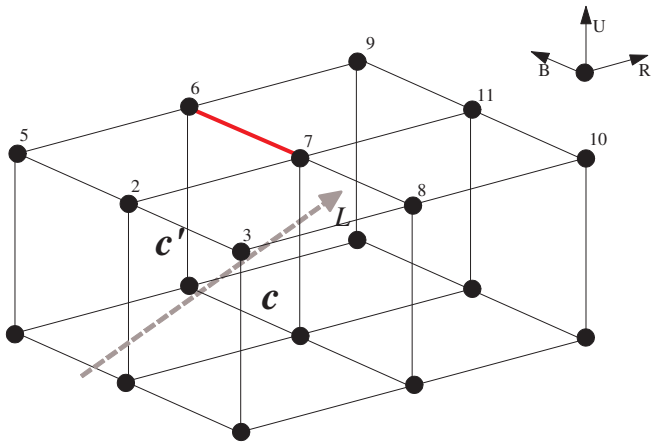


Figure C.5: Back Step ($\widehat{Q} = \underline{\text{B}}$)

C.2.2.2 Back Step ($\widehat{Q} = \underline{\text{B}}$)

The step before $\widehat{Q} = \underline{\text{B}}$ can only be $\underline{\text{P}} = \underline{\text{R}}$ or $\underline{\text{P}} = \underline{\text{U}}$ due to the Step Sequence Property ($\underline{\text{BB}}$ and <start>$\underline{\text{B}}$ are not possible). Thus, we analyze the following two values of $\underline{\text{P}}\widehat{Q}$: $\underline{\text{RB}}$ and $\underline{\text{UB}}$. Let c' be the grid cell with a top face defined by 5, 6, 7, 2 (that is, its assigned coordinates correspond to vertex 6) and c be the grid cell with a top face defined by 2, 7, 8, 3 (that is, its assigned coordinates correspond to vertex 7) in Figure C.5.

- $\underline{\text{P}}\widehat{Q} = \underline{\text{RB}}$**:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the $\underline{\text{R}}$ in the $\underline{\text{RB}}$ sequence must be either 2, 3 or 7. The $\underline{\text{R}}$ was either processed by the move sequence algorithm, in which case storeR = 1 and $w = 2$ or $w = 3$, or it was deleted when the preceding $\underline{\text{U}}$ was processed, in which case storeR = 0, storeB = 0 and $w = 7$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 2$**:** The algorithm processed the $\underline{\text{R}}$, set storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 2. The algorithm processes the $\underline{\text{B}}$, sets storeB := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 2. Due to the current vertex and the algorithm state, the lemma holds (vertex 6 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 1 and the current vertex is 2).
  - $w = 3$**:** The algorithm processed the $\underline{\text{R}}$, set storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 3. The algorithm processes the $\underline{\text{B}}$, appends an rb move which traverses c, the state of the algorithm is (storeR = 0, storeB = 1) and the current vertex is 7. Due to the current vertex, the algorithm state and the fact that the rb move traversed unblocked grid cell c, the lemma holds (vertex 6 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 1 and the current vertex is 7).

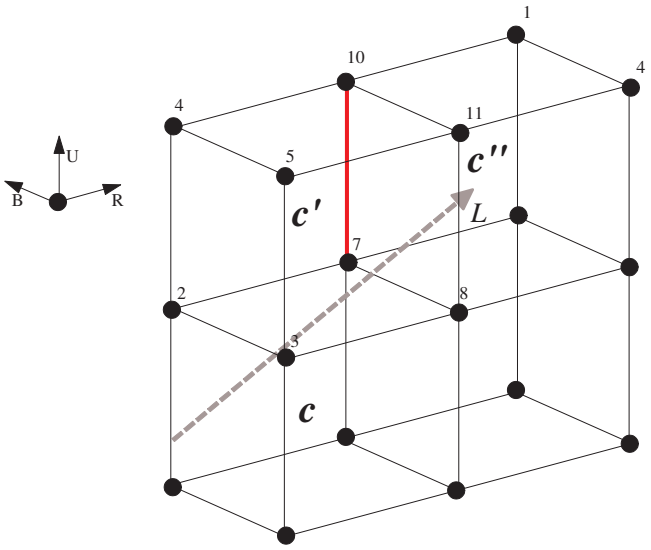


Figure C.6: Up Step ($\widehat{Q} = \underline{\text{U}}$)

- $w = 7$ **(The <u>R</u> in the <u>RB</u> sequence was deleted as a result of the <u>U</u> that immediately preceded it):** The algorithm deleted the <u>R</u>, set storeR := 0, storeB := 0, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 7. The algorithm processes the <u>B</u>, sets storeB := 1, the state of the algorithm is (storeR = 0, storeB = 1) and the current vertex is 7. Due to the current vertex and the algorithm state, the lemma holds (vertex 6 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 1 and the current vertex is 7).

- $\underline{\text{P}}\widehat{Q} = \underline{\text{UB}}$**:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the <u>U</u> in the <u>UB</u> sequence must be 7. When the move sequence algorithm processes a <u>U</u>, storeR = 0, storeB = 0 and thus $w = 7$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 7$**:** The algorithm processed the <u>U</u>, set storeR := 0, storeB := 0, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 7. The algorithm processes the <u>B</u>, sets storeB := 1, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 7. Due to the current vertex and the algorithm state, the lemma holds (vertex 6 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 1 and the current vertex is 7).

C.2.2.3 Up Step ($\widehat{Q} = \underline{\text{U}}$)

The step before $\widehat{Q} = \underline{\text{U}}$ can only be a <u>P</u> = <u>B</u> or <u>P</u> = <u>R</u> due to the Step Sequence Property (<u>UU</u> and <start><u>U</u> are not possible). Thus, we analyze the following two values of $\underline{\text{P}}\widehat{Q}$: <u>RU</u> and <u>BU</u>. Let c' be the unblocked grid cell with a top face defined by 4, 10, 11, 5 (that is, its assigned

coordinates correspond to vertex 10) and c be the unblocked grid cell with a top face defined by 2, 7, 8, 3 (that is, its assigned coordinates correspond to vertex 7) in Figure C.6.

- $\underline{\mathbf{P}}\widehat{Q} = \underline{\mathbf{RU}}$**:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the R in the RU sequence must be 3. When the move sequence algorithm processes an R, storeR = 1 and thus $w = 2$ or $w = 3$. However, $w = 2$ is impossible because a B must occur between any two Us or the start and a U and, after a B, storeB = 1. URU is impossible for the same reason and thus the R isn't being deleted as a result of the preceding U and thus it can not be the case that $w = 7$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 3$**:** The algorithm processed the R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 3. The algorithm processes the U, appends an rbu move which traverses c', the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 10. Due to the current vertex, the algorithm state and the fact that the rbu move traversed unblocked grid cell c', the lemma holds (vertex 10 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 10).

- $\underline{\mathbf{P}}\widehat{Q} = \underline{\mathbf{BU}}$**:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the B in the BU sequence must be either 3 or 8. When the move sequence algorithm processes a B, storeB = 1 and thus $w = 3$ or $w = 8$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 3$**:** The algorithm processed the B, set storeB := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 3. The algorithm processes the U, appends an rbu move which traverses c', the state of the algorithm is (storeR = 0, storeB = 0) and the current vertex is 10. Due to the current vertex, the algorithm state and the fact that the rbu move traversed unblocked grid cell c', the lemma holds (vertex 10 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 10).
  - $w = 8$**:** The algorithm processed the B, set storeB := 1, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 8. The algorithm processes the U, appends an rbu move which traverses grid cell c'' with a top face defined by 10, 1, 4, 11, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 1. The following R is then deleted (thus c'' must be unblocked). Due to the current vertex, the algorithm state and the fact that the rbu move traversed unblocked grid cell c'', the lemma holds (vertex 1 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 1).

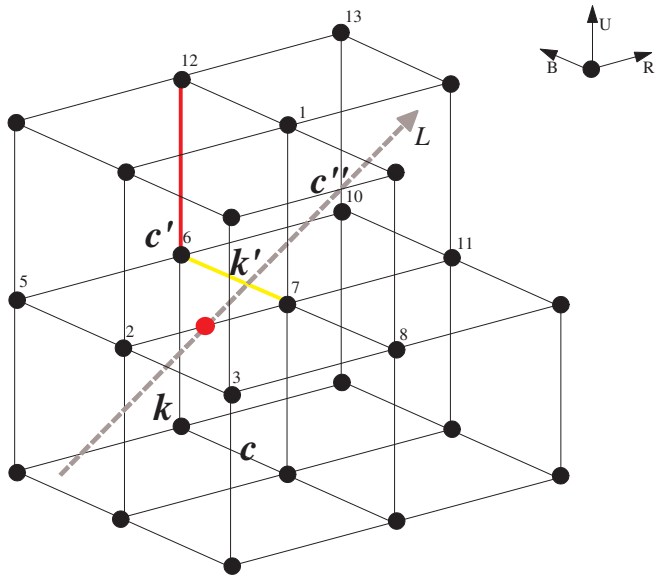


Figure C.7: Back Up Step ($\widehat{Q} = \text{B}\underline{\text{U}}$) [Right Side]

C.2.3 Category 2

In this section, we consider the 3 steps associated in Category 2, namely B$\underline{\text{U}}$, B$\underline{\text{R}}$ and R$\underline{\text{U}}$. L traverses consecutive unblocked grid cells c and c', and the coordinates of c and c' are connected by a solid yellow line followed by the solid red line in Figures C.7, C.8, and C.9, which depict B$\underline{\text{U}}$, B$\underline{\text{R}}$ and R$\underline{\text{U}}$ steps, respectively. The hamming distance between the coordinates of two consecutive grid cells traversed by L can only be 2 if L intersects one of the 12 defining sides of both grid cells (4 in the top face, 4 in the bottom face and 4 connecting the two faces). Therefore, grid cells c and c' must share a defining side (the red dots in Figures C.7, C.8, and C.9 lie on the intersected side). Let k and k' be the two other grid cells that share the same defining side that c and c' share. The blockage status of k and k' are unknown and we label them solely to ensure that L' does not traverse them. Each grid cell has 12 defining sides, but there are only 3 types of sides in a cubic grid, one for each dimension, namely **right sides** (the difference between the x-coordinates of the two endpoint vertices is 1 and the difference between the y-coordinates and z-coordinates of the two endpoint vertices is 0 (red line in Figure C.4)), **back sides** (the difference between the z-coordinates of the two endpoint vertices is 1 and the differences between the x-coordinates and y-coordinates of the two endpoint vertices is 0 (red line in Figure C.5)), and **up sides** (the difference between the y-coordinates of the two endpoint vertices is 1 and the differences between the x-coordinates and z-coordinates of the two endpoint vertices is 0 (red line in Figure C.6)). Category 2 cases only occur when L intersects the defining side of a cubic grid cell. There are only three types of defining sides that L can intersect and each of the three cases in Category 2 addresses one of them.

C.2.3.1 Back Up Step ($\widehat{Q}$ = BU) [Right Side]

The step before $\widehat{Q}$ = BU can only be P = R due to the Step Sequence Property (BBU, UBU and <start>BU are impossible). Thus, we analyze the following value of P$\widehat{Q}$: RBU. Suppose L intersects the right side which is implied by a BU (for example, side (2,7) in Figure C.7). Let c be the grid cell with a top face defined by 2, 7, 8, 3 (that is, its assigned coordinates correspond to vertex 7), c' be the grid cell with a bottom face defined by 5, 6, 7, 2 (that is, its assigned coordinates correspond to vertex 12), k be the grid cell with a top face defined by 5, 6, 7, 2 (that is, its assigned coordinates correspond to vertex 6) and k' be the grid cell with a bottom face defined by 2, 7, 8, 3 (that is, its assigned coordinates correspond to vertex 1).

- **P$\widehat{Q}$ = RBU:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the R in the RBU sequence must be either 2, 3 or 7. The R was either processed by the move sequence algorithm, in which case storeR = 1 and $w = 2$ or $w = 3$, or it was deleted when the preceding U was processed, in which case storeR = 0, storeB = 0 and $w = 7$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 2$**:** The algorithm processed the R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 2. The algorithm processes the B, sets storeB := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 2. The algorithm processes the U, appends an rbu move which traverses c', the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 12. Due to the current vertex, the algorithm state and the fact that the rbu move traversed the unblocked grid cell c', the lemma holds (vertex 12 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 12).
  - $w = 3$**:** The algorithm processed the R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 3. The algorithm processes the B, appends an rb move which traverses c, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 7. The algorithm processes the U, appends an rbu move which traverses c'' with a bottom face defined by 6, 10, 11, 7 (the grid cell is unblocked due to the R that has to follow the U, that is, BUB, BUU and BU<end> are not possible), deletes the R that follows the U, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 13. Due to the current vertex, the algorithm state and the fact that the rb move traversed the unblocked grid cell c, the rbu move traversed the unblocked grid cell c'', the lemma holds (vertex 13 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 13).
  - $w = 7$ **(The R in the RBU sequence was deleted as a result of the U that immediately preceded it):** The algorithm deleted the R, set storeR := 0, storeB := 0, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 7. The algorithm processes the B, sets storeB := 1, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 7. The algorithm processes the U, appends an rbu which traverses c'' with a bottom face defined by 6, 10, 11, 7 (c'' is unblocked due to the R that has to

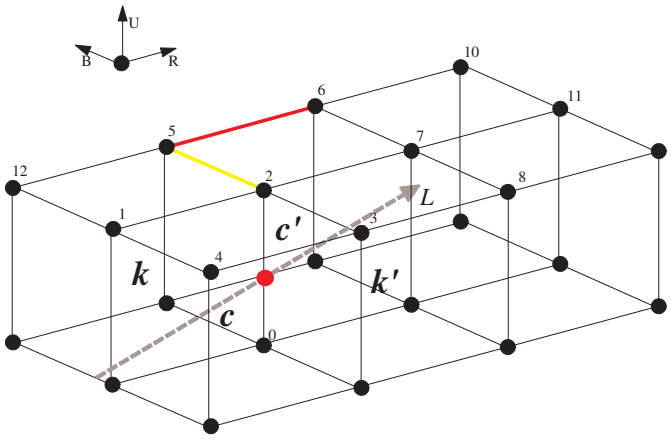


Figure C.8: Back Right Step ($\widehat{Q} = \underline{\text{BR}}$) [Up Side]

follow the U, that is, BUB, BUU and BU<end> are not possible), the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 13. Due to the current vertex, the algorithm state and the fact that the rbu move traversed the unblocked grid cell c'', the lemma holds (vertex 13 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 13).

C.2.3.2 Back Right Step ($\widehat{Q} = \underline{\text{BR}}$) [Up Side]

The step before $\widehat{Q} = \underline{\text{BR}}$ can only be $\underline{\text{P}} = \underline{\text{R}}$ or $\underline{\text{P}} = \underline{\text{U}}$ due to the Step Sequence Property (BBR and <start>BR are impossible). Thus, we analyze the following two values of $\underline{\text{P}}\widehat{Q}$: UBR and RBR. Suppose L intersects an up side which is implied by a BR (for example, side $(2, 0)$ in Figure C.8). Let c be the grid cell with a top face defined by 1, 2, 3, 4 (that is, its assigned coordinates correspond to vertex 2), c' be the grid cell with a top face defined by 5, 6, 7, 2 (that is, its assigned coordinates correspond to vertex 6), k be the grid cell with a top face defined by 12, 5, 2, 1 (that is, its assigned coordinates correspond to vertex 5) and k' be the grid cell with a top face 2, 7, 8, 3 (that is, its assigned coordinates correspond to vertex 7) in Figure C.8.

- $\underline{\text{P}}\widehat{Q} = \underline{\text{RBR}}$**:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the first R in the RBR sequence must be either 1, 4 or 2. The first R was either processed by the move sequence algorithm, in which case storeR = 1 and $w = 1$ or $w = 4$, or it was deleted when the preceding U was processed, in which case storeR = 0, storeB = 0 and $w = 2$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 1$**:** The algorithm processed the first R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 1. The algorithm processes the B, sets storeB := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 1. The algorithm processes the second R, appends an r move which traverses c, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 2. Due to the

current vertex, the state of the algorithm and the fact that the r move traversed the unblocked grid cell c, the lemma holds (vertex 6 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 1 and the current vertex is 2).

- $w = 4$**:** The algorithm processed the first R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 4. The algorithm processes the B, appends an rb move which traverses c, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 2. The algorithm processes the second R, sets storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 2. Due to the current vertex, the algorithm state and the fact that the rb move traversed the unblocked grid cell c, the lemma holds (vertex 6 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 1 and the current vertex is 2).
- $w = 2$ **(The first R in the RBR sequence was deleted as a result of the U that immediately preceded it):** The algorithm deleted the first R, set storeR := 0, storeB := 0, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 2. The algorithm processes the B, set storeB := 1, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 2. The algorithm processes the second R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 2. Due to the current vertex and the state of the algorithm, the lemma holds (vertex 6 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 1 and the current vertex is 2).

- **P**$\widehat{Q} =$ **UBR:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the U in the UBR sequence must be 2. When the move sequence algorithm processes a U, storeR = 0, storeB = 0 and thus $w = 2$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 2$**:** The algorithm processed the U, set storeR := 0, storeB := 0, the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 2. The algorithm processes the B, sets storeB := 1, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 2. The algorithm processes the R, sets storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 2. Due to the current vertex, the algorithm state, the lemma holds (vertex 6 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 1, storeB = 1 and the current vertex is 2).

C.2.3.3 Right Up Step ($\widehat{Q} =$ **RU**) [Back Side]

The step before $\widehat{Q} =$ RU can only be P = B or P = R due to the Step Sequence Property (URU, and <start>RU are impossible). Thus, we analyze the following two values of P$\widehat{Q}$: BRU and RRU. Suppose L intersects a back side which is implied by an RU (for example, side (6,7) in Figure C.9). Let c be the grid cell with a top face defined by 5, 6, 7, 2 (that is, its assigned coordinates

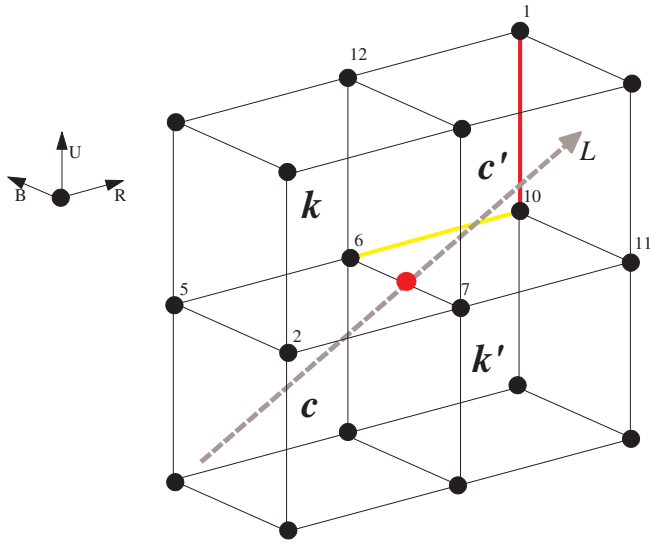


Figure C.9: Right Up Step ($\widehat{Q} = \text{R}\underline{\text{U}}$) [Back Side]

correspond to 6), c' be the grid cell with a bottom face defined by 6, 10, 11, 7 (that is, its assigned coordinates correspond to 1), k be the grid cell with a bottom face defined by 5, 6, 7, 2 (that is, its assigned coordinates correspond to 12) and k' be the grid cell with a top face defined by 6, 10, 11, 7 (that is, its assigned coordinates correspond to 10) in Figure C.9.

- **$\underline{\text{P}}\widehat{Q} = \underline{\text{BRU}}$:** Due to the Position Property and the induction assumption we known that the current vertex w after the algorithm has finished processing the $\underline{\text{B}}$ (that is, grid cell c) in the $\underline{\text{BRU}}$ sequence must be either 7 or 2, When the move sequence algorithm processes a $\underline{\text{B}}$, storeB = 1 and thus $w = 7$ or $w = 2$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to one of the potential vertices w and is unblocked.

  - $w = 7$**:** The algorithm processed the $\underline{\text{B}}$, set storeB := 1, the algorithm state is (storeR = 0, storeB = 1) and the current vertex is 7. The algorithm processes the R, sets storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 7. The algorithm processes the $\underline{\text{U}}$, appends an rbu move which traverses c', the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 1. Due to the current vertex, the algorithm state and the fact that the rbu move traversed the unblocked grid cell c', the lemma holds (vertex 1 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 1).
  - $w = 2$**:** The algorithm processed the $\underline{\text{B}}$, set storeB := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 2. The algorithm processes the R, appends an r move which traverses c, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 7. The algorithm processes the $\underline{\text{U}}$, appends an rbu move which traverses grid cell c', the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 1. Due to the current vertex, the algorithm state and the fact that the r move traversed unblocked grid cell c, the rbu move traversed the unblocked grid cell c', the

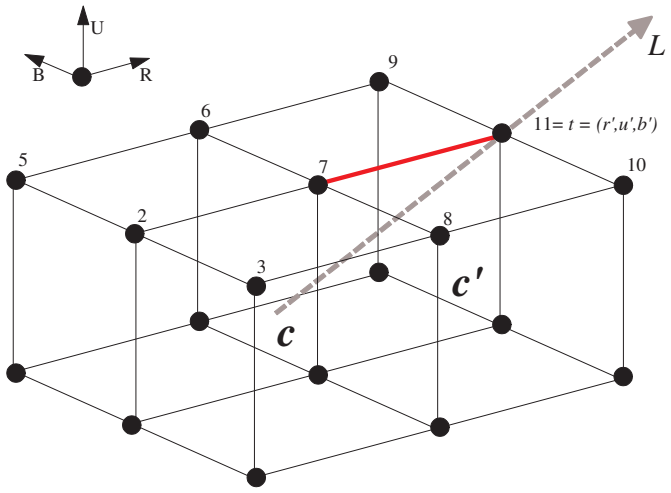


Figure C.10: <end> Step

lemma holds (vertex 1 is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 1).

- $\underline{\mathbf{P}}\widehat{Q} = \underline{\mathbf{RR}}\underline{\mathbf{U}}$**:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing the first R in the RRU sequence must be either 5 or 2. When the move sequence algorithm processes the first R, storeR = 1 and thus $w = 2$ or $w = 5$. URRU is not possible due to the Step Sequence Property and thus it cannot be the case that $w = 6$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to each potential value w and is unblocked.

  - $w = 5$**:** The algorithm processed the first R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 5. The algorithm processes the second R, appends an r move which traverses c, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 6. At this point, we have U. But we know that storeB = 0 is impossible before U. Thus, this case is impossible.

  - $w = 2$**:** The algorithm processed the first R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 2. The algorithm processes the second R, appends an r move which traverses c, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 7. The algorithm processes the U, appends an rbu move which traverses c', the algorithm state is (storeR = 0, storeB = 0) and the current vertex is the vertex is 1. Due to the current vertex, the algorithm state and the fact that the r move traversed unblocked grid cell c, the rbu move traversed unblocked grid cell c', the lemma holds (vertex 1 reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 1).

We have shown that the lemma holds after each step in $\widehat{L}$, other than the last step, is processed by the move sequence algorithm. When the last step in $\widehat{L}$ is processed the additional statements in the "then" clause of Line 6 are executed and thus we must verify that the lemma holds after these statements are executed. Let c and c' be the last two unblocked grid cells traversed by L. In Figure C.10, grid cell c' has a top face defined by 7, 11, 10, 8 (that is, its assigned coordinates correspond to vertex 11) and grid cell c has a top face defined by 2, 7, 8, 3 (that is, its assigned coordinates correspond to vertex 7). The last step in $\widehat{L}$ must be an R due to the Step Sequence Property. Thus we analyze the following value of $\underline{\mathrm{P}}\widehat{Q}$: R<end>.

- $\underline{\mathrm{P}}\widehat{Q}$ = **R<end>:** Due to the Position Property and the induction assumption we know that the current vertex w of grid cell c after the algorithm has finished processing R must be either 7 or 8. When the move sequence algorithm processes the R, storeR = 1 and thus $w = 7$ or $w = 8$. UR<end> is impossible because vertex t cannot be connected with any point on the bottom face of grid cell c' such that $r' > b' \geq u' \geq 1$ and thus it cannot be the case that $w = 11$. Due to the Unblocked Path Property and the induction assumption we know that L' moves from coordinates $(0, 0, 0)$ to each potential vertex w and is unblocked.
  - $w = 7$**:** The algorithm processed R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 0) and the current vertex is 7. The algorithm executes the then clause on Line 6, appends an r move which traverses c', the algorithm state is (storeR = 0, storeB = 0) and the current vertex is $11 = t = (r', u', b')$. Due to the current vertex, the algorithm state and the fact that the r move traversed the unblocked grid cell c', the lemma holds (vertex $11 = t$ at coordinates (r', u', b') is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 11).
  - $w = 8$**:** The algorithm processed R, set storeR := 1, the algorithm state is (storeR = 1, storeB = 1) and the current vertex is 8. The algorithm executes the "then" clause on Line 6, appends an rb move which traverses c', the algorithm state is (storeR = 0, storeB = 0) and the current vertex is 11 is equal to vertex t at coordinates (r', u', b'). Due to the current vertex, the algorithm state and the fact that the rb move traversed unblocked grid cell c', the lemma holds (vertex $11 = t$ at coordinates (r', u', b') is reached from coordinates $(1, 1, 1)$ after the execution of all steps deleted so far, storeR = 0, storeB = 0 and the current vertex is 11).

Therefore the lemma holds. □

Appendix D

Phi* and Incremental Phi* Proofs

In this appendix, we prove that Phi* and Incremental Phi* are complete and correct.

D.1 Notation and Definitions

- For any vertex $s \in V$, $G_p(s)$ is a sequence of pairwise distinct vertices $[v_0, v_1, \ldots, v_n]$ where $v_0 = s$, $v_{i+1} = \textit{local}(v_i)$ for all $0 \leq i < n$ and $v_n = s_{\textit{start}}$. We refer to $G_p(s)$ as a local path.

- For any vertex $s \in V$, $G_p(s, \textit{parent}(s)) \subseteq G_p(s)$ is a sequence of pairwise distinct vertices $[v_0, v_1, \ldots, v_n]$ where $v_0 = s$, $v_{i+1} = \textit{local}(v_i)$ for all $0 \leq i < n$, $\textit{local}(v_n) = \textit{parent}(s)$, $\textit{parent}(v_i) = \textit{parent}(s)$ for all $0 \leq i \leq n$ and $\Phi(v_{i+1}, \textit{parent}(s), v_i) \in [\textit{lb}(v_{i+1}), \textit{ub}(v_{i+1})]$ for all $0 \leq i < n$. We refer to $G_p(s, \textit{parent}(s))$ as a local parent path.

- For any vertex $s \in V$, $A_p(s) \subseteq G_p(s)$ is a sequence of pairwise distinct vertices $[v_0, v_1, \ldots, v_n]$ where $v_0 = s$, $v_{i+1} = \textit{parent}(v_i)$ for all $0 \leq i < n$, $v_n = s_{\textit{start}}$ and v_i has line-of-sight to its parent for all $0 \leq i < n$. We refer to $A_p(s)$ as a path.

- The proofs assume that, for any vertex $s \in V$, $A_p(s)$, $G_p(s)$ and $G_p(s, \textit{parent}(s))$ are sequences of pairwise distinct vertices. This is trivial to show because procedure ComputeCost assigns parents and local parents that point from vertices that are not in the closed list to vertices that are in the closed list.

- For any vertex $s \in V$, vertex s has been modified iff its membership in the open or closed lists changes or any of the following values change: the g-value of vertex s, the parent of vertex s, the local parent of vertex s, or the angle range of vertex s.

D.2 Proofs

We use induction on the number of calls to procedure ComputeShortestPath to show that Incremental Phi* is complete and correct. In order to do this we define a set of **Core Properties** that are required for Incremental Phi* to operate correctly. First, we examine the base case. We show in Section D.2.2 that the Core Properties hold at the conclusion of procedure Initialize (Line 206). Second, we examine the inductive step. In Section D.2.4, we examine an arbitrary call to procedure ComputeShortestPath (Line 208) during the while loop on Line 207. We show that

if the Core Properties hold at the start of procedure ComputeShortestPath then they hold at the conclusion of procedure ComputeShortestPath. We then use these Core Properties to show that **(1)** procedure ComputeShortestPath is complete and correct (Theorem 8) and that **(2)** it obeys the Local Parent Path Property (Theorem 6). In Section D.2.5, we examine an arbitrary call to procedure PreProcess (Lines 210-214) during the while loop on Line 207. We refer to Lines 210-214 as procedure PreProcess even though it is not explicitly encompassed within a procedure. We show that procedure PreProcess terminates. We then show that if the Core Properties hold at the start of procedure PreProcess then the Core Properties hold at the conclusion of procedure PreProcess. Thus, during the subsequent call to procedure ComputeShortestPath on Line 208, Theorem 8 and Theorem 6 hold and thus Incremental Phi* is complete and correct.

D.2.1 Core Properties

In this section, we define the set of Core Properties that are required for Incremental Phi* to be complete and correct.

Property 1. *The parent of the start vertex is the start vertex, the local parent of the start vertex is the start vertex, the g-value of the start vertex is 0, the angle range of the start vertex is* $[-\infty, \infty]$ *and the start vertex is in the open or closed lists.*

Property 2. *For any vertex* $t \in V$ *that is in the open or closed lists,* $G_p(t, \text{parent}(t)) = [t_0 = t, t_1, \ldots, t_n]$ *is well defined. Furthermore, for all* $1 \leq i \leq n$, $t_i \in G_p(t, \text{parent}(t))$ *is in the closed list.*

Property 3. *For any vertex* $t \in V$ *that is in the open or closed lists,* $G_p(t) = [t_0 = t, t_1, \ldots, t_n = s_{\text{start}}]$ *is well defined. Furthermore, for all* $1 \leq i \leq n$, $t_i \in G_p(t)$ *is in the closed list.*

Property 4. *For any vertex* $t \in V$ *such that procedure InitializeVertex*(t) *has been called at some point during the execution of procedure Main, the g-value of vertex* t *is finite iff vertex* t *is in the open or closed lists.*

Property 5. *For any vertex* $t \in V$ *that is in the closed list, the visible neighbors of vertex* t *are in the open or closed lists.*

Property 6. *For any vertex* $t \in V \setminus \{s_{\text{start}}\}$ *that is in the open or closed lists and whose* $\angle(t, \text{parent}(t))$ *is a multiple of 45 degrees, the parent of vertex* t *is a visible neighbor of vertex* t*.*

Property 7 (Line-of-Sight Property)**.** *For any vertex* $t \in V$ *that is in the open or closed lists, vertex* t *has line-of-sight to its parent.*

Property 8. *For any vertex* $t \in V$ *that is in the open or closed lists,* $A_p(t) = [t_0 = t, t_1, \ldots, t_n = s_{\text{start}}]$ *is well defined and thus is an unblocked path from the start vertex to vertex* t *in reverse and the g-value of vertex* t *is its length.*

D.2.2 Initialize

In this section, we show the following:

- The Core Properties hold at the conclusion of procedure Initialize (Line 206) and thus hold at the start of the first call to procedure ComputeShortestPath (Line 208).

D.2.2.1 Core Properties

Property D.2.2.1. *(At the conclusion of procedure Initialize:) The parent of the start vertex is the start vertex, the local parent of the start vertex is the start vertex, the g-value of the start vertex is 0, the angle range of the start vertex is* $[-\infty, \infty]$ *and the start vertex is in the open or closed lists.*

Proof. The g-value of the start vertex is set to 0 on Line 153, the parent of the start vertex is set to the start vertex on Line 154, the local parent of the start vertex is set to the start vertex on Line 155 and the angle range of the start vertex is set to $[-\infty, \infty]$ on Line 151. The start vertex is then added to the open list on Line 156. Therefore, at the conclusion of procedure Initialize, the g-value of the start vertex is 0, the parent of the start vertex is the start vertex, the local parent of the start vertex is the start vertex, the angle range of the start vertex is $[-\infty, \infty]$ and the start vertex is in the open or closed lists. □

Property D.2.2.2. *(At the conclusion of procedure Initialize:) For any vertex* $t \in V$ *that is in the open or closed lists,* $G_p(t, \text{parent}(t)) = [t_0 = t, t_1, \ldots, t_n]$ *is well defined. Furthermore, for all* $1 \leq i \leq n$*,* $t_i \in G_p(t, \text{parent}(t))$ *is in the closed list.*

Proof. There are no vertices in the closed list at the conclusion of procedure Initialize. The only vertex in the open list at the conclusion of procedure Initialize is the start vertex and the local parent and parent of the start vertex is the start vertex. Therefore, at the conclusion of procedure Initialize, the start vertex is the only vertex in the open or closed lists. $G_p(s_{start}, s_{start})$ has all the required properties since the local parent and parent of the start vertex is the start vertex. (We do not need to prove the parts of Property 2 that require $n > 0$). □

Property D.2.2.3. *(At the conclusion of procedure Initialize:) For any vertex* $t \in V$ *that is in the open or closed lists,* $G_p(t) = [t_0 = t, t_1, \ldots, t_n = s_{\text{start}}]$ *is well defined. Furthermore, for all* $1 \leq i \leq n$*,* $t_i \in G_p(t)$ *is in the closed list.*

Proof. There are no vertices in the closed list at the conclusion of procedure Initialize. The only vertex in the open list at the conclusion of procedure Initialize is the start vertex and the local parent of the start vertex is the start vertex. Therefore, at the conclusion of procedure Initialize, the start vertex is the only vertex in the open or closed lists and $G_p(s_{start})$ has all of the required properties since the local parent of the start vertex is the start vertex. (We do not need to prove the parts of Property 3 that require $n > 0$). □

Property D.2.2.4. *(At the conclusion of procedure Initialize:) For any vertex* $t \in V$ *such that procedure InitializeVertex*(t) *has been called at some point during the execution of procedure Main, the g-value of vertex* t *is finite iff vertex* t *is in the open or closed lists.*

Proof. During procedure Initialize, procedure InitializeVertex is called on Line 151 with the start vertex as the argument and on Line 152 with the goal vertex as the argument. The g-value of the start vertex is set to 0 on Line 153 and the start vertex is added to the open list on Line 156. The g-value of the goal vertex is set to infinity on Line 152 and the goal vertex is not added to the open or closed lists. Therefore, at the conclusion of procedure Initialize, the property holds. □

Property D.2.2.5. *(At the conclusion of procedure Initialize:) For any vertex* $t \in V$ *that is in the closed list, the visible neighbors of vertex* t *are in the open or closed lists.*

Proof. There are no vertices in the closed list at the conclusion of procedure Initialize. □

Property D.2.2.6. *(At the conclusion of procedure Initialize:) For any vertex $t \in V \setminus \{s_{\text{start}}\}$ that is in the open or closed lists and whose $\angle(t, \text{parent}(t))$ is a multiple of 45 degrees, the parent of vertex t is a visible neighbor of vertex t.*

Proof. There are no vertices in the closed list at the conclusion of procedure Initialize. The only vertex in the open list at the conclusion of procedure Initialize is the start vertex, which is explicitly omitted by the statement of the property. □

Property D.2.2.7 (Line-of-Sight Property)**.** *(At the conclusion of procedure Initialize:) For any vertex $t \in V$ that is in the open or closed lists, vertex t has line-of-sight to its parent.*

Proof. There are no vertices in the closed list at the conclusion of procedure Initialize. The only vertex in the open list at the conclusion of procedure Initialize is the start vertex. The parent of the start vertex is set to the start vertex on Line 154. Therefore, at the conclusion of procedure Initialize, the start vertex is the only vertex in the open or closed lists and it has line-of-sight to its parent. □

Property D.2.2.8. *(At the conclusion of procedure Initialize:) For any vertex $t \in V$ that is in the open or closed lists, $A_p(t) = [t_0 = t, t_1, \ldots, t_n = s_{\text{start}}]$ is well defined and thus is an unblocked path from the start vertex to vertex t in reverse and the g-value of vertex t is its length.*

Proof. There are no vertices in the closed list at the conclusion of procedure Initialize. The only vertex in the open list at the conclusion of procedure Initialize is the start vertex. The parent of the start vertex is set to the start vertex on Line 154 and the g-value of the start vertex is set to 0 on Line 153. Therefore, at the conclusion of procedure Initialize, the start vertex is the only vertex in the open or closed lists. $A_p(s_{start}, s_{start})$ is an unblocked path from the start vertex to the start vertex and the g-value of the start vertex, which is 0, is its length. □

D.2.3 Procedures ComputeCost and UpdateVertex

Procedures ComputeShortestPath (Section D.2.4) and PreProcess (Section D.2.5) both rely heavily on procedures ComputeCost and UpdateVertex. In this section, we prove lemmata about procedures ComputeCost and UpdateVertex.

Lemma 9. *For any vertex $s' \in V$ that was modified during procedure ComputeCost on Line 23, the following properties hold at the conclusion of procedure ComputeCost:*

- **(1)** *If vertex s' was updated according to Path 2 and the parent of vertex s' is in the closed list then $\Phi(\text{local}(s'), \text{parent}(s'), s') \in [\text{lb}(\text{local}(s')), \text{ub}(\text{local}(s'))]$.*
- **(2)** *If $\angle(s', \text{parent}(s'))$ is a multiple of 45 degrees, then the parent of vertex s' is a visible neighbor of vertex s'.*
- **(3)** $\text{g}(s') = \text{g}(\text{parent}(s')) + c(\text{parent}(s'), s')$.
- **(4)** *Vertex s' has line-of-sight to its parent.*

Proof. A call to procedure ComputeCost is preceded by a call to procedure UpdateVertex on Line 171 or Line 233 and, in either case, vertex s' is a visible neighbor of vertex s due to Lines 167 or 231. Property 1 ensures that vertex s' is not the start vertex because the g-value of the start vertex is always zero, $c(x, y) \geq 0$ and $g(x) \geq 0$ for any $x, y \in V$.[1] Thus the conditions on Lines 188 and 199 cannot be satisfied. Therefore, the start vertex cannot be modified during procedure ComputeCost.

- **(1)** Vertex s' was updated according to Path 2. A Path 2 update implies that vertex s' must have set its local parent to vertex s on Line 191 and its parent to $\mathit{parent}(s)$ on Line 189. The conditions on Line 186 must have been satisfied and the third condition on Line 186 ensures that $\Phi(\mathit{local}(s'), \mathit{parent}(s'), s') \in [\mathit{lb}(\mathit{local}(s')), \mathit{ub}(\mathit{local}(s'))]$.

  $\Phi(s, \mathit{parent}(s), s')$ is undefined if any pair of the following three vertices coincide: vertex s, vertex s' and the parent of vertex s'. Thus we must ensure that such a case is never encountered. Vertex s cannot equal vertex s' because they are visible neighbors. Vertex s can only equal the parent of vertex s if vertex s is the start vertex because the parent of a vertex is only set to that same vertex on Line 154. The parent of a vertex cannot be set to that same vertex on Line 200 since vertex s and vertex s' are visible neighbors and Line 188 ensures that the parent of a vertex cannot be set to that same vertex on Line 189. If vertex s is the start vertex then all visible neighbors of vertex s are updated according to Path 1 due to the first condition on Line 186 and the fact that the clauses in the if statement are evaluated from left to right with short circuit evaluation. Vertex s' cannot equal the parent of vertex s because vertex s' cannot be both the parent of vertex s and a visible neighbor of vertex s that is not in the closed list because we know, due to the statement of the lemma, that the parent of vertex s is in the closed list.

- **(2)** Vertex s' must have been updated according to either Path 1 or Path 2. Since $\angle(s', \mathit{parent}(s'))$ is a multiple of 45 degrees we know that vertex s' was updated according to Path 1. This follows directly from the first condition on Line 186. This ensures that the parent of vertex s' is a visible neighbor of vertex s' because the parent of vertex s' is vertex s due to Line 200 and vertex s' is a visible neighbor of vertex s due to Lines 167 or 231.

- **(3)** Vertex s' must have been updated according to either Path 1 or Path 2. If vertex s' was updated according to Path 1 then its parent (that is, $\mathit{parent}(s')$) must have been set on Line 200. Immediately after the parent of vertex s' is set to vertex s on Line 200 the g-value of vertex s' is set to $g(\mathit{parent}(s')) + c(\mathit{parent}(s'), s')$ on Line 201. If vertex s' was updated according to Path 2 then its parent (that is, $\mathit{parent}(s')$) must have been set on Line 189. Immediately after the parent of vertex s' is set to the parent of vertex s on Line 189 the g-value of vertex s' is set to $g(\mathit{parent}(s')) + c(\mathit{parent}(s'), s')$ on Line 190.

- **(4)** Vertex s' must have been updated according to either Path 1 or Path 2. If vertex s' was updated according to Path 1 then its parent (that is, $\mathit{parent}(s')$) must have been set to vertex s on Line 200 in which case vertex s' is a visible neighbor of vertex s due to Lines 167 or 231. Therefore, vertex s' has line-of-sight to its parent. If vertex s' was updated according to Path 2 then its parent (that is, $\mathit{parent}(s')$) must have been set to the parent of vertex s on

[1] Property 1 holds at any point during procedures ComputeShortestPath and ClearSubtree and thus Property 1 holds anytime procedure ComputeCost is called.

Line 189 in which case vertex s' has line-of-sight to its parent due to the second condition on Line 186 which explicitly ensures that vertex s' has line-of-sight to its parent.

□

Lemma 10. *For any vertex $s' \in V$ that was added to the open list during procedure UpdateVertex on Line 27, the following properties hold at the conclusion of procedure UpdateVertex:*

- **(1)** *If vertex s' was updated according to Path 2 and the parent of vertex s' is in the closed list then $\Phi(\text{local}(s'), \text{parent}(s'), s') \in [\text{lb}(\text{local}(s')), \text{ub}(\text{local}(s'))]$.*
- **(2)** *If $\angle(s', \text{parent}(s'))$ is a multiple of 45 degrees, then the parent of vertex s' is a visible neighbor of vertex s'.*
- **(3)** $\text{g}(s') = \text{g}(\text{parent}(s')) + c(\text{parent}(s'), s')$.
- **(4)** *Vertex s' has line-of-sight to its parent.*

Proof. If a vertex s' is added to the open list during procedure UpdateVertex on Line 27 then it must have satisfied the condition on Line 24 , namely the g-value of vertex s' is less than g_{old}. Since g_{old} was set to the g-value of vertex s' at the start of procedure ComputeCost, vertex s' must have been modified during procedure ComputeCost and Lemma 9 ensures that the lemma hold. □

D.2.4 Procedure ComputeShortestPath

In this section, we show the following:

- Procedure ComputeShortestPath terminates.
- If the Core Properties hold at the start of procedure ComputeShortestPath (Line 208) then the Core Properties hold at the conclusion of procedure ComputeShortestPath.
- Procedure ComputeShortestPath is complete and correct.
- Procedure ComputeShortestPath obeys the Local Parent Path Property.

Figure D.1 provides a summary of the dependencies of the lemmata, theorems and properties used in this section.

D.2.4.1 Helper Lemmata

We use the following helper lemmata:

Lemma 11. *(At any point during the execution of procedure ComputeShortestPath:) Once a vertex is added to the closed list, it is neither removed from the closed list nor added to the open list.*

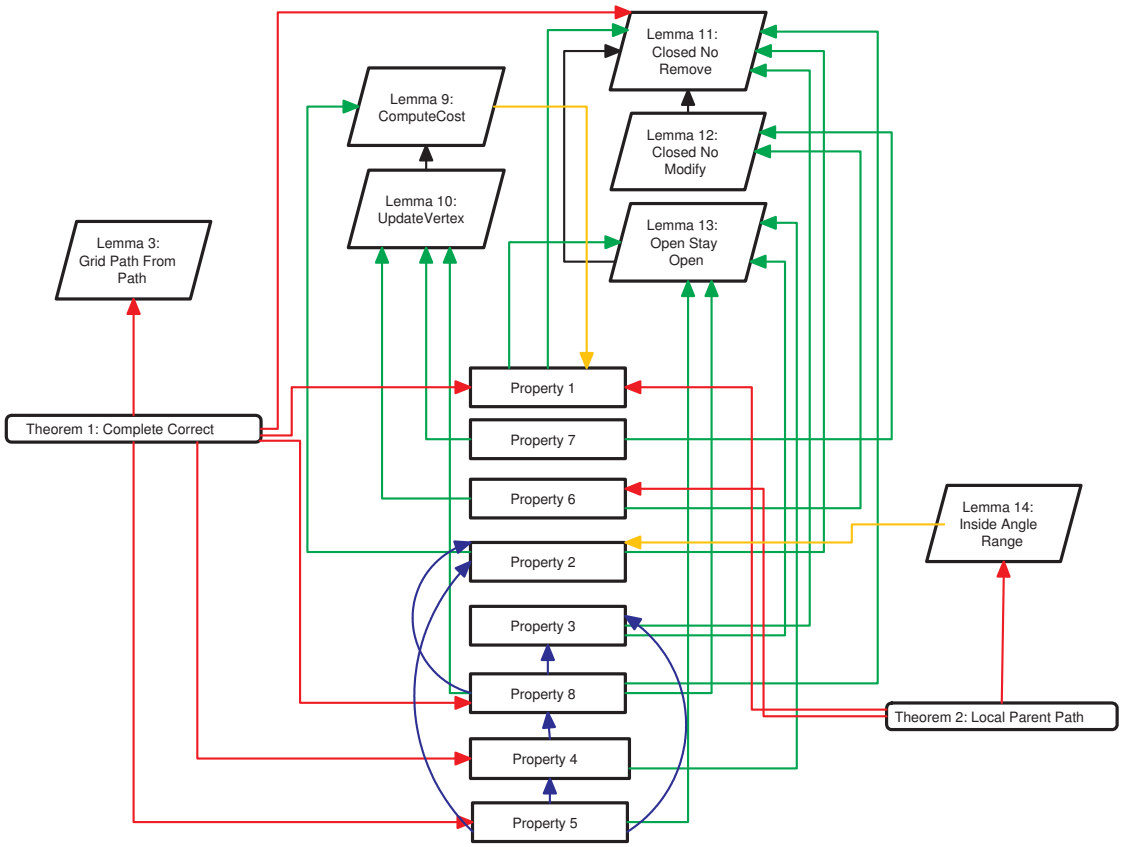


Figure D.1: Summary of Procedure ComputeShortestPath Dependencies

Proof. Consider any vertex t in the closed list. Vertex t can never be removed from the closed list because there is no statement in procedure ComputeShortestPath (or in any of the procedures called by procedure ComputeShortestPath) that removes a vertex from the closed list. Furthermore, because vertex t can never be removed from the closed list, Line 168 ensures that vertex t cannot be added to the open list at any point in the future. Thus, once a vertex is added to the closed list it can neither be removed from the closed list nor added to the open list. □

Lemma 12. *(At any point during the execution of procedure ComputeShortestPath:) Once a vertex t is added to the closed list, the g-value of vertex t, the parent of vertex t, the local parent of vertex t and the angle range of vertex t are never modified.*

Proof. Consider any vertex t in the closed list. Lemma 11 ensures that vertex t cannot be removed from the closed list. Line 168 ensures that, as long as vertex t is in the closed list, the g-value of vertex t, the parent of vertex t, the local parent of vertex t and the angle range of vertex t are never modified. This follows from the fact that the g-value of vertex t, the parent of vertex t, the local parent of vertex t and the angle range of vertex t can only be modified during a call to procedure InitializeVertex (Line 170) or procedure UpdateVertex (Line 171), which can only be executed if vertex t is not in the closed list (Line 168). Thus, once a vertex t is added to the closed

list, the g-value of vertex t, the parent of vertex t, the local parent of vertex t and the angle range of vertex t are never modified. □

Lemma 13. *(At any point during the execution of procedure ComputeShortestPath:) Once a vertex t is added to the open list it continues to be a member of the open or closed lists.*

Proof. Consider any vertex t in the open list. Vertex t can only be removed from the open list on Lines 26 or 165 after which it is immediately re-inserted into the open list or added to the closed list on Lines 27 and 166, respectively. Once a vertex is added to the closed list it remains in the closed list according to Lemma 11. Thus, once a vertex t is added to the open list, it continues to be a member of the open or closed lists. □

D.2.4.2 Core Properties

Consider an arbitrary iteration of the while loop on Line 207, which calls procedure ComputeShortestPath. Due to our induction assumption we know that the Core Properties hold at the start of procedure ComputeShortestPath. Therefore, we only need to validate the Core Properties for vertices that are modified during procedure ComputeShortestPath.

Property D.2.4.1. *(At any point during the execution of procedure ComputeShortestPath:) The parent of the start vertex is the start vertex, the local parent of the start vertex is the start vertex, the g-value of the start vertex is 0, the angle range of the start vertex is $[-\infty, \infty]$ and the start vertex is in the open or closed lists.*

Proof. Due to our induction assumption regarding the Core Properties, specifically Property 1, we know that the start vertex is in the open or closed lists at the start of procedure ComputeShortestPath and therefore Lemmata 11 and 13 ensure that it continues to be a member of the open or closed lists. Due to our induction assumption regarding the Core Properties, specifically Property 1, we also know that the g-value of the start vertex is 0, the parent of the start vertex is the start vertex, the local parent of the start vertex is the start vertex and the angle range of the start vertex is $[-\infty, \infty]$ at the start of procedure ComputeShortestPath. The g-value of the start vertex, the parent of the start vertex, the local parent of the start vertex and the angle range of the start vertex can only be modified during a call to either procedure InitializeVertex (Line 170) or procedure ComputeCost (Line 23). In the former case, we know that the parent of the start vertex, the local parent of the start vertex, the g-value of the start vertex and the angle range of the start vertex are never modified because we just showed that the start vertex is in the open or closed lists throughout procedure ComputeShortestPath and thus Lines 168 and 169 ensure that procedure InitializeVertex(s_{start}) is never called. In the latter case, we know that the parent of the start vertex, the local parent of the start vertex, the g-value of the start vertex and the angle range of the start vertex can only be modified if the conditions on either Line 188 or 199 are satisfied, but this cannot happen because the g-value of the start vertex is 0 and $c(x, y) \geq 0$ and $g(x) \geq 0$ for any $x, y \in V$. □

Property D.2.4.2. *(At any point during the execution of procedure ComputeShortestPath:) For any vertex $t \in V$ that is in the open or closed lists, $G_p(t, \text{parent}(t)) = [t_0 = t, t_1, \ldots, t_n]$ is well defined. Furthermore, for all $1 \leq i \leq n$, $t_i \in G_p(t, \text{parent}(t))$ is in the closed list.*

Proof. The property holds for any vertex $t \in V$ that is in the open or closed lists at the start of procedure ComputeShortestPath due to our induction assumption regarding the Core Properties, specifically Property 2. Consider any vertex t that becomes a member of the open or closed lists during procedure ComputeShortestPath. There must exist some sequence of vertices $X = [t_0 = t, \ldots, t_{n+1} = \textit{parent}(t)]$ such that vertex t_{i-1} is a visible neighbor of vertex t_i for all $1 \leq i \leq n+1$ and the parent of vertex t_i is equal to vertex $\textit{parent}(t)$ for all $1 \leq i \leq n$ when vertex t_i was expanded (expanded vertices are added to the closed list and thus vertex t_i is in the closed list for all $1 \leq i \leq n+1$ according to Lemma 11). If this was not the case then vertex t could not possibly set its parent to vertex $\textit{parent}(t)$.

First, we show that the local parent of vertex t_i is equal to vertex t_{i+1} for all $0 \leq i \leq n$ and that the parent of vertex t_i is equal to vertex $\textit{parent}(t)$ for all $0 \leq i \leq n$. We prove by induction on i that for any vertex t_i, where $0 \leq i \leq n$, the property holds. First, we examine the base case. When vertex $\textit{parent}(t)$ is expanded, some visible neighbor t_n of vertex $\textit{parent}(t)$ must have set its parent and local parent to vertex $\textit{parent}(t)$ on Lines 200 and 202, respectively. This follows from the fact that $\angle(t_n, \textit{parent}(t))$ must be a multiple of 45 degrees because vertex t_n and vertex $\textit{parent}(t)$ are visible neighbors. Thus, the first condition on Line 186 ensures that the parent and local parent of vertex t_n must have been set on Lines 200 and 202, respectively. Now we examine the induction step. Consider any vertex $t_i \in X$, where $0 \leq i < n$. When vertex t_{i+1} is expanded, the visible neighbor t_i of vertex t_{i+1} must set its parent to vertex $\textit{parent}(t)$ and thus vertex t_i must be updated according to Path 2. If it were updated according to Path 1 then the parent of vertex t_i would be vertex t_{i+1}. Therefore, vertex t_i must set its parent to vertex $\textit{parent}(t)$ on Line 189 after which vertex t_i must set its local parent to vertex t_{i+1} on Line 191. There are no other ways in which the parent or local parent of a vertex can change.

Second, we show that $\Phi(t_{i+1}, \textit{parent}(t), t_i) \in [\textit{lb}(t_{i+1}), \textit{ub}(t_{i+1})]$ for all $0 \leq i < n$. Consider any vertex $t_i \in X$ where $0 \leq i < n$. We have already argued that vertex t_i must have been updated according to Path 2 during procedure ComputeCost and that vertex $\textit{parent}(t)$ is in the closed list. Therefore, $\Phi(t_{i+1}, \textit{parent}(t), t_i) \in [\textit{lb}(t_{i+1}), \textit{ub}(t_{i+1})]$ according to Lemma 9. There are no other ways in which the angle range of a vertex can change.

It follows that $[t_0, \ldots, t_n] = G_p(t, \textit{parent}(t))$ and the property holds. We know that the property continues to hold because the g-value, the parent, the local parent and the angle range of vertices in the closed list are never modified according to Lemmata 11 and 12.

□

Property D.2.4.3. *(At any point during the execution of procedure ComputeShortestPath:) For any vertex $t \in V$ that is in the open or closed lists, $G_p(t) = [t_0 = t, t_1, \ldots, t_n = s_{\text{start}}]$ is well defined. Furthermore, for all $1 \leq i \leq n$, $t_i \in G_p(t)$ is in the closed list.*

Proof. We prove by induction on the ordered sequence of vertices in $G_p(t)$ that the property holds. The property holds initially for any vertex $t \in V$ that is in the open or closed lists because it must hold at the start of any call to procedure ComputeShortestPath due to our induction assumption regarding the Core Properties, specifically Property 3. We now show that the statement continues to hold whenever a vertex changes its local parent or its membership in the open or closed lists. Once a vertex is in the open or closed lists, it continues to be a member (Lemmata 11 and 13). A vertex can become a member of the open or closed lists only when procedure ComputeShortestPath expands some vertex t and updates the local parent of a visible neighbor t' of vertex t in procedure UpdateVertex. Vertex t' is not in the closed list due to Line 168. Vertex t is

in the closed list due to Line 166 and thus $G_p(t)$ has all of the required properties according to our induction assumption. If procedure UpdateVertex updates vertex t' then the property continues to hold since the local parent of vertex t' is vertex t when vertex t' is updated according to either Path 1 or Path 2. Thus $G_p(t')$ has all of the required properties. Once vertex t' is in the open list Line 169 ensures that procedure InitializeVertex is never called. If vertex t' becomes a member of the closed list then Line 168 ensures that procedure InitializeVertex is never called. There are no other ways in which the local parent of a vertex can change. □

Property D.2.4.4. *(At any point during the execution of procedure ComputeShortestPath:) For any vertex $t \in V$ such that procedure InitializeVertex(t) has been called at some point during the execution of procedure Main, the g-value of vertex t is finite iff vertex t is in the open or closed lists.*

Proof. At the start of procedure ComputeShortestPath the property holds for any vertex $t \in V$ such that procedure InitializeVertex(t) has been called due to our induction assumption regarding the Core Properties, specifically Property 4. The property can become invalid if procedure InitializeVertex is called for the first time, if a vertex changes its membership in the open or closed lists or if a vertex changes its g-value. If procedure InitializeVertex is called for the first time then the vertex was not in the open or closed lists (Lines 168 and 169) and its g-value is set to infinity (Line 158) so the property holds. Now consider any vertex t that changes is membership in the open or closed lists or its g-value. We must show the following: **(1)** if vertex t becomes a member of the open or closed lists during procedure ComputeShortestPath then the g-value of vertex t is finite and **(2)** if the g-value of vertex t is set to a finite value during procedure ComputeShortestPath then vertex t becomes a member of the open or closed lists. Lemma 13 ensures that once a vertex is added to the open list it continues to be a member of the open or closed lists and Lines 168 and 169 ensure that the g-value of a vertex in the open or closed list is never set to infinity. Therefore, we do not examine the case in which a vertex is removed from the open or closed lists and we do not examine the case in which the g-value of a vertex is set to infinity.

- **(1)** Consider any vertex t that becomes a member of the open or closed lists during procedure ComputeShortestPath. From Property 8, we know that path $A_p(t)$ is a path from the start vertex to vertex t in reverse and the g-value of vertex t is its length. Therefore, the g-value of vertex t is finite because the number of vertices in V is finite.

- **(2)** Consider any vertex t such that the g-value of vertex t is set to a finite value during procedure ComputeShortestPath. If the g-value of vertex t is set to a finite value on Line 190 or 201 then it decreases according to Line 188 or 199, respectively. Thus, the condition on Line 24 is satisfied and vertex t becomes a member of the open list on Line 27 (if it was not a member already). Finally, the g-value of vertex t can never increase on Lines 190 and 201 due to Lines 188 and 199, respectively.

□

Property D.2.4.5. *(At any point during the execution of procedure ComputeShortestPath:) For any vertex $t \in V$ that is in the closed list, the visible neighbors of vertex t are in the open or closed lists.*

Proof. The property holds for any vertex $t \in V$ that is in the closed list at the start of procedure ComputeShortestPath due to our induction assumption regarding the Core Properties, specifically Property 5. Consider any vertex t that is added to the closed list during procedure ComputeShortestPath. Immediately after vertex t was added to the closed list the visible neighbors of vertex t must have been iterated over due to Line 167. If some visible neighbor t' of vertex t was not already in the open or closed lists then the g-value of vertex t' is infinity immediately prior to Line 23 due to Lines 168 and 169, which ensures that the condition on Line 24 must be satisfied. This follows from the fact that either the condition on Line 199 or 188 must be satisfied. In the case of Line 199 we know that vertex t is in the closed list and therefore Property 4 ensures that the g-value of vertex t is finite and thus $g(t) + c(t, t') < g(t') = \infty$ (procedure InitializeVertex must have been called on any vertex in the open or closed lists). In the case of Line 188 we know that vertex t is in the closed list and therefore from Properties 2 and 3 together we know that the parent of vertex t is in the closed list. Therefore, from Property 4, it follows that the g-value of the parent of vertex t is finite and thus $g(\textit{parent}(t)) + c(\textit{parent}(t), t') < g(t') = \infty$ (procedure InitializeVertex must have been called on any vertex in the open or closed lists). Therefore vertex t' must be updated according to either Path 1 or Path 2 and, in either case, the g-value of vertex t' decreases and thus the condition on Line 24 is satisfied and vertex t' is added to the open list (if it was not a member already). Finally, Lemma 13 ensures that once a vertex is added to the open list it continues to be a member of the open or closed lists and thus the property holds. □

Property D.2.4.6. *(At any point during the execution of procedure ComputeShortestPath:) For any vertex $t \in V \setminus \{s_{\text{start}}\}$ that is in the open or closed lists and whose $\angle(t, \text{parent}(t))$ is a multiple of 45 degrees, the parent of vertex t is a visible neighbor of vertex t.*

Proof. The property holds for any vertex $t \in V \setminus \{s_{start}\}$ that is in the open or closed lists at the start of procedure ComputeShortestPath due to our induction assumption regarding the Core Properties, specifically Property 6. Consider any vertex t, other than the start vertex, that was added to the open list during procedure ComputeShortestPath. It is possible that vertex t is being added to the open list after having been removed on Line 26. A vertex can only be added to the open list if it was modified during a call to procedure UpdateVertex on Line 171 and thus the property holds according to Lemma 10. There are no other ways in which a vertex in the open list can change its parent. Now consider any vertex t that was added to the closed list during procedure ComputeShortestPath. A vertex can only be added to the closed list on Line 166 after being removed from the open list on Line 165 and we just showed that the property holds for any vertex in the open list. Therefore, since vertex t is added to the closed list on Line 166 immediately after it is removed from the open list, it must be the case that the property holds for any vertex in the closed list. Finally, Lemma 12 ensures that this property continues to hold for the remainder of procedure ComputeShortestPath. □

Property D.2.4.7 (Line-of-Sight Property)**.** *(At any point during the execution of procedure ComputeShortestPath:) For any vertex $t \in V$ that is in the open or closed lists, vertex t has line-of-sight to its parent.*

Proof. The property holds for any vertex $t \in V$ that is in the open or closed lists at the start of procedure ComputeShortestPath due to our induction assumption regarding the Core Properties, specifically Property 7. Consider any vertex t that was added to the open list during procedure ComputeShortestPath. It is possible that vertex t is being added to the open list after having been

removed on Line 26. A vertex can only be added to the open list if it was modified during a call to procedure UpdateVertex on Line 171 and thus the property holds according to Lemma 10. There are no other ways in which a vertex in the open list can change its parent. Now consider any vertex t that was added to the closed list during procedure ComputeShortestPath. A vertex can only be added to the closed list on Line 166 after being removed from the open list on Line 165 and we just showed that the property holds for any vertex in the open list. Therefore, since vertex t is added to the closed list on Line 166 immediately after it is removed from the open list, it must be the case that the property holds for any vertex in the closed list. Finally, Lemma 12 ensures that this property continues to hold for the remainder of procedure ComputeShortestPath. □

Property D.2.4.8. *(At any point during the execution of procedure ComputeShortestPath:) For any vertex $t \in V$ that is in the open or closed lists, $A_p(t) = [t_0 = t, t_1, \ldots, t_n = s_{\text{start}}]$ is well defined and thus is an unblocked path from the start vertex to vertex t in reverse and the g-value of vertex t is its length.*

Proof. We prove by induction on the ordered sequence of vertices in $A_p(t)$ that the property holds. The property holds initially for any vertex $t \in V$ that is in the open or closed lists because it must hold at the start of procedure ComputeShortestPath due to our induction assumption regarding the Core Properties, specifically Property 8. We now show that the statement continues to hold whenever a vertex changes either its g-value and parent or its membership in the open or closed lists. Once a vertex is a member of the open or closed lists, it continues to be a member (Lemmata 11 and 13). A vertex can become a member of the open or closed lists only when procedure ComputeShortestPath expands some vertex t and updates the g-value and parent of a visible neighbor t' of vertex t in procedure UpdateVertex. Vertex t' is not in the closed list due to Line 168. Vertex t is in the closed list due to Line 166 and its parent is in the closed list according to Properties 2 and 3 together. Thus, $A_p(t)$ (or $A_p(\mathit{parent}(t))$) is an unblocked path from the start vertex to vertex t (or its parent, respectively) in reverse, and the g-value of vertex t is its length (or the g-value of its parent, respectively) according to the induction assumption. If procedure UpdateVertex updates vertex t' then we know that the parent of vertex t' is either vertex t or the parent of vertex t and, in either case, the property continues to hold according to Lemma 10 which ensures that vertex t' has line-of-sight to its parent and that $g(t') = g(\mathit{parent}(t')) + c(\mathit{parent}(t'), t')$. Furthermore, since vertex t' is in the open list and the parent of vertex t' is in the closed list, no vertex can have vertex t' as its parent and thus it cannot be the case that the g-value of a vertex no longer represents the length of the path from the start vertex to that vertex because vertex t' changed its g-value and parent (this also ensures that a cycle can never be introduced). Thus, $A_p(t')$ has all of the required properties and thus is an unblocked path from the start vertex to vertex t' in reverse, and the g-value of vertex t' is its length. The only other place where the g-value and parent of a vertex can be modified is during procedure InitializeVertex. Once vertex t' is in the open list Line 169 ensures that procedure InitializeVertex is never called. If vertex t' becomes a member of the closed list then Line 168 ensures that procedure InitializeVertex is never called. There are no other ways in which the g-value and parent of a vertex can change. □

D.2.4.3 Completeness and Correctness

In this section, we show that if the Core Properties hold at the start of procedure ComputeShortestPath then procedure ComputeShortestPath is complete and correct.

Theorem 8. *Procedure ComputeShortestPath terminates and path extraction retrieves an unblocked path from the start vertex to the goal vertex, if such a path exists. Otherwise, procedure ComputeShortestPath terminates and reports that no unblocked path exists.*

Proof. The following properties prove the theorem. Their proofs utilize the fact that procedure ComputeShortestPath terminates iff the open list is empty or it is about to expand a vertex whose f-value is greater than or equal to the f-value of the goal vertex and the fact that procedure InitializeVertex must have been called on the goal vertex (Line 152) and any vertex in the open or closed lists.

- Property A: Procedure ComputeShortestPath terminates. It iterates until it either is about to remove a vertex from the open list whose f-value is greater than or equal to that of the goal vertex, or until the open list is empty (Line 164). During each iteration, exactly one vertex is removed from the open list (Line 165) and added to the closed list (Line 166) which ensures that it can never again be added to the open list according to Lemma 11. Since the number of vertices is finite, the open list eventually becomes empty and procedure ComputeShortestPath has to terminate if it has not already terminated because it was about to expand a vertex whose f-value is greater than or equal to that of the goal vertex.

- Property B: If procedure ComputeShortestPath terminates because the open list is empty, then there does not exist an unblocked path from the start vertex to the goal vertex. We prove the contrapositive. Assume there exists an unblocked path from the start vertex to the goal vertex. We prove by contradiction that procedure ComputeShortestPath does not terminate because the open list is empty. Thus, assume by contradiction that procedure ComputeShortestPath terminates because the open list is empty. Then, there exists an unblocked grid path $[s_0 = s_{start}, s_1, \ldots, s_n = s_{goal}]$ from the start vertex to the goal vertex according to Lemma 3. Choose vertex s_i to be the first vertex on the path that is not in the closed list when procedure ComputeShortestPath terminates. The goal vertex is not in the closed list when procedure ComputeShortestPath terminates since procedure ComputeShortestPath would otherwise have terminated when it was about to expand the goal vertex. Thus, vertex s_i exists. Vertex s_i is not the start vertex since the start vertex must be in the closed list according to Property 1 and the fact that procedure ComputeShortestPath terminated because the open list is empty. Thus, vertex s_i has a predecessor that is a visible neighbor of vertex s_i on the path. This predecessor is in the closed list when procedure ComputeShortestPath terminates since vertex s_i is the first vertex on the path that is not in the closed list when procedure ComputeShortestPath terminates. Since vertex s_i has a predecessor in the closed list, vertex s_i must be in the open list according to Property 5 and the fact that it is not in the closed list. Thus, vertex s_i is still in the open list when procedure ComputeShortestPath terminates. But then procedure ComputeShortestPath could not have terminated because the open list is empty, which is a contradiction.

- Property C: If procedure ComputeShortestPath terminates because it is about to expand a vertex whose f-value is greater than or equal to the f-value of the goal vertex, then path extraction retrieves an unblocked path from the start vertex to the goal vertex in reverse. If procedure ComputeShortestPath terminates because it is about to expand a vertex whose f-value is greater than or equal to the f-value of the goal vertex then the g-value (and thus also the f-value) of the expanded vertex is finite according to Property 4 and thus the f-value

(and thus also the g-value) of the goal vertex is also finite. If the g-value of the goal vertex is finite then the goal vertex must be in the open or closed lists according to Property 4 and if the goal vertex is in the open or closed lists then path extraction retrieves an unblocked path from the start vertex to the goal vertex in reverse according to Property 8 when following parents from the goal vertex to the start vertex.

□

D.2.4.4 Local Parent Path Property

In this section, we show that the Local Parent Path Property holds.

Theorem 6. *(At any point during the execution of procedure ComputeShortestPath and at the conclusion of any iteration of the foreach loop on Line 210:) The local parent path* $G_p(s, \text{parent}(s))$ *of any vertex* $s \in V$ *that is in the open or closed lists contains at least one corner of each grid cell that the path segment* $\overline{s, \text{parent}(s)}$ *traverses.*

The Core Properties used to prove Theorem 6 hold at any point during the execution of procedure ComputeShortestPath (as we saw in the previous section) and at the conclusion of any iteration of the foreach loop on Line 210 (as we show in Section D.2.5).

We use the following helper lemma:

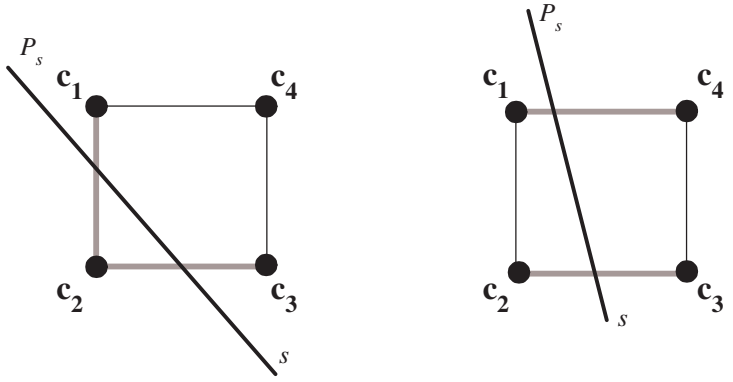
Lemma 14. *(At any point during the execution of procedure ComputeShortestPath and at the conclusion of any iteration of the foreach loop on Line 210:) Consider any vertex* $s \in V$ *that is in the open or closed lists and the sequence of pairwise distinct vertices* $G_p(s, \text{parent}(s)) = [v_0, v_1, \ldots, v_n]$. *For all* $0 \leq i < j \leq n$, $\Phi(v_j, \text{parent}(s), v_i) \in [\text{lb}(v_j), \text{ub}(v_j)]$.

Proof. Consider any vertex $s \in V$ that is in the open or closed lists. Let $G_p(s, \textit{parent}(s)) = [v_0, v_1, \ldots, v_n]$ according to Property 2. We want to show that $\Phi(v_j, \textit{parent}(s), v_i) \in [\textit{lb}(v_j), \textit{ub}(v_j)]$ for all $0 \leq i < j \leq n$. We do this using **Process A** which is defined as follows: If $i = n - 1$ then the lemma holds according to Property 2. If $0 \leq i < n - 1$ then there must exist three consecutive vertices v_i, v_{i+1} and v_{i+2} in $G_p(s, \textit{parent}(s))$. We know that $\Phi(v_{i+1}, \textit{parent}(s), v_i) \in [\textit{lb}(v_{i+1}), \textit{ub}(v_{i+1})]$ according to Property 2. Thus, it must be the case that $\Phi(v_{i+1}, \textit{parent}(s), v_i) + \delta \in [\textit{lb}(v_{i+1}) + \delta, \textit{ub}(v_{i+1}) + \delta]$ where δ is any angle. Assume $\delta = \Phi(v_{i+2}, \textit{parent}(s), v_{i+1})$. We know that $\Phi(v_{i+2}, \textit{parent}(s), v_i) = \Phi(v_{i+1}, \textit{parent}(s), v_i) + \delta$. We also know that the angle range of v_{i+1} was intersected with the angle range of v_{i+2} when vertex v_{i+1} inherited parent $\textit{parent}(s)$ from v_{i+2} (Lines 195 and 196). Thus, $\textit{lb}(v_{i+1}) \geq \textit{lb}(v_{i+2}) - \Phi(v_{i+2}, \textit{parent}(s), v_{i+1})$ and $\textit{ub}(v_{i+1}) \leq \textit{ub}(v_{i+2}) - \Phi(v_{i+2}, \textit{parent}(s), v_{i+1})$ or equivalently, $\textit{lb}(v_{i+2}) \leq \textit{lb}(v_{i+1}) + \Phi(v_{i+2}, \textit{parent}(s), v_{i+1})$ and $\textit{ub}(v_{i+2}) \geq \textit{ub}(v_{i+1}) + \Phi(v_{i+2}, \textit{parent}(s), v_{i+1})$. Therefore, we know that $\Phi(v_{i+2}, \textit{parent}(s), v_i) \in [\textit{lb}(v_{i+2}), \textit{ub}(v_{i+2})]$.

We can use Process A repeatedly to show that $\Phi(v_j, \textit{parent}(s), v_i) \in [\textit{lb}(v_j), \textit{ub}(v_j)]$ for any two vertices $v_i, v_j \in G_p(s, \textit{parent}(s))$ where $0 \leq i < j \leq n$. Thus, it follows that for all $0 \leq i < j \leq n$, $\Phi(v_j, \textit{parent}(s), v_i) \in [\textit{lb}(v_j), \textit{ub}(v_j)]$. □

We use the following notation and definitions:

For brevity, let $P_x = \textit{parent}(x)$ and $Q_p(s) = G_p(s, P_s)$. $\overline{s, P_s}$ intersects a line a' before a line (or a point) b', if when following $\overline{s, P_s}$ from vertex P_s to vertex s, we encounter the intersection of



(a) Orthogonal Case (b) Parallel Case

Figure D.2: Orthogonal and Parallel Cases

$\overline{s, P_s}$ and a' before we encounter the intersection of $\overline{s, P_s}$ and b'. Similarly, $\overline{s, P_s}$ intersects a line a' after a line b', if when following $\overline{s, P_s}$ from vertex P_s to vertex s, we encounter the intersection of $\overline{s, P_s}$ and b' before we encounter the intersection of $\overline{s, P_s}$ and a'.

Proof. The theorem trivially holds if vertex s is the start vertex according to Property 1 which ensures that the parent of the start vertex is the start vertex. Consider any vertex $s \in V \setminus \{s_{start}\}$ that is in the open or closed lists and the path segment $\overline{s, P_s}$.

If $\angle(s, P_s)$ is a multiple of 45 degrees (and not a multiple of 90 degrees) then the theorem trivially holds according to Property 6. Property 6 ensures that, if $\angle(s, P_s)$ is a multiple of 45 degrees, then the parent of vertex s is a visible neighbor of vertex s and thus $\overline{s, P_s}$ traverses one grid cell. Vertex s is both a member of $Q_p(s)$ and a corner of the traversed grid cell. If $\angle(s, P_s)$ is a multiple of 90 degrees then $\overline{s, P_s}$ does not traverse any grid cells.

If $\angle(s, P_s)$ is not a multiple of 45 degrees then we consider two cases and, in each case, we use contradiction to show that the theorem holds.

- **Orthogonal Case:** $\overline{s, P_s}$ intersects two orthogonal sides of grid cell c with upper left corner c_1, lower left corner c_2, lower right corner c_3 and upper right corner c_4 (Figure D.2(a)). Due to symmetry, we only need to consider one of the eight cases in which $\overline{s, P_s}$ intersects two orthogonal sides of grid cell c. We consider the case in which $\overline{s, P_s}$ intersects $\overline{c_1, c_2}$ before it intersects $\overline{c_2, c_3}$. If $\overline{s, P_s}$ intersects a side of c and a corner of c then it could be considered either the orthogonal case or the parallel case, we consider it the former. $\overline{s, P_s}$ may intersect one of the two orthogonal sides at the endpoint of a side (that is not the common endpoint). For example, in Figure D.2(a), $\overline{s, P_s}$ may intersect $\overline{c_1, c_2}$ before it intersects c_3. In the orthogonal case, the two intersection points and the common endpoint of the two orthogonal sides (that is, c_2 in Figure D.2(a)) form a right triangle with one angle strictly greater than 45 degrees and one angle strictly less than 45 degrees.

- **Parallel Case:** $\overline{s, P_s}$ intersects two parallel sides of grid cell c with upper left corner c_1, lower left corner c_2, lower right corner c_3 and upper right corner c_4 (Figure D.2(b)). Due to symmetry, we only need to consider one of the four cases in which $\overline{s, P_s}$ intersects two parallel sides of grid cell c. We consider the case in which $\overline{s, P_s}$ intersects $\overline{c_1, c_4}$ before it

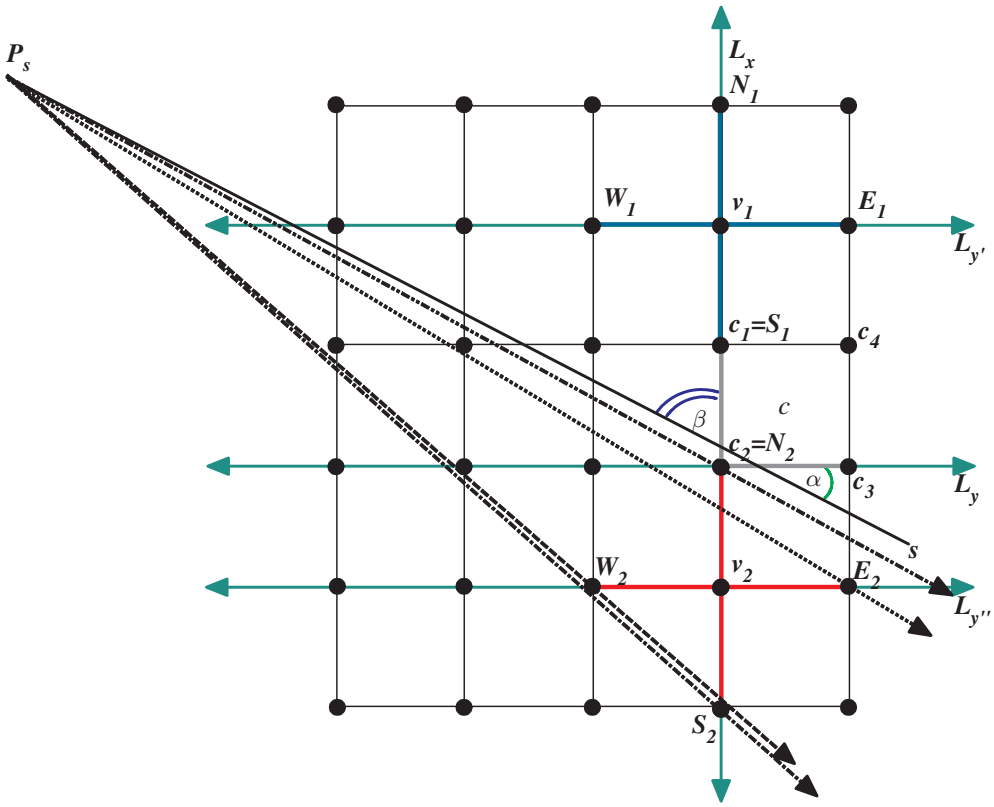


Figure D.3: Orthogonal Case ($\beta > 45$ degrees)

intersects $\overline{c_2, c_3}$. In the parallel case, the smaller angle formed by the intersection of $\overline{s, P_s}$ and either of the two parallel sides must be strictly greater than 45 degrees.

First, we examine the orthogonal case and then we examine the parallel case.

Orthogonal Case: Consider any path segment $\overline{s, P_s}$ that intersects two orthogonal sides $\overline{c_1, c_2}$ and $\overline{c_2, c_3}$ of some grid cell c with upper left corner c_1, lower left corner c_2, lower right corner c_3 and upper right corner c_4 (Figures D.3 and D.4). Let L_x be the vertical line that passes through c_1 and c_2 and L_y be the horizontal line that passes through c_2 and c_3 (Figures D.3 and D.4). Consider the smaller angle β of the two angles formed by $\overline{s, P_s}$ and L_x and the smaller angle α of the two angles formed by $\overline{s, P_s}$ and L_y. We know that $\overline{s, P_s}$ and the two orthogonal sides of c that are intersected by $\overline{s, P_s}$ must form a right triangle. Therefore, we know that $\beta + \alpha = 90$ degrees. Furthermore, we know that $\max(\beta, \alpha)$ is strictly greater than 45 degrees and that $\min(\beta, \alpha)$ is strictly less than 45 degrees.

First, consider the case where the angle that is strictly greater than 45 degrees is defined by $\overline{s, P_s}$ and L_x (that is, β in Figure D.3). In order for our contradictory assumption to hold there must exist some vertex $s' \in Q_p(s)$ on L_x that is not a corner of c such that $\Phi(s', P_s, s) \in [lb(s'), ub(s')]$. If some vertex $s' \in Q_p(s)$ on L_x did not exist then vertex s could not have parent P_s. If vertex s' is a corner of c then our assumption is not contradictory. If vertex s' inherits vertex P_s then $\Phi(s', P_s, s) \in [lb(s'), ub(s')]$ according to Lemma 14 (where $s' = v_j$ and $s = v_i$). The angle range of vertex s', $[lb(s'), ub(s')]$ can be at most as wide as the angle

range defined by vertex P_s, vertex s', and the four crossbar neighbors of vertex s' (Lines 195 and 196) (Section 6.4.2). We consider two vertices $s' = v_1$ or $s' = v_2$ on L_x, where vertex v_1 is immediately to the north of c_1 and vertex v_2 is immediately to the south of c_2. If we can show that $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ and $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$, then we have demonstrated a contradiction (neither vertex v_1 nor vertex v_2 is a corner of c, $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ and $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$ and thus vertex s can only inherit parent vertex P_s if Lemma 14 doesn't hold, contradiction). We do not need to consider any other vertices on L_x as potential values for vertex s'. After we demonstrate a contradiction for the vertices v_1 and v_2 we show that any vertex on L_x north of vertex v_1 and any vertex on L_x south of vertex v_2 also demonstrates a contradiction. Let the four crossbar neighbors to the north, east, south and west of vertex v_2 be $N_2 = c_2$, E_2, S_2 and W_2, respectively (red crossbar in Figure D.3) and the four crossbar neighbors to the north, east, south and west of vertex v_1 be N_1, E_1, $S_1 = c_1$ and W_1, respectively (blue crossbar in Figure D.3). Let $L_{y'}$ be the horizontal line that passes through W_1 and E_1 (and vertex v_1) and $L_{y''}$ be the horizontal line that passes through W_2 and E_2 (and vertex v_2). We say that $\overline{s, P_s}$ travels east faster than it travels south, if when following $\overline{s, P_s}$ from vertex P_s to vertex s, it travels less than one unit (of arbitrary length) south for every unit it travels east. Similarly, we say that, in the reverse direction, $\overline{s, P_s}$ travels west faster than it travels north, if when following $\overline{s, P_s}$ from vertex s to vertex P_s, it travels less than one unit (of arbitrary length) north for every unit it travels west.

- v_2**:** Since $\overline{s, P_s}$ is counter-clockwise of $\overline{v_2, P_s}$ we only need to show that $\overline{s, P_s}$ is also counter-clockwise of $\overline{c_2 = N_2, P_s}$ (dash-dot-dot line in Figure D.3), $\overline{E_2, P_s}$ (dotted line in Figure D.3), $\overline{S_2, P_s}$ (dash-dot line in Figure D.3) and $\overline{W_2, P_s}$ (dashed line in Figure D.3) as that would mean that $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$ where $lb(v_2) \geq \min_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$, $ub(v_2) \leq \max_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$ and $nghbr_4(v_2) = \{N_2, E_2, S_2, W_2\}$. We show that $\Phi(v_2, P_s, s) > \max_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$, which implies $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$.
  - N_2**:** Since $\overline{s, P_s}$ intersects $\overline{c_1, c_2 = N_2}$ we know that $\overline{N_2, P_s}$ intersects L_x south of where $\overline{s, P_s}$ intersects L_x. $\overline{s, P_s}$ cannot intersect c_2 because if it did then it would not traverse grid cell c. $\overline{s, P_s}$ and $\overline{N_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_x south of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_2, P_s}$ and thus $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, N_2)$.
  - E_2**:** Since β is strictly greater than 45 degrees, α is strictly less than 45 degrees and thus $\overline{s, P_s}$ travels east faster than it travels south. Therefore, after $\overline{s, P_s}$ intersects $\overline{c_2 = N_2, c_3}$ it intersects $L_{y''}$ east of where $\overline{E_2, P_s}$ intersects $L_{y''}$. $\overline{s, P_s}$ and $\overline{E_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y''}$ west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{E_2, P_s}$ and thus $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, E_2)$.
  - S_2**:** Since $\overline{s, P_s}$ intersects $\overline{c_1, c_2 = N_2}$ we know that $\overline{S_2, P_s}$ intersects L_x south of where $\overline{s, P_s}$ intersects L_x. $\overline{s, P_s}$ and $\overline{S_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_x south of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{S_2, P_s}$ and thus $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, S_2)$.
  - W_2**:** Since β is strictly greater than 45 degrees, α is strictly less than 45 degrees and thus $\overline{s, P_s}$ travels east faster than it travels south. Therefore, after $\overline{s, P_s}$ intersects

$\overline{c_2 = N_2, c_3}$ it intersects $L_{y''}$ east of where $\overline{W_2, P_s}$ intersects $L_{y''}$. $\overline{s, P_s}$ and $\overline{W_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y''}$ west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{W_2, P_s}$ and thus $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, W_2)$.

Since $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, N_2)$, $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, E_2)$, $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, S_2)$ and $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, W_2)$, it must be the case that $\Phi(v_2, P_s, s) > \max_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$ and thus $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$.

Now, consider any vertex v_s on L_x that is south of vertex v_2 and let its four cross-bar neighbors be N_s, E_s, S_s and W_s. N_s, E_s, S_s and W_s are all directly south of N_2, E_2, S_2 and W_2, respectively. Therefore we know that $\overline{N_2, P_s}$, $\overline{E_2, P_s}$, $\overline{S_2, P_s}$ and $\overline{W_2, P_s}$ are counter-clockwise of $\overline{N_s, P_s}$, $\overline{E_s, P_s}$, $\overline{S_s, P_s}$ and $\overline{W_s, P_s}$, respectively. Since we just showed that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_2, P_s}$, $\overline{E_2, P_s}$, $\overline{S_2, P_s}$ and $\overline{W_2, P_s}$, it must also be the case that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_s, P_s}$, $\overline{E_s, P_s}$, $\overline{S_s, P_s}$ and $\overline{W_s, P_s}$. Therefore, since $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, N_s)$, $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, E_s)$, $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, S_s)$ and $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, W_s)$, it must be the case that $\Phi(v_s, P_s, s) > \max_{s'' \in nghbr_4(v_s)} \Phi(v_s, P_s, s'')$ and thus $\Phi(v_s, P_s, s) \notin [lb(v_s), ub(v_s)]$.

- v_1**:** Since $\overline{s, P_s}$ is clockwise of $\overline{v_1, P_s}$ we only need to show that $\overline{s, P_s}$ is also clockwise of $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$ as that would mean that $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ where $lb(v_1) \geq \min_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_s, s'')$, $ub(v_1) \leq \max_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_s, s'')$ and $nghbr_4(v_1) = \{N_1, E_1, S_1, W_1\}$. We show that $\Phi(v_1, P_s, s) < \min_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_s, s'')$, which implies $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$.

  - N_1**:** Since $\overline{s, P_s}$ intersects $\overline{c_1 = S_1, c_2}$ we know that $\overline{N_1, P_s}$ intersects L_x north of where $\overline{s, P_s}$ intersects L_x. $\overline{s, P_s}$ and $\overline{N_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_x north of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{N_1, P_s}$ and thus $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, N_1)$.
  - E_1**:** Since β is strictly greater than 45 degrees, α is strictly less than 45 degrees and thus, in the reverse direction, $\overline{s, P_s}$ travels west faster than it travels north. Therefore, before $\overline{s, P_s}$ intersects $\overline{c_1 = S_1, c_2}$ it intersects $L_{y'}$ west of where $\overline{E_1, P_s}$ intersects $L_{y'}$. $\overline{s, P_s}$ and $\overline{E_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y'}$ east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{E_1, P_s}$ and thus $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, E_1)$.
  - S_1**:** Since $\overline{s, P_s}$ intersects $\overline{c_1 = S_1, c_2}$ we know that $\overline{S_1, P_s}$ intersects L_x north of where $\overline{s, P_s}$ intersects L_x. $\overline{s, P_s}$ cannot intersect c_1 because, if it did, then it could not intersect $\overline{c_2, c_3}$ because β is strictly greater than 45 degrees and thus α is strictly less than 45 degrees. $\overline{s, P_s}$ and $\overline{S_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_x north of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{S_1, P_s}$ and thus $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, S_1)$.
  - W_1**:** Since β is strictly greater than 45 degrees, α is strictly less than 45 degrees and thus, in the reverse direction, $\overline{s, P_s}$ travels west faster than it travels north. Therefore, before $\overline{s, P_s}$ intersects $\overline{c_1 = S_1, c_2}$ it intersects $L_{y'}$ west of where $\overline{W_1, P_s}$ intersects

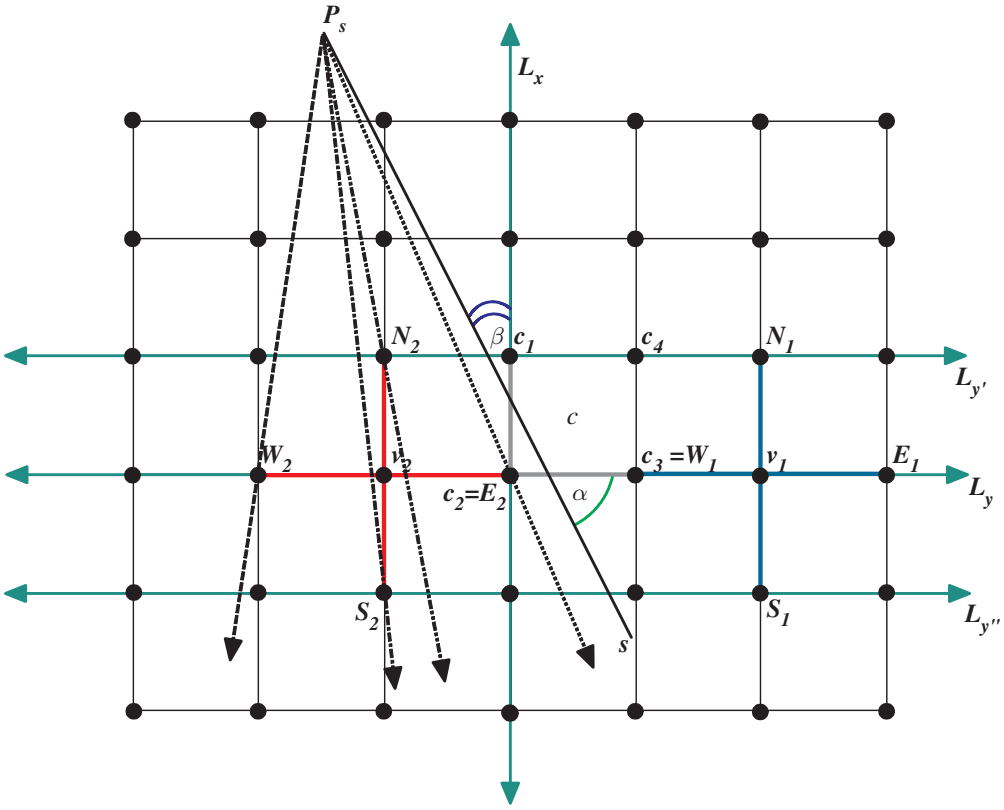


Figure D.4: Orthogonal Case ($\alpha > 45$ degrees)

$L_{y'}$. $\overline{s, P_s}$ and $\overline{W_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y'}$ east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{W_1, P_s}$ and thus $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, W_1)$.

Since $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, N_1)$, $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, E_1)$, $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, S_1)$ and $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, W_1)$, it must be the case that $\Phi(v_1, P_s, s) < \min_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_s, s'')$ and thus $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$.

Now, consider any vertex $v_{n'}$ on L_x that is north of vertex v_1 and let its four crossbar neighbors be $N_{n'}$, $E_{n'}$, $S_{n'}$ and $W_{n'}$. $N_{n'}$, $E_{n'}$, $S_{n'}$ and $W_{n'}$ are all directly north of N_1, E_1, S_1 and W_1, respectively. Therefore we know that $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$ are clockwise of $\overline{N_{n'}, P_s}$, $\overline{E_{n'}, P_s}$, $\overline{S_{n'}, P_s}$ and $\overline{W_{n'}, P_s}$, respectively. Since we just showed that $\overline{s, P_s}$ is clockwise of $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$, it must also be the case that $\overline{s, P_s}$ is clockwise of $\overline{N_{n'}, P_s}$, $\overline{E_{n'}, P_s}$, $\overline{S_{n'}, P_s}$ and $\overline{W_{n'}, P_s}$. Therefore, since $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, N_{n'})$, $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, E_{n'})$, $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, S_{n'})$ and $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, W_{n'})$, it must be the case that $\Phi(v_{n'}, P_s, s) < \min_{s'' \in nghbr_4(v_{n'})} \Phi(v_{n'}, P_s, s'')$ and thus $\Phi(v_{n'}, P_s, s) \notin [lb(v_{n'}), ub(v_{n'})]$.

Contradiction.

Second, consider the case where the angle that is strictly greater than 45 degrees is defined by $\overline{s, P_s}$ and L_y (that is, α in Figure D.4).[2] In order for our contradictory assumption to hold there must exist some vertex $s' \in Q_p(s)$ on L_y that is not a corner of c such that $\Phi(s', P_s, s) \in [lb(s'), ub(s')]$. If some vertex $s' \in Q_p(s)$ on L_y did not exist then vertex s could not have parent P_s. If vertex s' is a corner of c then our assumption is not contradictory. If vertex s' inherits vertex P_s then $\Phi(s', P_s, s) \in [lb(s'), ub(s')]$ according to Lemma 14 (where $s' = v_j$ and $s = v_i$). The angle range of s', $[lb(s'), ub(s')]$ can be at most as wide as the angle range defined by vertex P_s, vertex s', and the four crossbar neighbors of vertex s' (Lines 195 and 196) (Section 6.4.2). We consider the two vertices $s' = v_1$ or $s' = v_2$ on L_y, where vertex v_1 is immediately to the east of c_3 and vertex v_2 is immediately to the west of c_2. If we can show that $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ and $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$, then we have demonstrated a contradiction (neither vertex v_1 nor vertex v_2 is a corner of c, $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ and $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$ and thus vertex s can only inherit parent vertex P_s if Lemma 14 doesn't hold, contradiction). We do not need to consider any other vertices on L_y as potential values for vertex s'. After we demonstrate a contradiction for the vertices v_1 and v_2 we show that any vertex on L_y east of vertex v_1 and any vertex on L_y west of vertex v_2 also demonstrates a contradiction. Let the four crossbar neighbors to the north, east, south and west of vertex v_2 be N_2, $E_2 = c_2$, S_2 and W_2, respectively (red crossbar in Figure D.4) and the crossbar neighbors to the north, east, south and west of vertex v_1 be N_1, E_1, S_1 and $W_1 = c_3$, respectively (blue crossbar in Figure D.4). Let $L_{y'}$ be the horizontal line that passes through N_1 and N_2 and $L_{y''}$ be the horizontal line that passes through S_1 and S_2. We say that $\overline{s, P_s}$ travels south faster than it travels east if, when following $\overline{s, P_s}$ from vertex P_s to vertex s, it travels less than one unit (of arbitrary length) east for every unit it travels south. Similarly, we say that, in the revere direction, $\overline{s, P_s}$ travels north faster than it travels west if, when following $\overline{s, P_s}$ from vertex s to vertex P_s, it travels less than one unit (of arbitrary length) west for every unit it travels north.

- v_2**:** Since $\overline{s, P_s}$ is counter-clockwise of $\overline{v_2, P_s}$ we only need to show that $\overline{s, P_s}$ is also counter-clockwise of $\overline{N_2, P_s}$ (dash-dot-dot line in Figure D.4), $\overline{c_2 = E_2, P_s}$ (dotted line in Figure D.4), $\overline{S_2, P_s}$ (dash-dot line in Figure D.4) and $\overline{W_2, P_s}$ (dashed line in Figure D.4) as that would mean that $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$ where $lb(v_2) \geq \min_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$, $ub(v_2) \leq \max_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$ and $nghbr_4(v_2) = \{N_2, E_2, S_2, W_2\}$. We show that $\Phi(v_2, P_s, s) > \max_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$, which implies $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$.
  - N_2**:** Since α is strictly greater than 45 degrees, β is strictly less than 45 degrees and thus, in the reverse direction, $\overline{s, P_s}$ travels north faster than it travels west. Therefore, before $\overline{s, P_s}$ intersects $\overline{c_1, c_2 = E_2}$ it intersects $L_{y'}$ east of where $\overline{N_2, P_s}$ intersects $L_{y'}$. $\overline{s, P_s}$ and $\overline{N_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y'}$ west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_2, P_s}$ and thus $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, N_2)$.
  - E_2**:** Since $\overline{s, P_s}$ intersects $\overline{c_3, c_2 = E_2}$ we know that $\overline{E_2, P_s}$ intersects L_y west of where $\overline{s, P_s}$ intersects L_y. $\overline{s, P_s}$ cannot intersect c_2 because if it did then it would not traverse grid cell c. $\overline{s, P_s}$ and $\overline{E_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_y west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{E_2, P_s}$ and thus $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, E_2)$.

[2] This analysis is nearly identical to the analysis performed in the prior case.

- S_2: Since α is strictly greater than 45 degrees, β is strictly less than 45 degrees and thus $\overline{s, P_s}$ travels south faster than it travels east. Therefore, after $\overline{s, P_s}$ intersects $\overline{c_3, c_2 = E_2}$ it intersects $L_{y''}$ east of where $\overline{S_2, P_s}$ intersects $L_{y''}$. $\overline{s, P_s}$ and $\overline{S_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y''}$ west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{S_2, P_s}$ and thus $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, S_2)$.
- W_2: Since $\overline{s, P_s}$ intersects $\overline{c_3, c_2 = E_2}$ we know that $\overline{W_2, P_s}$ intersects L_y west of where $\overline{s, P_s}$ intersects L_y. $\overline{s, P_s}$ and $\overline{W_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_y west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{W_2, P_s}$ and thus $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, W_2)$.

Since $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, N_2)$, $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, E_2)$, $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, S_2)$ and $\Phi(v_2, P_s, s) > \Phi(v_2, P_s, W_2)$, it must be the case that $\Phi(v_2, P_s, s) > \max_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$ and thus $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$.

Now, consider any vertex v_s on L_y that is west of vertex v_2 and let its four cross-bar neighbors be N_s, E_s, S_s and W_s. N_s, E_s, S_s and W_s are all directly west of N_2, E_2, S_2 and W_2, respectively. Therefore we know that $\overline{N_2, P_s}$, $\overline{E_2, P_s}$, $\overline{S_2, P_s}$ and $\overline{W_2, P_s}$ are counter-clockwise of $\overline{N_s, P_s}$, $\overline{E_s, P_s}$, $\overline{S_s, P_s}$ and $\overline{W_s, P_s}$, respectively. Since we just showed that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_2, P_s}$, $\overline{E_2, P_s}$, $\overline{S_2, P_s}$ and $\overline{W_2, P_s}$, it must also be the case that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_s, P_s}$, $\overline{E_s, P_s}$, $\overline{S_s, P_s}$ and $\overline{W_s, P_s}$. Therefore, since $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, N_s)$, $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, E_s)$, $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, S_s)$ and $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, W_s)$, it must be the case that $\Phi(v_s, P_s, s) > \max_{s'' \in nghbr_4(v_s)} \Phi(v_s, P_s, s'')$ and thus $\Phi(v_s, P_s, s) \notin [lb(v_s), ub(v_s)]$.

- v_1: Since $\overline{s, P_s}$ is clockwise of $\overline{v_1, P_s}$ we only need to show that $\overline{s, P_s}$ is also clockwise of $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$ as that would mean that $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ where $lb(v_1) \geq \min_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_s, s'')$, $ub(v_1) \leq \max_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_s, s'')$ and $nghbr_4(v_1) = \{N_1, E_1, S_1, W_1\}$. We show that $\Phi(v_1, P_s, s) < \min_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_s, s'')$, which implies $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$.
  - N_1: Since α is strictly greater than 45 degrees, β is strictly less than 45 degrees and thus, in the reverse direction, $\overline{s, P_s}$ travels north faster than it travels west. Therefore, before $\overline{s, P_s}$ intersects $\overline{c_1, c_2}$ it intersects $L_{y'}$ west of where $\overline{N_1, P_s}$ intersects $L_{y'}$. $\overline{s, P_s}$ and $\overline{N_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y'}$ east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{N_1, P_s}$ and thus $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, N_1)$.
  - E_1: Since $\overline{s, P_s}$ intersects $\overline{c_2, c_3 = W_1}$ we know that $\overline{E_1, P_s}$ intersects L_y east of where $\overline{s, P_s}$ intersects L_y. $\overline{s, P_s}$ and $\overline{E_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_y east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{E_1, P_s}$ and thus $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, E_1)$.
  - S_1: Since α is strictly greater than 45 degrees, β is strictly less than 45 degrees and thus $\overline{s, P_s}$ travels south faster than it travels east. Therefore, after $\overline{s, P_s}$ intersects $\overline{c_2, c_3 = W_1}$ it intersects $L_{y''}$ west of where $\overline{S_1, P_s}$ intersects $L_{y''}$. $\overline{s, P_s}$ and $\overline{S_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y''}$ east

of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{S_1, P_s}$ and thus $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, S_1)$.

– W_1**:** Since $\overline{s, P_s}$ intersects $\overline{c_2, c_3 = W_1}$ we know that $\overline{W_1, P_s}$ intersects L_y east of where $\overline{s, P_s}$ intersects L_y. $\overline{s, P_s}$ cannot intersect c_3 because if it did then it could not intersect $\overline{c_1, c_2}$ because β is strictly less than 45 degrees and thus α is strictly greater than 45 degrees. $\overline{s, P_s}$ and $\overline{W_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_y east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{W_1, P_s}$ and thus $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, W_1)$.

Since $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, N_1)$, $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, E_1)$, $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, S_1)$ and $\Phi(v_1, P_s, s) < \Phi(v_1, P_s, W_1)$, it must be the case that $\Phi(v_1, P_s, s) < \min_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_s, s'')$ and thus $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$.

Now, consider any vertex $v_{n'}$ on L_y that is east of vertex v_1 and let its four crossbar neighbors be $N_{n'}$, $E_{n'}$, $S_{n'}$ and $W_{n'}$. $N_{n'}$, $E_{n'}$, $S_{n'}$ and $W_{n'}$ are all directly east of N_1, E_1, S_1 and W_1, respectively. Therefore we know that $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$ are clockwise of $\overline{N_{n'}, P_s}$, $\overline{E_{n'}, P_s}$, $\overline{S_{n'}, P_s}$ and $\overline{W_{n'}, P_s}$, respectively. Since we just showed that $\overline{s, P_s}$ is clockwise of $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$, it must also be the case that $\overline{s, P_s}$ is clockwise of $\overline{N_{n'}, P_s}$, $\overline{E_{n'}, P_s}$, $\overline{S_{n'}, P_s}$ and $\overline{W_{n'}, P_s}$. Therefore, since $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, N_{n'})$, $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, E_{n'})$, $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, S_{n'})$ and $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, W_{n'})$, it must be the case that $\Phi(v_{n'}, P_s, s) < \min_{s'' \in nghbr_4(v_{n'})} \Phi(v_{n'}, P_s, s'')$ and thus $\Phi(v_{n'}, P_s, s) \notin [lb(v_{n'}), ub(v_{n'})]$.

Contradiction.

Parallel Case: Consider any path segment $\overline{s, P_s}$ that intersects two parallel sides $\overline{c_1, c_4}$ and $\overline{c_2, c_3}$ of some grid cell c with upper left corner c_1, lower left corner c_2, lower right corner c_3 and upper right corner c_4 (Figure D.5). Let L_y be the horizonal line that passes through c_2 and c_3, $L_{y''}$ be the horizonal line that passes through c_1 and c_4 and $L_{y'}$ be the horizontal line immediately to the south of L_y (Figure D.5). Consider the smaller angle α of the two angles formed by $\overline{s, P_s}$ and L_y (Figure D.5). α is strictly greater than 45 degrees because $\overline{s, P_s}$ intersects $\overline{c_1, c_4}$ and then intersects $\overline{c_2, c_3}$. Without loss of generality, by rotation and reflection, we consider the case in which $\overline{s, P_s}$ travels south and east (Figure D.5), that is, as you follow $\overline{s, P_s}$ from vertex P_s to vertex s $\overline{s, P_s}$ travels south and east. In order for our contradictory assumption to hold there must exist some vertex $s' \in Q_p(s)$ on L_y that is not a corner of c and $\Phi(s', P_s, s) \in [lb(s'), ub(s')]$. If some vertex $s' \in Q_p(s)$ on L_y did not exist than s could not have parent P_s. If vertex s' is corner of c then our assumption is not contradictory. If s' inherits P_s then $\Phi(s', P_s, s) \in [lb(s'), ub(s')]$ according to Lemma 14 (where vertex $s' = v_j$ and vertex $s = v_i$). The angle range of vertex s', $[lb(s'), ub(s')]$ can be at most as wide as the angle range defined by vertex P_s, vertex s', and the four crossbar neighbors of s' (Lines 195 and 196) (Section 6.4.2). We consider the two vertices $s' = v_1$ or $s' = v_2$ on L_y, where vertex v_1 is immediately to the west of c_2 and vertex v_2 is immediately to the east of c_3. If we can show that $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ and $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$, then we have demonstrated a contradiction (neither vertex v_1 nor vertex v_2 is a corner of c, $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ and $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$ and thus vertex s can only inherit parent vertex P_s if Lemma 14 doesn't hold, contradiction). We do not need to consider any other vertices on L_y as potential values for vertex s'. After we

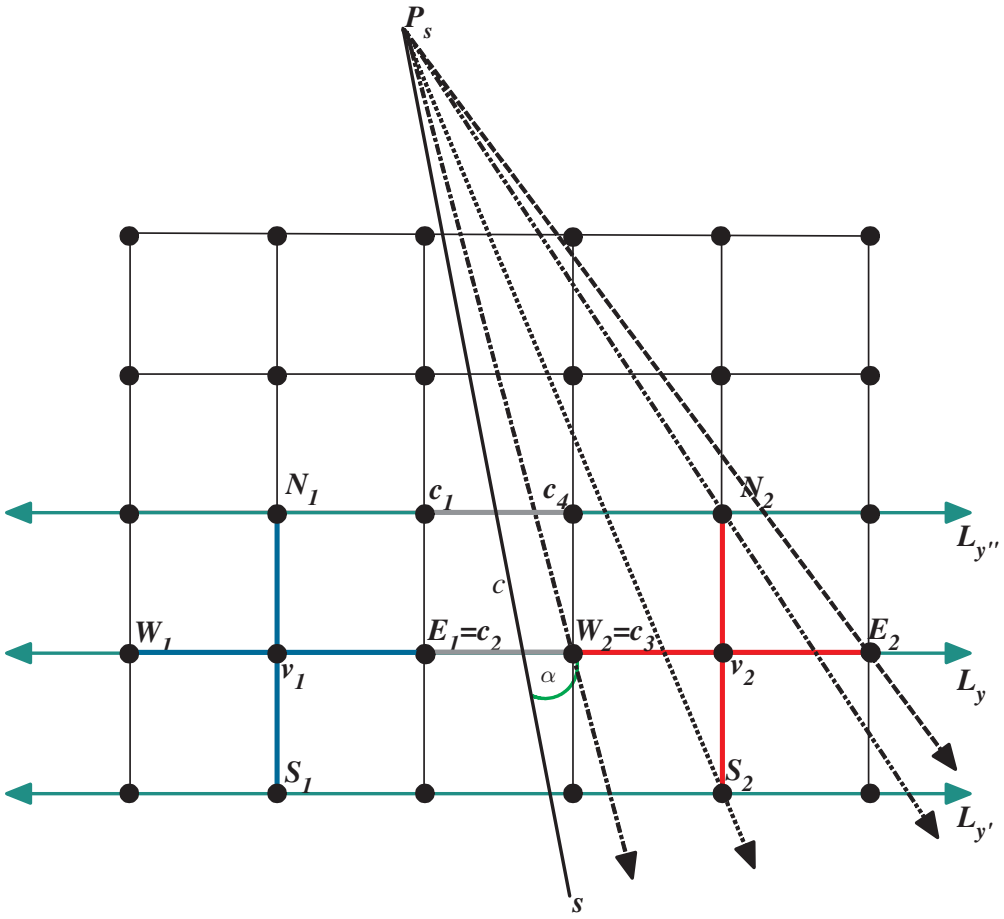


Figure D.5: Parallel Case

demonstrate a contradiction for the vertices v_1 and v_2 we show that any vertex on L_y west of vertex v_1 and any vertex on L_y east of vertex v_2 also demonstrates a contradiction. Let the four crossbar neighbors to the north, east, south and west of vertex v_2 be N_2, E_2, S_2 and $W_2 = c_3$, respectively (red crossbar in Figure D.5) and the four crossbar neighbors to the north, east, south and west of vertex v_1 be N_1, $E_1 = c_2$, S_1 and W_1, respectively (blue crossbar in Figure D.5). We say that $\overline{s, P_s}$ travels south faster than it travels east if when following $\overline{s, P_s}$ from vertex P_s to vertex s it travels less than one unit (of arbitrary length) east for every unit it travels south.

- v_2**:** Since $\overline{s, P_s}$ is clockwise of $\overline{v_2, P_s}$ we only need to show that $\overline{s, P_s}$ is also clockwise of $\overline{N_2, P_s}$ (dash-dot-dot line in Figure D.5), $\overline{E_2, P_s}$ (dashed line in Figure D.5), $\overline{S_2, P_s}$ (dotted line in Figure D.5) and $\overline{W_2, P_s}$ (dash-dot line in Figure D.5) as that would mean that $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$ where $lb(v_2) \geq \min_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$, $ub(v_2) \leq \max_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$ and $nghbr_4(v_2) = \{N_2, E_2, S_2, W_2\}$. We show that $\Phi(v_2, P_s, s) < \min_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$, which implies $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$.

– N_2: Since $\overline{s, P_s}$ intersects $\overline{c_1, c_4}$ we know that $\overline{s, P_s}$ intersects $L_{y''}$ west of where $\overline{N_2, P_s}$ intersects $L_{y''}$. $\overline{s, P_s}$ and $\overline{N_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y''}$ east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{N_2, P_s}$ and thus $\Phi(v_2, P_s, s) < \Phi(v_2, P_s, N_2)$.

– E_2: Since $\overline{s, P_s}$ intersects $\overline{c_2, c_3}$ we know that $\overline{s, P_s}$ intersects L_y west of where $\overline{E_2, P_s}$ intersects L_y. $\overline{s, P_s}$ and $\overline{E_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_y east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{E_2, P_s}$ and thus $\Phi(v_2, P_s, s) < \Phi(v_2, P_s, E_2)$.

– S_2: Since α is strictly greater than 45 degrees we know that $\overline{s, P_s}$ travels south faster than it travels east. Therefore, after $\overline{s, P_s}$ intersects $\overline{c_2, c_3}$, we know that $\overline{s, P_s}$ intersects $L_{y'}$ west of where $\overline{S_2, P_s}$ intersects $L_{y'}$. $\overline{s, P_s}$ and $\overline{S_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y'}$ east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{S_2, P_s}$ and thus $\Phi(v_2, P_s, s) < \Phi(v_2, P_s, S_2)$.

– W_2: Since $\overline{s, P_s}$ intersects $\overline{c_2, c_3}$ (exclusive of the endpoints) we know that $\overline{s, P_s}$ intersects L_y west of where $\overline{W_2, P_s}$ intersects L_y. $\overline{s, P_s}$ and $\overline{W_2, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_y east of the former. It follows that $\overline{s, P_s}$ is clockwise of $\overline{W_2, P_s}$ and thus $\Phi(v_2, P_s, s) < \Phi(v_2, P_s, W_2)$.

Since $\Phi(v_2, P_s, s) < \Phi(v_2, P_s, N_2)$, $\Phi(v_2, P_s, s) < \Phi(v_2, P_s, E_2)$, $\Phi(v_2, P_s, s) < \Phi(v_2, P_s, S_2)$ and $\Phi(v_2, P_s, s) < \Phi(v_2, P_s, W_2)$, it must be the case that $\Phi(v_2, P_s, s) < \min_{s'' \in nghbr_4(v_2)} \Phi(v_2, P_s, s'')$ and thus $\Phi(v_2, P_s, s) \notin [lb(v_2), ub(v_2)]$.

Now, consider any vertex $v_{n'}$ on L_y that is east of vertex v_2 and let its four cross-bar neighbors be $N_{n'}$, $E_{n'}$, $S_{n'}$ and $W_{n'}$. $N_{n'}$, $E_{n'}$, $S_{n'}$ and $W_{n'}$ are all directly east of N_2, E_2, S_2 and W_2, respectively. Therefore we know that $\overline{N_2, P_s}$, $\overline{E_2, P_s}$, $\overline{S_2, P_s}$ and $\overline{W_2, P_s}$ are clockwise of $\overline{N_{n'}, P_s}$, $\overline{E_{n'}, P_s}$, $\overline{S_{n'}, P_s}$ and $\overline{W_{n'}, P_s}$, respectively. Since we just showed that $\overline{s, P_s}$ is clockwise of $\overline{N_2, P_s}$, $\overline{E_2, P_s}$, $\overline{S_2, P_s}$ and $\overline{W_2, P_s}$ it must also be the case that $\overline{s, P_s}$ is clockwise of $\overline{N_{n'}, P_s}$, $\overline{E_{n'}, P_s}$, $\overline{S_{n'}, P_s}$ and $\overline{W_{n'}, P_s}$. Therefore, since $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, N_{n'})$, $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, E_{n'})$, $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, S_{n'})$ and $\Phi(v_{n'}, P_s, s) < \Phi(v_{n'}, P_s, W_{n'})$, it must be the case that $\Phi(v_{n'}, P_s, s) < \min_{s'' \in nghbr_4(v_{n'})} \Phi(v_{n'}, P_s, s'')$ and thus $\Phi(v_{n'}, P_s, s) \notin [lb(v_{n'}), ub(v_{n'})]$.

- v_1: Since $\overline{s, P_s}$ is counter-clockwise of $\overline{v_1, P_s}$ we only need to show that $\overline{s, P_s}$ is also counter-clockwise of $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$ as that would mean that $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$ where $lb(v_1) \geq \min_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_{s'}, s'')$, $ub(v_1) \leq \max_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_{s'}, s'')$ and $nghbr_4(v_1) = \{N_1, E_1, S_1, W_1\}$. We show that $\Phi(v_1, P_s, s) > \max_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_{s'}, s'')$, which implies $\Phi(v_1, P_s, s) \notin [lb(v_1), ub(v_1)]$.

  – N_1: Since $\overline{s, P_s}$ intersects $\overline{c_1, c_4}$ we know that $\overline{s, P_s}$ intersects $L_{y''}$ east of where $\overline{N_1, P_s}$ intersects $L_{y''}$. $\overline{s, P_s}$ and $\overline{N_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y''}$ west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_1, P_s}$ and thus $\Phi(v_1, P_s, s) > \Phi(v_1, P_s, N_1)$.

  – E_1: Since $\overline{s, P_s}$ intersects $\overline{c_2, c_3}$ (exclusive of the endpoints) we know that $\overline{s, P_s}$ intersects L_y east of where $\overline{E_1, P_s}$ intersects L_y. $\overline{s, P_s}$ and $\overline{E_1, P_s}$ have a common

endpoint, vertex P_s and the latter line intersects L_y west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{E_1, P_s}$ and thus $\Phi(v_1, P_s, s) > \Phi(v_1, P_s, E_1)$.

- S_1**:** Since α is strictly greater than 45 degrees we know that $\overline{s, P_s}$ travels south faster than it travels east. Therefore, after $\overline{s, P_s}$ intersects $\overline{c_2, c_3}$, we know that $\overline{s, P_s}$ intersects $L_{y'}$ east of where $\overline{S_1, P_s}$ intersects $L_{y'}$. $\overline{s, P_s}$ and $\overline{S_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects $L_{y'}$ west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{S_1, P_s}$ and thus $\Phi(v_1, P_s, s) > \Phi(v_1, P_s, S_1)$.
- W_1**:** Since $\overline{s, P_s}$ intersects $\overline{c_2, c_3}$ we know that $\overline{s, P_s}$ intersects L_y east of where $\overline{W_1, P_s}$ intersects L_y. $\overline{s, P_s}$ and $\overline{W_1, P_s}$ have a common endpoint, vertex P_s and the latter line intersects L_y west of the former. It follows that $\overline{s, P_s}$ is counter-clockwise of $\overline{W_1, P_s}$ and thus $\Phi(v_1, P_s, s) > \Phi(v_1, P_s, W_1)$.

Since $\Phi(v_1, P_s, s) > \Phi(v_1, P_s, N_1)$, $\Phi(v_1, P_s, s) > \Phi(v_1, P_s, E_1)$, $\Phi(v_1, P_s, s) > \Phi(v_1, P_s, S_1)$ and $\Phi(v_1, P_s, s) > \Phi(v_1, P_s, W_1)$ it must be the case that $\Phi(v_1, P_s, s) > \max_{s'' \in nghbr_4(v_1)} \Phi(v_1, P_{s'}, s'')$ and thus $\Phi(v_1, P_s, s) \notin [\mathit{lb}(v_1), \mathit{ub}(v_1)]$.

Now, consider any vertex v_s on L_y that is west of vertex v_1 and let its four cross-bar neighbors be N_s, E_s, S_s and W_s. N_s, E_s, S_s and W_s are all directly west of N_1, E_1, S_1 and W_1, respectively. Therefore we know that $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$ are counter-clockwise of $\overline{N_s, P_s}$, $\overline{E_s, P_s}$, $\overline{S_s, P_s}$ and $\overline{W_s, P_s}$, respectively. Since we just showed that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_1, P_s}$, $\overline{E_1, P_s}$, $\overline{S_1, P_s}$ and $\overline{W_1, P_s}$ it must also be the case that $\overline{s, P_s}$ is counter-clockwise of $\overline{N_s, P_s}$, $\overline{E_s, P_s}$, $\overline{S_s, P_s}$ and $\overline{W_s, P_s}$. Therefore, since $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, N_s)$, $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, E_s)$, $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, S_s)$ and $\Phi(v_s, P_s, s) > \Phi(v_s, P_s, W_s)$, it must be the case that $\Phi(v_s, P_s, s) > \max_{s'' \in nghbr_4(v_s)} \Phi(v_s, P_s, s'')$ and thus $\Phi(v_s, P_s, s) \notin [\mathit{lb}(v_s), \mathit{ub}(v_s)]$.

Contradiction.

Thus, the theorem holds. □

D.2.5 Procedure PreProcess

We refer to Lines 210-214 collectively as procedure PreProcess even though they are not explicitly encompassed within a procedure.

In this section, we show the following:

- Procedure PreProcess terminates.
- If the Core Properties hold at the start of procedure PreProcess (Line 210) then the Core Properties hold at the conclusion of procedure PreProcess.
- If the Core Properties that Theorem 6 references hold at the start of procedure PreProcess then the Core Properties that Theorem 6 references hold when they are referenced by Property 7.

This ensures that our induction assumption regarding the Core Properties holds and, thus, at the conclusion of the next call to procedure ComputeShortestPath, path extraction retrieves an unblocked path from the start vertex to the goal vertex in reverse, if such a path exists.

Figure D.6 provides a summary of the dependencies of the lemmata, theorems and properties used in this section.

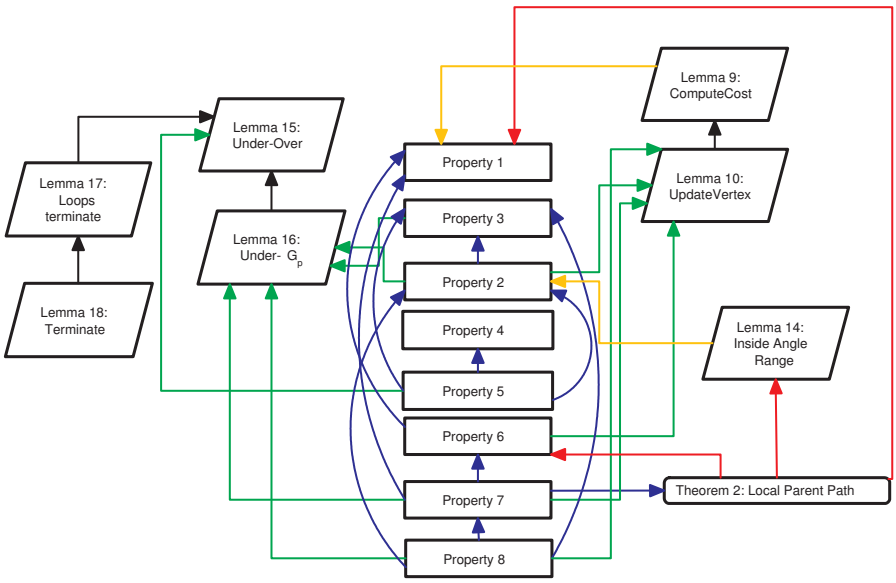


Figure D.6: Summary of Procedure PreProcess Dependencies

D.2.5.1 Helper Lemmata

We use the following helper lemmata:

Lemma 15. *The set of vertices iterated over during the while loop on Line 218 is the same set of vertices that are iterated over during the while loop on Line 229 during each call to procedure ClearSubtree.*

Proof. This follows from the fact that the latter loop iterates over the vertices in the over list until it is empty and the former iterates over the vertices in the under list until it is empty. Both sets are FIFO queues that are initialized to the empty set at the beginning of each call to procedure ClearSubtree on Line 216 and every vertex that is removed from the under list on Line 219 is added to the over list on Line 220. □

Lemma 16. *For any vertex s in the under list, every vertex s' with $s \in G_p(s')$ is iterated over by both the while loop on Line 218 and the while loop on Line 229 during each call to procedure ClearSubtree.*

Proof. Suppose for contradiction that there existed a largest non-empty set R of vertices and a vertex s in the under list such that each vertex $s'' \in R$ satisfied $s \in G_p(s'')$ and was not iterated over by either while loop. We consider the vertex $s' \in R$ that is the fewest number of local parent hops from vertex s (ties are broken arbitrarily). R cannot be empty because $s \in G_p(s)$ and thus some vertex s' must exist. Furthermore, we know that vertex s must have been iterated over because vertex s is in the under list and the while loop on Line 218 iterates until the under list is empty. Therefore, we know that vertex s' does not equal vertex s and that vertex s' must have a local parent. It must be the case that the local parent of vertex s' was iterated over because vertex s' is the fewest number of local parent hops from vertex s. Therefore, the condition on Line 227 ensures that vertex s' would be added to the under list on Line 228. The local parent of any

vertex must be a visible neighbor of that vertex (the start vertex is an exception that we discuss in Property 1), due to Lines 167 and 231 and the visible neighbors of a vertex are a subset of the 8 compass neighbors of that vertex. Thus, it must be the case that vertex s' would be iterated over because the while loop on Line 218 iterates until the under list becomes empty. Finally, Lemma 15 ensures that the while loop on Line 229 iterates on the exact set of vertices that were iterated over during the while loop on Line 218. Contradiction. □

Lemma 17. *The while loops on Lines 218 and 229 terminate.*

Proof. The bodies of the while loops on Lines 218 and 229 have a finite runtime. Thus, we show that the while loops terminate by arguing that they iterate a finite number of times. First we show that the while loop on Line 218 terminates. The while loop on Line 218 iterates over the vertices in the under list until it is empty. The under list is initialized to the empty set (Line 216), after which a vertex can only be added to the under list if it has a non-$NULL$ local parent (Line 227). Any vertex that is added to the under list has its local parent set to $NULL$ on Line 221. Thus, a vertex cannot be added to the under list more than once during the while loop on Line 218. Since the number of vertices in V is finite, the total number of vertices added to the under list must be finite and therefore the while loop on Line 218 terminates. Now we show that the while loop on Line 229 terminates. We just showed that the while loop on Line 218 iterates over a finite number of vertices and thus Lemma 15 guarantees that the while loop on Line 229 also iterates over a finite number of vertices. Therefore, it must be the case that the while loop on Line 229 terminates. □

Lemma 18. *Procedure PreProcess terminates.*

Proof. Procedure PreProcess is composed of six loops, four foreach loops (Lines 210, 212, 226 and 231) and two while loops (Lines 218 and 229). The four foreach loops trivially terminate and the two while loops terminate according to Lemma 17. Therefore, procedure PreProcess terminates. □

D.2.5.2 Core Properties

Consider an arbitrary iteration of the while loop on Line 207 which calls procedure PreProcess. Initially, we know that if the Core Properties hold at the start of procedure ComputeShortestPath then they hold at the conclusion of procedure ComputeShortestPath. Procedure PreProcess terminates according to Lemma 17. We show that if the Core Properties hold at the start of procedure PreProcess then they hold the conclusion of procedure PreProcess. We divide the Core Properties into two groups: **(1)** Properties 1-4 are properties that are independent of newly blocked grid cells. For these properties, we prove by induction that the Core Properties hold at the conclusion of any call to procedure ClearSubtree. **(2)** Properties 5-8 are properties that are *not* independent of newly blocked grid cells. For these properties, we prove by induction that the properties hold at the conclusion of any iteration of the foreach loop on Line 210. All of the Core Properties hold at the conclusion of procedure PreProcess.

Property D.2.5.1. *(At any point during procedure ClearSubtree:) The parent of the start vertex is the start vertex, the local parent of the start vertex is the start vertex, the g-value of the start vertex is 0, the angle range of the start vertex is* $[-\infty, \infty]$ *and the start vertex is in the open or closed lists.*

Proof. If the start vertex is not modified during procedure ClearSubtree then the property holds according to our induction assumption regarding the Core Properties, specifically Property 1. The start vertex can only be modified during procedure ClearSubtree if it is added to the under list. Vertices can only be added to the under list on Line 217 or Line 228. Line 213 explicitly prevents the start vertex from being added to the under list on Line 217. In order for a vertex to be added to the under list on Line 228 the condition on Line 227 must be satisfied, but we know from our induction assumption regarding the Core Properties, specifically Property 1, that the local parent of the start vertex is the start vertex and thus that condition cannot be satisfied. Therefore, the start vertex cannot be added to the under list which means that the start vertex cannot be removed from the open or closed lists and the parent of the start vertex, the local parent of the start vertex, the g-value of the start vertex and the angle range of the start vertex cannot be modified. □

Property D.2.5.2. *(At any point during the while loop on Line 229:) For any vertex $t \in V$ that is in the open or closed lists, $G_p(t, \text{parent}(t)) = [t_0 = t, t_1, \ldots, t_n]$ is well defined. Furthermore, for all $1 \leq i \leq n$, $t_i \in G_p(t, \text{parent}(t))$ is in the closed list.*

Proof. Consider the p^{th} time that procedure ClearSubtree is executed. First, we examine the while loop on Line 218. Consider any vertex $t \in V$ that is in the open or closed lists at the start of the while loop on Line 218. $G_p(t, \textit{parent}(t))$ must exist because $t \in G_p(t, \textit{parent}(t))$. Let vertex $t_i \in G_p(t, \textit{parent}(t))$ be the vertex that is the fewest number of local parent hops from the parent of vertex t in reverse, that is modified during the while loop on Line 218. If no such vertex exists then the property holds for vertex t due to our induction assumption regarding the Core Properties, specifically Property 2. Lemma 16 ensures that any vertex $t_j \in G_p(t, \textit{parent}(t)) \subseteq G_p(t)$ for all $0 \leq j \leq i$ must be iterated over during the while loop on Line 218 because we know that vertex t_i was modified and thus must have been added to the under list and iterated over. Each vertex iterated over must have set its parent and local parent to $NULL$, its lower bound to negative infinity and its upper bound to infinity on Line 221, after which it was removed from both the open and closed lists (Lines 223 and 225). Therefore, the property holds when the while loop on Line 218 terminates since vertex t is removed from the open and closed lists. It follows that the property holds for every vertex that is in the open or closed lists when the while loop on Line 218 terminates. Now, we examine the while loop on Line 229. We prove by induction on the ordered sequence of vertices in $G_p(t, \textit{parent}(t))$ that the property holds at the end of the while loop on Line 229. The property holds initially for any vertex $t \in V$ that is in the open or closed lists due to our induction assumption. We now show that the statement continues to hold whenever a vertex changes its parent, its local parent, its angle range or its membership in the open or closed lists. A vertex t' can become a member of the open or closed lists only when its visible neighbors are being iterated over during the foreach loop on Line 231 and procedure UpdateVertex(t, t') updates the parent, local parent and angle range of vertex t', where vertex t is a visible neighbor of vertex t'. Vertex t' is not in the closed list because every vertex iterated over during the while loop on Line 229 was removed from the closed list during the while loop on Line 218. Vertex t is in the closed list due to Line 232 and thus we know that $G_p(t, \textit{parent}(t))$ has all the required properties due to our induction assumption. The parent of vertex t is in the closed list due to Property 3 and our induction assumption. If vertex t' is updated according to Path 2 then the parent of vertex t' is equal to the parent of vertex t, the local parent of vertex t' is equal to vertex t, and the property holds according to Lemma 10. If vertex t' is updated according to Path 1 then the parent of vertex t' is vertex t and the local parent of vertex t' is vertex t and the property holds

because $n = 0$ and vertex t is in the closed list (we do not need to prove the parts of Property 2 that require $n > 0$). If vertex t' is not updated according to either Path 1 or Path 2 then it did not become a member of the open or closed lists. There are no other ways in which a vertex can change its membership in the open or closed lists or its parent, local parent and angle range. □

Property D.2.5.3. *(At any point during the while loop on Line 229:) For any vertex $t \in V$ that is in the open or closed lists, $G_p(t) = [t_0 = t, t_1, \ldots, t_n = s_{\text{start}}]$ is well defined. Furthermore, for all $1 \leq i \leq n$, $t_i \in G_p(t)$ is in the closed list.*

Proof. Consider the p^{th} time that procedure ClearSubtree is executed. First, we examine the while loop on Line 218. Consider any vertex $t \in V$ that is in the open or closed lists at the start of the while loop on Line 218. $G_p(t)$ must exist because $t \in G_p(t)$. Let vertex $t_i \in G_p(t)$ be the vertex that is the fewest number of local parent hops from the start vertex to vertex t in reverse, that is modified during the while loop on Line 218. If no such vertex exists then the property holds for vertex t due to our induction assumption regarding the Core Properties, specifically Property 3. Lemma 16 ensures that any vertex $t_j \in G_p(t)$ for all $0 \leq j \leq i$ must be iterated over during the while loop on Line 218 because we know that vertex t_i was modified and thus must have been added to the under list and iterated over. Each vertex iterated over must have set its local parent to $NULL$ on Line 221, after which it was removed from both the open and closed lists (Lines 223 and 225). Therefore the property holds when the while loop on Line 218 terminates since vertex t is removed from the open and closed lists. It follows that the property holds for every vertex that is in the open or closed lists when the while loop on Line 218 terminates. Now, we examine the while loop on Line 229. We prove by induction on the ordered sequence of vertices in $G_p(t)$ that the property holds at the end of the while loop on Line 229. The property holds initially for any vertex $t \in V$ that is in the open or closed lists due to our induction assumption. We now show that the statement continues to hold whenever a vertex changes its local parent or its membership in the open or closed lists. A vertex t' can become a member of the open or closed lists only when its visible neighbors are being iterated over during the foreach loop on Line 231 and procedure UpdateVertex(t, t') updates the local parent of vertex t', where vertex t is a visible neighbor of vertex t'. Vertex t' is not in the closed list because every vertex iterated over during the while loop on Line 229 was removed from the closed list during the while loop on Line 218. Vertex t is in the closed list due to Line 232 and thus we know that the property holds for $G_p(t)$ due to our induction assumption. Therefore, if vertex t' is updated according to either Path 1 or Path 2, then the property holds for $G_p(t')$ because the local parent of vertex t' is vertex t (we do not need to prove the parts of Property 3 that require $n > 0$). If vertex t' is not updated according to either Path 1 or Path 2 then it did not become a member of the open or closed lists. There are no other ways in which a vertex can change its membership in the open or closed lists or its local parent. □

Property D.2.5.4. *(At any point during the while loop on Line 229:) For any vertex $t \in V$ such that procedure InitializeVertex(t) has been called at some point during the execution of procedure Main, the g-value of vertex t is finite iff vertex t is in the open or closed lists.*

Proof. Consider the p^{th} time that procedure ClearSubtree is executed and any vertex t that was modified during procedure ClearSubtree. If vertex t was not modified during procedure ClearSubtree then the property holds according to our induction assumption regarding the Core Properties,

specifically Property 4. InitializeVertex cannot be called on a vertex for the first time during procedure PreProcess. First, we examine vertices that were modified during the while loop on Line 218. Lines 221-225 ensure both that any vertex that is removed from the open or closed lists has its g-value set to infinity and that any vertex that has its g-value set to infinity is removed from the open or closed lists. The while loop on Line 218 does not add vertices to the open or closed lists and does not set the g-values of vertices to finite values. Now, we examine vertices that were modified during the while loop on Line 229. We must show that **(1)** if vertex t becomes a member of the open or closed lists during procedure PreProcess then the g-value of vertex t is finite and we must show that **(2)** if the g-value of vertex t is set to a finite value during procedure PreProcess then vertex t must be in the open or closed lists. The while loop on Line 229 does not remove vertices from the open or closed lists and does not set the g-values of vertices to infinity.

- **(1)** Procedure PreProcess does not add vertices to the closed list. Consider any vertex t that was added to the open list. Procedure ClearSubtree only adds vertices to the open list during procedure UpdateVertex on Line 233. Any vertex added to the open list during procedure UpdateVertex must have had its g-value set on either Line 190 or Line 201. This follows from the fact that vertex t was added to the open list and thus the condition on Line 24 was satisfied which means that the condition on either Line 188 or 199 was satisfied as well. In either case, the condition on either Line 188 or 199 ensures that the g-value of vertex t must have been set to a finite value.

- **(2)** Consider any vertex t such that the g-value of vertex t is set to a finite value during procedure ClearSubtree. The g-value of vertex t can only be set to a finite value on Line 190 or 201. In either case, the condition on Line 24 must be satisfied because it was satisfied on either Line 188 or 199 immediately prior to the g-value of vertex t being set to a finite value on either Line 190 or 201, respectively and thus vertex t is added to the open list. Finally, the g-value of vertex t can never increase on Lines 190 and 201 due to Lines 188 and 199.

□

Property D.2.5.5. *(At the conclusion of any iteration of the foreach loop on Line 210:) For any vertex $t \in V$ that is in the closed list, the visible neighbors of vertex t are in the open or closed lists.*

Proof. We prove this property by induction on the number of iterations of the foreach loop on Line 210. Due to our induction assumption regarding the Core Properties, specifically Property 5, we know that the property to holds at the conclusion of procedure ComputeShortestPath, after which grid cells became blocked. Consider the k^{th} iteration of the foreach loop on Line 210. Since procedure PreProcess does not add vertices to the closed list and newly blocked grid cells can only reduce the number of visible neighbors that a vertex has, we only need to ensure that every vertex that is removed from the open or closed lists is re-inserted into the open list if it has a visible neighbor in the closed list. Suppose for contradiction that at the conclusion of the k^{th} iteration of the foreach loop on Line 210 a vertex t with a visible neighbor in the closed list was removed from the open or closed lists but was not re-inserted into the open list. Vertex t must have been removed from the open or closed list and set its g-value to infinity during the while loop on Line 218. Therefore, Lemma 15 ensures that vertex t must have been iterated over during

the while loop on Line 229. Our contradictory assumption ensures that there must exist some visible neighbor t' of vertex t such that vertex t' is in the closed list and thus the conditionals on Lines 231 and 232 must be satisfied and UpdateVertex(t', t) must be called. Since vertex t' is in the closed list, Properties 2 and 3 together ensure that the parent of vertex t' is in the closed list, and thus, Property 4 ensures that the g-value of vertex t' and the g-value of the parent of vertex t' are finite and since the g-value of vertex t is infinity it must be the case that the condition on either Line 188 or 199 is satisfied (procedure InitializeVertex must have been called on any vertex in the open or closed lists). Therefore vertex t must be updated according to either Path 1 or Path 2 and in either case the g-value of vertex t decreases and thus the condition on Line 24 is satisfied and vertex t is added to the open list on Line 233 during procedure UpdateVertex. Contradiction. □

Property D.2.5.6. *(At the conclusion of any iteration of the foreach loop on Line 210:) For any vertex $t \in V \setminus \{s_{\text{start}}\}$ that is in the open or closed lists and whose $\angle(t, \text{parent}(t))$ is a multiple of 45 degrees, the parent of vertex t is a visible neighbor of vertex t.*

Proof. We prove this property by induction on the number of iterations of the foreach loop on Line 210. Due to our induction assumption regarding the Core Properties, specifically Property 6, we know that the property holds at the conclusion of procedure ComputeShortestPath, after which grid cells became blocked. Consider the k^{th} iteration of the foreach loop on Line 210, some vertex $t \in V \setminus \{s_{start}\}$ that is in the open or closed lists and a grid cell c_k in the arbitrarily ordered sequence of newly blocked grid cells. Property 6 can only become invalid during procedure PreProcess either **(1)** if a vertex and its parent are no longer visible neighbors due to a newly blocked grid cell c_k or **(2)** if a vertex changes its parent or membership in the open or closed lists during procedure PreProcess such that the property is violated. For every other vertex the property holds due to our induction assumption regarding the Core Properties.

- **(1)** The property might no longer hold if the parent of vertex t is no longer a visible neighbor of vertex t due to a newly blocked grid cell c_k. However, then both vertex t and the parent of vertex t must be corners of c_k because they are neighbors and thus Line 212 ensures that both vertex t and the parent of vertex t are added to the under list and iterated over during the while loop on Line 218, where they are removed from both the open and closed lists (Lines 223 and 225) and have their parents set to $NULL$ on Line 221.

- **(2)** The property might no longer hold if a vertex t changes its parent or membership in the open or closed lists during procedure PreProcess. Vertices are only modified during procedure ClearSubtree. First, we examine vertices that are modified during the while loop on Line 218. Lines 221-225 ensure that any vertex that changes its parent is removed from the open or closed lists. Now we examine vertices that are modified during the while loop on Line 229. The while loop on Line 229 does not add vertices to the closed list. Consider any vertex t, other than the start vertex, that becomes a member of the open list during procedure PreProcess. A vertex can only become a member of the open list during procedure UpdateVertex and thus the property holds according to Lemma 10. The start vertex is not modified during procedure PreProcess due to Property 1. There are no other ways in which a vertex can change its parent or its membership in the open or closed lists.

□

Property D.2.5.7 (Line-of-Sight Property)**.** *(At the conclusion of any iteration of the foreach loop on Line 210:) For any vertex $t \in V$ that is in the open or closed lists, vertex t has line-of-sight to its parent.*

Proof. We prove this property by induction on the number of iterations of the foreach loop on Line 210. Due to our induction assumption regarding the Core Properties, specifically Property 7, we know that the property holds at the conclusion of procedure ComputeShortestPath, after which grid cells became blocked. Consider the k^{th} iteration of the foreach loop on Line 210, some vertex $t \in V$ that is in the open or closed lists and a grid cell c_k in the arbitrarily ordered sequence of newly blocked grid cells. Property 7 can only become invalid during procedure PreProcess either **(1)** if a vertex t no longer has line-of-sight to its parent due to a newly blocked grid cell c_k or **(2)** if a vertex changes its parent or membership in the open or closed lists during procedure PreProcess such that the property is violated. For every other vertex the property holds due to our induction assumption regarding the Core Properties.

- **(1)** The property might no longer hold because a newly blocked grid cell c_k can prevent a vertex from having line-of-sight to its parent. We show that, by calling ClearSubtree(t') for all $t' \in corners(c_k)$, any vertex t that is in the open or closed lists, such that vertex t no longer has line-of-sight to its parent because c_k became blocked, is removed from both the open and closed lists and sets its parent to $NULL$. There are two cases in which a vertex t no longer has line-of-sight to its parent: **(a)** $\overline{t, parent(t)}$ traverses the interior of a blocked grid cell, or **(b)** $\overline{t, parent(t)}$ passes between two blocked grid cells that share a side.

  - **(a)** $\overline{t, parent(t)}$ traverses c_k: From Theorem 6, we know that, for any vertex t that is in the open or closed list, if $\overline{t, parent(t)}$ traverses c_k then some corner $t' \in corners(c_k)$ must be in $G_p(t, parent(t))$. Line 212 ensures that ClearSubtree(t') is called and thus vertex t' is added to the under list and iterated over during the while loop on Line 218. Therefore, since vertex $t' \in G_p(t, parent(t)) \subseteq G_p(t)$, Lemma 16 ensures that vertex t is iterated over during the while loop on Line 218. Thus vertex t is removed from both the open and closed list (Lines 223 and 225) and the parent of vertex t is set to $NULL$ (Line 221).
  - **(b)** $\overline{t, parent(t)}$ passes between two blocked grid cells that share a side, one of which has an endpoint that is also a corner c_k: We know that vertex t is not the start vertex due to Property 1 which ensures that the parent of the start vertex is the start vertex. Due to Property 6, any vertex that does not have line-of-sight to its parent because $\overline{t, parent(t)}$ passes between two blocked grid cells that share a side, one of which has an endpoint that is also a corner c_k, would also be removed from both the open or closed list and have set its parent to $NULL$. This follows from the fact that any such $\overline{t, parent(t)}$ must have an absolute angle $\angle(t, parent(t))$ that is a multiple of 45 degrees and thus vertex t and the parent of vertex t are visible neighbors. This means that vertex t must be a corner of c_k and thus is added to the under list on Line 217 and iterated over during the while loop on Line 218. Thus, vertex t is removed from the open and closed lists (Lines 223 and 225) and the parent of vertex t is set to $NULL$ (Line 221).

At this point we have shown that every vertex that no longer has line-of-sight to its parent due to the newly blocked grid cell c_k is removed from the open and closed lists and the parent of vertex t is $NULL$.

- **(2)** The property might no longer hold if a vertex t changes its parent or membership in the open or closed lists during procedure PreProcess. Vertices are only modified during procedure ClearSubtree. First, we examine vertices that are modified during the while loop on Line 218. Lines 221-225 ensure that any vertex that changes its parent is removed from the open or closed lists. Now we examine vertices that are modified during the while loop on Line 229. The while loop on Line 229 does not add vertices to the closed list. Consider any vertex t, that becomes a member of the open list during procedure PreProcess. A vertex can only become a member of the open list during procedure UpdateVertex and thus the property holds according to Lemma 10. There are no other ways in which a vertex can change its parent or its membership in the open or closed lists.

□

Property D.2.5.8. *(At the conclusion of any iteration of the foreach loop on Line 210:) For any vertex $t \in V$ that is in the open or closed lists, $A_p(t) = [t_0 = t, t_1, \ldots, t_n = s_{\text{start}}]$ is well defined and thus is an unblocked path from the start vertex to vertex t in reverse and the g-value of vertex t is its length.*

Proof. At the conclusion of any iteration of the foreach loop on Line 210, every vertex that is in the open or closed lists has line-of-sight to its parent according to Property 7 and, therefore, we only have to show that for any vertex t that is in the open or closed lists, $A_p(t)$ is a path from the start vertex to vertex t in reverse and the g-value of vertex t is its length. Consider the p^{th} time that procedure ClearSubtree is executed. First, we examine the while loop on Line 218. Consider any vertex $t \in V$ that is in the open or closed lists at the start of the while loop on Line 218. $A_p(t)$ must exist because $t \in A_p(t)$. Let vertex $t_i \in A_p(t)$ be the vertex that is the fewest number of parent hops from the start vertex to vertex t in reverse, that is modified during the while loop on Line 218. If no such vertex exists then the property holds for vertex t due to our induction assumption regarding the Core Properties, specifically Property 8. Lemma 16 ensures that any vertex $t_j \in A_p(t)$ for all $0 \leq j \leq i$ must be iterated over during the while loop on Line 218 because we know that vertex t_i was modified and thus must have been added to the under list and iterated over. Each vertex iterated over must have set its parent to $NULL$ and its g-value to infinity on Line 221, after which it was removed from both the open and closed lists (Lines 223 and 225). Therefore, the property holds when the while loop on Line 218 terminates since vertex t is removed from the open and closed lists. It follows that the property holds for every vertex that is in the open or closed lists when the while loop on Line 218 terminates. Now, we examine the while loop on Line 229. We prove by induction on the ordered sequence of vertices in $A_p(t)$ that the property holds at the end of the while loop on Line 229. The property holds initially for any vertex $t \in V$ that is in the open or closed lists due to our induction assumption. We now show that the statement continues to hold whenever a vertex changes its g-value, its parent or its membership in the open or closed lists. A vertex t' can become a member of the open or closed lists only when its visible neighbors are being iterated over during the foreach loop on Line 231 and procedure UpdateVertex(t, t') updates the parent and g-value of vertex t', where vertex t is

a visible neighbor of vertex t'. Vertex t' is not in the closed list because every vertex iterated over during the while loop on Line 229 was removed from the closed list during the while loop on Line 218. Vertex t is in the closed list due to Line 232, and its parent is in the closed list according to Properties 2 and 3 together and thus we know that the property holds for $A_p(t)$ (and $A_p(\mathit{parent}(t))$) due to the fact that we just showed that the property holds for any vertex $t \in V$ that is in the open or closed lists when the while loop on Line 218 terminates (the while loop on Line 229 does not remove vertices from the closed list). Thus, $A_p(t)$ (or $A_p(\mathit{parent}(t))$) is a path from the start vertex to vertex t (or its parent, respectively) in reverse, and the g-value of vertex t is its length (or the g-value of its parent, respectively) according to the induction assumption. If procedure UpdateVertex updates vertex t' then we know that the parent of vertex t' is either vertex t or the parent of vertex t and, in either case, the property continues to hold according to Lemma 10 which ensures that $g(t') = g(\mathit{parent}(t')) + c(\mathit{parent}(t'), t')$. Furthermore, since vertex t' is in the open list and the parent of vertex t' is in the closed list, no vertex can have vertex t' as its parent and thus it cannot be the case that the g-value of a vertex no longer represents the length of the path from the start vertex to that vertex because vertex t' changed its g-value and parent (this also ensures that a cycle can never be introduced). If vertex t' is not updated according to either Path 1 or Path 2 then it did not become a member of the open or closed lists. There are no other ways in which a vertex can change its membership in the open or closed lists, its parent or its g-value. Thus, $A_p(t')$ is an unblocked path from the start vertex to vertex t' in reverse and the g-value of vertex t' is its length. □

We just showed that procedure PreProcess terminates and that the Core Properties hold at the conclusion of procedure PreProcess if they hold at the start of procedure PreProcess. We showed earlier that procedure ComputeShortestPath terminates and that the Core Properties hold at the conclusion of procedure ComputeShortestPath if they hold at the start of procedure ComputeShortestPath. We also showed earlier that the Core Properties hold at the conclusion of procedure Initialize (which terminates trivially because it contains no loops or recursive calls). Therefore, the Core Properties hold at the conclusion of any call to procedure ComputeShortestPath or procedure PreProcess. Theorem 6 and Theorem 8 hold if the Core Properties hold and, therefore, when procedure ComputeShortestPath terminates if the g-value of the goal vertex is not equal to infinity then path extraction retrieves an unblocked path from the start vertex to the goal vertex in reverse, and if the g-value of the goal vertex is equal to infinity then there does not exist an unblocked path from the start vertex to the goal vertex.

Appendix E

Impact of Any-Angle Path Planning

In this appendix, we discuss the impact of any-angle path planning. We focus on any-angle path planning research that references, extends or performs experimental comparisons with the new any-angle find-path algorithms introduced in this dissertation. In Section E.1, we list research that cites the contributions made in this dissertation. In Section E.2, we discuss research that extends the contributions made in this dissertation. Finally, in Section E.3, we discuss research that introduces new any-angle find-path algorithms that are compared experimentally with the any-angle find-path algorithms introduced in this dissertation.

E.1 Citations of Basic Theta* and its Variants

In this section, we list some of the research that references the new any-angle find-path algorithms introduced in this dissertation. This research comes from five different areas of computer science and engineering: robotics, artificial intelligence, aerospace, video games and oceanography. While the relevance of the different citations varies, they highlight the fact that researchers in many different areas of computer science and engineering are interested in any-angle path planning.

E.1.1 Robotics Conferences

- Practical Search Techniques in Path Planning for Autonomous Driving (Dolgov, Thrun, Montemerlo, & Diebel, 2008).
- Continuous-Field Path Planning with Constrained Path-Dependent State Variables (Mills-Tettey, Stentz, & Dias, 2008).
- Dynamic Control in Real-Time Heuristic Search (Bulitko, Luštrek, Schaeffer, Björnsson, & Sigmundarson, 2008)
- Planning with Uncertainty in Position Using High-Resolution Maps (Gonzalez, 2008).
- Autonomous Driving in Semi-Structured Environments: Mapping and Planning (Dolgov & Thrun, 2009).
- Smooth Path Planning in Constrained Environments (Rufli, Ferguson, & Siegwart, 2009).

- Autonomous Robot Path Planning (Crous, 2009).
- Autonomous Driving in a Multi-Level Parking Structure (Kümmerle, Hähnel, Dolgov, Thrun, & Burgard, 2009).
- Path Planning for Autonomous Vehicles in Unknown Semi-Structured Environments (Dolgov, Thrun, Montemerlo, & Diebel, 2010).
- Curvature-Continuous Trajectory Generation with Corridor Constraint for Autonomous Ground Vehicles (Choi et al., 2010).
- Fast Any-Angle Path Planning on Grid Maps with Non-Collision Pruning (Choi, Lee, & Yu, 2011).
- Any-Angle Path Planning on Non-Uniform Costmaps (Choi & Yu, 2011).

E.1.2 Artificial Intelligence Conferences

- Dynamic Control in Real-Time Heuristic Search (Bulitko et al., 2008).
- Accelerated A* Trajectory Planning: Grid-Based Path Planning Comparison (Šišlák et al., 2009b).
- Block A*: Database-Driven Search with Applications in Any-Angle Path-Planning (Yap et al., 2011).
- Trajectory Planning on Grids: Considering Speed Limit Constraints (Chrpa, 2011).
- Smoothed Hex-Grid Trajectory Planning Using Helicopter Dynamics (Chrpa & Komenda, 2011).

E.1.3 Aerospace Conferences

- On-Board Multi-Objective Mission Planning for Unmanned Aerial Vehicles (Wu, Campbell, & Merz, 2009).

E.1.4 Video Game Conferences

- Variable Resolution A* (Walsh & Banerjee, 2008).
- Optimizing Motion-Constrained Pathfinding (Sturtevant, 2009).

E.1.5 Oceanography Conferences

- Path Planning for Gliders Using Regional Ocean Models: Application of Pinzon Path Planner with the ESEOAT Model and the RU27 Trans-Atlantic Flight Data (Fernandez-Perdomo, Cabrera-Gamez, Hernandez-Sosa, Isern-Gonzalez, Dominguez-Brito, Redondo, Coca, Ramos, Fanjul, & Garcia, 2010).

E.2 Extensions of Basic Theta*

In this section, we discuss research that extends Basic Theta*. Specifically, we discuss two papers that modify procedure LineOfSight, which is called on Line 44 of Algorithm 3. The first paper introduces techniques that can reduce the runtime of procedure LineOfSight. The second paper introduces techniques that allow Basic Theta* to find paths on square grids that contain unblocked grid cells with non-uniform traversal costs as described in Section 4.7.2. All of the research discussed in this section was designed for finding paths in known 2D environments that have been discretized into grid graphs constructed from square grids.

		FD*	Basic Theta*	AP Theta*	A* on Visibility Graphs (true shortest paths)	A* on Octile Graphs (grid paths)	A* PS
100×100	Random Grids 0%	51.88	51.80	51.80	51.80	54.63	51.80
	Random Grids 5%	48.83	48.74	48.74	48.69	51.24	48.99
	Random Grids 10%	50.64	50.53	50.54	50.45	53.11	50.91
	Random Grids 20%	48.65	48.54	48.55	48.43	50.86	49.04
	Random Grids 30%	50.19	50.10	50.11	49.98	52.25	50.61
500×500	Random Grids 0%	259.65	259.24	259.24	N/A	273.11	259.24
	Random Grids 5%	257.19	256.58	256.60	N/A	270.40	259.14
	Random Grids 10%	259.37	258.62	258.65	N/A	271.77	261.62
	Random Grids 20%	258.71	257.88	257.93	N/A	270.60	261.36
	Random Grids 30%	266.49	265.84	265.90	N/A	277.57	269.60

Table E.1: Path Lengths

		FD*	Basic Theta*	AP Theta*	A* on Visibility Graphs (true shortest paths)	A* on Octile Graphs (grid paths)	A* PS
100×100	Random Grids 0%	0.0109	0.0026	0.0034	0.0020	0.0015	0.0057
	Random Grids 5%	0.0097	0.0022	0.0038	0.0311	0.0013	0.0048
	Random Grids 10%	0.0120	0.0028	0.0051	0.1173	0.0014	0.0054
	Random Grids 20%	0.0135	0.0034	0.0065	0.4594	0.0019	0.0054
	Random Grids 30%	0.0153	0.0045	0.0081	1.0769	0.0028	0.0058
500×500	Random Grids 0%	0.1231	0.0288	0.0113	N/A	0.0045	0.1688
	Random Grids 5%	0.1538	0.0390	0.0523	N/A	0.0053	0.1747
	Random Grids 10%	0.1795	0.0577	0.0870	N/A	0.0108	0.1783
	Random Grids 20%	0.2219	0.0882	0.1384	N/A	0.0273	0.1871
	Random Grids 30%	0.3207	0.1244	0.2000	N/A	0.0628	0.1951

Table E.2: Runtimes

		FD*	Basic Theta*	AP Theta*	A* on Visibility Graphs (true shortest paths)	A* on Octile Graphs (grid paths)	A* PS
100×100	Random Grids 0%	183	87	63	1	46	580
	Random Grids 5%	196	116	104	12	47	499
	Random Grids 10%	244	166	153	33	61	549
	Random Grids 20%	286	213	209	90	91	518
	Random Grids 30%	355	292	289	177	169	540
500×500	Random Grids 0%	3312	903	314	N/A	231	14671
	Random Grids 5%	4323	2013	1627	N/A	284	14697
	Random Grids 10%	5131	3123	2769	N/A	646	14553
	Random Grids 20%	6837	4674	4522	N/A	1572	14653
	Random Grids 30%	8907	6818	6688	N/A	3827	14797

Table E.3: Number of Vertex Expansions

E.2.1 Faster Line-of-Sight Checks

In the work of Choi et al. (2011), techniques are introduced which reduce the runtime of the line-of-sight checks performed by Basic Theta* (Choi et al., 2011). These techniques are designed to be easily integrated into a line-of-sight check based on the standard Bresenham line-drawing algorithm from computer graphics (Bresenham, 1965), such as the one provided in Appendix A. The authors are able to reduce the runtime of a line-of-sight check by introducing a set of pruning rules which allow a line-of-sight check to terminate earlier than it normally would. These rules are based on the idea that, if three ordered vertices s, s', s'' are collinear (or close to collinear), then whether or not one pair of the three vertices have line-of-sight can reduce the runtime required to determine whether or not another pair of the three vertices have line-of-sight. For example, if vertex s and vertex s' have line-of-sight then one only needs to check whether or not vertex s' and vertex s'' have line-of-sight in order to determine whether or not vertex s and vertex s'' have line-of-sight. These pruning rules can be overcautious and thus sometimes the line-of-sight check will return false even though the two vertices actually have line-of-sight. While determining whether or not three vertices are collinear (or close to collinear) requires additional computation, the results provided by Choi et al. (2011) suggest that these pruning rules can reduce the runtime of Basic Theta* by up to a factor of ≈ 2 without a significant increase in the lengths of the paths found by Basic Theta*. Personal communication with video game developers suggests that the runtime of the line-of-sight check provided in Appendix A can be reduced by up to one order of magnitude for certain compilers *without* the optimizations suggested in the work of Choi et al. (2011).

E.2.2 Non-Uniform Traversal Costs

In the work of Choi and Yu (2011), techniques are introduced which allow Basic Theta* to find paths on 8-neighbor square grid graphs that contain unblocked grid cells with non-uniform traversal costs (Choi & Yu, 2011). These techniques are designed to be easily integrated into a line-of-sight check based on the standard Bresenham line-drawing algorithm from computer graphics (Bresenham, 1965), such as the one provided in Appendix A. Computing the exact cost of a path segment connecting two vertices that passes through several grid cells with different traversal costs can be more time consuming than determining whether or not two vertices have line-of-sight. Choi and Yu (2011) introduce techniques that approximate the cost of a path segment that passes through several grid cells with different traversal costs and techniques that use optimizations to efficiently compute the exact cost of a path segment that passes through several grid cells with different traversal costs. Their approximation techniques use two different equations for computing an average of the traversal costs of the grid cells that are traversed by a path segment.

The results provided by Choi and Yu (2011) suggest that approximating costs and computing exact costs provides a tradeoff with respect to the runtime of the search and the cost of the resulting path. More specifically, their results suggest: **(1)** that computing exact costs is $\approx 10 - 30\%$ slower than computing approximate costs and **(2)** that the paths found when approximating costs are only $\approx 1 - 2\%$ larger than the paths found when computing exact costs (because underestimates and overestimates cancel each other out). No experimental comparison was performed with the technique we introduced in Section 4.7.2.

		A* on Visibility Graphs (true shortest paths)	Basic Theta*	AA*
100×100	Random Grids 5%	54.21	54.35	54.21
	Random Grids 10%	53.19	53.43	53.19
	Random Grids 20%	53.30	53.62	53.30
	Random Grids 30%	53.21	53.61	53.21

Table E.4: Path Lengths (Šišlák et al., 2009b)

E.3 Experimental Comparisons with Basic Theta*

In this section, we discuss research that introduces new any-angle find-path algorithms that are compared experimentally with Basic Theta*. Specifically, we discuss two new any-angle find-path algorithms that were developed after Basic Theta* and provide different tradeoffs than Basic Theta* with respect to the runtime of the search and the length of the resulting path: Accelerated A* (Šišlák et al., 2009b, 2009a) finds shorter paths than Basic Theta* but finds them more slowly and Block A* (Yap et al., 2011) finds longer paths than Basic Theta* but finds them faster. Both of these experimental comparisons were performed on path-planning problems similar to those that were used in the experimental comparisons performed on random grids in Section 4.6 with two differences: **(1)** the start vertex and goal vertex were both chosen randomly from the corners of unblocked grid cells (as was the case for game maps) and **(2)** there was *not* a one-unit border of unblocked grid cells around the square grid. Other than these two differences the path-planning problems used in their experimental comparisons are identical to the path-planning problems used in the experimental comparisons performed in Section 4.6. These path-planning problems are similar to the path-planning problems used in the experimental comparison performed in the work of Nash et al. (2007) and thus we include the experimental results from that paper in Tables E.1-E.3. These tables are annotated in the same manner as Tables 4.1-4.4 in Section 4.6. The values in Tables E.1-E.3 are different from the values in Tables 4.1-4.4, but the relationships between the different any-angle find-path algorithms are similar. For example, in Table E.1, the paths found by A* on random 500×500 grids with 10 percent blocked grid cells are $\approx 5.08\%$ longer than the paths found by Basic Theta* on average and in Table 4.1, the paths found by A* on random 500×500 grids with 10 percent blocked grid cells are $\approx 5.03\%$ longer than the paths found by Basic Theta* on average. The relationships are similar for the other find-path algorithms, for grids with different percentages of blocked grid cells and for different metrics.[1]

E.3.1 Accelerated A* (AA*)

Accelerated A* (AA*) (Šišlák et al., 2009b, 2009a) is a new any-angle find-path algorithm that provides a different tradeoff than Basic Theta* with respect to the runtime of the search and the length of the resulting path. AA* finds paths that are shorter than those found by Basic Theta*, but it finds them more slowly than Basic Theta*.

There are two significant differences between AA* and Basic Theta*:

[1]The values in Tables E.1-E.3 are typically smaller than the values in Tables 4.1-4.4 because, on average, the start vertex and goal vertex are closer together for the path-planning problems used to generate Tables E.1-E.3.

		A* on Visibility Graphs (true shortest paths)	Basic Theta*	AA*
100×100	Random Grids 5%	10.0840	0.0230	0.0790
	Random Grids 10%	11.8960	0.0260	0.0820
	Random Grids 20%	18.4760	0.0360	0.1010
	Random Grids 30%	31.4930	0.0490	0.1290

Table E.5: Runtimes (Šišlák et al., 2009b)

		A* on Visibility Graphs (true shortest paths)	Basic Theta*	AA*
100×100	Random Grids 5%	56	172	138
	Random Grids 10%	102	216	162
	Random Grids 20%	198	302	238
	Random Grids 30%	294	372	324

Table E.6: Number of Vertex Expansions (Šišlák et al., 2009b)

- AA* chooses the four visible neighbors, $nghbr_{vis}(s)$, of an expanded vertex s by constructing what the authors refer to as a maximum unblocked square. A maximum unblocked square for a vertex s is constructed by expanding a square centered on vertex s that does not contain any blocked grid cells, until one side of the square touches either the goal vertex or a blocked grid cell. The four visible neighbors of vertex s are the vertices that are in the middle of the four sides that define the maximum unblocked square. The idea behind using this technique is to eliminate the examination of vertices that are not close to blocked grid cells since true shortest paths only contain heading changes at the corners of blocked grid cells.

- AA* does not consider Path 1 or Path 2 when updating the g-value and parent of an unexpanded visible neighbor s' of vertex s. Instead it uses a technique that is similar in spirit to the key vertices technique discussed in Section 4.9.2. When AA* updates the g-value and parent of an unexpanded visible neighbor s' of vertex s, it considers the paths from the start vertex to each key vertex and from them to vertex s' in a straight line. AA* updates the g-value and parent of vertex s' if any of the paths considered is shorter than the shortest path from the start vertex to vertex s' found so far. The set of key vertices is determined by constructing an ellipse that is defined by the start vertex, vertex s' and the g-value of vertex s' (see Šišlák et al. (2009b) for a more detailed description). Every vertex that is both within the ellipse and a member of the closed list is considered a key vertex.

AA* has a longer runtime than Basic Theta* because the number of key vertices it considers can be quite large, but it finds shorter paths than Basic Theta*. In fact, in all of the more than two thousand path-planning problems that were considered in the experimental comparison performed in the work of Šišlák et al. (2009b), AA* always found a true shortest path. The complete results on random grids from the work of Šišlák et al. (2009b) can be found in Tables E.4-E.6, which are

		A* on Octile Graphs (grid paths)	Basic Theta*	Block A*
500×500	Random Grids 0%	274.70	260.80	261.80
	Random Grids 10%	275.30	261.60	262.50
	Random Grids 20%	276.40	263.30	264.30
	Random Grids 30%	277.50	265.40	266.60
	Random Grids 40%	282.70	271.50	273.00
	Random Grids 50%	296.90	286.20	287.80

Table E.7: Path Lengths (Yap et al., 2011)

		A* on Octile Graphs (grid paths)	Basic Theta*	Block A*
500×500	Random Grids 0%	0.0048	0.0065	0.0010
	Random Grids 10%	0.0049	0.0042	0.0014
	Random Grids 20%	0.0050	0.0049	0.0019
	Random Grids 30%	0.0052	0.0063	0.0024
	Random Grids 40%	0.0058	0.0090	0.0032
	Random Grids 50%	0.0083	0.0148	0.0047

Table E.8: Runtimes (Yap et al., 2011)

annotated in same way as Tables E.1-E.3.[2] There were two differences between the experimental setup used to generate Tables E.4-E.6 and the experimental setup used by Šišlák et al. (2009b) to generate Tables E.1-E.3: they did not perform experimental comparisons on random 500×500 grids and they did not perform experimental comparisons on random 100×100 grids with 0 and 30 percent blocked grid cells. The results in these tables suggest that AA* does indeed find paths that are shorter than those found by Basic Theta* and that it does so more slowly than Basic Theta*. In addition, much like the results in Tables E.1 and 4.1, the results in Table E.4 demonstrate that Basic Theta* finds paths that are only slightly longer than true shortest paths. For example, in Table E.1, the paths found by Basic Theta* on random 100×100 grids with 10 percent blocked grid cells are $\approx 0.16\%$ longer than the true shortest paths, in Table 4.1, the paths found by Basic Theta* on random 100×100 grids with 10 percent blocked grid cells are $\approx 0.17\%$ longer than the true shortest paths and, in Table E.4, the paths found by Basic Theta* on random 100×100 grids with 10 percent blocked grid cells are $\approx 0.45\%$ longer than the true shortest paths. Results on random grids with different percentages of blocked grid cells are similar.

E.3.2 Block A*

Block A* (Yap et al., 2011) is a new any-angle find-path algorithm that provides a different tradeoff than Basic Theta* with respect to the runtime of the search and the length of the resulting path. Block A* finds paths that are longer than those found by Basic Theta*, but it finds them

[2] Additional experiments (not included here) were performed on six selected grids that were motivated by the work of LaValle (2006).

		A* on Octile Graphs (grid paths)	Basic Theta*	Block A*
500×500	Random Grids 0%	14957	918	638
	Random Grids 10%	15039	4439	845
	Random Grids 20%	15351	6229	1159
	Random Grids 30%	15889	8536	1617
	Random Grids 40%	18025	12603	2407
	Random Grids 50%	26146	22721	4476

Table E.9: Number of Vertex Expansions (Yap et al., 2011)

faster than Basic Theta*. Our description of Block A* is very similar to the one provided in the work of Yap et al. (2011).

Block A* uses a pre-computed database, a Local Distance Database (LDDB), to find paths faster. Prior to the search, Block A* divides a square grid into "blocks" of $m \times n$ contiguous grid cells. Assume that x and y are vertices on the boundary of a block. For every possible arrangement of blocked and unblocked grid cells within an $m \times n$ block of continuous grid cells, the LDDB stores a path between each (x, y) pair. The paths stored in an LDDB can be grid paths or any-angle paths. During the search, Block A* operates on blocks rather than individual vertices, although each vertex still maintains a g-value and parent. A block b is inserted into the open list (rather than an individual vertex) with its ingress vertices, that is, vertices around the perimeter of b whose g-values decreased when b was added to the open list. When b is expanded, the g-values and parents of the vertices along the perimeter of a block b' that is adjacent to b are updated. Two blocks are adjacent if they share a side. Block A* updates the egress vertices of b, that is, the vertices along the side that is common to both b and b'. Block A* considers the paths from the start vertex to each ingress vertex of b and from them to the ingress vertices of b'. The lengths of the paths from the ingress vertices of b to the ingress vertices of b' do not need to be computed because they are stored in the LDDB. b' and its ingress vertices are then added to the open list with a key equal to the smallest f-value among the f-values of the ingress vertices.

The operation of Block A* is similar to the operation of A* and Basic Theta*, although there are two important differences that make Block A* more difficult to implement and understand: **(1)** LDDBs can be quite large. For an $n \times n$ block the number of possible arrangements of blocked and unblocked grid cells is $2^{n \times n}$ and the number of total entries in the LDDB is $2^{n \times n} \times 4(n - 1) \times (4(n - 1) - 1)$. Each entry in the LDDB is a path that must be computed which can be time consuming if n is large. The authors acknowledge this problem and suggest that symmetry compression can be used to reduce the size of an LDDB. **(2)** The start block and goal block (that is, the blocks that contain the start vertex and goal vertex, respectively) must be handled differently than all of the other blocks.

Block A* has a shorter runtime than Basic Theta* because it uses an LDDB, but it finds longer paths than Basic Theta* (and is more difficult to implement and understand). The complete results on random grids from the work of Yap et al. (2011) can be found in Tables E.7-E.9, which are annotated in the same way as Tables E.1-E.3.[3] There were the following differences between the experimental setup used by Yap et al. (2011) to generate Tables E.7-E.9 and the experimental

[3] A vertex expansion in A* and Basic Theta* is very different from a vertex expansion in Block A*.

setup used to generate Tables E.1-E.3: they used the straight-line Euclidean distances as h-values instead of the octile distances for A*, they performed experimental comparisons on many more path-planning problems for each percentage of blocked grid cells, they did not perform experimental comparisons on random 100×100 grids, they performed experimental comparisons on random 500×500 grids with 40 and 50 percent blocked grid cells and they did not perform experimental comparisons on random 500×500 grids with 5 percent blocked grid cells. The results in these tables suggest that Block A* does indeed find paths that are longer than those found by Basic Theta* and that it finds them faster than Basic Theta*.[4] In addition, much like the results in Tables E.1 and 4.1, the results in Table E.7 and E.8 demonstrate that A* finds paths that are $\approx 4 - 5\%$ longer than the paths found by Basic Theta* for all percentages of blocked grid cells on average and that Basic Theta* is faster than A* with the straight-line Euclidean distances as h-values on random 500×500 grids with 10 and 20 percent blocked grid cells.

In a short period of time, the work in this dissertation has garnered a significant amount of interest from roboticists and video game developers. Any-angle find-path algorithms are a class of algorithms that researchers are interested in because they address the problems presented by traditional edge-constrained find-path algorithms without any significant drawbacks. Furthermore, this brief survey suggests that Basic Theta* is an important benchmark any-angle find-path algorithm with which new any-angle find-path algorithms will be compared. This is not only because Basic Theta* quickly finds short and realistic looking paths, but because it is simple to implement and understand.

[4]The experiments in the work of Yap et al. (2011) were performed with an optimized version of A* and the line-of-sight check provided in Appendix A (Yap, 2011). In their implementation of Basic Theta* the runtime of the line-of-sight check became a significant bottleneck. However, as we mentioned in Section E.2.1, the line-of-sight check provided in Appendix A can be optimized, which would reduce the runtime of Basic Theta* relative to A* and Block A*. Furthermore, Lazy Theta*, which they did not compare against, can be used to significantly reduce the total number of line-of-sight checks performed.