Efficient Bounded-Suboptimal Multi-Agent Path Finding and Motion Planning via Improvements to Focal Search

by

Liron Cohen

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

August 2020

Copyright 2020 Liron Cohen

To Avi, Orna, Shir and Gil

Acknowledgements

This work would not be possible without the support of my lovely adviser, Prof. Sven Koenig. I am grateful for your time, effort, patience, dedication and life lessons. More than anything, thank you for your fierce and constructive criticism which was invaluable in pushing me to develop my ideas and communicate them clearly (both in papers and in presentations).

To my committee members, Prof. Maxim Likachev, Prof. Bistra Dilkina and Prof. Peng Shi, thank you for your helpful comments and thoughtful suggestions. I also thank Prof. Fei Sha and Prof. Nora Ayanian for helpful discussions on an earlier version of this dissertation. I was lucky to have Prof. Ariel Felner and Prof. Carlos Hernandez visiting our lab for an extended period of time. I thank both of you for interesting and fruitful discussions on heuristic search.

I was fortunate to work with extremely bright collaborators. First and foremost, I thank Dr. Tansel Uras and Prof. Satish Kumar for the countless hours of brainstorming, debugging ideas and valuable insights. More than your professional support, I treasure your friendship. I also thank Dr. Wolfgang Hoenig, Dr. Hang Ma, Dr. Hong Xu and (soon to be Dr.) Jiaoyang Li for helpful discussions.

Finally, I would like to thank my family for their unconditional love. To my parents, Avi and Orna, I am grateful for your ever lasting mental and financial support. To my siblings, Shiran and Gilran, I am proud that you consistently chose to spend all your vacations with me in Los Angeles, and always found the time to lift my spirit over Skype. To my dog, Paz, who kept me in good shape and took me on daily walks to clear my mind. No words can express how much you all mean to me.

The research presented in this dissertation was supported by NSF under grant numbers 1409987, 1724392, 1319966, 1817189 and 1837779, NASA via Stinger Ghaffarian Technologies and a gift from Amazon.

Table of Contents

Acknowledgements **v**

List Of Tables **ix**

List Of Figures **xi**

Abstract **xviii**

Chapter 1: Introduction **1**
1.1 Contributions 3
1.2 Outline 5

Chapter 2: Background **7**
2.1 The Multi-Agent Path Finding Problem 7
2.2 Multi-Agent Path-Finding Solvers 11
2.2.1 Optimal MAPF Solvers 11
2.2.1.1 Conflict-Based Search 15
2.2.2 Suboptimal MAPF Solvers 16
2.2.3 Bounded-Suboptimal MAPF Solvers 18
2.2.4 Others 18

Chapter 3: Anytime MAPF **21**
3.1 Introduction 22
3.2 Background 24
3.2.1 Bounded-Suboptimal Search and Bounded-Cost Search 24
3.2.2 Focal Search 24
3.3 Anytime Focal Search 25
3.3.1 Priority Functions 27
3.3.2 Anytime Bounds 30
3.3.3 Pseudocode 31
3.3.4 Theoretical Properties 32
3.4 Anytime Conflict-Based Search 35
3.5 Experimental Results 36
3.5.1 The MAPF Domain 36
3.5.2 The Generalized Covering Traveling Salesman Problem Domain 45
3.6 Conclusions and Future Work 47

Chapter 4: The Highway Heuristic **49**
4.1 Introduction . . . 50
4.2 Background . . . 52
4.2.1 Experience Graphs . . . 52
4.2.2 Enhanced Conflict-Based Search . . . 52
4.3 Focal Search with Inflated Heuristics . . . 53
4.4 The Highways Heuristic . . . 55
4.5 Conflict-Based Search with Highways . . . 60
4.5.1 Enhanced Conflict-Based Search with Highways . . . 62
4.5.2 Experiments . . . 63
4.5.3 M* with Highways . . . 81
4.6 Automatically Generating Highways . . . 83
4.6.1 Crisscross Highways . . . 84
4.6.2 Graphical Model-Based Highways . . . 85
4.6.3 Heat Map-Based Highways . . . 94
4.6.4 Limitations . . . 100
4.7 Conclusions and Future Work . . . 101

Chapter 5: Multi-Agent Motion Planning **103**
5.1 Introduction . . . 104
5.2 Background . . . 106
5.2.1 Motion Primitives and State Lattices . . . 106
5.2.2 Safe Interval Path Planning . . . 107
5.3 Focal Search with Reasoning About Wait Actions in Bulk . . . 107
5.4 The Multi-Agent Motion-Planning Problem . . . 114
5.5 Adapting Conflict-Based Search to Multi-Agent Motion Planning . . . 116
5.5.1 Reservation Table . . . 119
5.5.2 Reasoning About Wait Actions in Bulk with Timeintervals . . . 123
5.5.3 Soft Collisions Interval Path Planning . . . 133
5.5.3.1 Viewing SCIPP as a w-suboptimal Version of SIPP . . . 137
5.6 Experiments . . . 139
5.7 Conclusions and Future Work . . . 147

Chapter 6: Conclusions and Future Work **149**

Reference List **154**

List Of Tables

1.1 Acronyms 6

3.1 Success rates of anytime BCBS, CBS and ECBS for different numbers of agents in the $32{\times}32_20\%$, Drones and Kiva environments. Success rate* indicates success rate over the MAPF problem instances that CBS failed to solve. 40

4.1 Shows the different factors of our GM. The first digit in the value of X_i indicates whether ('1') or not ('0') there is a collision in the cell. The second digit indicates whether the magnitude of DV(i) is greater than 1/2 ('1') or not ('0'). The third digit indicates whether the direction of DV(i) is in the eastern quadrant $[315°, 45°)$ ('1'), northern quadrant $[45°, 135°)$ ('2'), western quadrant $[135°, 225°)$ ('3') or southern quadrant $[225°, 315°)$ ('4'). '?' represents a wildcard. 85

4.2 ASL values for the null hypothesis "ECBS+GM is not more efficient than ECBS" for each environment and number of agents. 93

4.3 ASL values for the null hypothesis "ECBS+HWY is not more efficient than ECBS+GM" for each environment and number of agents. 93

4.4 ASL values for the null hypothesis "ECBS is not more efficient than ECBS+HM" for the Kiva environment and "ECBS+HM is not more efficient than ECBS" for the Drones and Roundabout environments for each number of agents. "–" appears in all entries for which the success rate of ECBS+HM is 0%. 99

4.5 ASL values for the null hypothesis "ECBS+HWY is not more efficient than ECBS+HM" for each environment and number of agents. 99

List Of Figures

1.1 Shows real-world application domains of autonomous agents navigating in different environments. (a) Amazon fulfillment center (Photo credit: The Boston Globe). (b) Autonomous aircrafts conducting operations in low-altitude airspace above a city (Photo credit: NASA). (c) Autonomous forklifts in a warehouse (Photo credit: SME Group). . . . 2

1.2 Summarizes the contributions of this dissertation. 5

2.1 Shows a MAPF problem instance with two agents, red and blue. (a) A 4-neighbor grid environment with two agents. The start and goal cells of the agents, s^1, s^2, g^1 and g^2, are specified. (b) The graph representation of the given environment with arbitrary vertex labels. For these labels, the start and goal vertices are: $s^1 = v_4$, $g^1 = v_{10}$, $s^2 = v_2$ and $g^2 = v_{11}$. (c) Feasible and collision-free paths for the agents (depicted with their respective colors). Two discs in the same vertex represent a wait action. 8

2.2 Illustrates vertex and edge collisions. 9

2.3 Illustrates the high-level search tree of CBS for a MAPF problem instance with two agents, red and blue, in a 4×4 4-neighbor grid environment. Actions are assumed to have unit cost. The start and goal cells of the red agent are $[1, 0]$ and $[2, 3]$, respectively. The start and goal cells of the blue agent are $[0, 1]$ and $[3, 2]$, respectively. Each dot represents a timestep and, therefore, two dots in the same cell represent a wait action. The cost of a high-level state is the total arrival time defined by its paths. The collision that CBS chooses to resolve in each high-level state is specified below it, and the constraints specified in a high-level state are specified above it. 17

3.1 Illustrates FOCAL (black) and OPEN (black+grey) for BSS and BCS in (a) and (b), respectively. 25

3.2 (a) Shows an $h_{\texttt{FOCAL}}$ that is not w-admissible. The h-value of each state is specified with the state (in blue) and $c(s_i, s_j)$ is specified near the edge (s_i, s_j). (b) Shows that FS may require re-expansions to guarantee bounded-suboptimality. 28

3.3 Illustrates the anytime effect on FOCAL for the three different ways of updating the bounds. 30

3.4 Shows the 32×32_20% environment. The start and goal cells of 100 agents in this MAPF problem instance are shown in cyan and green, respectively. 37

3.5 Shows the Drones environment. The start and goal cells of 100 agents in this MAPF problem instance are shown in cyan and green, respectively. 38

3.6 Shows the Kiva environment. The start and goal cells of 100 agents in this MAPF problem instance are shown in cyan and green, respectively. 39

3.7 (a), (b), (c) and (d) show the typical behavior of anytime BCBS on a few MAPF problem instances in the 32×32_20% environment with 50, 70, 90 and 110 agents each, respectively. (e) shows the aggregate behavior of anytime BCBS for numbers of agents with more than 50% success rate in the 32×32_20% environment. 41

3.8 (a), (b), (c) and (d) show the typical behavior of anytime BCBS on a few MAPF problem instances in the Drones environment with 10, 20, 30 and 40 agents each, respectively. (e) shows the aggregate behavior of anytime BCBS for numbers of agents with more than 50% success rate in the Drones environment. 42

3.9 (a), (b), (c) and (d) show the typical behavior of anytime BCBS on a few MAPF problem instances in the Kiva environment with 10, 20, 30 and 40 agents each, respectively. (e) shows the aggregate behavior of anytime BCBS for numbers of agents with more than 50% success rate in the Kiva environment. 43

3.10 Shows the behaviors of anytime BSS and BCS in the GCTSP domain. The left column shows behaviors on typical medium and large size problem instances. The right column shows aggregate behaviors on 69 medium and 14 large problem instances. BKS stands for best known solution. 46

4.1 Illustrates that the agents' start (s^i) and goal (g^i) cells influence whether the given highways are helpful. Red arrows represent the edges of the given highways. (a) shows a MAPF problem instance with 10 agents, all of which need to move from right to left. In this case, the given highways are not helpful. (b) shows a MAPF problem instance with 20 agents, of which 10 need to move from right to left and the other 10 need to move from left to right. In this case, the given highways are helpful. 51

4.2 Differences between $h_{\texttt{SP}}$ and $h_{\texttt{HWY}}$ in a simple environment. (a) shows a 4×4 4-neighbor grid environment with no obstacles, and two start and goal cells for the red and blue agents. (b) and (c) show $h_{\texttt{SP}}$ values for the red and blue agents, respectively. (d) shows the highways. (e) and (f) show $h_{\texttt{HWY}}$ values for the highways in (d) and an inflation factor of $w = 2$ for the red and blue agents, respectively. 55

4.3 Illustrates how agents spread out in ECBS for the MAPF problem instance depicted in Figure 4.1(b). All paths shown are the ones in the high-level root state of ECBS(2). (a) and (b) show the paths that ECBS(2) computes for two agents early in the process of building the high-level root state. When these paths are computed, the reservation table is almost empty. Thus, these paths are the corresponding optimal paths from their start to their goal cells. (c) and (d) show the paths that ECBS(2) computes for two other agents later in the process of building the high-level root state. When these paths are computed, the reservation table contains many other agents' paths. Thus, these later paths meander. . 56

4.4 Average runtimes (left column) and success rates (right column) of ECBS with suboptimality bounds $w = \{1.1, 1.2, 1.5, 2.2, 3, 6.6\}$ for the Kiva, Drones and Roundabout environments. 58

4.5 Illustrates the effect of using $h_{\mathtt{HWY}}$ instead of $h_{\mathtt{SP}}$ for the MAPF problem instance depicted in Figure 4.1(b). Like in Figure 4.3, it shows the paths computed by ECBS(2) in its high-level root state for the first two agents ((a) and (b)) and last two agents ((c) and (d)) it plans for. Instead of $h_{\mathtt{SP}}$, however, ECBS(2) uses $h_{\mathtt{HWY}}$ for the highway specified in Figure 4.1(b) and an inflation factor of 2. While the paths in (a) and (b) are similar to the ones of ECBS(2) with $h_{\mathtt{SP}}$, the paths in (c) and (d) are less meandering and, unlike in Figure 4.3(c) and (d), utilize the lower row of the corridor. 60

4.6 Illustrates the high-level tree of CBS+HWY(2) for the example in Figure 4.2(a). 61

4.7 Shows the Roundabout environment. The start and goal cells of all agents in a MAPF problem instance with 100 agents are shown in cyan and green, respectively. 64

4.8 Shows the human-specified highways for the Kiva, Drones and Roundabout environments in (a), (b) and (c), respectively. 66

4.9 Shows the average runtimes and success rates of ECBS(1.1)+HWY, ECBS(1.2)+HWY and ECBS(1.5)+HWY for different inflation factors in the Kiva environment. 68

4.10 Shows the average runtimes and success rates of ECBS(1.1)+HWY, ECBS(1.2)+HWY and ECBS(1.5)+HWY for different inflation factors in the Drones environment. 69

4.11 Shows the average runtimes and success rates of ECBS(1.1)+HWY, ECBS(1.2)+HWY and ECBS(1.5)+HWY for different inflation factors in the Roundabout environment. 70

4.12 Shows the average runtimes and success rates of 2.2-, 3.0- and 6.6-suboptimal MAPF solvers in the Kiva environment. 72

4.13 Shows the average runtimes and success rates of 2.2-, 3.0- and 6.6-suboptimal MAPF solvers in the Drones environment. 73

4.14 Shows the average runtimes and success rates of 2.2-, 3.0- and 6.6-suboptimal MAPF solvers in the Roundabout environment. 74

4.15 Shows the average runtimes and success rates of the most efficient values of the ones we experimented with for the Kiva, Drones and Roundabout environments. 76

4.16 Shows the solution cost (x-axis) and runtime (y-axis) of each MAPF problem instance for a given number of agents of ECBS(2.2) and ECBS(1.5)+HWY(3). 77

4.16 Shows the solution cost (x-axis) and runtime (y-axis) of each MAPF problem instance for a given number of agents of ECBS(2.2) and ECBS(1.5)+HWY(3). 78

4.17 Shows the solution cost (x-axis) and runtime (y-axis) of each MAPF problem instance for a given number of agents of ECBS(3) and ECBS(1.5)+HWY(6). 79

4.18 Shows the solution cost (x-axis) and runtime (y-axis) of each MAPF problem instance for a given number of agents of ECBS(3) and ECBS(1.5)+HWY(6). 80

4.19 (a), (b) and (c) show the average solution cost of ECBS(2.2) and ECBS(1.5)+HWY(3) (ECBS(3) and ECBS(1.5)+HWY(6)) for the Kiva environmnet (Drones and Roundabout environments). . . . 82

4.20 Shows the average runtimes and success rates M* and M*+HWY(2) in the Kiva environment. 83

4.21 Shows the CC highways for the Kiva environment and $h_{\texttt{HWY}}$ values for goal cell [18,51] with $w = 2$. . . . 86

4.22 Shows the CC highways for the Kiva environment and $h_{\texttt{HWY}}$ values for goal cell [18,51] with $w = 6$. . . . 87

4.23 (a) shows a GM over a 4x4 grid environment. (b) shows the mapping from angles to discrete values. . . . 88

4.24 Shows the highways generated by the GM-based approach for MAPF problems with 150 agents in the Kiva environment (a), 120 agents in the Drones environment (b) and 150 agents in the Roundabout environment (c). . . . 90

4.25 Shows the average runtimes and success rates of ECBS(2.2), ECBS(1.5)+HWY(3) and ECBS(1.5)+GM(3) (ECBS(3), ECBS(1.5)+HWY(6) and ECBS(1.5)+GM(6)) for the Kiva environmnet (Drones and Roundabout environments). . . . 92

4.26 Shows the highways generated by the HM-based approach for MAPF problems with 150 agents in the Kiva environment (a), 120 agents in the Drones environment (b) and 150 agents in the Roundabout environment (c). . . . 96

4.27 Shows the average runtimes and success rates of ECBS(2.2), ECBS(1.5)+HWY(3) and ECBS(1.5)+HM(3) (ECBS(3), ECBS(1.5)+HWY(6) and ECBS(1.5)+HM(6)) for the Kiva environmnet (Drones and Roundabout environments). . . . 98

4.28 Shows two environments with the same start and goal cells of the agents (in cyan and green, respectively), and their GM-based highways (in red). In (a), straight corridors connect open spaces, and the GM-based highways utilize (almost) all of the edges in the corridors. (The only cell in the corridor without a highway edge is the one in which the corridor is not straight.) In (b), zig-zag corridors connect open spaces, and the GM-based highways do not utilize many edges in the corridors. . . . 100

4.29 Shows an environment with a central corridor and a peripheral corridor. Assuming that all the start and goal cells are not inside corridors, only the optimal paths for the start and goal cells on the rightmost column or the bottommost row (corresponding to the blue colored cells in (a)) may utilize the peripheral corridor, and all other optimal paths necessarily utilize the central corridor. In (b), the start and goal cells (in cyan and green, respectively) from Figure 4.28 are specified. Only agent 8 may utilize the peripheral corridor. Red arrows show the GM-based highways, which, as expected, do not utilize the peripheral corridor. . . . 101

5.1 Illustrates the inefficiency of reasoning about wait actions "one ϵ at a time". Assume that any path to the goal configuration involves the action (v, v') and that $\{(v, t, v', t+d), (v, t+\epsilon, v', t+d+\epsilon), \ldots, (v, t+566\epsilon, v', t+d+566\epsilon)\} \in$ `constraints`. When FS expands the state (v, t), it can only generate $(v, t+\epsilon)$ for a (single) wait action. Thus, FS has to expand and generate 567 states (and possibly many other states) in order to find a path to state $(v, t+567\epsilon)$, from which the actions (v, v') does not violate any constraint. 111

5.2 Illustrates bulk generation and bulk expansion. The x-axis represents time, and the y-axis represents configurations. Crosses represent states, and lines represent actions between states. Black color represents traditional expansions or generations, and blue color represents additional expansions or generations that are done in bulk. Red dotted lines represent constraints. A red dot on the horizontal line for v (v') represents a constraint (v, τ) ((v', τ)) at the time τ of its x-coordinate, and a red dot between v and v' represents a constraint (v, t, v', t') at the time $\tau \in [t, t']$ of its x-coordinate. (a) illustrates bulk generated states in blue crosses. Each state is generated using wait actions after reaching v', which the parent pointers in blue represent. (b) illustrates that it is possible to aggregate all states generated in (a) using timeintervals (see Chapter 5.5.2). (c) and (d) illustrate bulk expanded states in blue crosses. Each state is generated by either: waiting in v before executing the action (illustrated in (c)) or arriving at v later and executing the action immediately (illustrated in (d)). Any combination of arriving at v later and waiting in v is also possible (this amounts to different paths of getting to v' with the same g-value). (e) illustrates that it is possible to aggregate all states generated in (c) and (d) using timeintervals (see more in Chapter 5.5.2). 112

5.3 (a) Illustrates an action from v (depicted by the bottom triangle) to v' (depicted by the top triangle) with a duration of 60. Denote by $e = (v, v')$ the edge representing this action. Thus, $w(e) = 60$. Each swept cell c_i^e is depicted by a circle, and $[lb_i^e, ub_i^e]$ is shown for four swept cells. Note that $\min_i lb_i^e = 0$ and $\max_i ub_i^e = 60$. (b) Illustrates two actions, red and blue, colliding at one swept cell since $[10, 15] \cap [10, 35] \neq \emptyset$. 115

5.4 (a) Shows two agents, red and blue, colliding in a swept cell c during $[2{,}000, 3{,}000]$. (b) A constraint $(c, [2{,}000, 3{,}000]$ forbids an optimal path in which the blue agent sweeps c during $[2{,}000, 2{,}500]$ and the red agent sweeps c during $[2{,}501, 3{,}001]$. 118

5.5 Illustrates different choices of a timestep τ for a constraint. Consider two agents, red and blue, that sweep a cell c during overlapping timeintervals. The blue agent sweeps c during $[100, 392]$, and the red agent sweeps c during $[81, 150]$. Therefore, the two agents collide in that swept cell during $[100, 150]$. (a)-(f) show the times during which the red and blue agents occupy the swept cell. (a) Choosing $\tau < 81$ or $\tau > 392$ will not enforce a change of path for any of the two agents and is thus not helpful. (b) Choosing $81 \leq \tau < 100$ or $150 < \tau \leq 392$ will enforce a change of path for just one of the two agents and is thus not helpful. (c) Choosing $\tau = 100$ will enforce a change of path for both agents. While this is helpful, the agents are likely to collide again during $[101, 150]$. (d) continues (c) with one additional constraint for $\tau = 101$. Once again, while this enforces a change of path for both agents, they are likely to collide again during $[102, 150]$. (e) Illustrates that continuing to choose $\tau = lb$ (for a given duration $[lb, ub]$ of a collision) is likely to lead to a (possibly very long) sequence of replanning operations, each of which results in the agent delaying the execution of the same action by only one timestep. (f) Illustrates our choice of $\tau = ub$ (in this example, $\tau = 150$). This choice guarantees that the replan operations for the red and blue agents cannot simply delay the execution of some actions in a way that causes them collide again in c during $[100, 150]$. 120

5.6 Illustrates the execution window that a constraint (c, τ) blocks for a given action. (a) shows an action $e = (v, v')$ that sweeps cell c (shown in black) during the timeinterval $[t_1, t_2]$. (b) illustrates a constraint that forbids an agent from occupying c at τ. Here, the x-axis represents time, and the y-axis represents the agent's progression through c (specifically, the bottom gray line signifies the agent entering c, and the top gray line signifies the agent leaving c). (c, τ) implies that the agent has to either: 1) leave c before τ (that is, it may occupy c at $\tau - 1$ at the latest); or 2) enter c after τ (that is, it may occupy c at $\tau + 1$ at the earliest). Thus, the agent has to start executing the action so that it either: 1) enters c before $\tau - (t_2 - t_1)$ (otherwise, it may not leave c before τ); or 2) leaves c after $\tau + (t_2 - t_1)$ (otherwise, it may enter c before τ). (c) is similar to (b) but shows the implication of (c, τ) on e, namely, starting to execute e during $[\tau - t_2, \tau - t_1]$ violates (c, τ). Thus, getting to v' using e during $[\tau - t_2 + d, \tau - t_1 + d]$, where d is the duration of e, is forbidden by (c, τ). 121

5.7 Shows three agents, red, blue and green, sweeping the encircled cell during overlapping timeintervals. The aggregate-on-overlap operation produces 4 timeintervals, each with an associated number of collisions, as indicated in black. 122

5.8 Illustrates the colliding timeinterval for a given action when another agent sweeps a cell c during the timeinterval $[\tau_1, \tau_2]$. (a) shows an action $e = (v, v')$ that sweeps c (shown in black) during the timeinterval $[t_1, t_2]$. (b) illustrates the timeinterval $[\tau_1, \tau_2]$ on c. Here, the x-axis represents time, and the y-axis represents the agent's progression through c (specifically, the bottom gray line signifies the agent entering c, and the top gray line signifies the agent leaveing c). Another agent sweeping c during $[\tau_1, \tau_2]$ implies that the agent executing e has to either: 1) leave c before τ_1 (that is, it may occupy c at $\tau_1 - 1$ at the latest); or 2) enter c after τ_2 (that is, it may occupy c at $\tau_2 + 1$ at the earliest). Thus, the agent has to start executing the action so that it either: 1) enters c before $\tau_1 - (t_2 - t_1)$ (otherwise, it may not leave c before τ_1); or 2) leaves c after $\tau_2 + (t_2 - t_1)$ (otherwise, it may enter c before τ_2). (c) is similar to (b) but shows the implication of another agent sweeping c during $[\tau_1, \tau_2]$ on e, namely, starting to execute e during $[\tau_1 - t_2, \tau_2 - t_1]$ results in a collision. Thus, getting to v' using e during $[\tau_1 - t_2 + d, \tau_2 - t_1 + d]$, where d is the duration of e, is associated with a collision. 125

5.9 Illustrates how *GenerateIntervals* works. (a) describes an action $e = (v, v')$ of duration $16{,}000$. The swept cells of e are shown in black circles. The timeintervals during which some cells are swept are also specified. A list of constraints and collisions is specified on the right side. Assume that the agent occupies cell $(8, C)$ while waiting in v and the cell $(2, B)$ while waiting in v'. (b) illustrates steps (1) and (2) of *GenerateIntervals*. v' is generated via e where waiting in v before execution is possible during $[10{,}000, 120{,}000]$. Thus, $I_e = [26{,}000, 136{,}000]$. The relevant constraints for (v, v') during I_e are shown in red dotted lines. Removing the blocked timeintervals results in the following intervals: $\{[26{,}000, 34{,}999], [38{,}001, 79{,}999], [82{,}000, 136{,}000]\}$. (c) illustrates step (3) of *GenerateIntervals*. The relevant collisions for (v, v') during I_e are shown in orange dotted lines. Aggregating-on-overlap the number of collisions results in the following annotated intervals: $\{([26{,}000, 34{,}999]; 0), ([38{,}001, 49{,}499]; 0), ([49{,}500, 56{,}000]; 2), ([56{,}001, 79{,}999]; 0), ([82{,}000, 103{,}999]; 0), ([104{,}000, 108{,}000]; 1)([108{,}001, 136{,}000]; 0)\}$. 129

5.9 Illustrates how *GenerateIntervals* works (continued). (d) illustrates steps (4) and (5) of *GenerateIntervals*. Since the earliest relevant constraint on v' during $[136, \infty]$ is 150, $I_w = [26{,}000, 150{,}000]$. The relevant constraints for v' during I_w are shown in red dotted lines. Removing them from I_w results in the following timeintervals: $\{[26{,}000, 79{,}999], [80{,}001, 149{,}999]\}$. (e) illustrates step (6) of *GenerateIntervals*. The relevant collisions for v' during I_w are shown in orange dotted lines. Aggregating-on-overlap the number of collisions results in the following timeintervals: $\{([26{,}000, 79{,}999]; 0), ([80{,}001, 103{,}999]; 0), ([104{,}000, 106{,}000]; 1), ([106{,}001, 139{,}999]; 0), ([140{,}000, 145{,}000]; 3), ([145{,}001, 149{,}999]; 0)\}$. (f) illustrates step (7). Each row shows one iteration, starting from L_e and ending at the returned annotated timeintervals: $\{([26{,}000, 79{,}999]; 0), ([80{,}001, 103{,}999]; 0), ([[104{,}000, 108{,}000]]; 1), ([108{,}001, 139{,}999]; 0), ([140{,}000, 150{,}000]; 3)$ . . . 130

5.10 Illustrates lemmas 35, 36 and 37 in (a), (b) and (c), respectively. Lemma 35 shows that the set of timesteps in L_e is equal to the set of timesteps generated by the bulk expansion part of FS-B. Lemma 36, shows that the set of timesteps in L_w is a superset of the set of timesteps generated by the bulk generation part of FS-B. Lemma 37 shows that the set of timesteps in L_{ew} is equal to the set of timesteps generated by FS-B (both the bulk expansion and bulk generation parts). . . . 131

5.11 Illustrates duplicate detection using *Merge* for two states having the same configuration and overlapping timeintervals. . . . 136

5.12 (a) shows an action (v, v') in blue. (b) illustrates the difference between SIPP and SCIPP for (a). . . . 138

5.13 Shows the Arena environment with 49×49 cells. Unblocked cells are black, and blocked cells are white. . . . 140

5.14 Shows the Den520d environment with 256×257 cells. Unblocked cells are black, and blocked cells are white. . . . 141

5.15 (a) and (b) show the Unicycle and the PR2 motion primitives, respectively, for (x, y, θ) in 5×18 free cells for the start state depicted in blue. For a motion primitive, a black line represents the trajectory of the center of the agent's footprint, and a red triangle at its end represents the orientation at the successor state. . . . 141

5.16 Shows the average runtimes (left column) and success rates (right column) of ECBS-CT with suboptimality bounds $w \in \{1, 1.2, 1.5, 2\}$ and the Unicycle motion primitives (top row) or the PR2 motion primitives (bottom row) in the Arena environment. . . . 143

5.17 Shows the average runtimes (left column) and success rates (right column) of ECBS-CT with suboptimality bounds $w \in \{1, 1.2, 1.5, 2\}$ and the Unicycle motion primitives (top row) or the PR2 motion primitives (bottom row) in the Den520d environment. . . . 144

Abstract

Cooperative autonomous agents navigating in different environments can be useful in real-world application domains such as warehouse automation, search-and-rescue, fighting forest fires, traffic control, computer games and mining. Agents are required to navigate in such complex environments while avoiding obstacles and each other. In Artificial Intelligence (AI), a simplified model of this problem is called the Multi-Agent Path Finding (MAPF) problem. In MAPF, time is discretized into timesteps and the environment is discretized into cells that individual agents can occupy exclusively at any given timestep. Along with the environment, the MAPF problem also specifies unique start and goal cells for each agent. A solution to the MAPF problem is a set of paths, one for each agent, that take the agents from their respective start cells to their respective goal cells without collisions. A path for an agent is a sequence of move (between adjacent cells) or wait (at a cell) actions, each with some duration in timesteps.

Different measures for the cost of a solution to the MAPF problem, such as sum-of-costs or makespan, are commonly used in AI. Unfortunately, finding an optimal solution to a MAPF problem according to any of these cost measures is NP-hard, thus explaining why optimal MAPF solvers are inefficient in many real-world application domains. On the other end of the spectrum, suboptimal MAPF solvers are efficient but can be ineffective or even incomplete. One framework that balances the trade-off between efficiency and effectiveness is that of bounded-suboptimality. A bounded-suboptimal solver takes a user-specified constant $w \geq 1$ and returns a solution with a cost guaranteed to be at most w times the cost of an optimal solution. On the one hand, the freedom to explore suboptimal solutions allows these solvers to be more efficient than optimal ones. On the other hand, and as opposed to suboptimal solvers, these solvers ensure that their solutions are effective.

Unfortunately, state-of-the-art bounded-suboptimal MAPF solvers, all of which rely heavily on a bounded-suboptimal heuristic search method called Focal Search (FS), have a few shortcomings. First, small changes in w can significantly affect these solvers' runtime. Thus, it is often difficult to determine a w such that an effective solution is found efficiently. Second, these solvers are inefficient when agents are huddled together and the environment has some structural components that create bottlenecks. One such example is a typical warehouse domain. Here, the environment has long narrow corridors connecting open spaces or other corridors, and agents have to move between different regions of the warehouse through the corridors. Third, these solvers are inefficient when move actions have different durations. One such example is actions that model kinodynamically feasible motions. These shortcomings make bounded-suboptimal MAPF solvers inefficient in many real-world application domains, thus making suboptimal MAPF solvers the only viable option. This inefficiency is problematic because a low solution cost is often important for performing the task at hand sufficiently well.

In this dissertation, we improve FS in ways that resolve the above shortcomings of bounded-suboptimal MAPF solvers. Specifically, we develop an anytime framework for FS, which alleviates the need to choose w carefully. An anytime bounded-suboptimal MAPF solver based on this framework finds a "good" solution quickly and refines it to better and better solutions if time allows. We also show that FS provides bounded-suboptimality guarantees even when it is used with inflated heuristics. We develop an inflated heuristic, called the highway heuristic, which improves the efficiency of bounded-suboptimal MAPF solvers when the domain's environment has structural components that create bottlenecks. The highway heuristic alters the paths that agents choose by biasing the agents away from their individual optimal paths and towards paths on shared "highways" (directional lanes) in the environment. This results in implicit coordination of the agents, which, in turn, increases the efficiency of bounded-suboptimal MAPF solvers. Finally, we develop a version of FS that reasons about wait durations "in bulk" by using timeintervals instead of timesteps. A bounded-suboptimal MAPF solver that uses this version is efficient even when move actions have different durations, thus making it suitable for solving MAPF problems with actions that model kinodynamically feasible motions.

On the theoretical side, we formally prove that our improvements are bounded-suboptimal. On the experimental side, we show that our improvements result in increased efficiency in domains inspired by real-world applications. The overall impact of this dissertation is twofold. The first impact is in opening up the possibility of using bounded-suboptimal MAPF solvers for new application domains. New application domains include ones in which existing bounded-suboptimal MAPF solvers are not applicable, such as when the agents are kinodynamically constrained (for example, forklifts), as well as ones in which existing bounded-suboptimal MAPF solvers are too inefficient, such as automated warehouses. Different benefits of using bounded-suboptimal MAPF solvers in such application domains include safety (that is, the solutions are guaranteed to be collision-free), completeness and effectiveness (since the cost of the solution is guaranteed to be at most w times the cost of an optimal solution). The second impact is in revitalizing FS. Compared to other heuristic search methods, such as A* and wA*, FS is less restricted in the states it can choose to expand and can take advantage of a broader family of heuristics. Nevertheless, A* has received significantly more attention than FS from the scientific community.[1] Our improvements to FS make it more applicable and call for revisiting its importance.

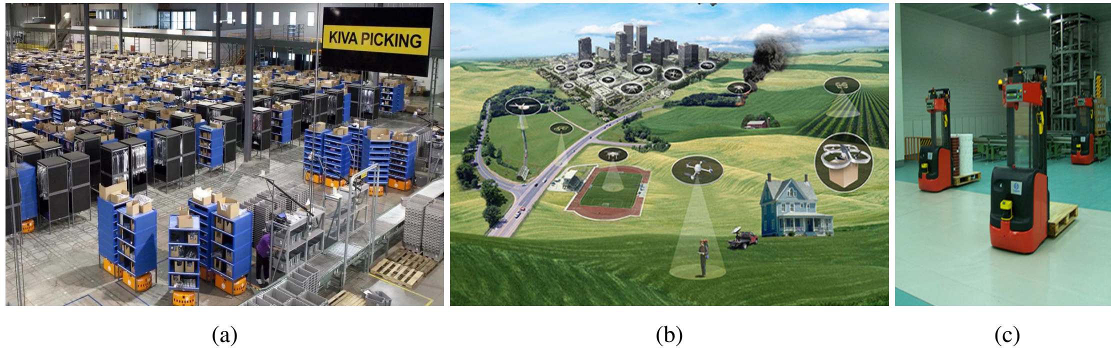
[1] According to Google Scholar, as of May 15, 2020, the first paper to introduce A* has been cited 9,522 times while the first paper to introduce FS has been cited 83 times.

Chapter 1

Introduction

Cooperative autonomous agents navigating in different environments without colliding with obstacles or with each other can be useful in real-world application domains such as warehouse automation, yard operations automation, delivery and transportation services, search-and-rescue, fighting forest fires, traffic control, computer games and mining. Figure 1.1 illustrates three such domains. Figure 1.1(a) shows an Amazon fulfillment center with multiple autonomous robots. Each robot can pick up, carry and put down a shelving unit containing products. Shelving units are normally stored in the inner part of the warehouse. Robots are tasked with moving shelving units to stations on the perimeter of the warehouse (where products are boxed and shipped to customers) and with moving shelving units back to storage. In order to increase throughput, the robots need to perform their tasks as quickly as possible [76]. Figure 1.1(b) illustrates autonomous aircraft conducting public safety, commercial and hobbyist operations in low-altitude airspace above a city. The Federal Aviation Administration forecasts that millions of such aircraft will fly in U.S. airspace within a decade. Thus, NASA is designing an automated traffic management system for such aircraft with the main requirements being safety and efficiency [63]. Finally, Figure 1.1(c) shows autonomous forklifts operating in a warehouse. The main difference between the robots in (a) and forklifts is their maneuverability model.

In Artificial Intelligence (AI), the Multi-Agent Path Finding (MAPF) problem is a simplified model of agents navigating in an environment without colliding with obstacles or with each other. In the MAPF



(a) (b) (c)

Figure 1.1: Shows real-world application domains of autonomous agents navigating in different environments. (a) Amazon fulfillment center (Photo credit: The Boston Globe). (b) Autonomous aircrafts conducting operations in low-altitude airspace above a city (Photo credit: NASA). (c) Autonomous forklifts in a warehouse (Photo credit: SME Group).

problem, time is discretized into timesteps and the environment is discretized into cells that individual agents can occupy exclusively at any given timestep. Along with the environment, the MAPF problem also specifies a unique start cell and a unique goal cell for each agent. A solution to the MAPF problem is a set of paths, one for each agent, that take the agents from their respective start cells to their respective goal cells without collisions. A path for an agent is a sequence of move (between adjacent cells) or wait (at a cell) actions, each with some duration in timesteps.

Different measures for the cost of a solution to the MAPF problem, such as sum-of-costs, defined to be the total number of timesteps across all of the paths, or makespan, defined to be the number of timesteps until all agents are at their goal cells, are commonly used in AI. Unfortunately, finding an optimal solution to the MAPF problem according to any of these cost measures is NP-hard, thus explaining why optimal MAPF solvers are inefficient (that is, have a long runtime) in many real-world application domains. On the other end of the spectrum, suboptimal MAPF solvers are efficient but can be ineffective (that is, result in high solution cost) or even incomplete (that is, may fail to find a solution when one exists). Using ineffective MAPF solvers in real-world application domains is problematic because the solution cost is essential for performing the task at hand sufficiently well.

One framework that balances the trade-off between efficiency and effectiveness is that of bounded-suboptimality. A bounded-suboptimal solver takes a user-specified constant $w \geq 1$ and returns a solution

with a cost guaranteed to be at most w times the cost of an optimal solution. On the one hand, the freedom to explore suboptimal solutions allows these solvers to be more efficient than optimal ones. On the other hand, and as opposed to suboptimal solvers, these solvers ensure that their solutions are effective. Unfortunately, state-of-the-art bounded-suboptimal MAPF solvers, all of which rely heavily on a bounded-suboptimal heuristic search method called Focal Search (FS), have the following shortcomings:

1. The runtime of bounded-suboptimal MAPF solvers is significantly affected by small changes in w. Thus, it is often challenging to determine w such that an effective solution is found efficiently.

2. Bounded-suboptimal MAPF solvers are inefficient when agents are huddled together and the environment has some structural components that create bottlenecks. One such example is a typical warehouse domain. Here, the environment has long narrow corridors connecting open spaces or other corridors, and agents have to move between different regions of the warehouse through the corridors.

3. Bounded-suboptimal MAPF solvers are inefficient when move actions have different durations. One such example is actions that model kinodynamically feasible motions.

These shortcomings make bounded-suboptimal MAPF solvers inefficient in many real-world application domains, thus making suboptimal MAPF solvers the only viable option. In this dissertation, we improve FS in ways that resolve the above shortcomings of bounded-suboptimal MAPF solvers. Using these improvements, we develop MAPF solvers that are both efficient[1] and effective, therefore allowing the agents to perform their tasks sufficiently well.

1.1 Contributions

In this dissertation, we make the following improvements to FS:

[1] Since the MAPF problem is NP-hard, it is unlikely that there exists a MAPF solver that can efficiently solve all MAPF problem instances. We use experiments in different environments inspired by real-world application domains to claim that the bounded-suboptimal MAPF solvers we develop are efficient. We base this claim on two arguments. First, we experimentally show that, in these environments, our bounded-suboptimal MAPF solvers are more efficient than other bounded-suboptimal MAPF solvers. Second, we experimentally show that, in these environments, our bounded-suboptimal MAPF solvers are efficient in finding solutions to MAPF problem instances with large numbers of agents, which makes them applicable to real-world application domains.

1. We develop Anytime FS (AFS), an anytime framework for FS which alleviates the need to choose w carefully.

2. We show that FS provides bounded-suboptimality guarantees even when it is used with inflated heuristics.

3. We develop FS-Bulk, a version of FS that reasons about wait durations "in bulk".

Based on the above improvements to FS, we develop the following MAPF solvers:

1. We develop Anytime Bounded Conflict-Based Search (Anytime BCBS), an anytime MAPF solver based on AFS. Anytime BCBS finds a "good" solution quickly and refines it to better and better solutions if time allows. Thus, Anytime BCBS is useful when deliberation time is limited. Moreover, Anytime BCBS provides bounded-suboptimality guarantees for each solution it computes.

2. We develop the highway heuristic, an inflated heuristic that improves the efficiency of MAPF solvers when the domain's environment has structural components that create bottlenecks. The highway heuristic alters the paths that agents choose by biasing the agents away from their (individual) optimal paths and towards paths on (shared) "highways" in the environment. This results in implicit coordination between the agents, which, in turn, increases the efficiency of MAPF solvers. We provide bounded-suboptimality guarantees for our MAPF solvers, such as Enhanced Conflict-Based Search+HWY (ECBS+HWY), based on the bounded-suboptimality guarantees of FS with inflated heuristics.

3. We develop ECBS-Contiguous Time (ECBS-CT), a bounded-suboptimal MAPF solver that uses an efficient implementation of FS-Bulk by using timeintervals instead of timesteps in the state description. ECBS-CT is efficient even when move actions have different durations, thus making it suitable for solving MAPF problems with actions that model kinodynamically feasible motions.

On the theoretical side, we formally prove that our improvements are bounded-suboptimal. On the experimental side, we show that our improvements result in increased efficiency in domains inspired by

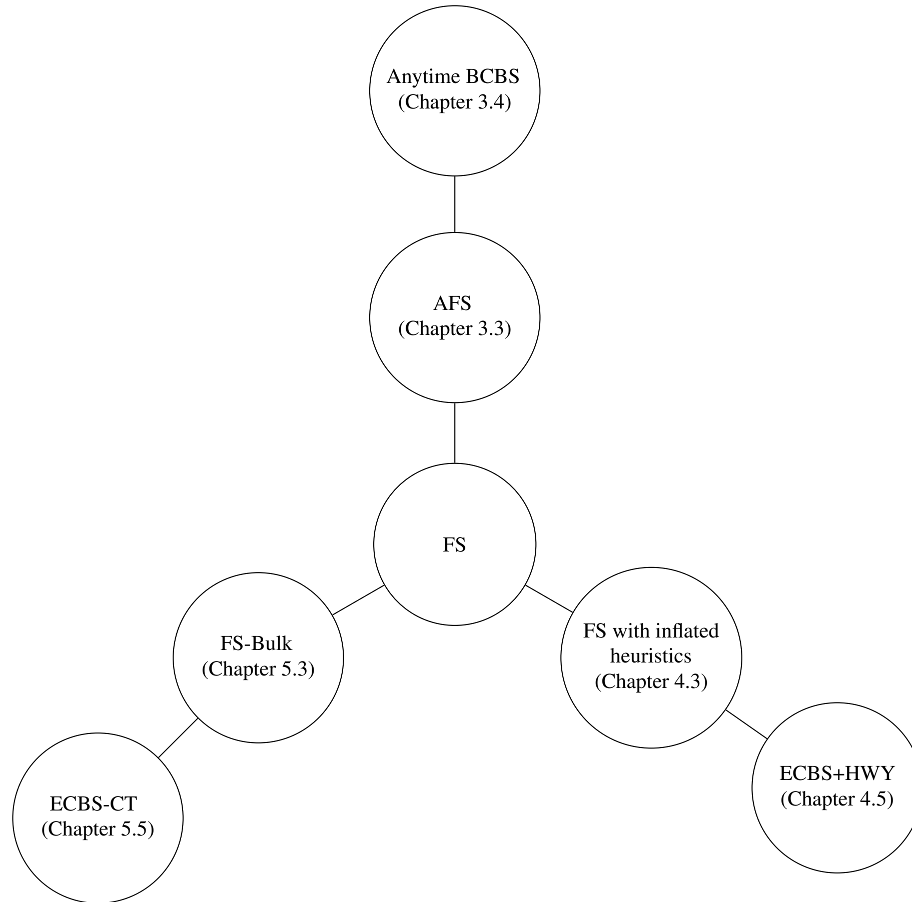


Figure 1.2: Summarizes the contributions of this dissertation.

real-world applications. Figure 1.2 summarizes the contributions in this dissertation. Finally, this dissertation makes some additional contributions, such as developing approaches to automatically generate highways and generalizing the MAPF problem to the Multi-Agent Motion Planning (MAMP) problem.

1.2 Outline

This dissertation is organized as follows. A formal definition of the MAPF problem and an overview of different approaches for solving the MAPF problem, including Conflict-Based Search, which we use extensively, is provided in chapter 2. In chapter 3, we provide an in-depth discussion of FS in the context of bounded-suboptimal and bounded-cost heuristic search, develop AFS and Anytime BCBS. In chapter

MAPF	Multi-agent path finding
MAMP	Multi-agent motion planning
BSS	Bounded-suboptimal search
BCS	Bounded-cost search
FS	Focal search
AFS	Anytime focal search
FS-B	Focal search bulk
OPEN	Open list
FOCAL	Focal list
PS	Potential search
ATPS	Anytime potential search
ANA*	Anytime non-parametric A*
wA*	Weighted A*
ARA*	Anytime repairing A*
EES	Explicit estimation search
GCTSP	Generalized cost traveling salesman problem
CBS	Conflict-based search
BCBS	Bounded conflict-based search
ECBS	Enhanced conflict-based search
ECBS-CT	Enhanced conflict-based search-contiguous timesteps
SIPP	Safe interval path planning
SCIPP	Soft collision interval path planning
EG	Experience graph
RRT	Rapidly exploring random tree
PRM	Probabilistic roadmap

Table 1.1: Acronyms

4, we show that FS with an inflated heuristic is bounded-suboptimal, develop the highway heuristic along with bounded-suboptimal MAPF solvers that use it and develop approaches to generate highways automatically. In chapter 5, we develop FS-Bulk and an efficient implementation of it, called SCIPP, that uses timeintervals instead of timesteps in the state description. We also define the MAMP problem, a generalization of the MAPF problem, and develop ECBS-CT, a bounded-suboptimal MAMP solver that uses SCIPP. In chapter 6, we summarize the contributions made in this dissertation and discuss directions for future work.

Chapter 2

Background

In this chapter, we provide a formal definition of the MAPF problem and survey optimal, suboptimal and bounded-suboptimal MAPF solvers.

2.1 The Multi-Agent Path Finding Problem

The problem of multiple agents navigating in a complex environment occurs in many different settings. In some settings, agents may be *indistinguishable*[1] while in other settings they are *distinguishable*[2]. Indistinguishable agents choose a (unique) goal cell from a given set of goal cells [93]. Distinguishable agents, however, have assigned goal cells that cannot be changed. Recently, a generalization to teams of agents (where agents in the same team are indistinguishable) was proposed in [49]. When all agents are on the same team, this generalization is equivalent to the indistinguishable agents setting. When all teams are of size one, this generalization is equivalent to the distinguishable agents setting.

In this dissertation, we focus on distinguishable agents for two reasons. First, agents in some of our application domains are inherently distinguishable. For example, two Kiva-like robots carrying different shelving units to different stations cannot simply switch their goal cells. Second, it turns out that the distinguishable case is computationally harder [49], thus novel ideas for improvements can have a more

[1] also called *unlabeled*
[2] also called *labeled*

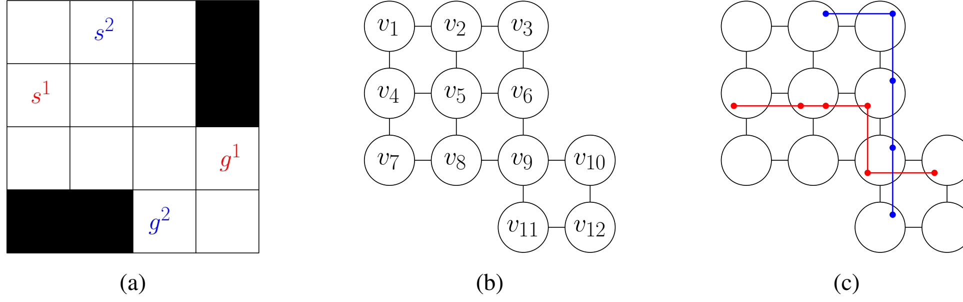


Figure 2.1: Shows a MAPF problem instance with two agents, red and blue. (a) A 4-neighbor grid environment with two agents. The start and goal cells of the agents, s^1, s^2, g^1 and g^2, are specified. (b) The graph representation of the given environment with arbitrary vertex labels. For these labels, the start and goal vertices are: $s^1 = v_4$, $g^1 = v_{10}$, $s^2 = v_2$ and $g^2 = v_{11}$. (c) Feasible and collision-free paths for the agents (depicted with their respective colors). Two discs in the same vertex represent a wait action.

significant impact on the applicability of state-of-the-art MAPF solvers. In the next subsection, we formally define the MAPF problem for distinguishable agents, henceforth referred to as the MAPF problem.

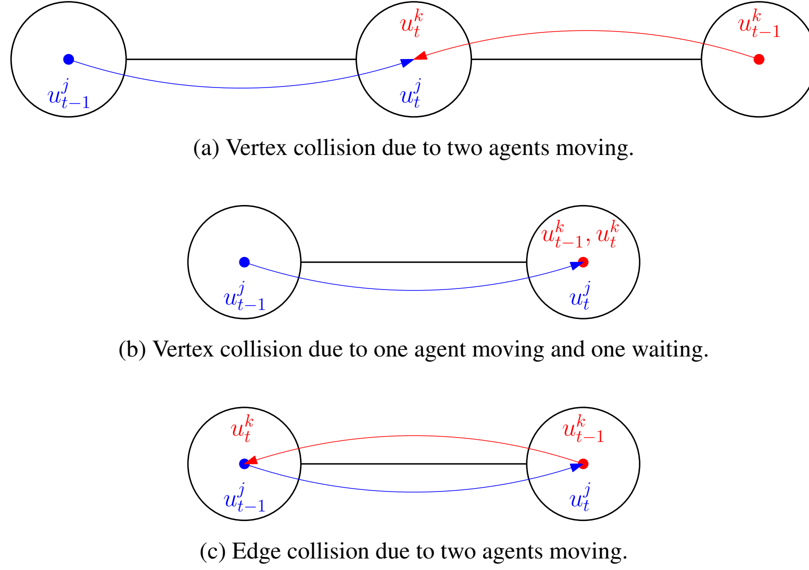
Definition 1 (*The MAPF problem*)**.** *The shared environment is represented by a graph $G = (V, E)$. In this graph, vertices represent cells that agents can occupy exclusively at any given timestep, and edges represent possible transitions between cells. There are K agents, labeled $1, \ldots, K$. Each agent j has a unique start cell $s^j \in V$ and a unique goal cell $g^j \in V$. In each timestep, every agent chooses to either: 1) move to an adjacent cell; or 2) wait at its current cell. In general, each wait or move action has a non-negative cost associated with it, $c : V \times V \to \mathbb{R}^+$. Unless explicitly stated otherwise, we assume that both wait and move actions have a unit cost.*

Figure 2.1(a) shows an instance of the MAPF problem.

Definition 2 (*A feasible path*)**.** *A path $u^j = \{u^j_0, \ldots, u^j_{T_j}, u^j_{T_j+1}, \ldots\}$ for agent j is feasible if and only if:*

1. *It starts at agent j's start cell, that is, $u^j_0 = s^j$.*

2. *It ends at agent j's goal cell where the agent remains indefinitely, that is, there exists an earliest T_j such that $u^j_{T_j} = g^j$ and, for each $t > T_j$, $u^j_t = g^j$. We call T_j the arrival time of agent j at its goal cell.*



(a) Vertex collision due to two agents moving.

(b) Vertex collision due to one agent moving and one waiting.

(c) Edge collision due to two agents moving.

Figure 2.2: Illustrates vertex and edge collisions.

3. *Every action is a legal move or wait action, that is, for all* $t \in \{0, 1, \ldots, T_j - 1\}$, $(u^j_t, u^j_{t+1}) \in E$ *or* $u^j_t = u^j_{t+1}$.

Definition 3 (*A collisions*)**.** *A collision between the paths of agents* j *and* k *can be one of the following:*

- *A* vertex collision *in cell* $v \in V$ *at timestep* t*, that is,* $v = u^j_t = u^k_t$*. We denote a vertex collision by* (j, k, v, t).
- *An* edge collision *in transition* $(v_1, v_2) \in E$ *at timestep* t*, that is,* $v_1 = u^j_t = u^k_{t+1}$ *and* $v_2 = u^j_{t+1} = u^k_t$*. We denote an edge collision by* (j, k, v_1, v_2, t).

Definition 4 (*A solution to a MAPF problem instance*)**.** *A solution to a MAPF problem instance is a set of feasible paths, one for each agent, such that no two paths are in collision.*

Given a set of paths, we can check if they are feasible and collision-free in polynomial time. For example, Figure 2.1(c) illustrates the paths of the red and blue agents. The red path for agent 1 is $u^1 = \{u^1_0 = v_4, u^1_1 = v_5, u^1_2 = v_5, u^1_3 = v_6, u^1_4 = v_9, u^1_5 = v_{10} \ldots\}$ and $T_1 = 5$. The blue path for agent 2 is $u^2 = \{u^2_0 = v_2, u^2_1 = v_3, u^2_2 = v_6, u^2_3 = v_9, u^2_4 = v_{11} \ldots\}$ and $T_2 = 4$. These paths are feasible and collision free. Hence, they are a solution to the MAPF problem instance given in Figure 2.1(a).

In some MAPF settings, agents may be *self-interested*, and the task is to devise a mechanism that will cause them to cooperate [6]. In the MAPF settings targeted in this dissertation, we are in full control of the agents. Thus, we assume that the agents are *fully collaborative*. When agents are fully collaborative, the solution cost is typically measured by some global cumulative cost function.

Definition 5 (*The cost of a solution*)**.** *Common functions used in the AI community to compute a global cumulative cost of K paths are:*

1. Total arrival time *is the summation of the individual arrival times, that is,* $\sum_{j=1}^{K} T_j$.

2. Makespan *is the maximal individual arrival time, that is,* $\max_{j=1}^{K} T_j$.

3. Sum-of-costs *is the summation of the individual path costs, that is,* $\sum_{j=1}^{K} \sum_{i=1}^{T_j-1} c(u_i^j, u_{i+1}^j)$.[3]

4. Total fuel cost *is the summation of the individual move action costs, that is,* $\sum_{j=1}^{K} \sum_{i=1}^{T_j-1} \mathbb{1}_{[u_i^j \neq u_{i+1}^j]} c(u_i^j, u_{i+1}^j)$.[4]

For example, the solution in Figure 2.1(c) has a total arrival time of 9 and a makespan of 5. Assuming that every action has unit cost, the paths in Figure 2.1(c) have a sum-of-costs of 9 and total fuel cost of 8. Unless explicitly stated otherwise, we use the total arrival time throughout this dissertation.

Unfortunately, finding a solution with minimum total arrival time, minimum makespan or minimum sum-of-costs is NP-hard[5]. Moreover, optimality cannot always be simultaneously achieved for minimum makespan and minimum total arrival time [95]. Other measures for the solution cost have been proposed in the AI community. For example, in [40], a vector of independent cost measures (one entry per agent) is used instead of a single scalar. Such a cost function is part of the broader field of multi-objective optimization and is out of the scope of this dissertation.

[3]Total arrival time is equivalent to sum-of-costs when all actions have unit costs.
[4]Also called total distance. Total fuel is equivalent to sum-of-costs when wait actions cost zero.
[5]While we believe that minimizing the total fuel cost is NP-hard as well, it has not been formally proved.

MAPF solvers generally fall into one of two categories: *distributed* MAPF solvers and *centralized* MAPF solvers. A distributed MAPF solver assumes that each agent has its own computing and communication capabilities. A centralized MAPF solver assumes a single computing platform with full knowledge[6] as well as reliable communication to and from all agents. Since digital communication is becoming increasingly reliable, we assume to have this capability at our disposal. Reliable communication, along with fully collaborating agents, naturally lends itself to centralized approaches. In the next section, we survey relevant centralized MAPF solvers.

2.2 Multi-Agent Path-Finding Solvers

In this section, we survey optimal, suboptimal and bounded-suboptimal MAPF solvers. We focus on centralized approaches for the fully collaborative setting.

2.2.1 Optimal MAPF Solvers

Optimal MAPF solvers normally operate in the *joint state space*. In the joint state space, the entire set of agents is treated as a single entity. For a graph with $|V|$ vertices and K agents, a *state* in the joint state space represents one of $|V|^K$ possible placements of agents at vertices. Similarly, a *valid state* represents one of $\binom{|V|}{K}$ possible placements of agents at vertices such that each vertex hosts at most one agent. Given a MAPF problem instance, the *start state* is the valid state for which each agent j is at its start vertex s^j, and the *goal state* is the valid state for which each agent j is at its goal vertex g^j.

We define the *neighbors* of a state to be the set of states that can be generated from it by any combination of K actions, one action per agent. Similarly, we define the *valid neighbors* of a valid state to be the set of valid states that can be generated from it by any combination of K actions, one action per agent, such that there is no collision (that is, neither a vertex collision nor an edge collision) between any two agents. For each state, its number of valid neighbors is bounded by its number of neighbors. For

[6] In general, "full knowledge" means the combined knowledge of all agents, such as their current cells, goal cells, the observed portion of the environment and planned paths. In this dissertation, we assume that the environment is fully known to the centralized MAPF solver.

example, consider K agents navigating on a 4-neighbor grid environment. The number of actions for each agent is 5 because the possible actions are move-up, move-down, move-right, move-left and wait. Thus, there are 5^K neighbors. The number of actions that do not result in a collision is at most 5 per agent. Naively, to compute the valid neighbors, we need to consider all neighbors and discard the ones resulting in a collision. It is also possible to frame the problem of computing the valid neighbors of a given state as a constraint satisfaction problem (CSP) [73]. In this CSP, each agent has a variable with domain values that correspond to its actions, and constraints between variables ensure that possible actions do not generate collisions. However, computing the valid neighbors of a given state may still require to examine a number of states that is exponentially increasing in K.

MAPF solvers that operate in the joint state space are called *coupled MAPF solvers*. For example, one can apply A* [25] to find paths for all agents simultaneously by searching the joint state space. Here, every state s has g, h and f-values. The g-value of s (also referred to as $g(s)$) is the cost of a path from the start state to s. The h-value of s (also referred to as $h(s)$) is the heuristic estimate of the cost of a path from s to a goal state. Finally, the f-value of s (also referred to as $f(s)$) equals $g(s) + h(s)$ and represents the estimated cost of the path from the start state to a goal state through s. While *expanding* s, we *generate* its children, which corresponds to generating a state for each valid neighbor of s.

As is always the case with A*, its runtime is heavily dependent on the heuristic being used. One admissible heuristic[7] is the sum of the individual Manhattan distance of each agent from its current cell to its goal cell. A more informed heuristic that is also admissible is the sum of the individual optimal path cost of each agent from its current cell to its goal cell while abstracting away all other agents [71]. This heuristic can be viewed as a perfect estimate if one ignores any potential interactions among agents. Moreover, the difference between this heuristic and the optimal solution cost is due only to the interaction between agents.

Unfortunately, using A* in the joint state space is infeasible for more than a few agents, even when using the above heuristics. One problem is that the number of valid neighbors of a state s may be exponential

[7]Heuristic h is said to be *admissible* if, for any state s, $h(s)$ does not overestimate the distance from s to a goal state.

in K. Moreover, many of the valid neighbors of s are states in which some agents move away from their goal cells, which means that their f-values are higher than $f(s)$. This makes many generated states less likely to be expanded, and thus does not contribute to advancing the search towards a solution. *Operator decomposition* [73] (OD) alleviates this problem by augmenting the joint state space with intermediate states as follows. Before A* begins, a fixed (arbitrary) order of the agents is chosen. When A* expands a state from the joint state space, it considers only actions of the first agent for generating successor intermediate states. When A* expands an intermediate state that was generated with an action of agent $i < K$, it considers only actions of the $(i+1)^{\text{th}}$ agent for generating successor intermediate states. Finally, when A* expands an intermediate state in which it considers only actions of the Kth agent, successor states from the joint state space are generated. OD reduces the branching factor to the number of actions per agent but increases the depth of a solution by a factor of K. Another approach for reducing the number of successors is that of Enhanced Partial Expansion A* [22] (EPEA*). EPEA* uses domain-specific knowledge to avoid generating *surplus states*, which include any state with an f-value larger than the cost of an optimal solution.

Another problem with applying A* in the joint state space is that its size grows exponentially with the number of agents. Thus, reducing the number of agents that A* needs to handle can result in an exponential speedup. *Independence detection* (ID) [73] leverages this insight by trying to decouple the MAPF problem by partitioning agents into smaller groups such that the optimal paths found for each group independently do not collide with the paths of other groups. It greedily decides on the partitions as follows. Initially, each agent is a group by itself. Then, ID uses A* to find paths for each group independently. If a collision between paths of two groups is found, then they are merged, and A* is used to find paths for the resulting group. This process continues iteratively until there are no more collisions between groups. ID effectively reduces the size of the state space from exponential in K to exponential in the size of the largest group.

ID is a simple example of a *dynamically coupled MAPF solver*. Whereas coupled MAPF solvers treat the agents as a single meta-agent and explore the joint state space, dynamically coupled MAPF solvers initially operate in a much smaller state space and grow it dynamically, if necessary, up to the full joint

state space. For many MAPF problem instances, a smaller state space suffices to find an optimal solution. Two state-of-the-art dynamically coupled optimal MAPF solvers are M* [86] and conflict-based search (CBS) [69]. M* is an A*-based algorithm that uses subdimensional expansion to initially create a one-dimensional search space embedded in the joint state space. When the search encounters a collision, the dimensionality of the search space is locally increased to ensure that an alternative path can be found. We provide more details on M* in Chapter 4.5.3. Like M*, CBS also tries to avoid operating in the joint state space. However, it does so using a two-level search. On the high level, a search is performed on a conflict tree that represents constraints on agents. Each high-level state represents a set of constraints imposed on the actions of individual agents. On the low level, single-agent searches are performed in conformance with the constraints imposed by the relevant high-level states. Since we extensively use and extend CBS throughout this dissertation, we discuss it in depth in Chapter 2.2.1.1.

Finally, optimal *reduction-based MAPF solvers* cast the MAPF problem as different combinatorial problems such as Boolean satisfiability (SAT) [77, 78, 79], integer linear programming (ILP) [94] and answer set programming (ASP) [17]. These combinatorial problems are known to be NP-hard, but can leverage efficient off-the-shelf solvers to find optimal solutions. The SAT encoding is based on a time-expanded graph (TEG). The TEG duplicates each cell in G for a given number of timesteps T (which normally represents the makespan). Edges in the TEG represent possible actions (of a single agent) between consecutive timesteps. The SAT encoding creates a Boolean variable for each combination of agent and vertex in the TEG. Essentially, if a variable (v, t) in the TEG of agent a is assigned a True value, it is interpreted as agent a occupying cell v at timestep t. Different constraints on the Boolean variables ensure that a satisfying assignment represents a solution to the MAPF problem. To guarantee optimality, a SAT solver is trying to find a satisfying assignment for an increasingly larger T. Rather than casting the MAPF problem as a Boolean satisfiability problem, the ILP-based MAPF solver casts the MAPF problem as an integer maximum multi-commodity flow problem. Here, we view different agents as different types of commodities flowing through a network. Similar to the usage of TEGs for the SAT encoding, a time-expanded flow network (TEN) is used to encode the MAPF problem as a flow problem. For a given

number of timesteps T, the TEN creates $2T + 1$ copies, $v_0, \ldots, v_{2T}$, for each vertex v in G. For each edge in G and each possible timestep, a gadget ensuring that no two agents travel in opposite directions on that edge and timestep is added to the TEN. Additionally, for every vertex v in G, the TEN has an edge (v_i, v_{i+1}) for every timestep $i < 2T$ to "allow" for wait actions. Unit capacity constraints on the edges of the TEN ensure that the maximum flow can be translated to a solution of the MAPF problem.

2.2.1.1 Conflict-Based Search

In this subsection, we formally describe conflict-based search (CBS) [69] because we extend it in different ways throughout this dissertation. CBS is an optimal MAPF solver that performs high-level and low-level searches. Each high-level state contains a set of constraints and, for each agent, a feasible path that respects the constraints. The high-level root state has no constraints. The high-level search of CBS is a best-first search that uses the costs of the high-level states as their f-values. In general, the cost of a high-level state is defined by an objective function similar to Definition 5. In this dissertation, we use the total arrival time (Definition 5(1)) unless a different objective function is explicitly specified. When CBS expands a high-level state N, it checks whether the state is a goal state. A high-level state is a goal state if and only if none of its paths collide. If N is a goal state, then CBS terminates successfully and outputs the paths in N as solution. Otherwise, at least two paths collide. CBS chooses a collision to resolve and generates two high-level child states of N, called N_1 and N_2. Both N_1 and N_2 inherit the constraints of N. If the chosen collision is a vertex collision (j, k, s, t), then CBS adds the vertex constraint (j, s, t) to N_1 (that prohibits agent j from occupying cell s at timestep t) and the vertex constraint (k, s, t) to N_2. If the chosen collision is an edge collision (j, k, s_1, s_2, t), then CBS adds the edge constraint (j, s_1, s_2, t) to N_1 (that prohibits agent j from moving from cell s_1 to cell s_2 between timesteps t and $t+1$) and the edge constraint (k, s_2, s_1, t) to N_2. During the generation of a high-level state N, CBS performs a low-level search for the agent i affected by the added constraint. The low-level search for agent i is a (best-first) A* search that ignores all other agents and finds an optimal path from the start cell of agent i to its goal cell that is both feasible and respects the constraints of N that involve agent i.

Figure 2.3 illustrates a part of the high-level tree of CBS for a 4×4 grid environment with two agents. For brevity, we use [row,column] to specify a cell in grid environments, where [0,0] is the top-left cell. Initially, the high-level root state contains no constraints, and CBS computes an optimal path for each agent (depicted by the red and blue lines). Since these paths collide, the high-level root state is not a goal state. CBS chooses a collision (specifically, cell [1,2] at timestep 2, depicted by a green cell), and generates two high-level child states with additional constraints that resolve this collision. In each high-level child state, the newly constrained agent replans an optimal path that does not violate the constraints specified in that high-level state. This process continues until, eventually, CBS expands a high-level goal state. Note that this example is a pathological case where CBS is extremely inefficient. This is so because there are four optimal paths for each agent, and each of the sixteen combinations has a collision in one of the four inner cells. Thus, CBS expands 16 high-level states before it returns a solution with total arrival time of 9.

2.2.2 Suboptimal MAPF Solvers

Coupled MAPF solvers, as described above, explore the joint state space. This allows them to guarantee completeness and optimality. However, the joint state space gets very large even for a small number of agents. Thus, the runtimes of coupled MAPF solvers are often prohibitively long. Unlike optimal MAPF solvers, suboptimal MAPF solvers can quickly find solutions for MAPF problem instances with many agents. Suboptimal MAPF solvers explore a space that represents only a small portion of the joint state space, and thus they are not necessarily complete and can be very ineffective.

Reactive MAPF solvers are suboptimal and rely on collision-avoidance to plan paths. Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation (ORCA) [5] is such a solver that plans a path for each agent independently while taking into account the other agents' locations, radiuses and velocities. One problem with ORCA is that it can be very inefficient in crowded environments. C-Nav [21] adds implicit coordination between agents, which improves the solver's runtime in crowded environments. Other reactive MAPF solvers are inspired by biological behaviors such as flocking and herding [64]. A fundamental problem with such solvers is that they are prone to livelocks.

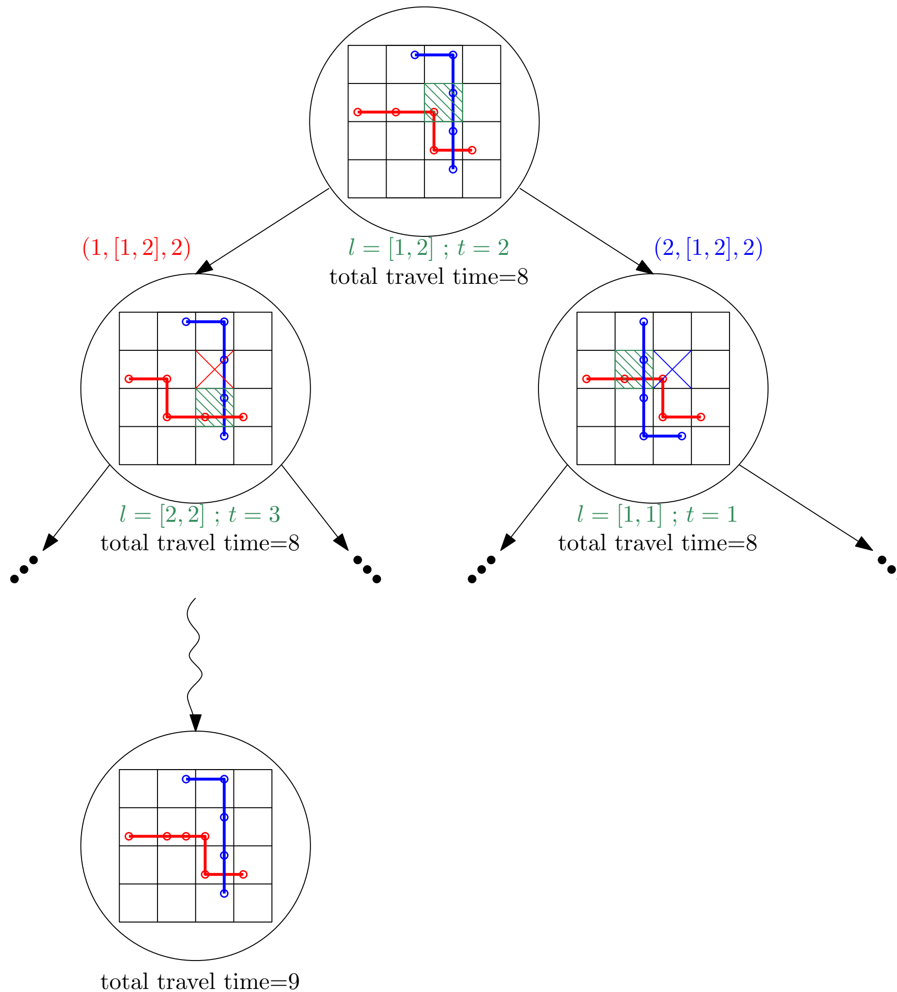


Figure 2.3: Illustrates the high-level search tree of CBS for a MAPF problem instance with two agents, red and blue, in a 4×4 4-neighbor grid environment. Actions are assumed to have unit cost. The start and goal cells of the red agent are $[1, 0]$ and $[2, 3]$, respectively. The start and goal cells of the blue agent are $[0, 1]$ and $[3, 2]$, respectively. Each dot represents a timestep and, therefore, two dots in the same cell represent a wait action. The cost of a high-level state is the total arrival time defined by its paths. The collision that CBS chooses to resolve in each high-level state is specified below it, and the constraints specified in a high-level state are specified above it.

Rule-based MAPF solvers, such as Push and swap [47] and push and rotate [89], are suboptimal. They plan for agents according to specific rules. These planners are fast but could result in very high solution costs since they allow only one agent to move at any timestep.

Hierarchical Cooperative A* (HCA*) [71] is a search-based suboptimal MAPF solver that plans for agents in some given priority. Here, agents plan one after the other, and each agent needs to accommodate

the paths of the agents that planned before it. Other MAPF solvers proposed to use flow restrictions in road networks, such that movement along cells in the grid is restricted to only one direction [88]. Additional rules are provided to ensure that no feasible path becomes infeasible. Another approach uses direction maps, which represent joint information about how agents have been moving on the grid, and leads to implicit cooperation during movement [32].

2.2.3 Bounded-Suboptimal MAPF Solvers

Optimally Solving the MAPF problem is NP-hard [95]. Optimal MAPF solvers, such as CBS [69] and M^* [86], therefore can be used only for low numbers of agents. Yet, finding low-cost solutions is important because the same throughput can then be achieved with fewer agents. It is therefore common to use bounded-suboptimal MAPF solvers for higher numbers of agents. In general, search algorithms are called *w-suboptimal* for a user-provided suboptimality bound w if and only if they return solutions of cost at most wOPT, where OPT is the cost of an optimal solution. Their solution costs typically increase with w while their runtimes typically decrease with w although this is not guaranteed [90]. w-suboptimal search algorithms are also called bounded-suboptimal search algorithms.

Two bounded-suboptimal MAPF solvers have been proposed in the literature, namely 1) a family of MAPF solvers that extend CBS, such as Weighted CBS and Enhanced CBS (ECBS) [4]; and 2) a family of MAPF solvers that extend M*, such as inflated-M*, inflated-rM*, inflated-ODrM*, inflated-EPEM* and EPErM* [86]. ECBS is faster than all other bounded-suboptimal MAPF algorithms compared in [4]. We discuss ECBS in detail in Chapter 4.2.2.

2.2.4 Others

Different variants of MAPF in the fully collaborative setting include agents that do not have an assigned goal cell, but instead choose one from a set of goal cells. In [93], agents are indistinguishable and each agent chooses a (unique) goal cell from a given set. This variant can be solved in polynomial time via a reduction to a maximum flow problem. More recently, a generalization to teams of agents (where agents

in the same team are indistinguishable) was proposed [49]. When all agents are in the same team, this generalization is equivalent to the indistinguishable agents case, and when all teams are of size one, this generalization is equivalent to our definition of the MAPF problem. We refer the reader to [50] for more details and a survey of generalizations of the MAPF problem to other real-world scenarios.

Chapter 3

Anytime MAPF

In this chapter, we develop an anytime version of focal search (FS). Like A*, FS uses an open list whose states are sorted in increasing order of their f-values. Unlike A*, FS also uses a focal list containing all states from the open list whose f-values are no larger than a suboptimality factor times the smallest f-value in the open list. Our anytime version of FS, called anytime FS (AFS), is useful when deliberation time is limited. AFS finds a "good" solution quickly and refines it to better and better solutions if time allows. It does this refinement efficiently by reusing previous search effort. On the theoretical side, we show that AFS is bounded suboptimal and that anytime potential search (ATPS/ANA*), a state-of-the-art anytime bounded-cost search (BCS) variant of A*, is a special case of AFS. In doing so, we bridge the gap between anytime search algorithms based on bounded-suboptimal search (BSS) and BCS. We also identify different properties of priority functions, used to sort the focal list, that may allow for efficient reuse of previous search effort. On the experimental side, we demonstrate the usefulness of AFS for solving hard combinatorial problems such as the MAPF problem.

This chapter is organized as follows: In Section 3.1, we discuss the limitations of anytime search-based algorithms and outline our contributions. In Section 3.2, we discuss the BSS and BCS frameworks and formally define FS. In Section 3.3, we discuss AFS along with different properties of priority functions and different approaches for setting bounds. In Section 3.4, we describe the usage of AFS in the CBS framework. In Section 3.5, we provide experimental results.

3.1 Introduction

A* [25] is a best-first search algorithm that continuously expands a state with minimal key from the open list (henceforth referred to as OPEN), where the key of state s is given by $f(s) = g(s) + h(s)$. Here, $g(s)$ is the distance of s from the start state computed and maintained by A*, and $h(s)$ is the state's cost-to-goal estimate (heuristic value). Despite the many successes of A*, it is known to be unviable for large combinatorial problems when the heuristic guidance is not perfect [27]. This has prompted the development of several variants of A* that have the freedom to produce suboptimal solutions since this freedom often increases efficiency [91]. Nevertheless, in many real-world domains, such as in robotics and probabilistic reasoning, the solution cost cannot be compromised beyond a reasonable factor.

Hence, subsequent work has focused on bounded-suboptimal search (BSS), that tries to trade-off efficiency with effectiveness. BSS algorithms produce solutions with costs at most w times the optimal cost, for some user-specified suboptimality bound $w \geq 1$. One such algorithm is weighted-A* (wA*) [61]. wA* differs from A* only in the keys it uses: It puts more weight on the heuristic value by inflating it with an inflation factor w, that is, $f(s) = g(s) + wh(s)$. wA* generates solutions faster than A* in many domains [7, 38]. However, increasing the weight of the heuristic value may also decrease efficiency, especially when the correlation between the heuristic values and the minimal number of edges-to-goal is weak [90].

Inflating heuristic values also allows for the development of anytime search algorithms [82]. Anytime algorithms are useful when deliberation time is limited. They are intended to generate an initial solution quickly and use any additional available time to generate better and better solutions. ARA* [45] is an anytime heuristic search algorithm that repeatedly runs wA* with decreasing values of w. ARA* reuses search effort from previous search iterations and is considered efficient since it expands each state at most once per search iteration. This efficiency property relies on bounded admissibility[1] [2].

Since ARA* is based on wA*, it is subject to a restriction: Like A*, it expands states greedily in order of increasing f-values from OPEN. Therefore, its intended trade-off between efficiency and effectiveness

[1] A state s is said to be *bounded admissible* iff $g(s) \leq wg^*(s)$ when it is selected for expansion, where $g^*(s)$ is the distance from the start state to s.

stems from the inflation of heuristic values rather than the freedom of expanding states with suboptimal f-values. Unlike A*, wA* or ARA*, focal search (FS) [56] leverages this freedom to expand states with suboptimal f-values. FS guarantees bounded suboptimality by using f-values in conjunction with arbitrary priorities to order state expansions. While the f-values determine a set of possible states (denoted `FOCAL`) that qualify for expansion, the arbitrary priorities are used to choose a particular state for expansion from `FOCAL`. FS has been successfully used to solve many combinatorial problems efficiently [4, 26].

In this chapter, we therefore develop an anytime version of FS, called anytime FS (AFS). Because the source of suboptimality in FS comes from the *flexibility* of expanding states with suboptimal f-values rather than the *inflation* of h-values, AFS works by iteratively tightening the flexibility rather than adopting ARA*'s strategy of iteratively decreasing the inflation factor. Like ARA*, AFS also reuses search effort from previous search iterations while guaranteeing the suboptimality bound. In addition, the mechanism that AFS uses to update `FOCAL` between consecutive search iterations is easy to implement and analyze.

For pedagogical reasons, we also relate our work to the bounded-cost search (BCS) framework and its anytime adaptations. In BCS, a cost bound C is given and the task is to find a solution with cost at most C as fast as possible. Two state-of-the-art anytime BCS algorithms, anytime potential search (ATPS) [75] and anytime non-parametric A* (ANA*) [84], have been shown to be equivalent [74]. Both ATPS and ANA* can be thought of as AFS that uses a specific mechanism for iteratively tightening `FOCAL` and a specific priority function, called the potential function, to sort it.

On the theoretical side, we show the bounded suboptimality of AFS, identify different ways to define `FOCAL` along with properties of priority functions used to sort it, and thus bridge the gap between anytime BSS and anytime BCS. On the experimental side, we demonstrate the usefulness of AFS for solving hard combinatorial problems, such as the MAPF problem and the generalized covering traveling salesman problem.

3.2 Background

In this section, we define the BSS and BCS frameworks, characterize FS under both frameworks and develop a unified view.

3.2.1 Bounded-Suboptimal Search and Bounded-Cost Search

Two prominent suboptimal search frameworks, BSS and BCS, are defined as follows.

Definition 6 (BSS). *Given a user-specified suboptimality bound $w \geq 1$, a BSS algorithm is guaranteed to find a solution of cost at most wP_{opt}, where P_{opt} is the cost of an optimal solution.*

Definition 7 (BCS). *Given a user-specified cost bound $C \geq 0$, a BCS algorithm is guaranteed to find a solution of cost at most C.*

3.2.2 Focal Search

Consider A*'s `OPEN` sorted in increasing order of $f(s) = g(s) + h(s)$ with a consistent h. We also define $f_{\min} = \min_{s \in \texttt{OPEN}} f(s)$ and $\texttt{head}(\texttt{OPEN}) = \arg\min_{s \in \texttt{OPEN}} f(s)$.

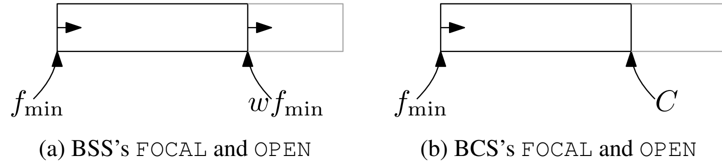
Definition 8 (Focal list (`FOCAL`)). *There are two ways to define `FOCAL` $\subseteq$ `OPEN`:*

1. *`FOCAL` $= \{s \in$ `OPEN` $: f(s) \leq wf_{\min}\}$ for a user-specified suboptimality bound $w \geq 1$.*

2. *`FOCAL` $= \{s \in$ `OPEN` $: f(s) \leq C\}$ for a user-specified cost bound $C \geq 0$.*

FS in the BSS framework is based on the following observation: While A* with admissible heuristic values might spend a long time identifying the best solution among many "good" solutions by expanding only states whose f-values equal $f_{\min}$, FS has the freedom to choose any "good enough" solution by expanding any state from `FOCAL` given in Definition 8(1). This flexibility allows FS to terminate earlier than A* while providing bounded suboptimality guarantees.

`FOCAL` is also useful in the BCS framework. Here, the largest f-value in `FOCAL` does not depend on $f_{\min}$. Instead, we are given a cost bound and the task is to find a solution as fast as possible whose



(a) BSS's FOCAL and OPEN (b) BCS's FOCAL and OPEN

Figure 3.1: Illustrates FOCAL (black) and OPEN (black+grey) for BSS and BCS in (a) and (b), respectively.

cost is no greater than this cost bound. Once again, we are free to expand any state from FOCAL given in Definition 8(2)[2] and are not constrained to states with minimum f-values only.

Figure 3.1 illustrates FOCAL in the BSS and BCS frameworks. In both frameworks, $f_{\min}$ represents the smallest f-value of all states in OPEN and, therefore, also in FOCAL. The difference between the two focal lists is in the largest f-value of all states in them. In (a), the largest such f-value increases when $f_{\min}$ increases (depicted by the right arrow), while, in (b), it remains fixed throughout the search.

Algorithm 1 presents pseudocode for the unified view of BSS and BCS (blue for BSS and red for BCS). Procedure *findPath* (line 4) implements FS, which starts with a singleton OPEN and FOCAL containing the start state s_{start}. The main loop of FS is conditioned on a non-empty FOCAL. Inside this loop, we first pop the head of FOCAL and remove it from OPEN as well (lines 6-10). If the popped state is a goal state, we return the solution found and terminate (lines 11-12). Otherwise, we generate its successors and add them to OPEN and possibly FOCAL (lines 13-29) (only if their g-values improve, that is, $g(s') > g(s) + c(s, s')$, where $c(s, s')$ is the transition cost from state s to its successor state s'). In case the f-value of the head of OPEN increases as a consequence of the above operations, we need to update FOCAL accordingly (lines 30-31). Finally, if FOCAL is empty, we report that no solution exists (line 32).

3.3 Anytime Focal Search

In this section, we present AFS. Like ARA* and ATPS/ANA*, AFS finds an optimal solution given enough time, provides suboptimality guarantees for each search iteration, and reuses previous search effort. However, AFS can compute tighter suboptimality bounds than ATPS/ANA* and, unlike ARA* and

[2]It is common to not maintain OPEN explicitly in the BCS framework because $f_{\min}$ does not play a role in its FOCAL. This has implications on its suboptimality bound, as we discuss in the context of Lemma 13.

Algorithm 1: Focal Search (FS).
s_{start} is the start state; $\texttt{isGoal}(s)$ is a predicate that returns true iff s is a goal state; $\texttt{succ}(s)$ returns a list of all successors of s; and w (or C) is the suboptimality bound (or cost bound). Blue (red) represents pseudocode relevant for BSS (BCS) only.

```
   Input: s_start, isGoal(s), succ(s), w (or C).
   Output: A solution.
1  OPEN = FOCAL = {s_start}
2  CLOSED = {}
3  return findPath(w (or C))
4  Procedure findPath(w (or C)):
5      while FOCAL ≠ ∅ do
6          f_min ← f(head(OPEN))
7          s ← head(FOCAL)
8          FOCAL ← FOCAL \ {s}
9          OPEN ← OPEN \ {s}
10         CLOSED ← CLOSED ∪ {s}
11         if isGoal(s) then
12             return a solution (that is, the path from s_start to s)
13         for each s' ∈ succ(s) do
14             if s' ∈ CLOSED or s' ∈ OPEN (or s' ∈ FOCAL) then
15                 if g(s') ≤ g(s) + c(s, s') then
16                     Continue to the next successor
17             g(s') ← g(s) + c(s, s')
18             if s' ∉ OPEN (or s' ∉ FOCAL) then
19                 OPEN ← OPEN ∪ {s'}
20                 if f(s') ≤ w f_min (or C) then
21                     FOCAL ← FOCAL ∪ {s'}
22             else
23                 Update the priority of s' in FOCAL
24                 Update the priority of s' in OPEN
25                 if f(s') ≤ w f_min then
26                     if s' ∈ FOCAL then
27                         Update the priority of s' in FOCAL
28                     else
29                         FOCAL ← FOCAL ∪ {s'}
30         if OPEN ≠ ∅ and f_min < f(head(OPEN)) then
31             updateLowerBound(w f_min, w f(head(OPEN)))
32     return "no solution exists"
33 Procedure updateLowerBound(old_b, new_b):
34     for each s ∈ OPEN do
35         if (f(s) > old_b) ∧ (f(s) ≤ new_b) then
36             FOCAL ← FOCAL ∪ {s}
```

ATPS/ANA*, AFS can use an arbitrary priority function to order FOCAL. Moreover, AFS may reuse previous search effort more efficiently than ARA* and ATPS/ANA* if the priority function it uses satisfies the "efficiently reusable" property, formally defined later.

We start by discussing priority functions and their properties. We then discuss different ways of changing the (suboptimality or cost) bound between consecutive search iterations. Finally, we present pseudocode for AFS and analyze it.

3.3.1 Priority Functions

The freedom to expand any state in FOCAL allows FS to find a suboptimal solution and terminate earlier than A*. Clearly, the efficiency is heavily dependent on the states we choose to expand and hence on the priority function $h_{\texttt{FOCAL}}$ used to sort FOCAL. Different instances of BSS and BCS use different priority functions. For example, wA* is a BSS that uses $h_{\texttt{FOCAL}}(s) = g(s) + wh(s)$, and Potential Search (PS) is a BCS that uses $h_{\texttt{FOCAL}}(s) = (C - g(s))/h(s)$ (henceforth referred to as the *potential function*).

It has already been shown that $h_{\texttt{FOCAL}}$ can be used in both definitions of FOCAL in the context of BSS and BCS [20]. This is also the case in this chapter, that is, $h_{\texttt{FOCAL}}$ can be used in both definitions of FOCAL in the context of anytime BSS and anytime BCS. However, not all priority functions are alike – some enable a more efficient search in any given iteration or more efficient reuse of previous search effort. Thus, we identify the following two properties of priority functions.

Definition 9 (w-admissible $h_{\texttt{FOCAL}}$)**.** *A priority function $h_{\texttt{FOCAL}}(s)$ is* w-admissible *iff $g(s) \leq wg^*(s)$ for every expanded state s, where $g^*(s)$ is the distance from the start state to s.*

A w-admissible $h_{\texttt{FOCAL}}$, such as in wA* [44], enables a more efficient search because the bounded-suboptimality is guaranteed even when every state is expanded at most once. Unfortunately, this is not the case for any $h_{\texttt{FOCAL}}$, as exemplified by the graph in Figure 3.2(a). Here, S denotes the start state, and G denotes the goal state. Assume that $w = 2$ and $h_{\texttt{FOCAL}}$ is the reverse alphabetical order. After expanding S, both A and C are in OPEN with $g(A) = 1$, $f(A) = 11$, $g(C) = 8$ and $f(C) = 16$. Since $f_{\min} = 11$,

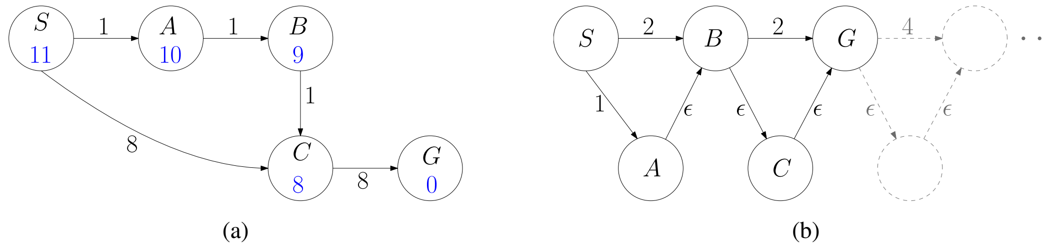


Figure 3.2: (a) Shows an $h_{\texttt{FOCAL}}$ that is not w-admissible. The h-value of each state is specified with the state (in blue) and $c(s_i, s_j)$ is specified near the edge (s_i, s_j). (b) Shows that FS may require re-expansions to guarantee bounded-suboptimality.

both A and C are in FOCAL. C is expanded next with $g(C) = 8$ as $h_{\texttt{FOCAL}}$ is the reverse alphabetical order. Observe that $g^*(C) = 3$, and thus we expand C with $g(C) > wg^*(C)$.

Avoiding state re-expansions in FS can violate the bounded suboptimality guarantee. This is exemplified by the graph in Figure 3.2(b). S denotes the start state, and G denotes the goal state. Assume that $w = 2$ and $h_{\texttt{FOCAL}}$ is the reverse alphabetical order. For simplicity, we assume that all heuristic values are zero. When expanding every state at most once, FS has the following trace: ('$\emptyset$' represents 'not in FOCAL' and '1' represents the head of FOCAL.)

	OPEN	$f(s)(= g(s))$	Order in FOCAL
Expand S	A	1	2
$(wf_{\min} = 2)$	B	2	1
Expand B	A	1	1
$(wf_{\min} = 2)$	C	$2 + \epsilon$	$\emptyset$
	G	4	$\emptyset$
Expand A	C	$2 + \epsilon$	2
$(wf_{\min} = 4 + 2\epsilon)$	G	4	1
Expand G	C	$2 + \epsilon$	1

Thus, FS terminates after expanding G with $g(G) = 4$, while the optimal solution's cost is $1 + 3\epsilon$ (for $\epsilon < 1$). We can easily choose ϵ so that the returned solution's cost is not within the suboptimality bound.

In fact, we can extend this example with additional gadgets (shown in Figure 3.2(b) in grey) to make the solution's suboptimality arbitrarily bad.

The w-admissible property affects the efficiency of any one iteration of FS (that is, one search episode). In the next subsection, we discuss the anytime setting which involves consecutive iterations of FS. The following property affects the efficiency of reusing search effort between such consecutive iterations.

Definition 10 (Efficiently reusable $h_{\texttt{FOCAL}}$)**.** *A priority function $h_{FOCAL}(s)$ is* efficiently reusable *iff it does not depend on w or C.*

The priority functions of wA* and PS are not efficiently reusable. Thus, any change to w or C may require reordering FOCAL, which is a costly operation. As we discuss in the next subsection, anytime algorithms repeatedly tighten their bounds. Thus, the efficiently reusable property can bear significant implications on their runtimes. For example, although ARA* is efficient due to the w-admissible property, its $h_{\texttt{FOCAL}}$ is not efficiently reusable and thus ARA* may still have to reorder its FOCAL between search iterations. Another example is the potential function, used in ATPS/ANA*, which is neither w-admissible nor efficiently reusable.

Another state-of-the-art BSS algorithm that is closely related to FS is explicit estimation search (EES) [83]. EES maintains three lists: The first list is $\texttt{OPEN}_f$, which is equivalent to OPEN as defined previously. The second list is $\texttt{OPEN}_{\hat{f}}$, which includes all states in $\texttt{OPEN}_f$ but is sorted according to $\hat{f}(s) = g(s) + \hat{h}(s)$, where $\hat{h}(s)$ is a (possibly inadmissible) estimate of the cost-to-goal. Denote by $\hat{f}_{\min} = \min_{s \in \texttt{OPEN}_{\hat{f}}} \hat{f}(s)$. The third list is $\texttt{FOCAL}_{\hat{d}}$, which includes all states in $\texttt{OPEN}_{\hat{f}}$ with $\hat{f}(s) \leq w\hat{f}_{\min}$ and is sorted according to $\hat{d}(s)$, a (possibly inadmissible) estimate of the edges-to-goal. Unlike FOCAL, one cannot simply expand states from $\texttt{FOCAL}_{\hat{d}}$ while maintaining suboptimality guarantees because $\hat{h}$ may be inadmissible. Thus, EES uses the following rule when expanding a state: If $f(\texttt{head}(\texttt{FOCAL}_{\hat{d}})) \leq wf_{\min}$, expand $\texttt{head}(\texttt{FOCAL}_{\hat{d}})$. Otherwise, if $f(\texttt{head}(\texttt{OPEN}_{\hat{f}})) \leq wf_{\min}$, expand $\texttt{head}(\texttt{OPEN}_{\hat{f}})$. Otherwise, expand $\texttt{head}(\texttt{OPEN}_f)$. Thus, EES does not fit our formulation of FS although it terminologically uses a focal list.

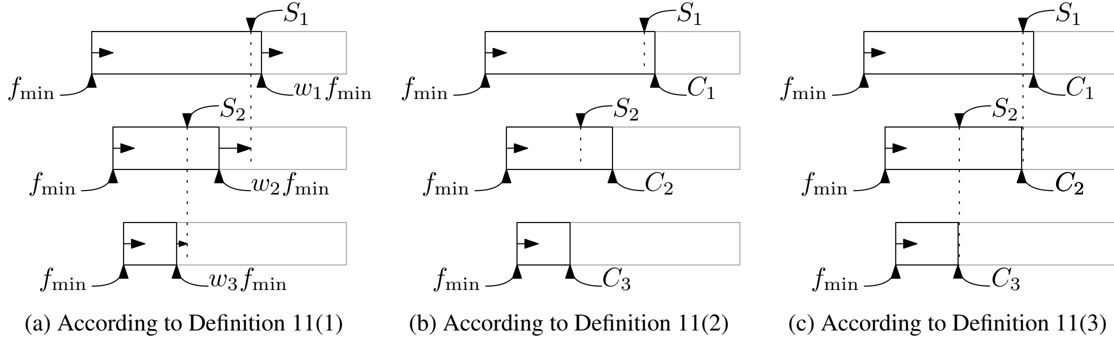


Figure 3.3: Illustrates the anytime effect on FOCAL for the three different ways of updating the bounds.

3.3.2 Anytime Bounds

Denote the costs of the solutions found in progressive search iterations of an anytime algorithm by $S_1, S_2, \ldots$.

Definition 11 (Bounds update scheme)**.** *Three possible ways of updating the (suboptimality or cost) bound between consecutive search iterations are as follows:*

1. *Given a sequence $w_1 > \ldots > w_K = 1$, search iteration i uses w_i as the suboptimality bound.*

2. *Given a sequence $C_1 = \infty > \ldots > C_K$, search iteration i uses C_i as the cost bound.*

3. *The first search iteration uses cost bound $C_1 = \infty$. In search iteration $i > 1$, we adaptively update the cost bound based on S_{i-1}. One common choice is $C_i = S_{i-1} - \epsilon$ (which is equivalent to suboptimality bound $w_i = \frac{S_{i-1}}{f_{\min}} - \epsilon$ when pruning surplus states, see below), where ϵ is a small positive number and $f_{\min}$ is the f-value of the head of OPEN when search iteration $i-1$ terminates.*

In the anytime BSS framework, Definition 11(1) is commonly used. Using $w_1 > \ldots > w_K$ guarantees a sequence of solutions, each with a better suboptimality guarantee than the previous ones. However, just using $w_1 > \ldots > w_K$ does not guarantee that the sequence of solutions has strictly decreasing costs, that is, it is not necessarily the case that $S_i > S_{i+1}$. From a user's perspective, it seems reasonable to expect that an anytime algorithm produces solutions with strictly decreasing costs as time progresses, and, ideally, with "diminishing returns," that is, the algorithm converges quickly to a "good" solution.

One way to accommodate this expectation is to use S_{i-1} as a cost bound for search iteration i. We can use this cost bound to prune *surplus states*, that is, in search iteration i, when generating a state with a cost higher than the current cost bound S_{i-1}, we do not add it to OPEN or FOCAL. Furthermore, when popping such a state from the head of FOCAL (lines 6-10 in Algorithm 1), we do not "process" it (lines 11-29 in Algorithm 1) and instead continue immediately to the next state in FOCAL. With these modifications, an anytime BSS algorithm guarantees that the sequence of solution costs are strictly decreasing, that is, $S_i > S_{i+1}$ for every i. Since $w_K = 1$, it is guaranteed to eventually find an optimal solution. Figure 3.3(a) illustrates the iterative behavior of FOCAL in this update scheme.

Definition 11(2) fits the anytime BCS framework. Unlike the anytime BSS framework, here, there is no guarantee to eventually find an optimal solution for an arbitrary sequence of cost bounds $C_1 > \ldots > C_K$. Figure 3.3(b) illustrates the iterative behavior of FOCAL in this update scheme.

Definition 11(3) is commonly used in the anytime BCS framework [74] when we want to guarantee that we eventually find an optimal solution. Here, too, we start with $C_1 = \infty$ but, in later search iterations, update the cost bound adaptively with respect to the cost of the best solution found so far. Figure 3.3(c) illustrates the iterative behavior of FOCAL in this update scheme. This update scheme essentially unifies the previous two update schemes if we prune surplus states. In fact, ATPS/ANA* uses this update scheme along with the potential function for prioritization of states in FOCAL. Thus, ATPS/ANA* is a special case of AFS.

3.3.3 Pseudocode

Algorithm 2 presents the pseudocode for AFS. AFS uses a specification of one of the update schemes in Definition 11. The main loop of AFS (line 2) is conditioned on the availability of runtime and the suboptimality of the best solution found so far[3]. Inside this loop, AFS calls FS with the current (suboptimality or cost) bound as an argument (line 7). After each search iteration terminates, FOCAL is updated to ensure

[3]When $f_{\min}$ equals the cost of the best solution found so far, we can terminate with the optimal solution.

Algorithm 2: Anytime Focal Search (AFS).
s_{start} is the start state; and getNextBound() is a specification of one of the update schemes in Definition 11. Blue (red) represents pseudocode relevant for BSS (BCS) only.

Input: s_{start}, getNextBound().
Output: Solution(s).

```
1  OPEN = FOCAL = {s_start}
2  while search not halted or optimal solution not found do
3      w (or C) ← getNextBound()
4      updateFocalBound(w f(head(OPEN)) (or C))
5      if h_FOCAL is not efficiently reusable then
6          reorder FOCAL
7      sol ← findPath(w (or C))
8      if sol = no-solution then
9          break
10     report sol
11 Procedure updateFocalBound(new_b):
12     for each s ∈ FOCAL do
13         if f(s) > new_b then
14             FOCAL ← FOCAL \ {s}
```

that all of its states are within the new (suboptimality or cost) bound (line 4). If h_{FOCAL} is not efficiently reusable, FOCAL is reordered (line 6).

3.3.4 Theoretical Properties

FS and AFS are different from ARA* in that they may require state re-expansions within the same search iteration to guarantee finding solutions with costs within the (suboptimality or cost) bound if h_{FOCAL} is not w-admissible. While state re-expansions may result in longer runtimes of some search iterations, they allow AFS to provide suboptimality guarantees for each search iteration. Such (suboptimality or cost) bounds are important since it is not known in advance when an anytime algorithm is forced to terminate. Moreover, AFS does not need to maintain any additional lists, such as INCONS[4] in ARA*. This makes AFS simpler to understand and implement.

[4]A state is *inconsistent* from the time it is generated with a smaller g-value than it previously had and until the time it is expanded. INCONS contains all inconsistent states generated during a search iteration, and thus they have to be put into OPEN before the next search iteration begins.

Theorem 12. *In each search iteration i with w_i, AFS is w_i-suboptimal for finite graphs with positive edge costs*[5].

Proof. Iteration i terminates: Since we assume that all edge costs are positive and the graph is finite, there are only finitely many cycle-free paths from the start state to any state s. Since FS re-expands states only when their g-values decrease, FS never explores paths with cycles and therefore expands s at most a finite number of times (no more than the number of cycle-free paths from the start state to s). Therefore, FS always terminates.

If search iteration i terminates with an empty OPEN, it reports 'no solution.' For $i = 1$, this means that the problem is unsolvable. For $i > 1$, this means that the solution found in iteration $i - 1$ is optimal. If search iteration i terminates with a non-empty OPEN, it terminates by expanding a goal state G_i. Assume that G_i is expanded with g-value S_i in search iteration i. We need to show that $S_i \leq w_i S^*$ (where S^* is the cost of the optimal solution). This follows directly from the w-suboptimality proof of FS: Let s' be the first state in OPEN on a cost-minimal path to G^* with $g(s) = g^*(s)$ (such a state always exists according to Lemma 22),

- Since h is admissible, $f(s') \leq S^*$.
- By definition of $f_{\min}$, $f_{\min} \leq f(s')$.
- By definition of FOCAL, $f(G_i) \leq w_i f_{\min}$.

Thus, $S_i = f(G_i) \leq w_i f_{\min} \leq w_i f(s') \leq w_i S^*$.

□

We now prove that AFS with the potential function computes suboptimality bounds that are tighter than those of ATPS/ANA*. This is so because FS maintains $f_{\min}$ at all times and its suboptimality bound is $S/f_{\min}$ while PS has a suboptimality bound of $\max_{s \in \texttt{FOCAL}}(C - g(s))/h(s)$ [74]. Here, S is the cost of

[5] It *could* be the case that the theorem also holds for infinite graphs if all edge costs are greater than some positive constant and a solution with finite cost exists.

the solution found by both FS and PS, $f_{\min}$ is the f-value of the head of OPEN when FS terminates, and C is the cost bound used by PS.

Lemma 13. *Let B_{PS} and B_{FS} be the suboptimality bounds computed by PS and FS, respectively. $B_{FS} \leq B_{PS}$.*

Proof.

$$B_{PS} = \max_{s \in \texttt{FOCAL}} \frac{C - g(s)}{h(s)}.$$

Since $\frac{C-g(s)}{h(s)} \geq 1$ and $g(s) \geq 0$ for every s in FOCAL,

$$B_{PS} \geq \max_{s \in \texttt{FOCAL}} \frac{C - g(s) + g(s)}{h(s) + g(s)} = \max_{s \in \texttt{FOCAL}} \frac{C}{f(s)}.$$

Since C is a constant and $C \geq S$,

$$B_{PS} \geq \frac{C}{\min_{s \in \texttt{FOCAL}} f(s)} = \frac{C}{f_{\min}} \geq \frac{S}{f_{\min}} = B_{FS}.$$

Hence, $B_{FS} \leq B_{PS}$. □

The fact that AFS computes tighter suboptimality bounds than ATPS/ANA* can have implications on the anytime behavior because it allows AFS to decrease the bound faster. AFS is also more general than ATPS/ANA* because ATPS/ANA* is a special case of AFS in which FOCAL is sorted according to a specific priority function (namely, the potential function), while AFS allows for arbitrary priorities. Unlike ATPS/ANA*, when the priorities for ordering FOCAL are efficiently reusable, AFS is not required to iterate over FOCAL and reorder it when the bound changes between search iterations. This could translate to substantial time savings when FOCAL is large or when solutions are found frequently. Moreover, AFS also facilitates anytime search in domains that have no well-defined heuristic function and, hence, no useful definition of potential function. Here, AFS is still viable but ATPS/ANA* is not. Finally, the flexibility

with arbitrary priorities in AFS allows incorporating domain-specific knowledge. This, in turn, can guide the search process better.

3.4 Anytime Conflict-Based Search

We base our anytime MAPF solver on Bounded Conflict-Based Search (BCBS). BCBS(w_L, w_H) is a bounded-suboptimal variant of CBS that uses a FS with suboptimality bound $w_L \geq 1$ instead of a best-first search for the low-level search, and a FS with suboptimality bound $w_H \geq 1$ instead of a best-first search for the high-level search. Both the low-level and high-level focal searches expand states with fewer collisions first by using a priority function that gives higher priorities to states with fewer collisions. Specifically, when generating a low-level state s during a low-level search for agent j, $h_{\texttt{FOCAL}}(s)$ is set to the number of collisions between the path from s^j to s and the paths of all agents other than j. When generating a high-level state s, $h_{\texttt{FOCAL}}(s)$ is set to the number of collisions between the paths specified in s. BCBS(w_L, w_H) is $w_L w_H$-suboptimal [4].

Our anytime MAPF solver is similar to BCBS but replaces its high-level FS with AFS. For the low-level FS, we set $w_L = 1$ to guarantee that our anytime MAPF solver eventually finds an optimal solution. Other values of w_L can be used for increased efficiency but, even if given enough time, the final solution is only w_L-suboptimal. We leave it for future work to explore an efficient use of AFS in both the low-level search and high-level search simultaneously.

At the time of publication [11], no non-trivial admissible heuristics[6] for the high-level search of CBS were known. This allowed us to exemplify the broader applicability of AFS compared to other anytime algorithms. Specifically, ATPS/ANA* is not applicable when $h \equiv 0$ because the potential function is undefined. Similarly, ARA* is not applicable when $h \equiv 0$ because $f(s) = g(s) + wh(s) = g(s)$; thus, inflating w has no effect, which means it cannot facilitate anytime search. Unlike ATPS/ANA* and ARA*,

[6] A trivial admissible heuristic assigns the value 0 to all states.

AFS can work even when $h \equiv 0$ because FOCAL is still well defined (that is, FOCAL $= \{s \in$ OPEN : $f(s) = g(s) \leq w f_{\min}\}$).

While admissible heuristics for the high-level search of CBS have been recently developed [18, 42], we leave it for future work to evaluate their impact on the efficiency of anytime MAPF solvers. Nevertheless, AFS can benefit from the availability of heuristic estimates, which make the f-values more informed. Finally, rather than simply inflating an admissible heuristic, AFS enjoys the advantage of using the number of collisions between paths of agents in a high-level state as the priority function. The freedom to choose an inadmissible and domain-specific priority function helps to guide the search towards a solution quickly. This priority function is not only informative but is also efficiently reusable.

3.5 Experimental Results

In this section, we describe experiments performed to evaluate AFS in two different domains. The first domain is the MAPF problem, in which we evaluate anytime BCBS in three different environments. The second domain is the generalized covering traveling salesman problem.

3.5.1 The MAPF Domain

We first describe the three environments—dubbed 32×32_20%, Drones and Kiva—and how we generate MAPF problems for them. We then evaluate anytime BCBS on these MAPF problems.

The 32×32_20% environment is depicted in Figure 3.4. It is a 32×32 four-neighbor grid environment with obstacles placed randomly (more specifically, this environmnet was generated with each cell having a 20% chance of being an obstacle). Such environments are commonly used in research papers [4, 86]. We generate MAPF problem instances as follows. For each number of agents between 50 and 120 in increments of 10, we randomly generate 50 MAPF problem instances. We generate each MAPF problem instance by choosing start and goal cells for every agent at random. We also ensure that no two agents

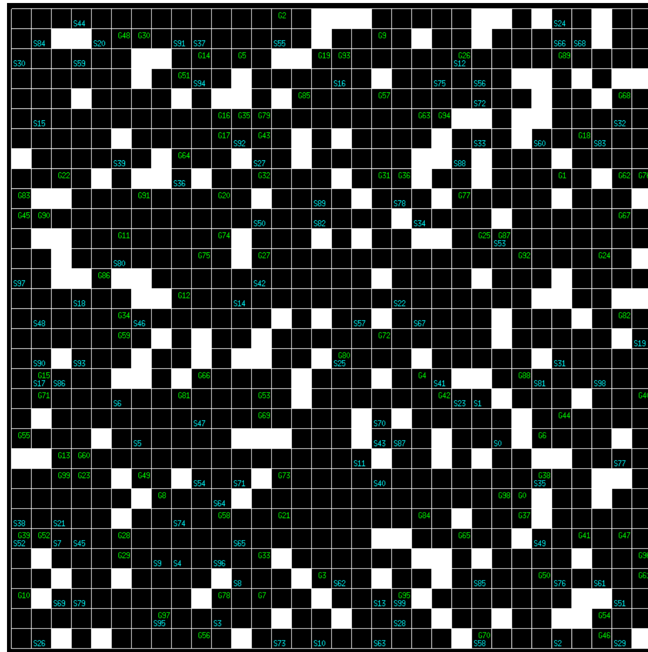


Figure 3.4: Shows the 32×32_20% environment. The start and goal cells of 100 agents in this MAPF problem instance are shown in cyan and green, respectively.

share the same start cell or the same goal cell. Figure 3.4 also shows the start and goal cells of 100 agents in a MAPF problem instance.

The Drones environment is depicted in Figure 3.5. It is a 55×120 four-neighbor grid environment with obstacles located such that the environment resembles a city's skyline. We aim to simulate drones taking off and landing in different parts of the city. Thus, we generate MAPF problem instances as follows. For each number of agents between 10 and 50 in increments of 10, we randomly generate 50 MAPF problem instances. We generate each MAPF problem instance by choosing start and goal cells for every agent in a random column at the bottommost non-obstacle row. We also ensure that no two agents share the same start cell or the same goal cell. Figure 3.5 also shows the start and goal cells of 100 agents in a MAPF problem instance.

The Kiva environment is depicted in Figure 3.6. It is a 22×54 four-neighbor grid environment with obstacles placed such that the environment resembles the layout of an automated warehouse. It contains corridors of width 1 connecting an open space of size 22×5 on the right side with an open space of size

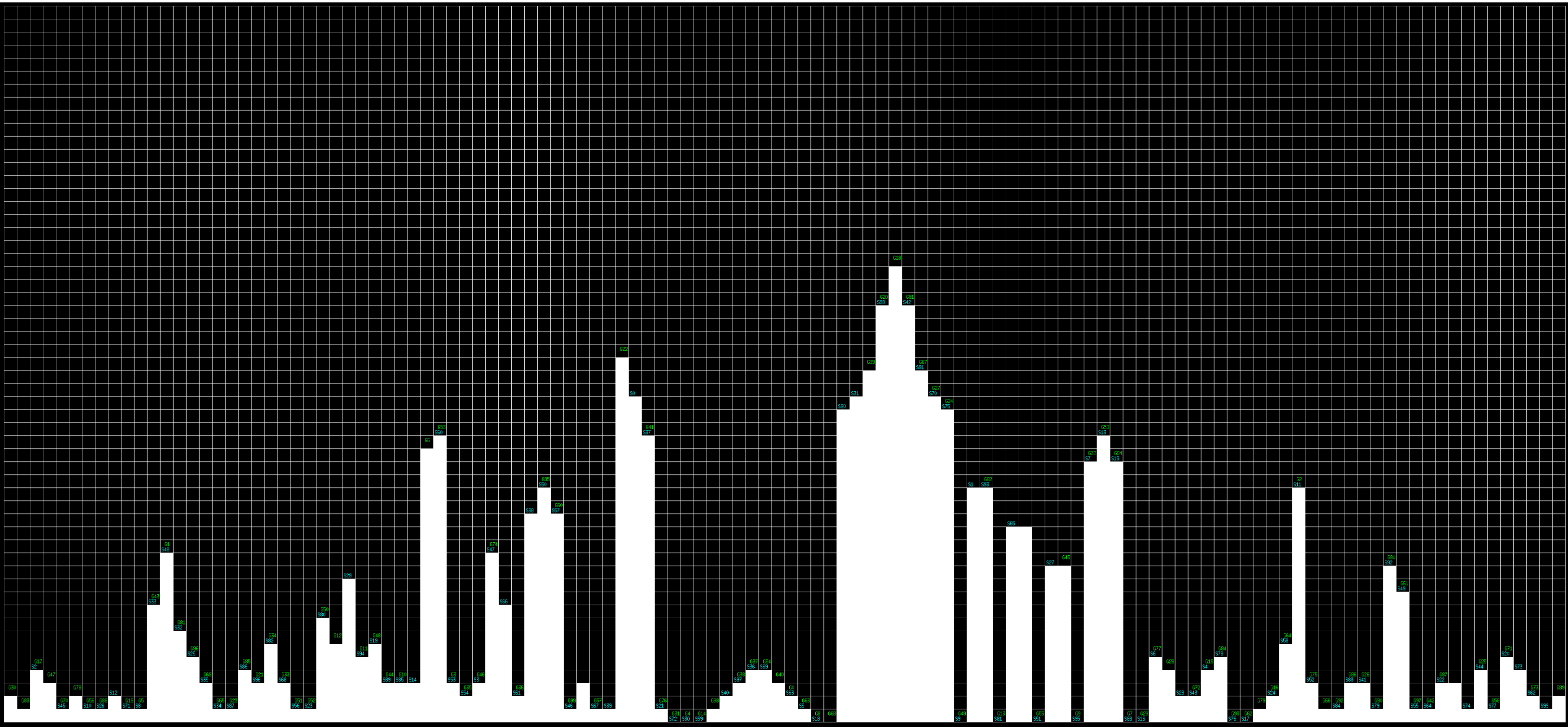


Figure 3.5: Shows the Drones environment. The start and goal cells of 100 agents in this MAPF problem instance are shown in cyan and green, respectively.

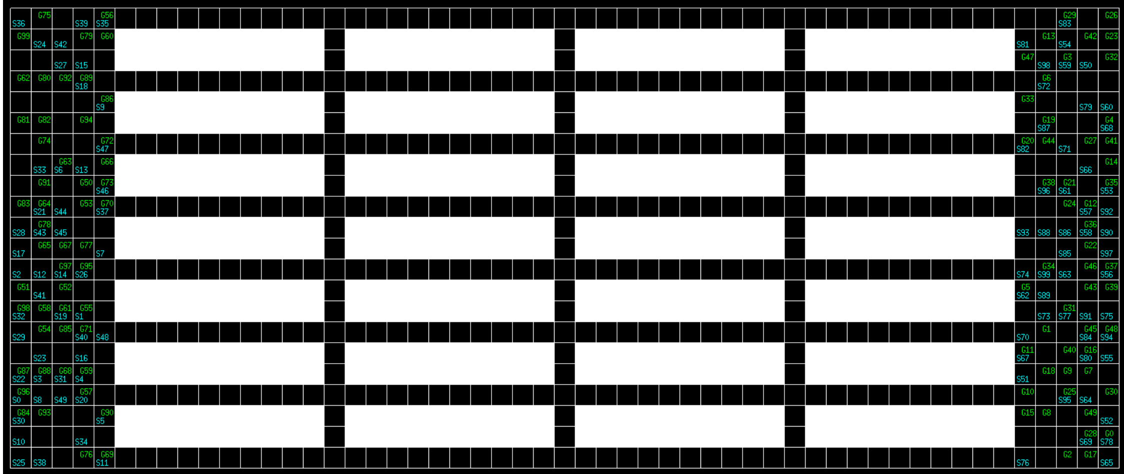


Figure 3.6: Shows the Kiva environment. The start and goal cells of 100 agents in this MAPF problem instance are shown in cyan and green, respectively.

22×5 on the left side. We aim to simulate robots carrying shelving units from one side of the warehouse to a pick-pack-and-ship station on the other side of the warehouse. Thus, we generate MAPF problem instances as follows. For each number of agents between 10 and 50 in increments of 10, we randomly generate 50 MAPF problem instances. Each MAPF problem instance is generated by randomly assigning a start cell in the left open space and a goal cell in the right open space to half of the agents, and vice versa for the other half of the agents. We also ensure that no two agents share the same start cell or the same goal cell. Figure 3.6 also shows the start and goal cells of 100 agents in a MAPF problem instance.

We evaluate anytime BCBS on the above environments. All experiments were run on an Ubuntu 19.04 desktop having an Intel(R) Core i7-7700 CPU @ 3.6GHz with four physical cores and 16GB RAM. Each experiment used one physical core and has a 100-second time limit; and four experiments ran on the desktop in parallel at any given moment. All solvers are implemented in C++. Definition 11(3) is used as the bound update scheme and $w_1 = 10$ for all experiments.

Table 3.1 reports the success rate of anytime BCBS for each environment and number of agents. Here, the success rate is defined to be the fraction of MAPF problems (out of 50) for which anytime BCBS finds at least one solution within the 100-second time limit. We observe that, in all environments, increasing the number of agents results in lower success rates. This is expected as adding more agents increases the

32x32_20% environment				Drones environment				Kiva environment			
Number of agents	Success rate anytime BCBS	Success rate CBS	Success rate* ECBS(*bb*)	Number of agents	Success rate anytime BCBS	Success rate CBS	Success rate* ECBS(*bb*)	Number of agents	Success rate anytime BCBS	Success rate CBS	Success rate* ECBS(*bb*)
50	1.00	0.06	0.14	10	1.00	0.90	0.10	10	1.00	1.00	N/A
60	1.00	0	0.26	20	0.96	0.70	0.15	20	1.00	0.82	0
70	1.00	0	0.48	30	0.82	0.28	0.11	30	0.94	0.04	0.04
80	0.98	0	0.71	40	0.62	0.04	0.13	40	0.76	0	0.10
90	0.98	0	0.71	50	0.44	-	-	50	0.12	-	-
100	0.94	0	0.95								
110	0.88	0	1								
120	0.44	-	-								

Table 3.1: Success rates of anytime BCBS, CBS and ECBS for different numbers of agents in the 32×32_20%, Drones and Kiva environments. Success rate* indicates success rate over the MAPF problem instances that CBS failed to solve.

number of collisions encountered by the MAPF solver, which in turn may exponentially increase the size of the high-level tree.

We now analyze the behavior of anytime BCBS for all numbers of agents for which it achieved a success rate higher than 50%. Figures 3.7 (a), (b), (c) and (d) show results for MAPF problem instances in the 32×32_20% environment with 50, 70, 90 and 110 agents, respectively. Each of these figures show the anytime profiles of six representative MAPF problem instances. The anytime profile of a MAPF problem instance shows the suboptimality bounds of the solutions[7] (y-axis) as a function of time (x-axis, in logarithmic scale). Figures 3.8 (3.9) (a), (b), (c) and (d) show results for MAPF problem instances in the Drones (Kiva) environment with 10, 20, 30 and 40 agents, respectively. Each of these figures show the anytime profiles of six representative MAPF problem instances. In all environments, higher numbers of agents tend to result in longer times needed to find the first solution as well as higher suboptimality bounds of the solutions found.

Finally, Figures 3.7(e), 3.8(e) and 3.9(e) report the average suboptimality bound (y-axis) as a function of time (x-axis, in logarithmic scale) for the 32×32_20%, Drones and Kiva environments, respectively. The average suboptimality bound in all three environments exhibits the diminishing returns property, that is characteristic of good anytime behavior [9, 80]. In all environments, higher numbers of agents result in longer times needed to find the first solution as well as higher suboptimality bounds of the solutions found.

[7] the ratio between the solution cost and $f_{\min}$

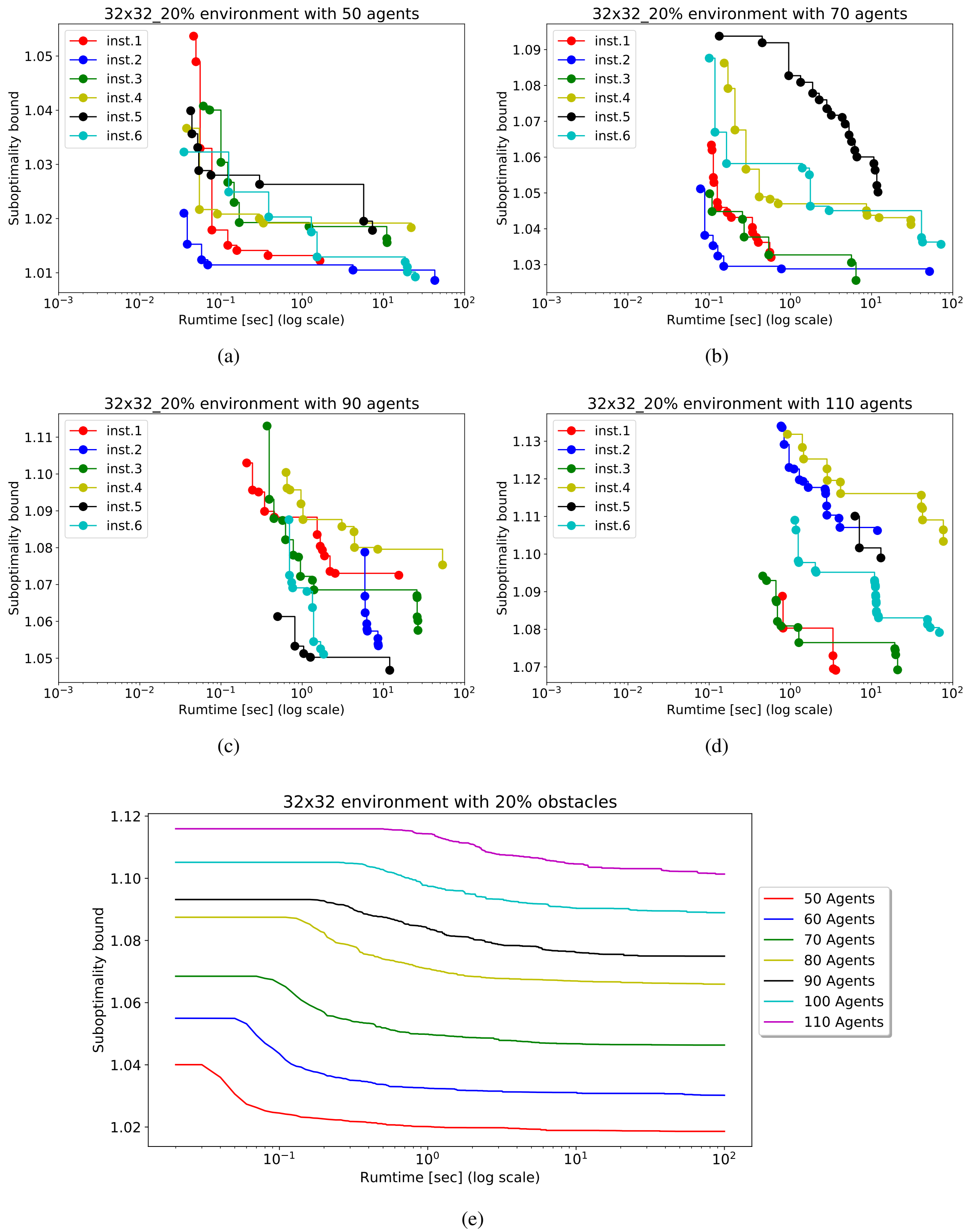


Figure 3.7: (a), (b), (c) and (d) show the typical behavior of anytime BCBS on a few MAPF problem instances in the 32×32_20% environment with 50, 70, 90 and 110 agents each, respectively. (e) shows the aggregate behavior of anytime BCBS for numbers of agents with more than 50% success rate in the 32×32_20% environment.

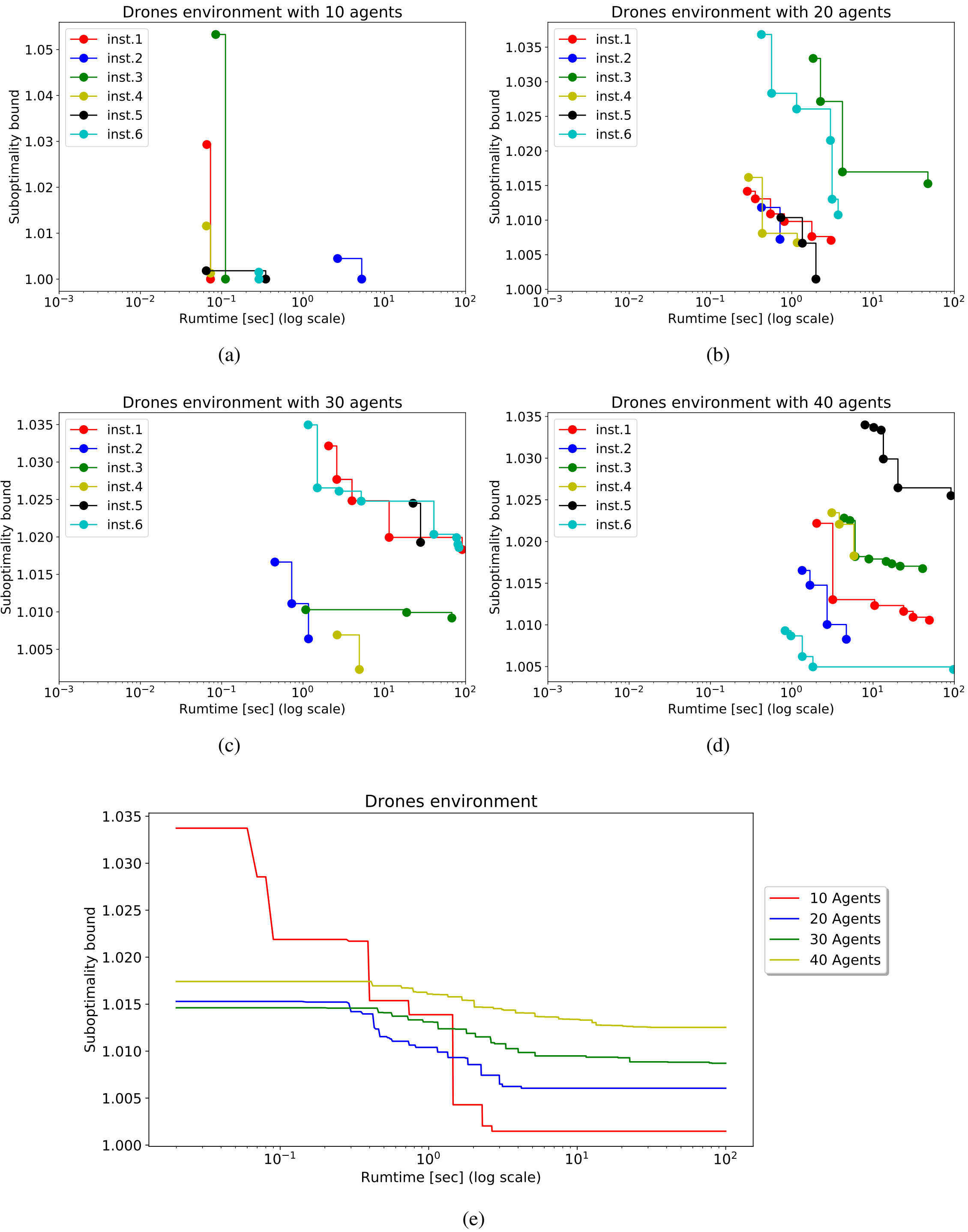


Figure 3.8: (a), (b), (c) and (d) show the typical behavior of anytime BCBS on a few MAPF problem instances in the Drones environment with 10, 20, 30 and 40 agents each, respectively. (e) shows the aggregate behavior of anytime BCBS for numbers of agents with more than 50% success rate in the Drones environment.

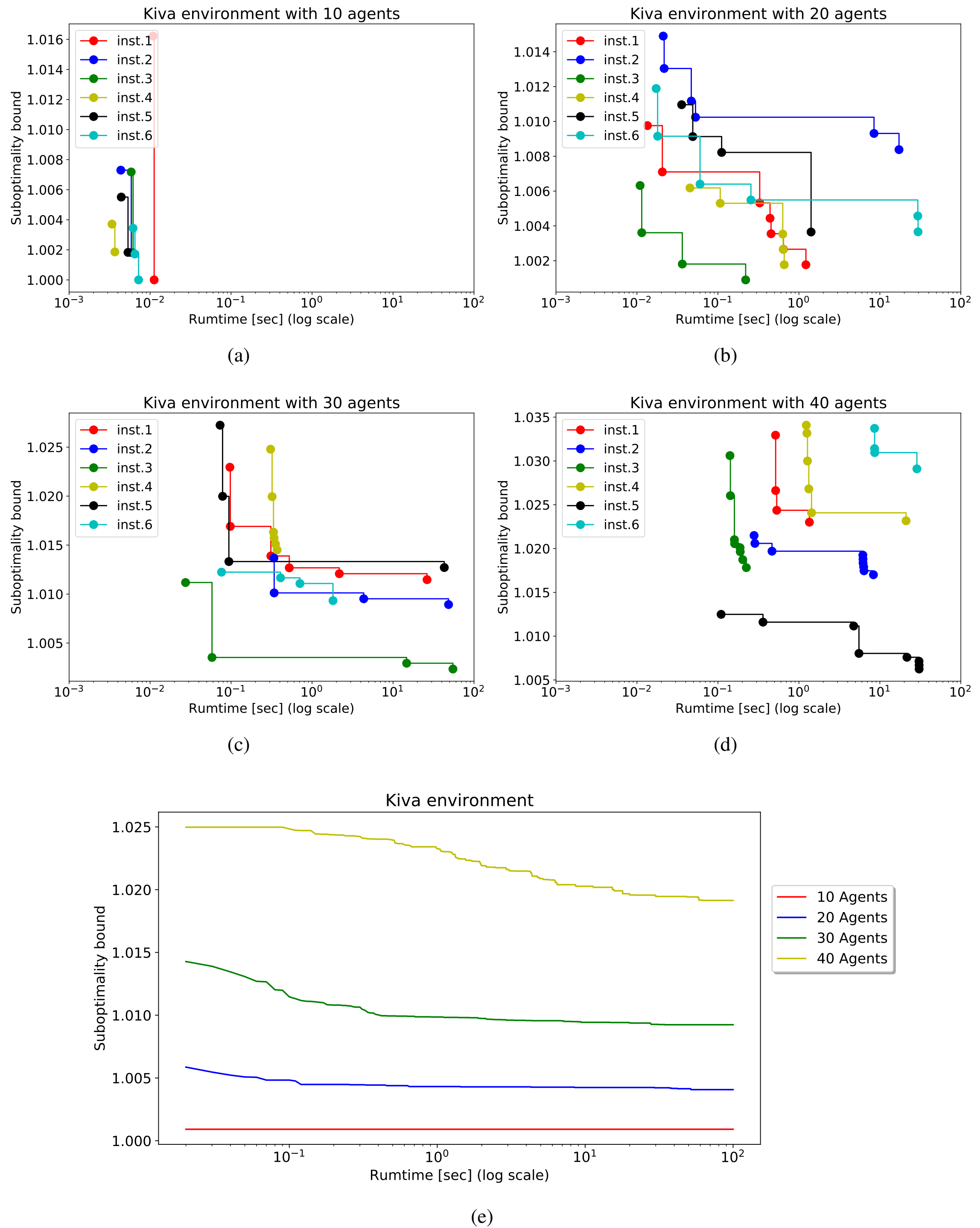


Figure 3.9: (a), (b), (c) and (d) show the typical behavior of anytime BCBS on a few MAPF problem instances in the Kiva environment with 10, 20, 30 and 40 agents each, respectively. (e) shows the aggregate behavior of anytime BCBS for numbers of agents with more than 50% success rate in the Kiva environment.

To the best of our knowledge, anytime BCBS is the first anytime MAPF solver and the only one that provides suboptimality guarantees. We conduct experiments to compare it with CBS and ECBS.

The first comparison is with CBS. We let CBS run on all MAPF problem instances with the same 100-second time limit. As reported in Table 3.1, the success rates of CBS are significantly lower than those of anytime BCBS. For example, the third row reports that the success rate in the 32×32_20% environment for 70 agents is 0% for CBS and 100% for anytime BCBS, the success rate in the Drones environment for 30 agents is 28% for CBS and 82% for anytime BCBS, and the success rate in the Kiva environment for 30 agents is 4% for CBS and 94% for anytime BCBS. This shows that anytime BCBS can solve MAPF problem instances with higher numbers of agents more efficiently than CBS.

The second comparison is with ECBS, a state-of-the-art bounded-suboptimal MAPF solver.[8] Here, we let ECBS run on all MAPF problem instances for which anytime BCBS finds at least one solution but CBS does not. For example, in the Drones environment with 30 agents, anytime BCBS finds at least one solution for 41 MAPF problem instances and CBS finds a solution for 14 MAPF problem instances. Thus, we evaluate ECBS on 27 (41 minus 14) MAPF problem instances for this environment and number of agents. For each such MAPF problem instance, we let ECBS run with a suboptimality bound that is set equal to bb, the suboptimality bound of the best solution produced by anytime BCBS. The success rate for ECBS(bb) is then the fraction of these MAPF problem instances that it solves, each with its bb suboptimality bound and the same 100-second time limit. We observe that, for higher numbers of agents in the 32×32_20% environment, the average value of bb is relatively high. Thus, ECBS(bb) has enough flexibility and finds solutions to most or all MAPF problem instances. However, for lower numbers of agents in the 32×32_20% environment, as well as for all numbers of agents in the Drones and Kiva environments, the average value of bb is relatively low. Thus, ECBS(bb) does not have enough flexibility and frequently fails to find solutions. This shows that anytime BCBS can be more effective than ECBS(bb).

Overall, anytime BCBS is more efficient than the optimal MAPF solver CBS as well as the bounded-suboptimal MAPF solver ECBS(bb). It also has the characteristic feature of finding "good" solutions

[8]ECBS is also based on FS but uses it differently. We provide a detailed description of ECBS in Chapter 4.2.2.

"quickly", that is, it frequently produces a solution within 2% suboptimality in less than 1 second when the other two MAPF solvers fail in the Drones and Kiva environments. However, the success rate of anytime BCBS falls beyond 40 agents in both environments. We believe that there are two main reasons for this. First, anytime BCBS uses FS only in its high-level search (that is, its low-level searches are optimal), which can make it inefficient for higher numbers of agents. Second, anytime BCBS does not take advantage of the specific structure present in these environments. In particular, the Kiva environment has a specific kind of structure characterized by narrow passageways. It is generally considered a hard environment for MAPF solvers because significant coordination between the agents is required when they travel through these narrow passageways. For such environments, it is important to take advantage of specific structure for improving efficiency. We discuss techniques for doing so in Chapter 4.

3.5.2 The Generalized Covering Traveling Salesman Problem Domain

We now demonstrate the usefulness of AFS for solving a different hard combinatorial problem, namely, the generalized covering traveling salesman problem (GCTSP). The GCTSP [68] is defined by an undirected graph that has one depot vertex and other vertices called facilities. Weighted edges between vertices represent distances. Each facility has a set of costumers associated with it, and customer i has a prize p_i. A customer can be covered by more than one facility. The task in GCTSP is to find a tour that starts at the depot and collects a specified minimum prize P while minimizing the total distance traveled. A tour collects the prizes of all customers associated with any of its facilities. In GCTSP's search space, each state represents a partial tour of facilities with its cumulative prize [62, 81]. A successor of a state augments a non-visited facility to the partial tour. We define the heuristic value of a state n with cumulative prize cp to be $h(n) = H(P - cp)$. $H(P - cp)$ is the minimum distance from the depot to any state with prize $P - cp$. This heuristic is admissible and pre-computed using Dijkstra's algorithm.

We evaluate AFS in the GCTSP domain on benchmark problem instances from [68]. We use 69 medium problem instances (between 100 and 200 vertices) and 14 large problem instances (between 535 and 1000 vertices). AFS uses the cumulative prize multiplied by the potential function as its priority

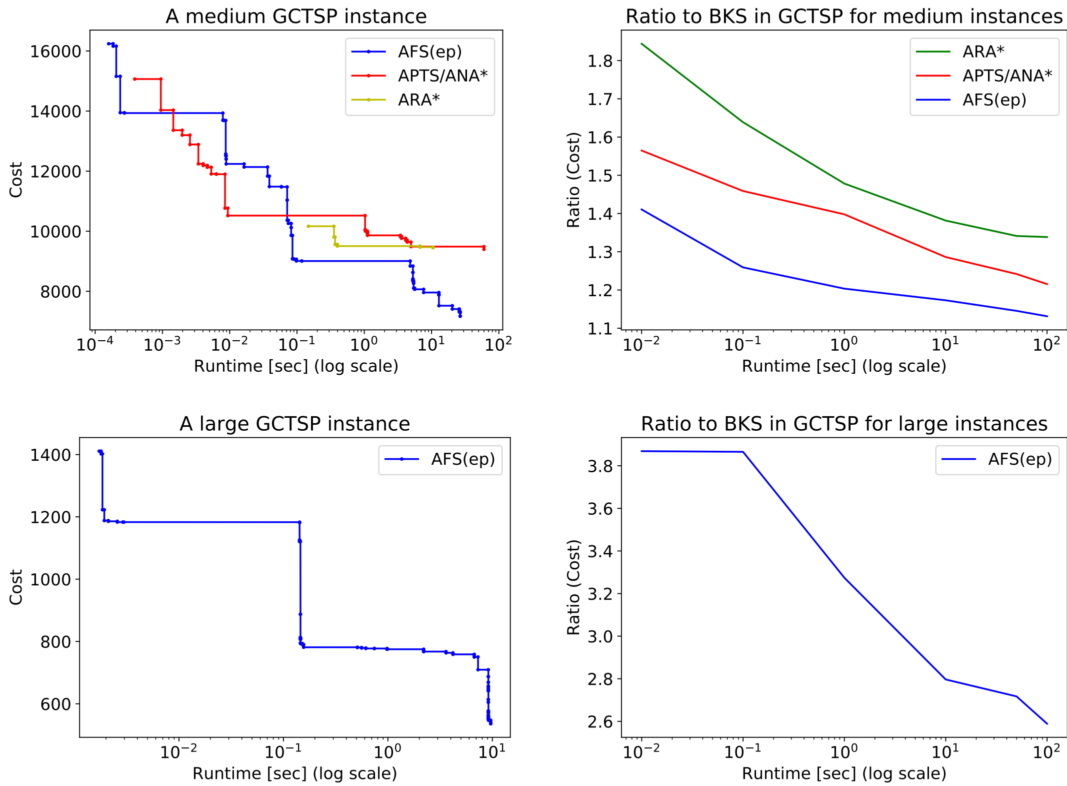


Figure 3.10: Shows the behaviors of anytime BSS and BCS in the GCTSP domain. The left column shows behaviors on typical medium and large size problem instances. The right column shows aggregate behaviors on 69 medium and 14 large problem instances. BKS stands for best known solution.

function. We also evaluate ATPS/ANA* and ARA*. All runs have a time limit of 100 seconds and use Definition 11(3) as the bound update scheme. Figure 3.10 shows the results. On medium-sized problem instances, AFS convincingly beats ATPS/ANA* and ARA*. More significantly, both ATPS/ANA* and ARA* fail to find any solution within the time limit for any of the 14 large-sized problem instances. These results suggest that adding domain-dependent knowledge to the priority function, as allowed in the general framework of AFS, has significant runtime benefits.

3.6 Conclusions and Future Work

In this chapter, we presented AFS, an anytime version of FS that unifies the anytime variants of BSS and BCS. We also emphasized the generality of AFS and showed how other state-of-the-art anytime search algorithms, like ARA* and ATPS/ANA*, are special cases of it. Theoretically, we proved the correctness and bounded suboptimality of AFS, the better quality of its bounds compared to ATPS/ANA*, and its ability to efficiently reuse previous search effort when it does not need to reorder FOCAL between search iterations. Empirically, we demonstrated the benefits of incorporating domain-specific knowledge in $h_{\texttt{FOCAL}}$.

The success of AFS in the GCTSP and MAPF domains is illustrative of a more general advantage of its framework. When admissible estimates of the costs-to-goal are available, AFS can always use them in h. When the available estimates are inadmissible but informative, AFS gives us the important option to use them in $h_{\texttt{FOCAL}}$. Indeed, for many hard combinatorial problems, efficient approximation algorithms produce such inadmissible but informative estimates of cost-to-goal. Moreover, abstractions and relaxations of search problems are admissible but not always informative. While their additive combinations may not be admissible, they are often informative and can be used in the AFS framework while providing bounded-suboptimality guarantees.

In future work, applying the ideas of AFS to Multi-Heuristic A* (MHA*) [3] can be exciting. While an anytime version of MHA* was recently proposed [55], it relies on iteratively decreasing the inflation factor used to inflate h-values. Instead, a version inspired by AFS would rely on iteratively decreasing the flexibility of expanding states with suboptimal f-values. In future work in the context of MAPF, developing an efficient anytime version of ECBS could potentially lead to a better anytime MAPF solver. Like anytime BCBS with $w_L > 1$, anytime ECBS enjoys the additional flexibility of bounded-suboptimal searches in both the high- and low-level searches. Unlike anytime BCBS with $w_L > 1$, anytime ECBS guarantees that it will eventually converge to an optimal solution, and is thus desirable.

Chapter 4

The Highway Heuristic

Focal Search (FS) is at the core of Enhanced Conflict-Based Search (ECBS), the state-of-the-art bounded-suboptimal MAPF solver. In this chapter, we show that, despite ECBS being a state-of-the-art MAPF solver, it is often inefficient in structured environments, such as ones with long narrow corridors. This inefficiency is a problem because many application domains, such as automated warehouses, have structured environments. Thus, we first develop bounded-suboptimal MAPF solvers that exploit the environment's structure. We propose to achieve this using the concept of "highways". Highways are user-provided edges that encourage a global coordination of agents in avoiding collisions with each other. We then develop two approaches, a graphical model-based approach and a heatmap-based approach, for automatically generating highways. On the theoretical side, we show that ECBS with highways is bounded-suboptimal and, more generally, that FS with inflated heuristics still provides suboptimality guarantees. On the experimental side, we show that ECBS with suitable highways is often more efficient and more effective than ECBS in a variety of environments.

This chapter is organized as follows: In Section 4.1, we motivate the use of highways. In Section 4.2, we provide relevant background on Experience Graphs and ECBS. In Section 4.3, we show that FS with inflated heuristics provides suboptimality guarantees. In Section 4.4, we discuss the limitations of ECBS with the shortest path heuristic and formally define the highway heuristic. In Section 4.5, we show that ECBS with the highway heuristic is bounded-suboptimal and provide experimental results to evaluate its

efficiency. Finally, in Section 4.6, we develop two approaches that generate highways automatically and provide experimental results to evaluate the efficiency of ECBS when it uses these automatically generated highways.

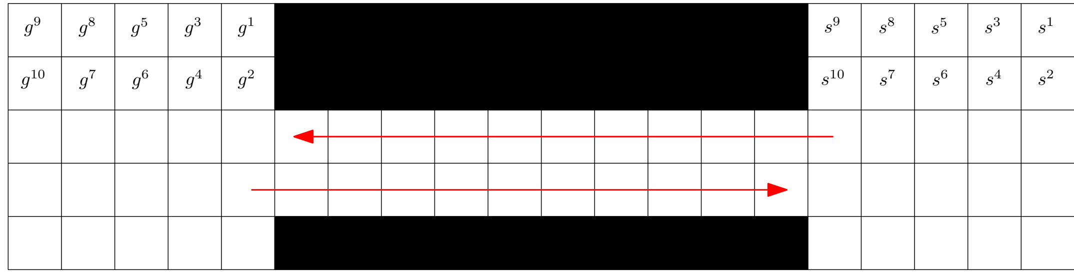
4.1 Introduction

In this chapter, we develop bounded-suboptimal MAPF solvers that have two benefits. First, our MAPF solvers are more efficient than existing state-of-the-art bounded-suboptimal MAPF solvers. Second, our MAPF solvers generate solutions that have more predictable paths. The predictability of agents' motions can be important for the safety of humans, for example, when agents share their workspace with them.

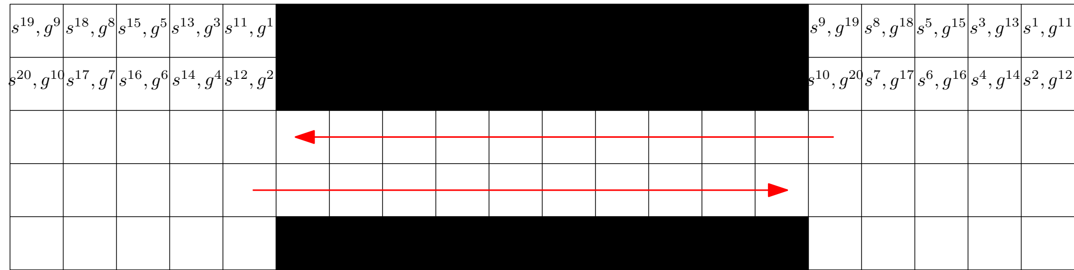
To achieve these benefits, our MAPF solvers enhance existing state-of-the-art MAPF solvers, such as CBS and M*, with a novel heuristic called the *highway heuristic* (henceforth referred to as $h_{\texttt{HWY}}$). $h_{\texttt{HWY}}$'s values are derived by a simple inflation scheme based on the ideas underlying Experience Graphs (EGs) [58]. Like EGs, $h_{\texttt{HWY}}$ is a w-admissible heuristic (Definition 18). Unlike EGs, $h_{\texttt{HWY}}$ does not rely on information from previous computations. Instead, a set of edges (called *highways*) in the environment's graph is used when computing $h_{\texttt{HWY}}$. Highways are either user-provided or automatically generated from a given MAPF problem instance.

In transportation, highways are used to restrict the travel direction of vehicles in order to avoid collisions with other vehicles that travel in the opposite direction. The motivation behind $h_{\texttt{HWY}}$ is similar – using it encourages agents to choose paths that include edges from the highways, which, in turn, promotes a global behavior of the agents that reduces collisions. Reducing the number of collisions between agents' paths translates to shorter runtimes of CBS or M*-based MAPF solvers because their runtimes are exponential in the number of collisions encountered [69, 86].

Since the runtimes of CBS or M*-based MAPF solvers can be exponential in the number of collisions encountered, "good" highways are those that help direct the agents so that there are fewer collisions



(a)



(b)

Figure 4.1: Illustrates that the agents' start (s^i) and goal (g^i) cells influence whether the given highways are helpful. Red arrows represent the edges of the given highways. (a) shows a MAPF problem instance with 10 agents, all of which need to move from right to left. In this case, the given highways are not helpful. (b) shows a MAPF problem instance with 20 agents, of which 10 need to move from right to left and the other 10 need to move from left to right. In this case, the given highways are helpful.

between them. Finding good highways requires us to reason about the structure of the MAPF problem instance, that is, both the environment's layout as well as the agents' start and goal cells. For example, Figure 4.1 shows two MAPF problem instances with the same environment and highways. In Figure 4.1(a), all agents need to move from left to right, and the highways are not helpful. However, in Figure 4.1(b), some agents need to move from right to left while other agents need to move from left to right, and the highways are helpful. When $h_{\texttt{HWY}}$ uses highways that are helpful for the MAPF problem instance at hand, we say that our MAPF solvers exploit the structure of the MAPF problem instance.

Other than $h_{\texttt{HWY}}$, we also develop two methods for automatically generating highways. The first method uses graphical models, and the second method uses heat maps. While these methods have some limitations and the highways produced by them are not always as helpful as user-provided highways, we show that they can still increase the efficiency of our MAPF solvers compared to ECBS.

4.2 Background

In this section, we provide relevant background on EGs and ECBS.

4.2.1 Experience Graphs

In domains where tasks are repetitive, it can be beneficial for a planner to reuse previously computed paths. An Experience Graph (EG) [58] is a set of paths that were generated by the planner previously or provided by a human. When a planner solves a new problem instance in the same environment, it can use the EG to bias the states chosen for expansion towards states in the EG that may help to expand a goal state faster. This bias is achieved by inflating heuristic values of states that are not part of the EG. Planners that use EGs have been shown to be efficient for motion planning of repetitive tasks, such as moving objects in a kitchen environment with mobile manipulators [57].

4.2.2 Enhanced Conflict-Based Search

ECBS(w) [4] is a bounded-suboptimal MAPF algorithm based on CBS that is faster than all other bounded-suboptimal MAPF algorithms compared in [4]. Like BCBS, the high-level and low-level searches of ECBS(w) are focal searches. However, unlike BCBS that has two user-provided parameters, $w_L \geq 1$ and $w_H \geq 1$, and suboptimality guarantees of $w_L w_H$, ECBS(w) has only one user-provided parameter, $w \geq 1$, and suboptimality guarantees of w.

The low-level search of ECBS(w) is identical to that of BCBS(w_L, w_H), that is, the low-level search for agent j is a FS with $\texttt{FOCAL}$ that contains all low-level states $s \in \texttt{OPEN}$ with $f(s) \leq w f_{\min}$, where $f_{\min}$ is the lowest f-value of any state in $\texttt{OPEN}$. The priority function it uses gives higher priorities to states with fewer collisions. More specifically, when generating a low-level state s for agent j, $h_{\texttt{FOCAL}}(s)$ is the number of collisions between the path from s^j to s and the paths of all agents other than j.

The high-level search of ECBS(w) is a FS with $\texttt{FOCAL}$ that contains all high-level states $N \in \texttt{OPEN}$ such that $\text{COST}_N \leq w\text{LB}$, where COST_N is the total arrival time of the paths in N and LB is the sum

of $f_{\min}$ values of all low-level searches that generated the paths in N. The priority function it uses gives higher priorities to states with fewer collisions. More specifically, when generating a high-level state N, $h_{\texttt{FOCAL}}(N)$ is the number of collisions between the paths specified in N.

4.3 Focal Search with Inflated Heuristics

In this section, we show that FS still provides suboptimality guarantees with an inflated heuristic. This will be key to providing bounded-suboptimality guarantees for our variant of ECBS that uses $h_{\texttt{HWY}}$.

Lemma 14. *FS with suboptimality bound $w_1 > 1$ that uses the f-values $f(s) = g(s) + w_2 h(s)$ for an admissible $h(s)$ and an inflation factor $w_2 > 1$ is $w_1 w_2$-suboptimal. Furthermore, the minimum f-value in its `OPEN` list is always at most w_2 times the cost of an optimal solution.*

Proof. We consider two focal searches:

1. FS^1 is a FS with suboptimality bound w_1 that uses the f-values $f(s) = g(s) + w_2 h(s)$ for an admissible $h(s)$ and an inflation factor $w_2 > 1$. Its `FOCAL` list thus is

   $$\text{FOCAL}^1 = \{s \in \texttt{OPEN} : g(s) + w_2 h(s) \leq w_1(g(s^1_{\min}) + w_2 h(s^1_{\min}))\},$$

   where $s^1_{\min} = \arg\min_{s \in \texttt{OPEN}}(g(s) + w_2 h(s))$.

2. FS^2 is a FS with suboptimality bound $w_1 w_2$ that uses the f-values $f(s) = g(s) + h(s)$. Its `FOCAL` list thus is

   $$\text{FOCAL}^2 = \{s \in \texttt{OPEN} : g(s) + h(s) \leq w_1 w_2(g(s^2_{\min}) + h(s^2_{\min}))\},$$

   where $s^2_{\min} = \arg\min_{s \in \texttt{OPEN}}(g(s) + h(s))$.

We show by induction that the OPEN lists of both searches can always be the same. Initially, both OPEN lists contain only the start state and thus are the same. Assume that they are the same at some point in time. Then, for each state $s \in \text{FOCAL}^1$, it holds that

$$
\begin{aligned}
g(s) + h(s) &\leq g(s) + w_2 h(s) \\
&\leq w_1(g(s^1_{\min}) + w_2 h(s^1_{\min})) \\
&\leq w_1(g(s^2_{\min}) + w_2 h(s^2_{\min})) \\
&\leq w_1(w_2 g(s^2_{\min}) + w_2 h(s^2_{\min})) \\
&\leq w_1 w_2 (g(s^2_{\min}) + h(s^2_{\min})).
\end{aligned}
$$

Therefore, $s \in \text{FOCAL}^2$ and, consequently, $\text{FOCAL}^1 \subseteq \text{FOCAL}^2$. Thus, if FS^1 expands s, FS^2 can be forced to expand s as well and their OPEN lists are then again the same. Thus, they eventually find the same solution. It is known that FS^2 is $w_1 w_2$-suboptimal [56]. Thus, FS^1 is $w_1 w_2$-suboptimal as well. Furthermore, it is known that the minimum f-value in the OPEN list of FS^2 is always at most the cost of the optimal solution *opt* [56]. Then, since the OPEN lists of both searches are always the same, it always holds that

$$
\begin{aligned}
\min_{s \in \text{OPEN}} (g(s) + h(s)) &\leq opt \\
w_2 \min_{s \in \text{OPEN}} (g(s) + h(s)) &\leq w_2 opt \\
\min_{s \in \text{OPEN}} (w_2 g(s) + w_2 h(s)) &\leq w_2 opt \\
\min_{s \in \text{OPEN}} (g(s) + w_2 h(s)) &\leq w_2 opt.
\end{aligned}
$$

Thus, the minimum f-value in the OPEN list of FS^1 is always at most w_2 times the cost of the optimal solution. $\square$

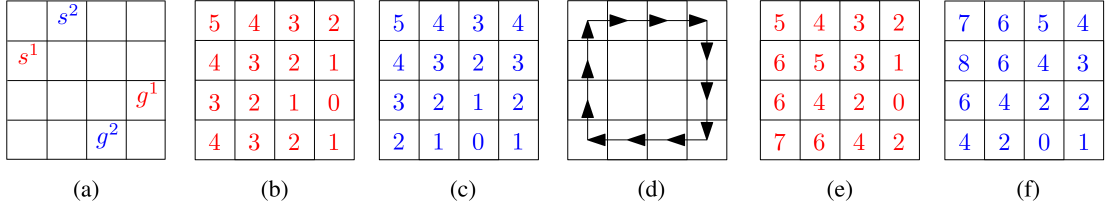


Figure 4.2: Differences between $h_{\texttt{SP}}$ and $h_{\texttt{HWY}}$ in a simple environment. (a) shows a 4×4 4-neighbor grid environment with no obstacles, and two start and goal cells for the red and blue agents. (b) and (c) show $h_{\texttt{SP}}$ values for the red and blue agents, respectively. (d) shows the highways. (e) and (f) show $h_{\texttt{HWY}}$ values for the highways in (d) and an inflation factor of $w = 2$ for the red and blue agents, respectively.

4.4 The Highways Heuristic

To a large extent, dynamically coupled MAPF solvers, such as CBS and M*, treat each agent individually. That is, a path is first computed for each agent independently and, if required, the search space of individual agents is altered to account for collisions with other agents. Thus, the ability to plan fast for individual agents is crucial. To this end, suitable per-agent heuristic guidance is utilized.

One admissible heuristic commonly used in 4-neighbor grid environments is the Manhattan distance[1]. A different admissible heuristic is the distance of each agent from its current cell to its goal cell while abstracting away all other agents. It is well defined for any graph and fast to compute.[2]

Definition 15 (Shortest path heuristic ($h_{\texttt{SP}}$))**.** *The shortest path heuristic value of state $s = (v, t)$ (that is, the agent is in cell $v \in V$ at timestep t) for agent j is*

$$h^j_{\texttt{SP}}(s) = \min_{\pi} \sum_{(u_i, u_{i+1}) \in \pi} 1,$$

where $\pi = \{v, \ldots, g^j\}$ is a feasible path that takes agent j from v to its goal cell.

It is easy to extend $h_{\texttt{SP}}$ to non unit-cost actions simply by replacing 1 with $c(u_i, u_{i+1})$.

Figures 4.2(b) and (c) show the $h_{\texttt{SP}}$ values for the MAPF problem instance depicted in Figure 4.2(a). $h^j_{\texttt{SP}}$ is a perfect estimate of the distance to agent j's goal cell, ignoring any potential interactions between

[1] The *Manhattan distance* from cell $[x_1, y_1]$ to cell $[x_2, y_2]$ is $|x_1 - x_2| + |y_1 - y_2|$.

[2] For example, given a graph $G = (V, E)$ and a goal cell, Dijkstra's algorithm computes $h_{\texttt{SP}}(v)$ in $O(|E| + |V| \log |V|)$ time for all $v \in V$ rooting the optimal path computation at the goal cell.

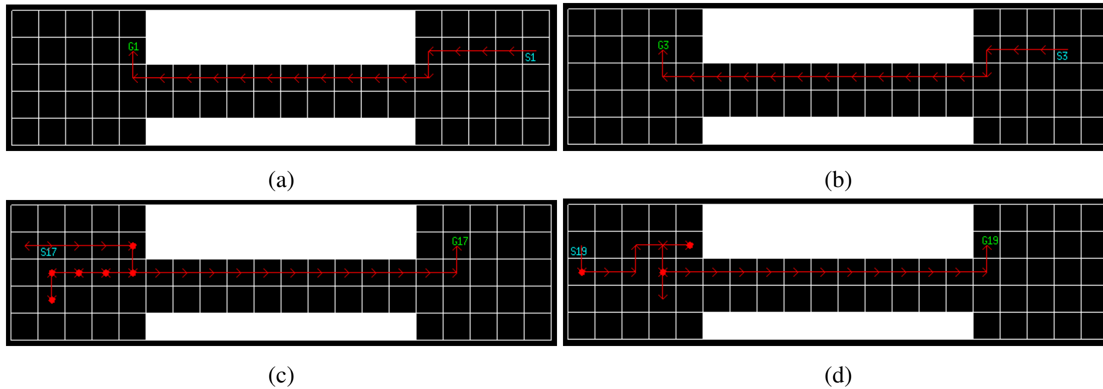


Figure 4.3: Illustrates how agents spread out in ECBS for the MAPF problem instance depicted in Figure 4.1(b). All paths shown are the ones in the high-level root state of ECBS(2). (a) and (b) show the paths that ECBS(2) computes for two agents early in the process of building the high-level root state. When these paths are computed, the reservation table is almost empty. Thus, these paths are the corresponding optimal paths from their start to their goal cells. (c) and (d) show the paths that ECBS(2) computes for two other agents later in the process of building the high-level root state. When these paths are computed, the reservation table contains many other agents' paths. Thus, these later paths meander.

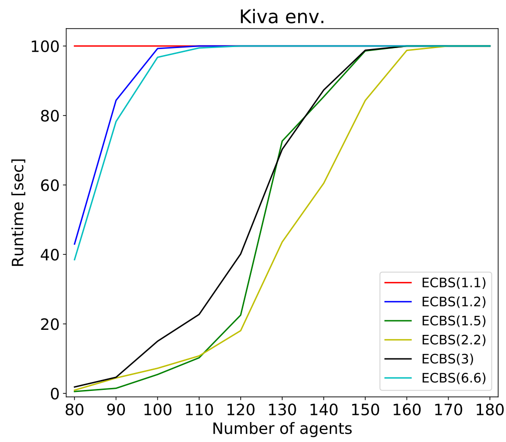
it and all other agents. Moreover, the difference between the heuristic and the optimal solution cost is due only to these interactions. Thus, h_{SP} is more informed than the Manhattan distance heuristic. However, since h_{SP} considers only the given agent's goal cell, using it in the context of MAPF can be far from ideal.

To illustrate why using h_{SP} can be inefficient in CBS, consider the MAPF problem depicted in Figure 4.1(b). Since all agents' start and goal cells are in the upper two rows, the h_{SP} values in the upper row of the corridor are smaller than the h_{SP} values in the bottom row of the corridor (for cells that are in the same column). Thus, each agent's low-level search would first choose paths that take the upper row of the corridor, which in turn causes many head-on collisions in the upper row of the corridor. The high-level search has to post numerous constraints to resolve these collisions, therefore making CBS inefficient. Moreover, head-on collisions are harder to resolve than other collisions for search-based planners because they cannot be resolved merely by a wait action but instead require planning a path for one of the agents around the other agent. As expected, CBS fails to find a solution for this problem within a 100 second time limit because its runtime can be exponential in the number of collisions the high-level search needs to resolve.

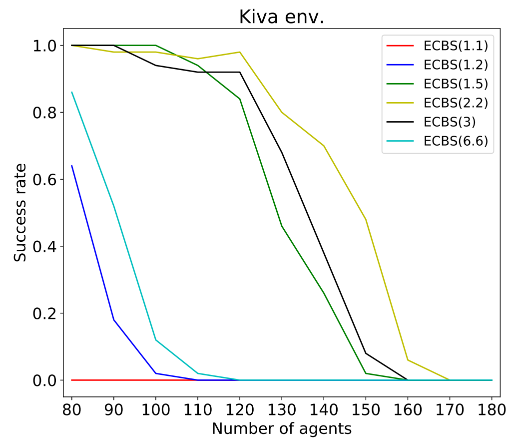
Interestingly, using $h_{\texttt{SP}}$ with ECBS can be inefficient, too. In ECBS, increasing w allows for longer paths, which provides agents with the flexibility to avoid collisions by waiting in place or moving around other agents. The high-level search then has to resolve fewer collisions and might terminate earlier. However, this also makes the agents spread out more. For example, Figure 4.3 illustrates how agents spread out for the MAPF problem depicted in Figure 4.1(b), resulting in meandering paths. When many agents are huddled together, meandering paths can result in many additional collisions, which makes ECBS inefficient. Consequently, larger values of w are not necessarily beneficial, as is evident in the experimental results we discuss next.

Figure 4.4 reports the average runtimes and success rates of ECBS with suboptimality bounds $w = \{1.1, 1.2, 1.5, 2.2, 3, 6.6\}$ for three domains, namely, Kiva, Drones and Roundabout (the first two are described in Chapter 3.5.1 and the third is described in Chapter 4.5.2). The experimental setup is identical to the one described in Chapter 3.5.1. We do not measure runtime differently when a MAPF solver fails to solve a MAPF problem instance within the 100 second time limit (that is, its runtime for this MAPF problem instance is 100 seconds). In the Kiva environment (upper row of Figure 4.4), agents navigate through long narrow corridors. Here, increasing w up to 2.2 results in higher efficiency and success rates. However, using $w = 3$ and $w = 6.6$ results in lower efficiency, closer to $w = 1.5$ and $w = 1.2$, respectively. In the Drones environment, agents are free to utilize the large open space in the upper part of the environment. Here, increasing w generally results in higher efficiency, and both $w = 3$ and $w = 6.6$ achieve a 100% success rate. In the Roundabout environment, agents navigate between four rooms through a roundabout. Here, increasing w up to 3 results in higher efficiency and success rates. However, using $w = 6.6$ results in lower efficiency for up to 220 agents. Finally, with 240 agents huddled together in this environment, $w = 6.6$ results in the highest efficiency.

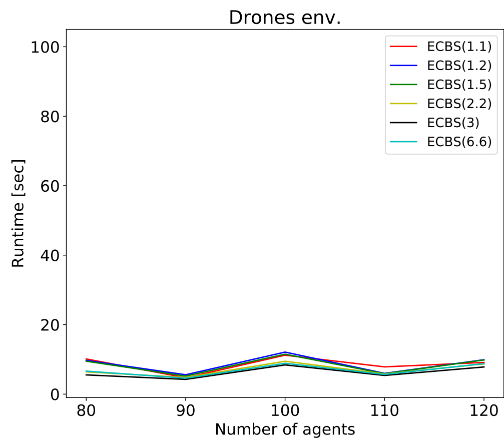
ECBS's failure to efficiently use the additional leeway that comes with higher suboptimality bounds in structured environments prompts us to develop a new heuristic, dubbed $h_{\texttt{HWY}}$. $h_{\texttt{HWY}}$ is defined with respect to a set of edges in graph G, called *highways*. $h_{\texttt{HWY}}$ inflates heuristic values non-uniformly in a way that encourages the low-level searches to return paths that include the highways' edges. Similar to



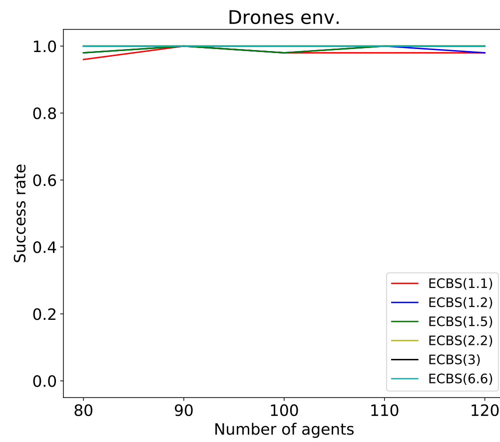
(a)



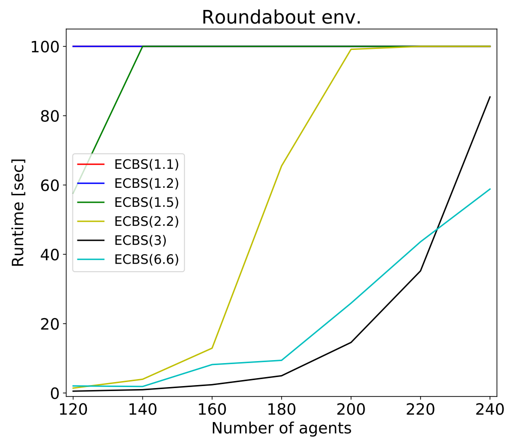
(b)



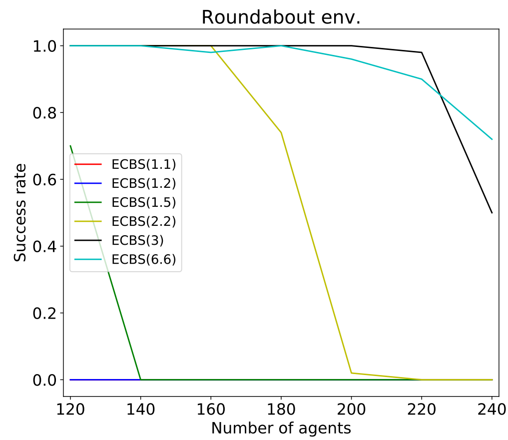
(c)



(d)



(e)



(f)

Figure 4.4: Average runtimes (left column) and success rates (right column) of ECBS with suboptimality bounds $w = \{1.1, 1.2, 1.5, 2.2, 3, 6.6\}$ for the Kiva, Drones and Roundabout environments.

the motivation of using highways in transportation, the motivation behind $h_{\texttt{HWY}}$ is to encourage a global behavior of the agents that reduces the number of collisions and, therefore, increases ECBS's efficiency. We now define highways and $h_{\texttt{HWY}}$ formally.

Definition 16 (Highways)**.** *Given a graph $G = (V, E)$,* highways *are a subset of the edges $E_{\texttt{HWY}} \subseteq E$.*

Definition 17 (Highway heuristic ($h_{\texttt{HWY}}$))**.** *Given a user provided inflation factor $w \geq 1$ and highways $E_{\texttt{HWY}}$, the highway heuristic value of state $s = (v, t)$ (that is, the agent is in cell $v \in V$ at timestep t) for agent j is*

$$h^j_{\texttt{HWY}}(s) = \min_{\pi} \sum_{(u_i, u_{i+1}) \in \pi} \begin{cases} 1 & \textit{if } (u_i, u_{i+1}) \in E_{\texttt{HWY}} \\ w & \textit{otherwise,} \end{cases}$$

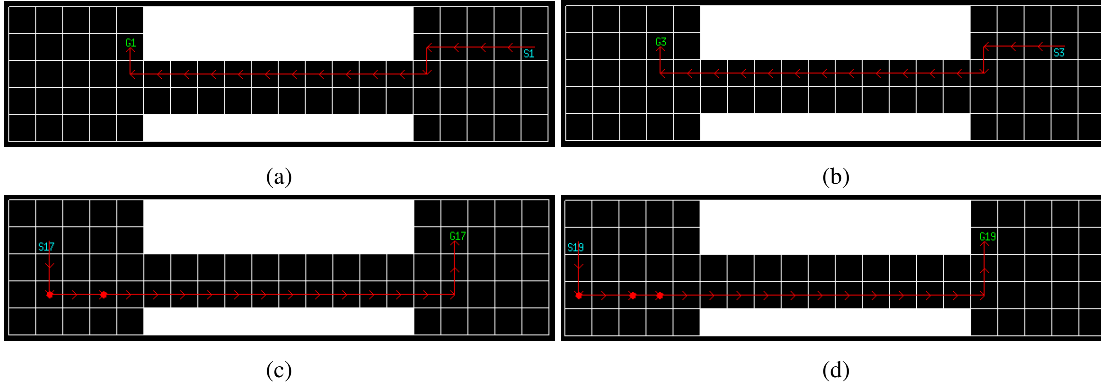
where $\pi = \{v, \ldots, g^j\}$ is a feasible path that takes agent j from v to its goal cell.

It is easy to extend $h_{\texttt{HWY}}$ to non unit-cost actions simply by replacing 1 with $c(u_i, u_{i+1})$ if $(u_i, u_{i+1}) \in E_{\texttt{HWY}}$ and replacing w with $wc(u_i, u_{i+1})$ otherwise.

Intuitively, the inflation factor $w > 1$ determines the level of encouragement for low-level searches to return paths that include the highways' edges. Figure 4.2(e) and (f) show $h_{\texttt{HWY}}$ values for the example in Figure 4.2(a) for $E_{\texttt{HWY}}$ given in Figure 4.2(d) with an inflation factor of $w = 2$. For instance, the $h_{\texttt{SP}}$ value for agent 1 and the cell in row 2 and column 3 is 2 (corresponding to moving east and south, each with cost one) while the $h_{\texttt{HWY}}$ value is 3 (corresponding to moving east with cost two and then following the highway south with cost one).[3]

Finally, Figure 4.5 illustrates how agents both spread out less as well as utilize the bottom part of the corridor when $h_{\texttt{HWY}}$ is used instead of $h_{\texttt{SP}}$ for the MAPF problem instance depicted in Figure 4.1(b). Using $h_{\texttt{HWY}}$ potentially reduces the number of collisions and thus can increase ECBS's efficiency. We will provide experimental evidence that this is indeed the case in Section 4.5.2, but first define the w-admissible

[3] We count rows from top to bottom and columns from left to right.



(a) (b)

(c) (d)

Figure 4.5: Illustrates the effect of using $h_{\texttt{HWY}}$ instead of $h_{\texttt{SP}}$ for the MAPF problem instance depicted in Figure 4.1(b). Like in Figure 4.3, it shows the paths computed by ECBS(2) in its high-level root state for the first two agents ((a) and (b)) and last two agents ((c) and (d)) it plans for. Instead of $h_{\texttt{SP}}$, however, ECBS(2) uses $h_{\texttt{HWY}}$ for the highway specified in Figure 4.1(b) and an inflation factor of 2. While the paths in (a) and (b) are similar to the ones of ECBS(2) with $h_{\texttt{SP}}$, the paths in (c) and (d) are less meandering and, unlike in Figure 4.3(c) and (d), utilize the lower row of the corridor.

property of a heuristic that $h_{\texttt{HWY}}$ satisfies. This property will be used in the next section to prove that our MAPF solvers are bounded-suboptimal.

Definition 18 (w-admissible heuristic)**.** *A heuristic h is w-*admissible *iff $h(s) \leq wopt(s)$ for every state s, where $opt(s)$ is the cost of an optimal path from s to any goal state.*

Proposition 19. *For all agents j, $h^{j}_{\texttt{HWY}}$ is w-admissible.*

4.5 Conflict-Based Search with Highways

Our first bounded-suboptimal MAPF solver, called CBS+HWY(w), is a version of CBS whose low-level searches use $h_{\texttt{HWY}}$ with inflation factor w instead of $h_{\texttt{SP}}$. Figure 4.6 shows CBS+HWY(2)'s high-level tree for the example in Figure 4.2(a). On the positive side, using $h_{\texttt{HWY}}$ affects the optimal paths found in the high-level root state such that there are no collisions, thus finding a solution faster by expanding only one high-level state. On the negative side, the cost of the solution found is 12, which is larger than the cost of the optimal solution. Nevertheless, the following theorem shows that CBS+HWY's solution cost cannot deteriorate too much.

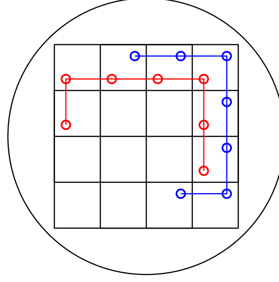


Figure 4.6: Illustrates the high-level tree of CBS+HWY(2) for the example in Figure 4.2(a).

We use the following notation throughout this chapter: OPT is the cost of an optimal solution of the MAPF problem. OPT_N^j is the cost of an optimal path for agent j that respects the constraints of high-level state N. COST_N^j is the cost of the path found by the low-level search. Let $\text{OPT}_N = \sum_{j=1}^{K} \text{OPT}_N^j$ and $\text{COST}_N = \sum_{j=1}^{K} \text{COST}_N^j$. LB_N^j is the minimum f-value in the OPEN list of the low-level search for agent j after it terminates when high-level state N is generated. Let $\text{LB}_N = \sum_{j=1}^{K} \text{LB}_N^j$ and $\text{LB} = \min_{N \in \texttt{OPEN}} \text{LB}_N$ for the OPEN list of the high-level search.

Theorem 20. *CBS+HWY(w) is w-suboptimal.*

Proof. Let M be any high-level state in the OPEN list of the high-level search of CBS+HWY(w) that contains an optimal solution of the MAPF problem (that is, $\text{OPT}_M \leq \text{OPT}$). Such an M exists according to Lemma 2 of [69]. The low-level searches of CBS+HWY(w) use w-admissible heuristic values according to Proposition 19. The low-level search for agent j when high-level state M is generated thus is guaranteed to find a path with a cost of at most $w\text{OPT}_M^j$, that is, $\text{COST}_M^j \leq w\text{OPT}_M^j$. The proof is identical to the one that proves the suboptimality guarantee of Weighted-A* [61]. Consequently, $\text{COST}_M \leq w\text{OPT}_M \leq w\text{OPT}$. The high-level search of CBS+HWY(w) is a best-first search and can thus never expand a high-level goal state with a cost of more than wOPT. Thus, CBS+HWY(w) is w-suboptimal. □

CBS+HWY(2) and ECBS(2) are both 2-suboptimal and thus suitable for comparison. We begin with an anecdotal comparison between the two MAPF solvers on the previous examples, namely, (i) the MAPF problem instance depicted in Figure 4.2(a), in which the environment is an open space; and (ii) the MAPF problem instance depicted in Figure 4.1(b), in which the environment contains a long narrow corridor. On

(i), CBS+HWY(2) finds a solution of cost 12 after expanding just one high-level state because there are no collisions in the root state. ECBS(2) finds a solution of cost 9 after expanding just one high-level state because a single wait action by one of the two agents prevents their paths from colliding. This example illustrates an advantage of ECBS, whose low-level searches' strength seems to be in reducing collisions either with a short sequence of wait actions or with a short detour circumnavigating other agents in open spaces. On (ii), however, CBS+HWY(2) finds a solution of cost 423 after expanding 27 high-level states while ECBS(2) finds a solution of cost 521 after expanding 67 high-level states.[4] This example illustrates the advantage of CBS+HWY, whose strength seems to be in reducing the number of head-on collisions in structured environments by agents following the highways. Following the highways further reduces collisions by encouraging agents to choose paths that meander less.

Although CBS+HWY(2) is more efficient and effective than ECBS(2) on the corridor example in Figure 4.1(b), CBS+HWY(2) fails to solve all MAPF problem instances in the Kiva environment (with 80 or more agents) using the highways in Figure 4.8(a) within 100 seconds. While $h_{\texttt{HWY}}$ guides the agents well through the corridors, each agent still has to move from its start cell in one open space into a corridor, and move out of a corridor to its goal cell in the other open space. In order to efficiently solve MAPF problem instances with many agents in environments like the Kiva environment, a MAPF solver capable of doing both is required. We develop such a MAPF solver in the next section.

4.5.1 Enhanced Conflict-Based Search with Highways

In this section, we develop ECBS(w_1)+HWY(w_2), a MAPF solver that combines ECBS with $h_{\texttt{HWY}}$. More specifically, ECBS(w_1)+HWY(w_2) is similar to ECBS(w_1) but instead of using $h_{\texttt{SP}}$ it uses $h_{\texttt{HWY}}$ with inflation factor w_2. We first prove that ECBS(w_1)+HWY(w_2) is $w_1 w_2$-suboptimal and then show experimentally that it is more efficient than ECBS(w) in three different environments.

Theorem 21. *ECBS(w_1)+HWY(w_2) is $w_1 w_2$-suboptimal.*

[4] CBS did not terminate within 300 seconds after expanding 1,950,616 high-level states. In fact, ECBS(1.2) also did not terminate within 300 seconds after expanding 889,812 high-level states.

Proof. Let M be any high-level state in the `OPEN` list of the high-level search of ECBS(w_1)+HWY(w_2) that contains an optimal solution of the MAPF problem (that is, $\text{OPT}_M \leq \text{OPT}$). Such an M exists analogously to Lemma 2 of [69]. Moreover, Lemma 14 asserts that the low-level searches of ECBS(w_1)+HWY(w_2) are w_1w_2-suboptimal and that $\text{LB}_N^j \leq w_2\text{OPT}_N^j$ for all high-level states N and agents j. Thus,

$$\text{LB} \leq \text{LB}_M = \sum_{j=1}^{K} \text{LB}_M^j \leq \sum_{j=1}^{K} w_2\text{OPT}_M^j = w_2\text{OPT}_M \leq w_2\text{OPT}.$$

The high-level search of ECBS(w_1)+HWY(w_2) is a focal search with `FOCAL` containing all states $N \in$ `OPEN` such that $\text{COST}_N \leq w_1\text{LB}$. Thus, it always expands a state whose cost is at most $w_1\text{LB} \leq w_1w_2\text{OPT}$. Consequently, the high-level search can never expand a high-level goal state with a cost of more than w_1w_2OPT, and ECBS(w_1)+HWY(w_2) is w_1w_2-suboptimal. □

4.5.2 Experiments

In this section, we describe experiments performed to evaluate our bounded-suboptimal MAPF solvers and show that using $h_{\texttt{HWY}}$ is beneficial in both the CBS and M* frameworks. The experimental setup is identical to the one described in Chapter 3.5.1. We conduct experiments in three environments, each having a different structure. The first two environments, Kiva and Drones, are described in Chapter 3.5.1. The Roundabout environment is depicted in Figure 4.7. It is a 24x48 four-neighbor grid environment with obstacles placed so that there are four open spaces, each of size 11x17, that are connected by a central roundabout. We generate MAPF problem instances as follows. For each number of agents between 120 and 240 in increments of 20, we randomly generate 50 MAPF problem instances. Each MAPF problem instance is generated by randomly assigning start cells to a quarter of the agents in each of the four open spaces. For each agent, a randomly assigned goal cell is then chosen in the open space diagonally opposite of the open space of its start cell. We generate MAPF problem instances such that no two agents share the

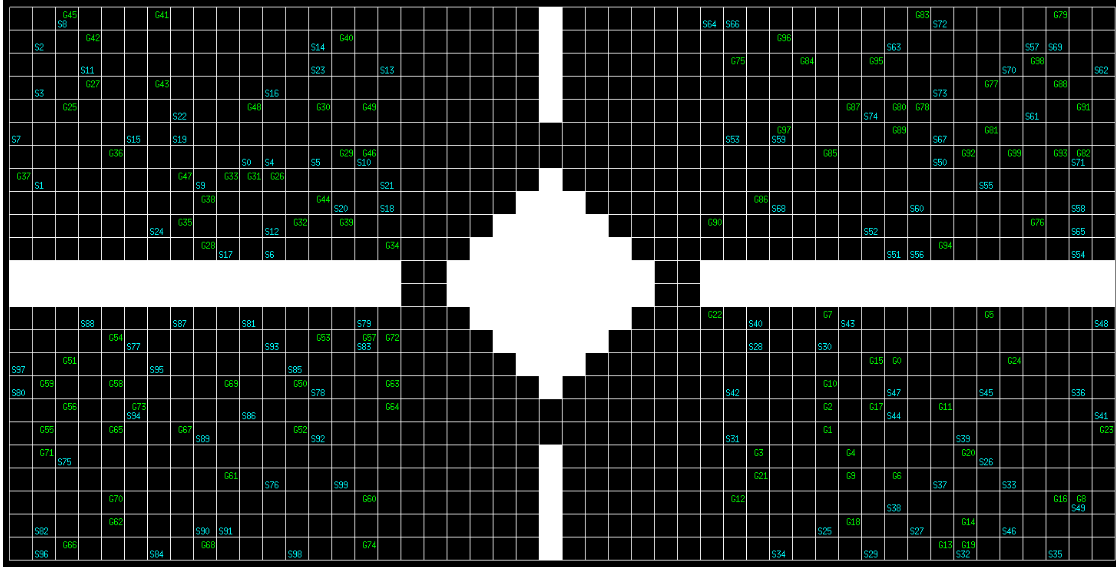


Figure 4.7: Shows the Roundabout environment. The start and goal cells of all agents in a MAPF problem instance with 100 agents are shown in cyan and green, respectively.

same start cell or the same goal cell. Figure 4.7 also shows the start and goal cells of a MAPF problem instance with 100 agents.

Figures 4.8(a), (b) and (c) show the edges of the human-specified highways for the Kiva, Drones and Roundabout environments, respectively, that we use in our experiments. In the remainder of this subsection, we discuss the following items for all three environments:

1. We show that ECBS(w_1)+HWY(w_2) (which is $w_1 w_2$-suboptimal) is more efficient than ECBS(w_1) (which is w_1-suboptimal) for different values of w_1 and w_2.

2. We compare ECBS(w_1)+HWY(w_2) with ECBS(w) such that $w_1 w_2 = w$ (that is, both are w-suboptimal) for different values of w, w_1 and w_2, and discuss the reasons for why one is more efficient than the other in different situations.

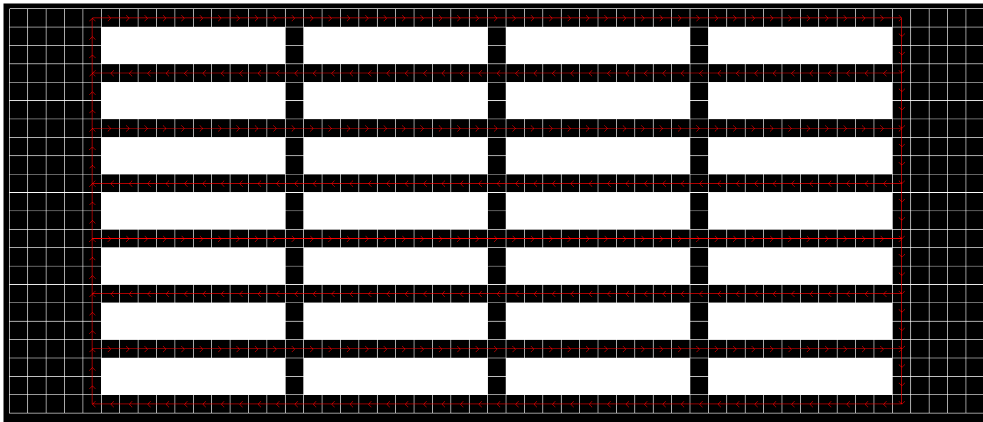
3. We show that, among all values for w, w_1 and w_2 that we use in our experiments, the most efficient[5] ECBS(w_1)+HWY(w_2) is more efficient than the most efficient ECBS(w). We back this claim up with a very strong statistical evidence supported by the (non-parametric) permutation test.

[5] most efficient is determined by highest success rate, breaking ties in favor of shorter average runtime

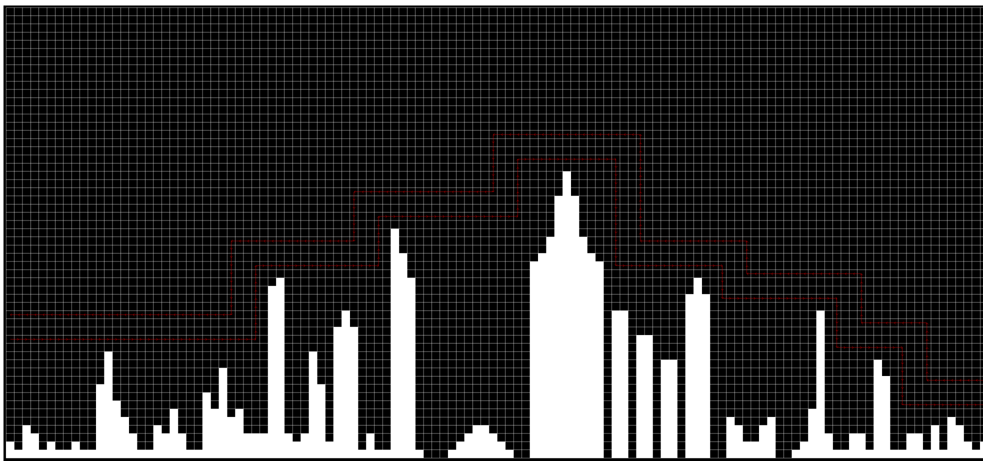
4. Finally, we show that, among all values for w, w_1 and w_2 that we use in our experiments, the most efficient ECBS(w_1)+HWY(w_2) is more effective than the most efficient ECBS(w) even though $w_1 w_2 > w$.

The first set of experiments compares ECBS(w_1)+HWY(w_2) with ECBS(w_1) for $w_1 \in \{1.1, 1.2, 1.5\}$ and $w_2 \in \{2, 3, 6\}$. Figures 4.9, 4.10 and 4.11 (for the Kiva, Drones and Roundabout environment, respectively) show that ECBS(w_1)+HWY(w_2) is more efficient (left columns) and, as a result, has higher success rates (right columns) than ECBS(w_1) for all w_1 and w_2. Thus, $h_{\texttt{HWY}}$ (with any w_2) results in higher efficiency compared to $h_{\texttt{SP}}$ in all three environments. We now discuss each environment in detail.

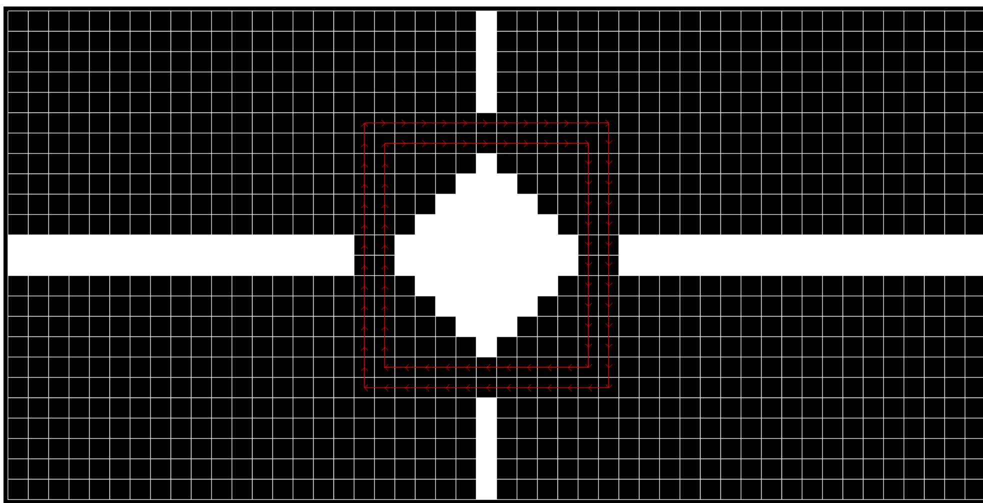
The Kiva environment requires significant coordination between the agents as they need to travel through narrow passageways to reach the opposite sides of the environment. Increasing the value of w_1 from 1.1 to 1.2 and from 1.2 to 1.5 (red curves in Figures 4.9 (a), (c) and (e), respectively) makes ECBS(w_1) more efficient, mainly because its low-level searches have more freedom to circumnavigate other agents (but not excessive freedom to meander too much). For all w_1 and w_2, using $h_{\texttt{HWY}}$ significantly improves the efficiency. For example, with 80 agents, ECBS(1.1) has an average runtime of 100 seconds and a success rate of 0% while ECBS(1.1)+HWY(2) has an average runtime of 1.07 seconds and a success rate of 100%. Furthermore, with 100 agents, ECBS(1.2) has an average runtime of 99.27 seconds and a success rate of 2% while ECBS(1.2)+HWY(2) has an average runtime of 0.93 seconds and a success rate of 100%. Finally, with 150 agents, ECBS(1.5) has an average runtime of 98.59 seconds and a success rate of 2% while ECBS(1.5)+HWY(3) has an average runtime of 57.21 seconds and a success rate of 62%. As the number of agents gets larger (for example, for 180 or more agents), the open spaces become heavily populated with agents and both ECBS(w_1) and ECBS(w_1)+HWY(w_2) fail to solve any MAPF problem instance within 100 seconds for all combinations of w_1 and w_2. We also observe that, on average, $w_2 = 3$ is most efficient in most cases. While encouraging the agents to follow the highways improves the efficiency overall, we also want them to be able to move off the highways when doing so helps to reduce the number of collisions with other agents. Thus, $w_2 = 6$ often diminish efficiency compared to $w_2 = 3$,



(a)



(b)



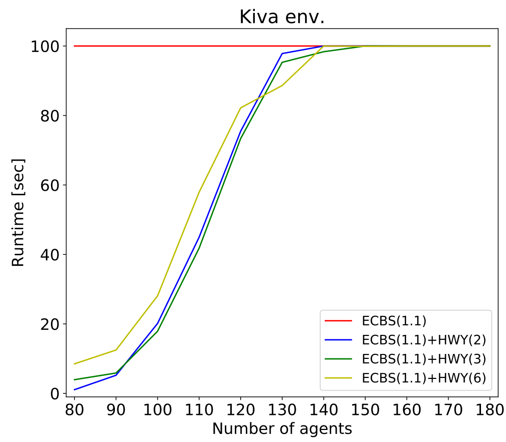
(c)

Figure 4.8: Shows the human-specified highways for the Kiva, Drones and Roundabout environments in (a), (b) and (c), respectively.

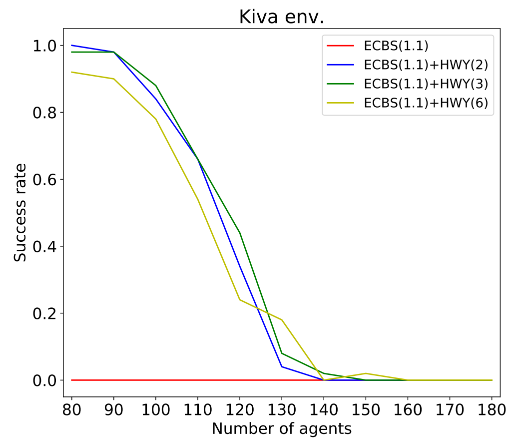
as is the case with ECBS(1.1)+HWY(6) for any number of agents for any number of agents below 130, ECBS(1.2)+HWY(6) for any number of agents and ECBS(1.5)+HWY(6) for any number of agents below 160.

Unlike the Kiva environment, the Drones environment does not have long narrow corridors but has a large open space above the city's skyline that an agent's low-level search can utilize to circumnavigate the other agents. Thus, significant coordination among the agents is not required, and ECBS is therefore quite efficient. This can be seen by the high success rates for all numbers of agents and w_1 in Figures 4.10 (b), (d) and (f). Nevertheless, for all w_1 and w_2, using $h_{\texttt{HWY}}$ significantly improves efficiency. For example, with 120 agents, ECBS(1.5) has an average runtime of 9.83 seconds and a success rate of 100% while ECBS(1.5)+HWY(6) has an average runtime of 0.53 seconds and a success rate of 100%.

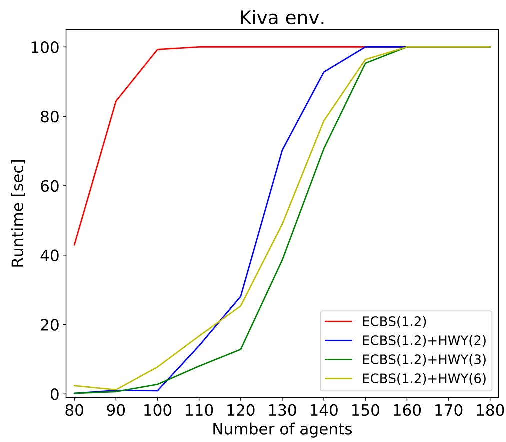
In the Roundabout environment, agents travel between diagonally opposite open spaces through a central roundabout. While it does not require significant coordination among the agents like in the Kiva environment, it does require more coordination between the agents than in the Drones environment. ECBS(1.1) and ECBS(1.2) fail to solve any MAPF problem instance within 100 seconds, and ECBS(1.5) fails to solve any MAPF problem instance with 140 or more agents within 100 seconds (red curves in Figures 4.11 (b), (d) and (f)). For all w_1 and w_2, using $h_{\texttt{HWY}}$ significantly improves efficiency. For example, with 120 agents, ECBS(1.1) has an average runtime of 100 seconds and a success rate of 0% while ECBS(1.1)+HWY(6) has an average runtime of 0.27 seconds and a success rate of 100%. Furthermore, with 180 agents, ECBS(1.2) has an average runtime of 100 seconds and a success rate of 0% while ECBS(1.2)+HWY(6) has an average runtime of 2.87 seconds and a success rate of 100%. Finally, with 240 agents, ECBS(1.5) has an average runtime of 100 seconds and a success rate of 0% while ECBS(1.5)+HWY(6) has an average runtime of 24.50 seconds and a success rate of 100%.



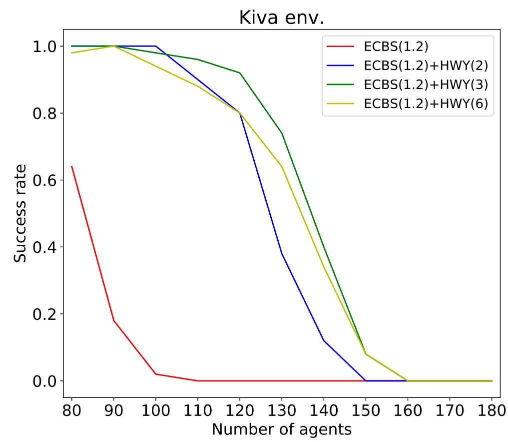
(a)



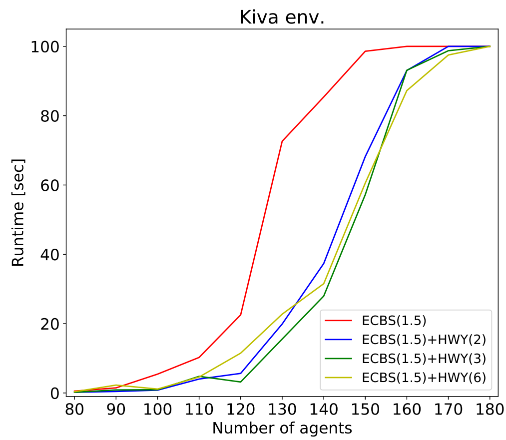
(b)



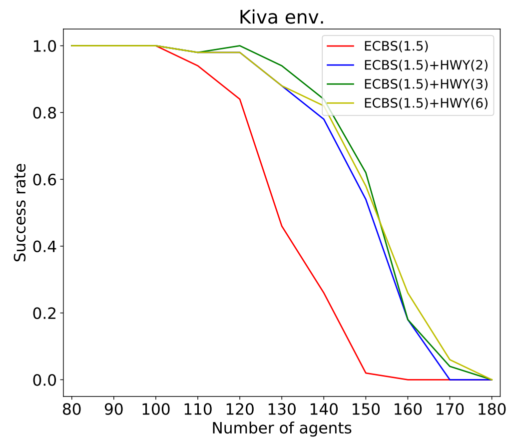
(c)



(d)

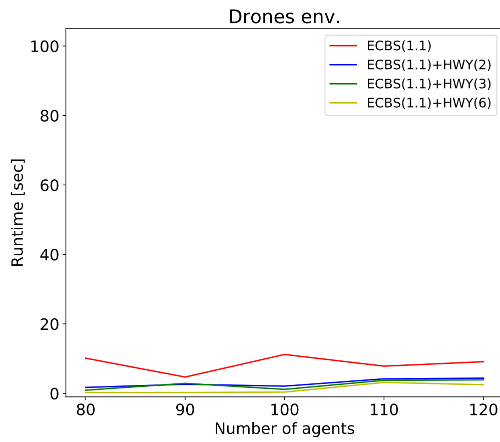


(e)

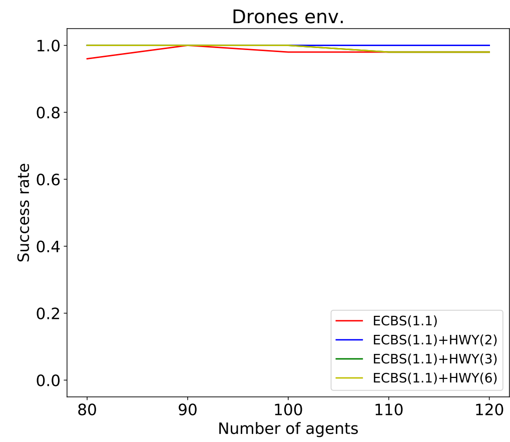


(f)

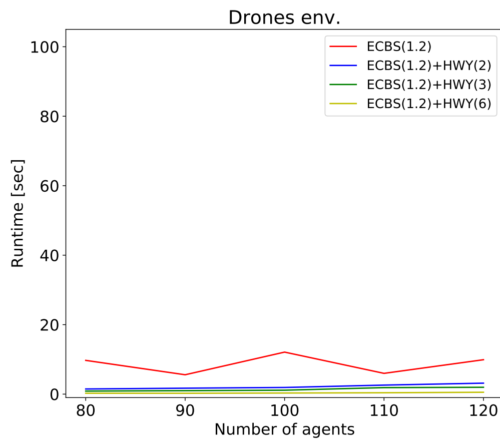
Figure 4.9: Shows the average runtimes and success rates of ECBS(1.1)+HWY, ECBS(1.2)+HWY and ECBS(1.5)+HWY for different inflation factors in the Kiva environment.



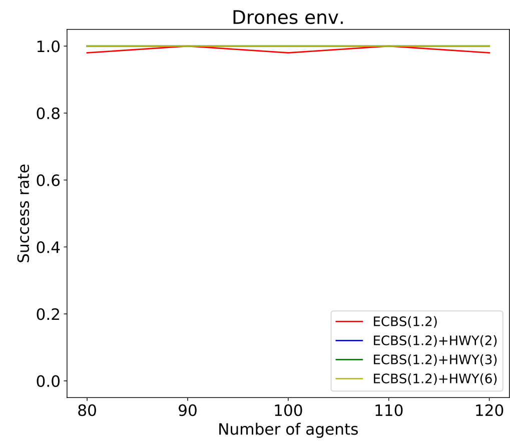
(a)



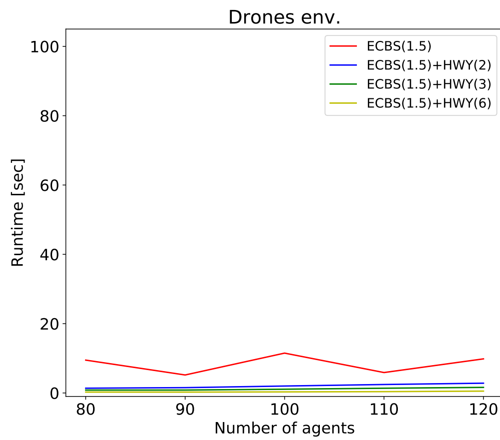
(b)



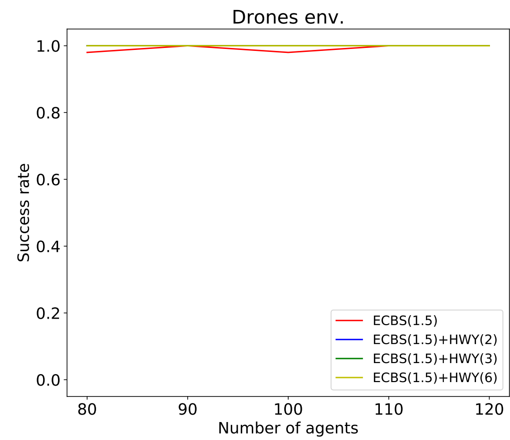
(c)



(d)

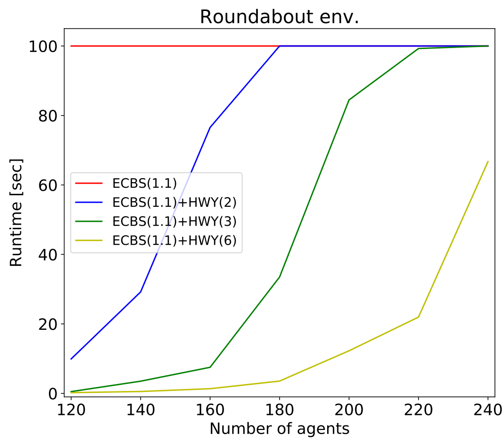


(e)

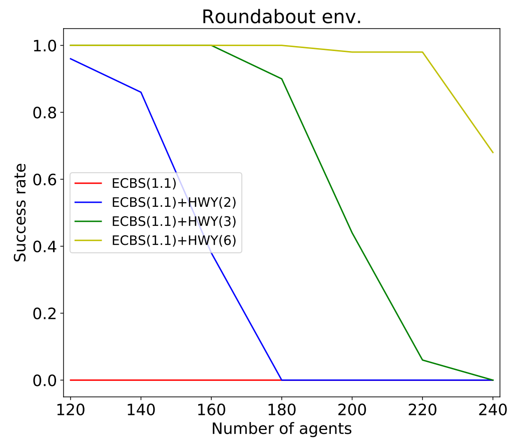


(f)

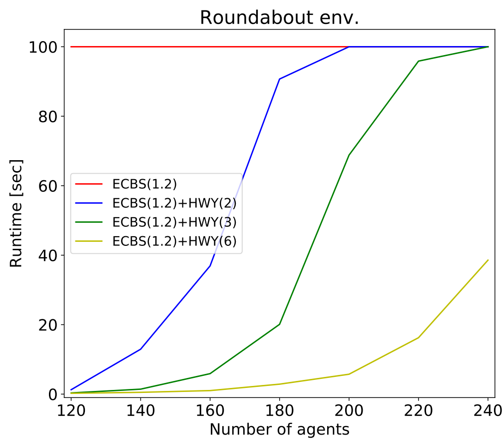
Figure 4.10: Shows the average runtimes and success rates of ECBS(1.1)+HWY, ECBS(1.2)+HWY and ECBS(1.5)+HWY for different inflation factors in the Drones environment.



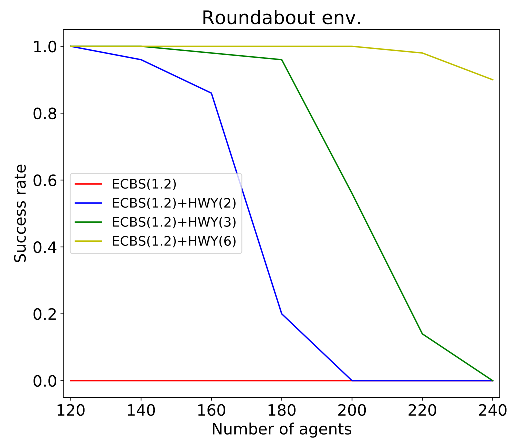
(a)



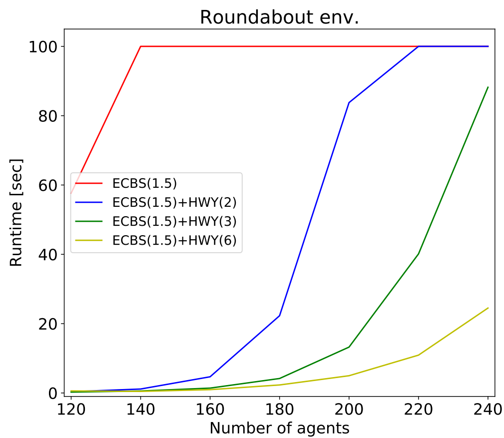
(b)



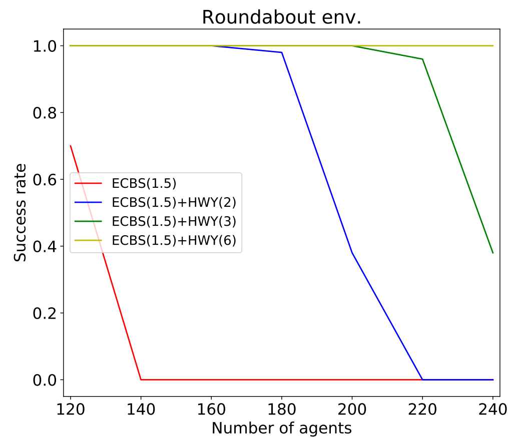
(c)



(d)



(e)



(f)

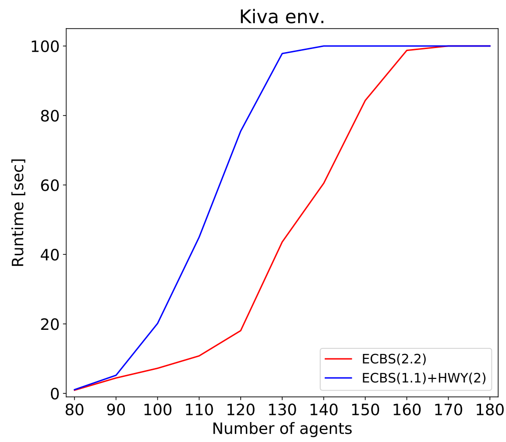
Figure 4.11: Shows the average runtimes and success rates of ECBS(1.1)+HWY, ECBS(1.2)+HWY and ECBS(1.5)+HWY for different inflation factors in the Roundabout environment.

The second set of experiments compares ECBS(w_1)+HWY(w_2) with ECBS(w) such that both MAPF solvers are w-suboptimal, that is, $w = w_1 w_2$. Figures 4.12, 4.13 and 4.14 (for the Kiva, Drones and Roundabout environment, respectively) show the average runtimes (left columns) and success rates (right columns) for: ECBS(1.1)+HWY(2) and ECBS(2.2) (upper rows); ECBS(1.5)+HWY(2) and ECBS(3) (middle rows); and 3) ECBS(1.1)+HWY(6), ECBS(2.2)+HWY(3) and ECBS(6.6) (bottom rows). We now discuss each suboptimality bound in detail.

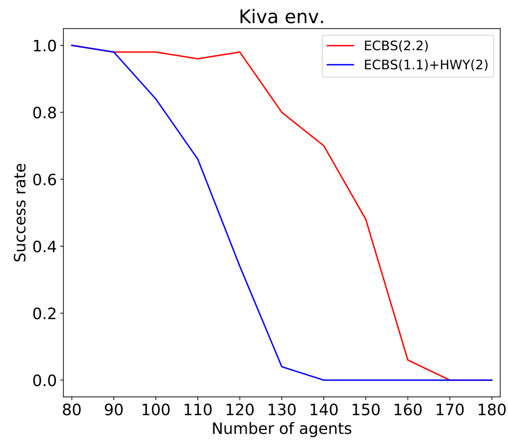
For the 2.2-suboptimal MAPF solvers, ECBS(1.1)+HWY(2) is more efficient than ECBS(2.2) in the Drones environment. However, this is not the case in the Kiva and Roundabout environments. Although the highways provide useful guidance in the Kiva and Roundabout environments, the focal suboptimality bound $w_1 = 1.1$ does not provide enough flexibility for the low-level searches to circumnavigate the other agents in open spaces. In the Kiva environment, this flexibility is helpful for agents navigating from their start cells into corridors, and out of corridors to their goal cells. Similarly, in the Roundabout environment, this flexibility is helpful for agents navigating from their start cells into the roundabout, and out of the roundabout to their goal cells.

For the 3-suboptimal MAPF solvers, ECBS(1.5)+HWY(2) is more efficient than ECBS(3) in the Drones and Kiva environments. However, this is not the case in the Roundabout environment. The usage of highways in the Roundabout environment is less crucial for solving MAPF problems efficiently than in the Kiva environment. Thus, the additional flexibility for agents to circumnavigate each other in open spaces that $w = 3$ provides turns out to be more helpful for efficiency than the flexibility provided by $w_1 = 1.5$ and $w_2 = 2$.

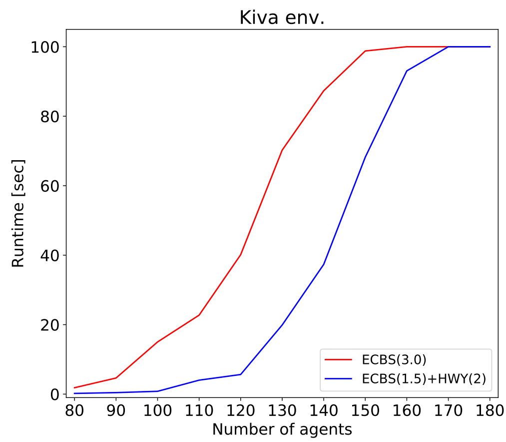
For the 6.6-suboptimal MAPF solvers, ECBS(2.2)+HWY(3) is more efficient than both ECBS(1.1)+HWY(6) and ECBS(6.6) in all environments. In the Kiva and Roundabout environments, in particular, $w_1 = 2.2$ provides enough flexibility for agents to circumnavigate each other in open spaces, and $w_2 = 3$ provides enough encouragement to follow the highways. This combination results in improved efficiency and significantly higher success rates for higher numbers of agents.



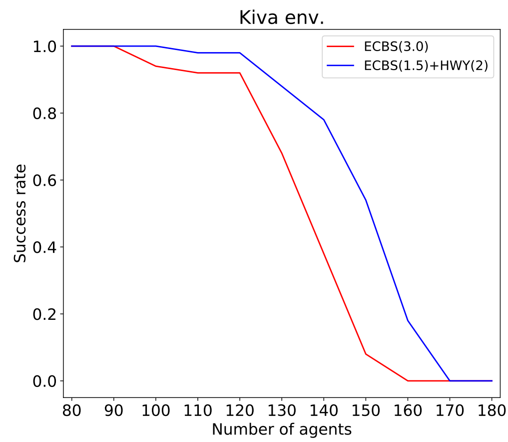
(a)



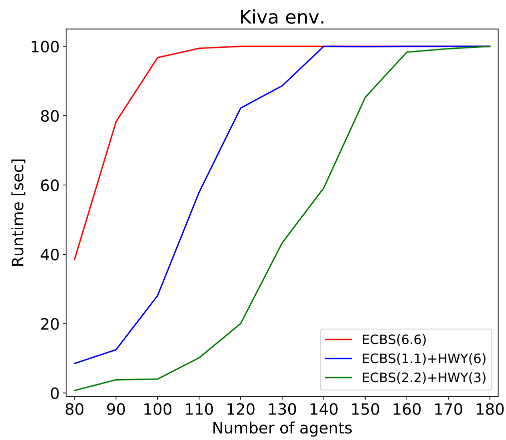
(b)



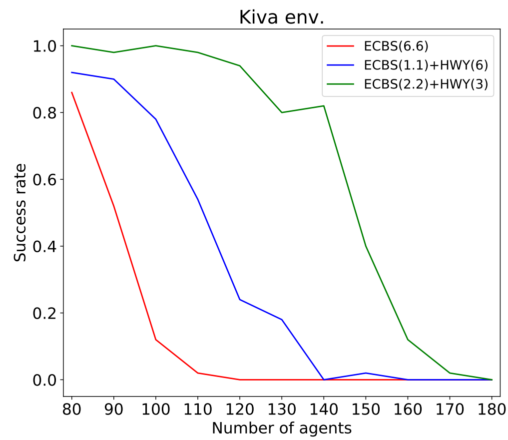
(c)



(d)

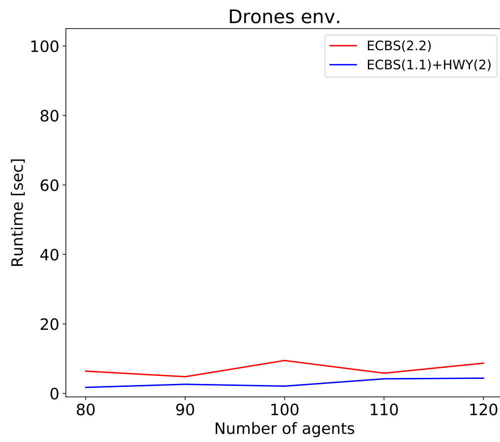


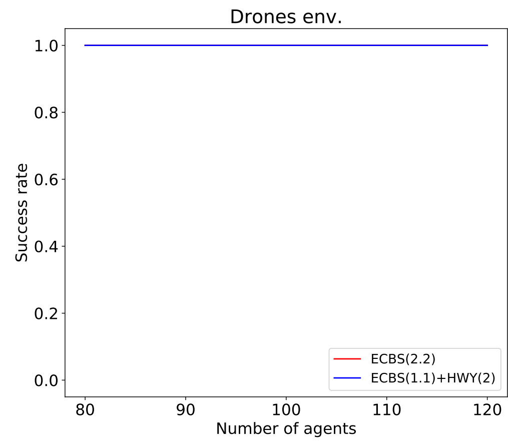
(e)



(f)

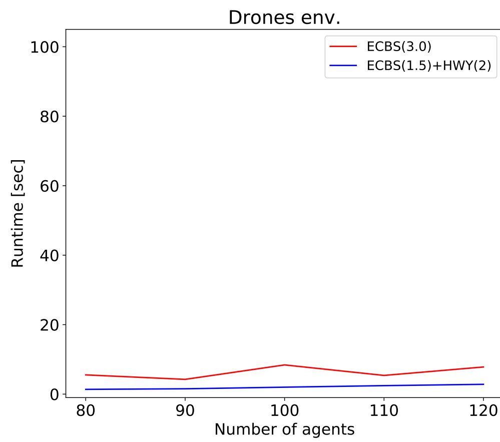
Figure 4.12: Shows the average runtimes and success rates of 2.2-, 3.0- and 6.6-suboptimal MAPF solvers in the Kiva environment.

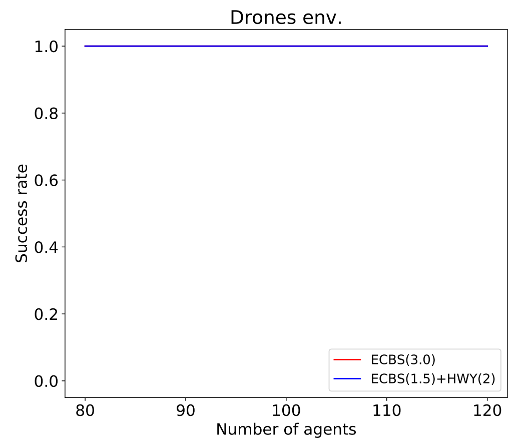




(a)

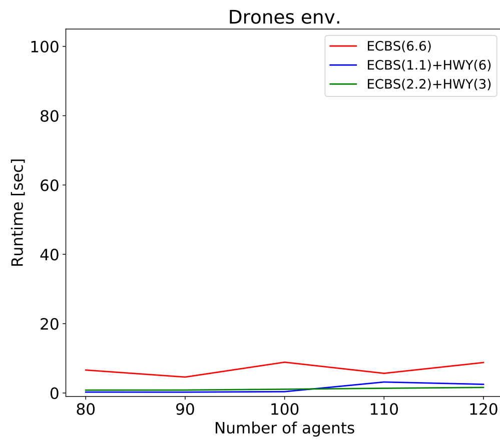
(b)

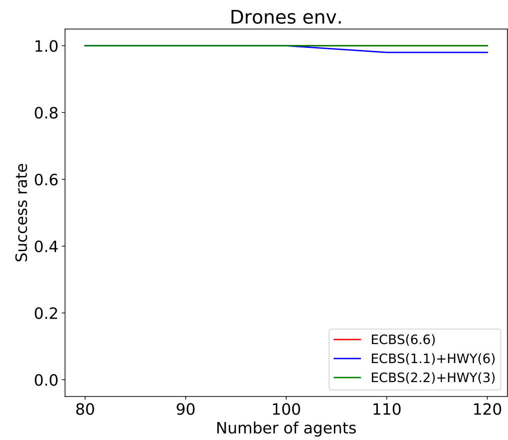




(c)

(d)

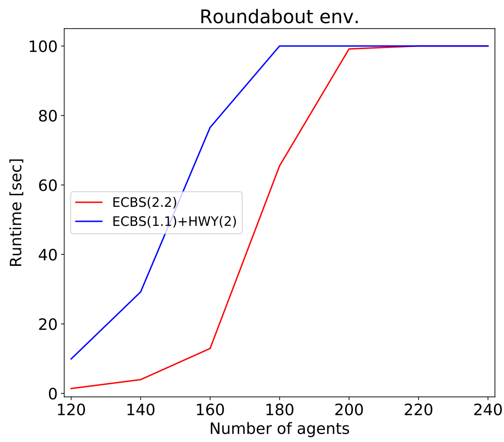




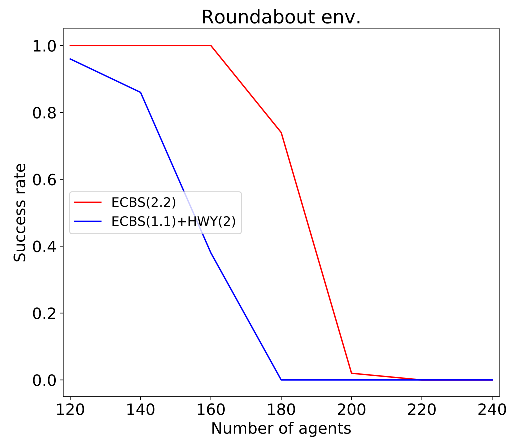
(e)

(f)

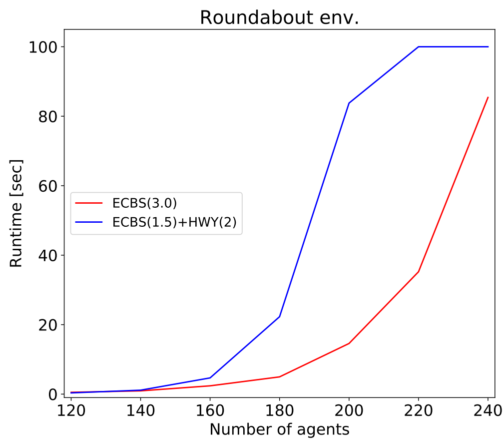
Figure 4.13: Shows the average runtimes and success rates of 2.2-, 3.0- and 6.6-suboptimal MAPF solvers in the Drones environment.



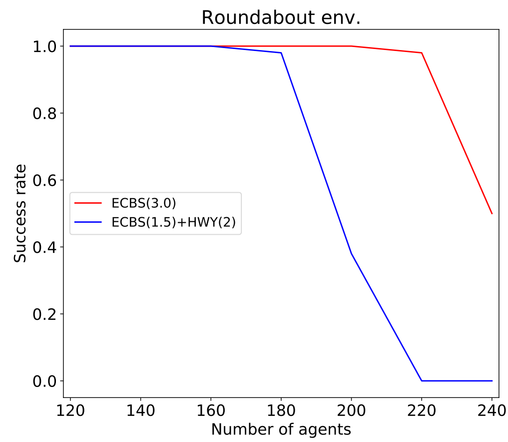
(a)



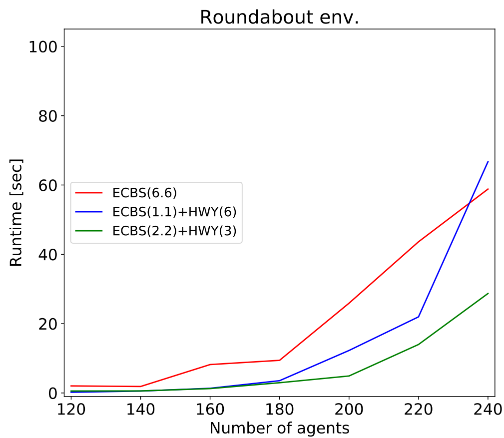
(b)



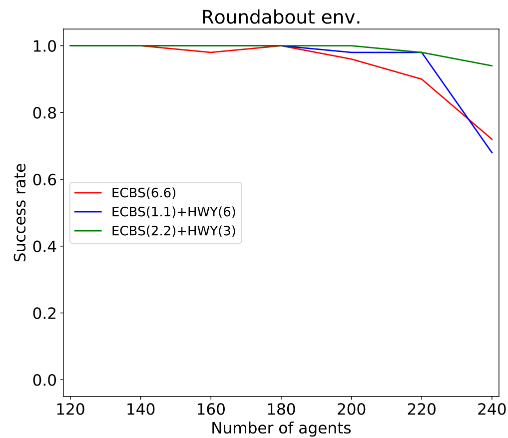
(c)



(d)



(e)



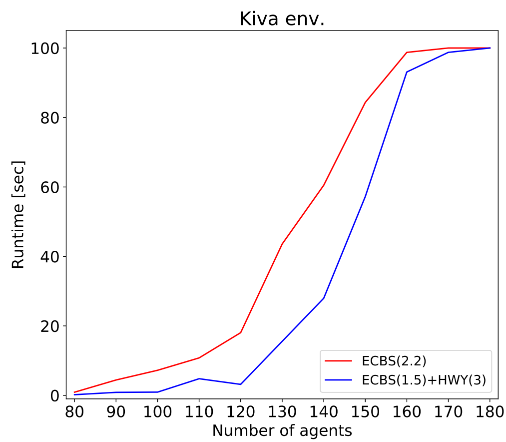
(f)

Figure 4.14: Shows the average runtimes and success rates of 2.2-, 3.0- and 6.6-suboptimal MAPF solvers in the Roundabout environment.

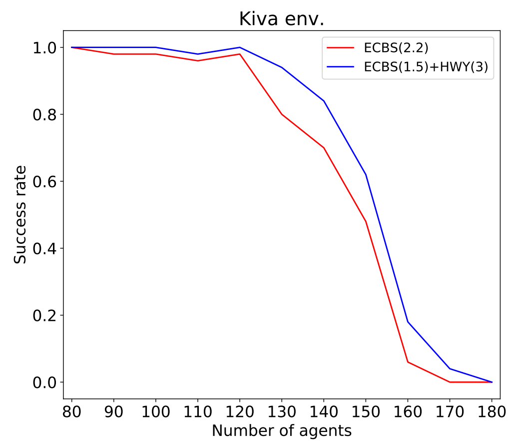
The third set of experiments compares the most efficient ECBS(w_1)+HWY(w_2) with the most efficient ECBS(w) among all values for $w_1 \in \{1.1, 1.2, 1.5\}$, $w_2 \in \{2, 3, 6\}$ and $w = \{1.1, 1.2, 1.5, 2.2, 3, 6.6\}$. More specifically, for each number of agents, we compare the average runtimes of ECBS(2.2) with ECBS(1.5)+HWY(3) in the Kiva environment, ECBS(3) with ECBS(1.5)+HWY(6) in the Drones environment and ECBS(3) with ECBS(1.5)+HWY(6) in the Roundabout environment. Figure 4.15 shows the average runtimes (left column) and success rates (right columns) for the Kiva (upper row), Drones (middle row) and Roundabout (bottom row) environments. In all environments, using $h_{\texttt{HWY}}$ improves efficiency. For example, in the Kiva environment with 140 agents, ECBS(2.2) has an average runtime of 60.49 seconds and a success rate of 70% while ECBS(1.5)+HWY(3) has an average runtime of 27.96 seconds and a success rate of 84%. In the Drones environment with 120 agents, ECBS(3) has an average runtime of 7.81 seconds and a success rate of 100% while ECBS(1.5)+HWY(6) has an average runtime of 0.53 seconds and a success rate of 100%. In the Roundabout environment with 240 agents, ECBS(3) has an average runtime of 85.4 seconds and a success rate of 50% while ECBS(1.5)+HWY(6) has an average runtime of 24.5 seconds and a success rate of 100%.

We continue to examine the efficiency of ECBS and ECBS+HWY for all environments and numbers of agents. Figures 4.16, 4.17 and 4.18 show scatter plots of solution costs (x-axis) and runtimes (y-axis) of ECBS (in red) and ECBS+HWY (in blue) for each number of agents in the Kiva, Drones and Roundabout environments, respectively. Each subfigure also reports the average runtime of ECBS and ECBS+HWY for the relevant 50 MAPF problem instances. We observe that, for all environments and numbers of agents, the average runtime of ECBS+HWY is smaller than the average runtime of ECBS. However, many scatter plots show an overlap in the runtimes of ECBS and ECBS+HWY. Thus, we use a statistical test to substantiate our claim that ECBS+HWY is more efficient than ECBS.

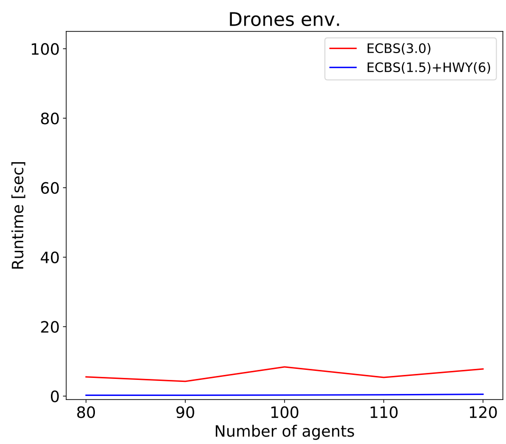
Our hypothesis testing asks how likely it is that the observed runtimes of ECBS and ECBS+HWY are generated from the same distribution. Our null hypothesis is, therefore, that ECBS+HWY is not more efficient than ECBS. Since there is evidence that the runtime of MAPF solvers from the CBS framework



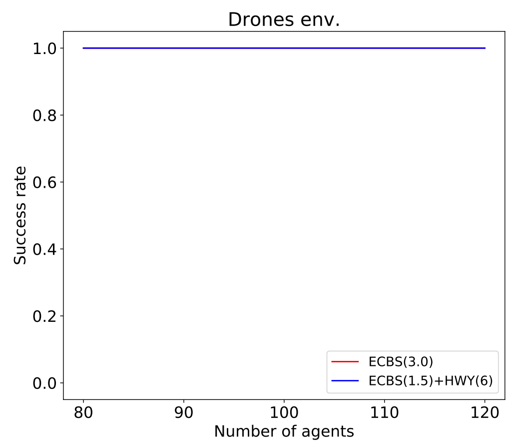
(a)



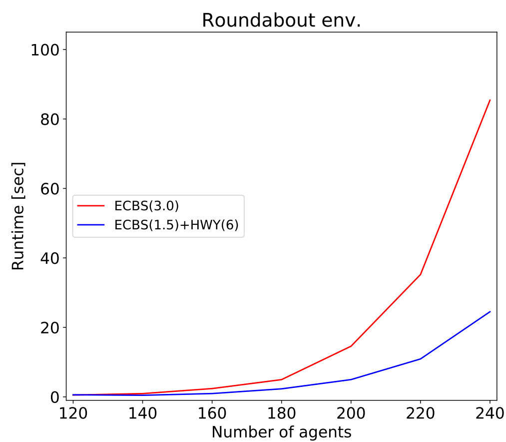
(b)



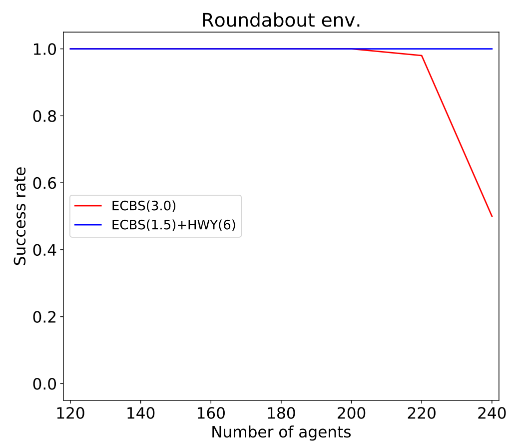
(c)



(d)



(e)



(f)

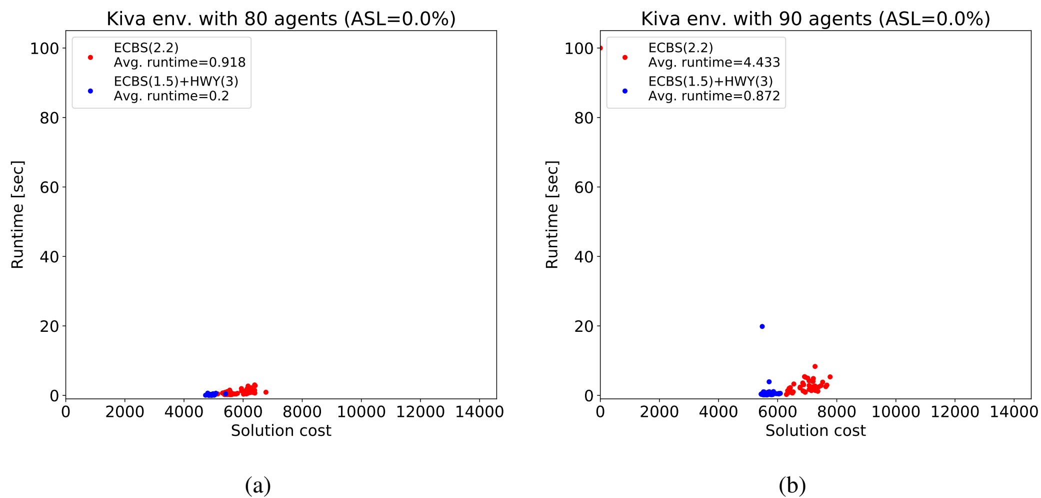
Figure 4.15: Shows the average runtimes and success rates of the most efficient values of the ones we experimented with for the Kiva, Drones and Roundabout environments.

is not distributed normally[6], we use a non-parametric statistical test called the permutation test [14]. The achieved significance level (ASL) is the probability that the null hypothesis holds. We follow the values in [16] for accepting or rejecting the null hypothesis, specifically:

- ASL <10% is borderline evidence against the null hypothesis.
- ASL <5% is reasonably strong evidence against the null hypothesis.
- ASL <2.5% is strong evidence against the null hypothesis.
- ASL <1% is very strong evidence against the null hypothesis.

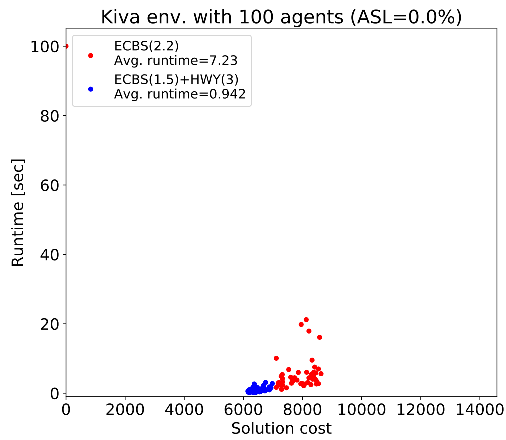
Each subfigure also reports the ASL computed for its runtimes. For all environments and numbers of agents but the Kiva environment with 110 agents, the ASL is 0%. In the Kiva environment with 110 agents, the ASL is 4.6%. Thus, we conclude that ECBS+HWY is more efficient than ECBS.

[6]In fact, the runtime distribution is likely heavy-tailed [13].

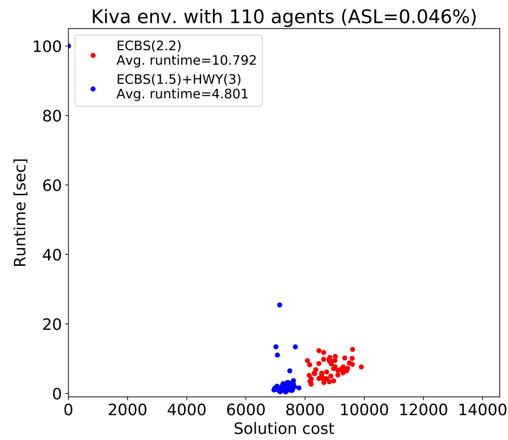


(a) (b)

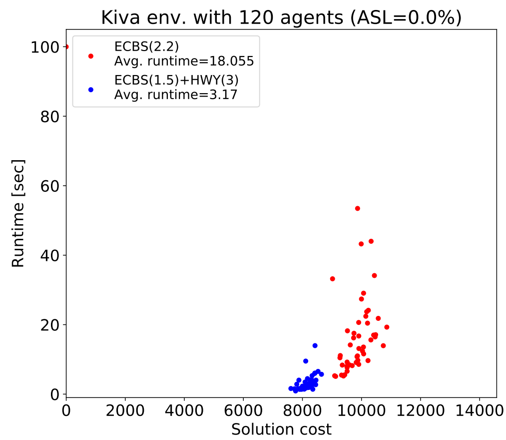
Figure 4.16: Shows the solution cost (x-axis) and runtime (y-axis) of each MAPF problem instance for a given number of agents of ECBS(2.2) and ECBS(1.5)+HWY(3).



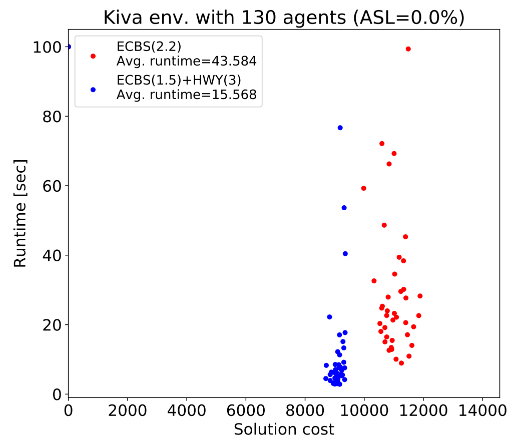
(c)



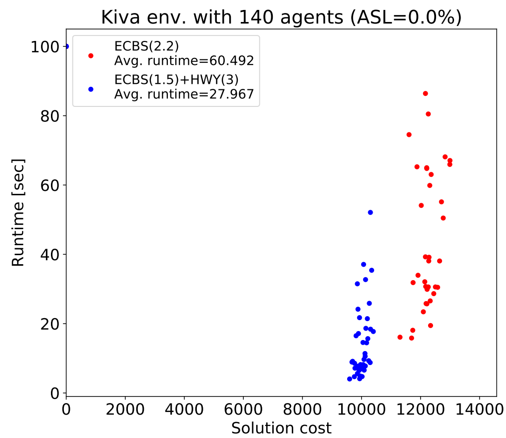
(d)



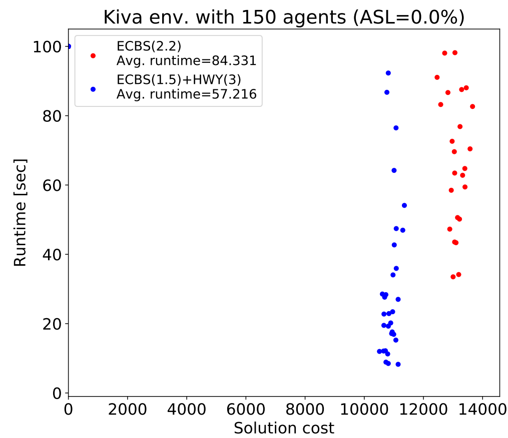
(e)



(f)

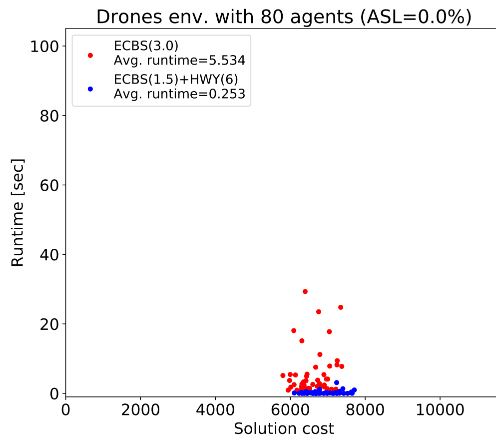


(g)

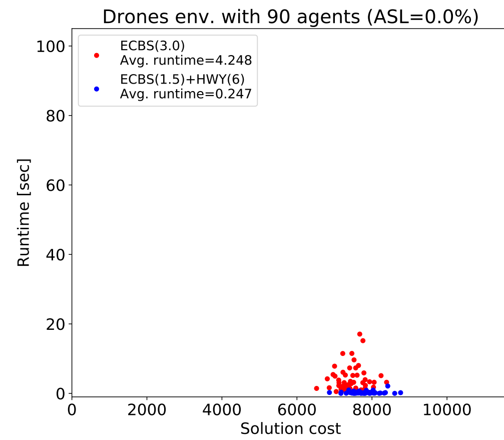


(h)

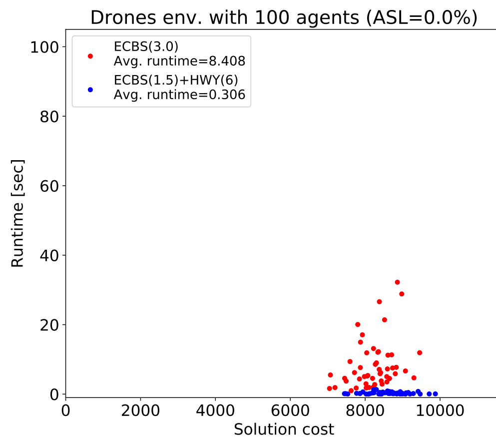
Figure 4.16: Shows the solution cost (x-axis) and runtime (y-axis) of each MAPF problem instance for a given number of agents of ECBS(2.2) and ECBS(1.5)+HWY(3).



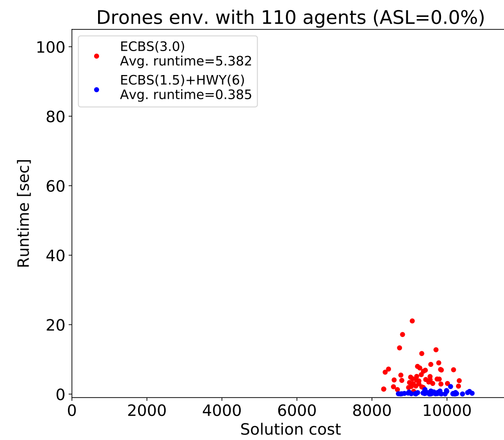
(a)



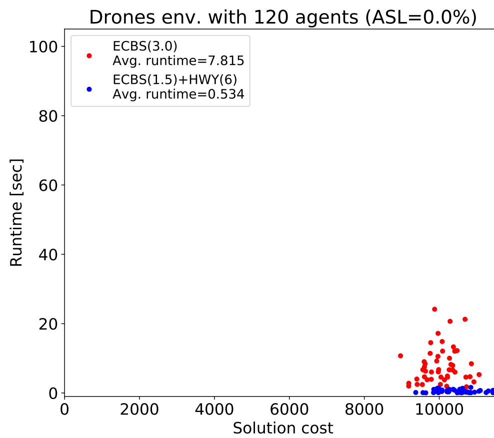
(b)



(c)

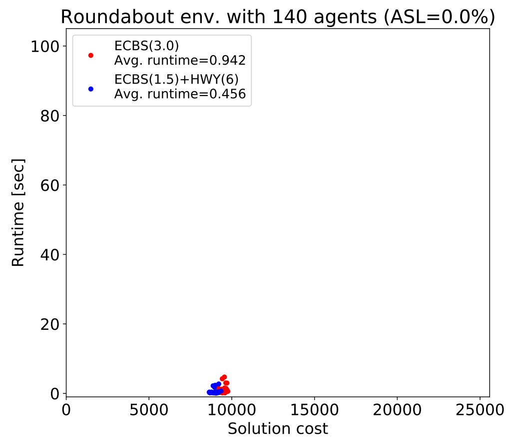


(d)

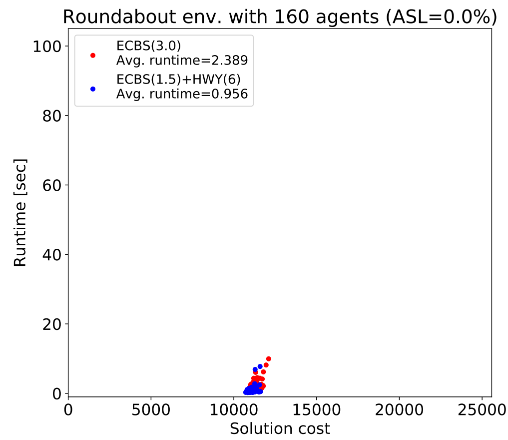


(e)

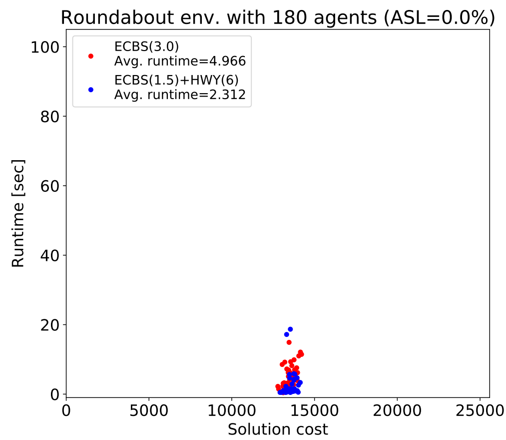
Figure 4.17: Shows the solution cost (x-axis) and runtime (y-axis) of each MAPF problem instance for a given number of agents of ECBS(3) and ECBS(1.5)+HWY(6).



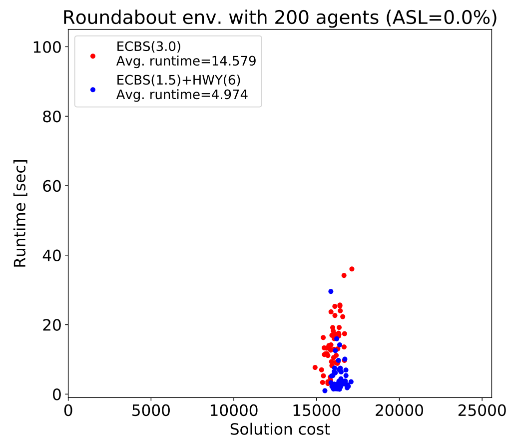
(a)



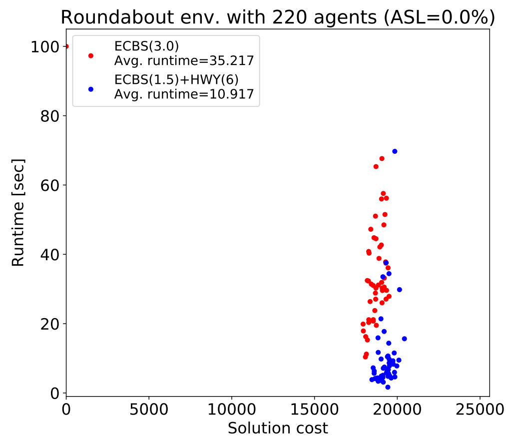
(b)



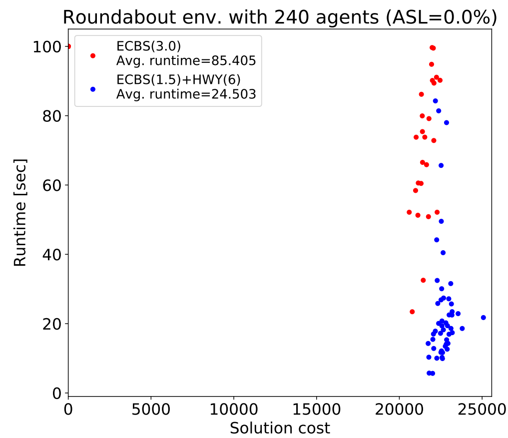
(c)



(d)



(e)



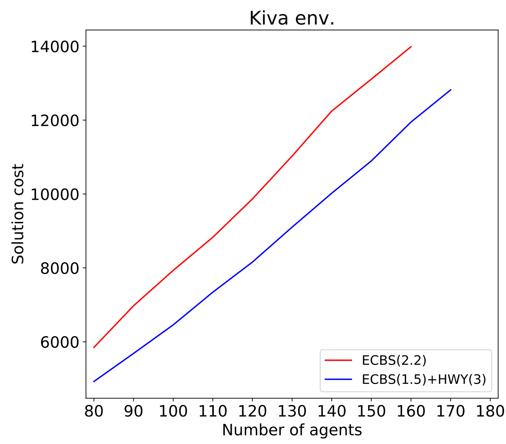
(f)

Figure 4.18: Shows the solution cost (x-axis) and runtime (y-axis) of each MAPF problem instance for a given number of agents of ECBS(3) and ECBS(1.5)+HWY(6).

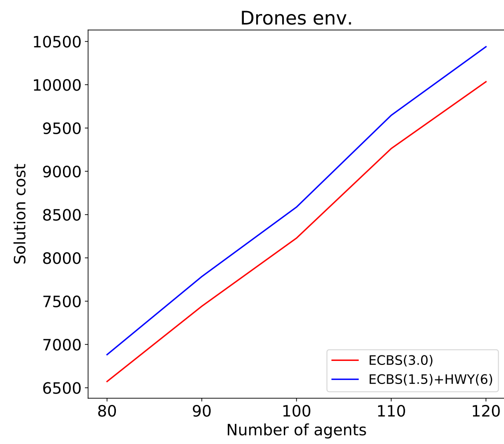
The fourth set of experiments compares the effectiveness of the most efficient ECBS(w_1)+HWY(w_2) with the most efficient ECBS(w) among all values for $w_1 \in \{1.1, 1.2, 1.5\}$, $w_2 \in \{2, 3, 6\}$ and $w = \{1.1, 1.2, 1.5, 2.2, 3, 6.6\}$. More specifically, for each number of agents, we compare the average solution cost of ECBS(2.2) with ECBS(1.5)+HWY(3) in the Kiva environment, ECBS(3) with ECBS(1.5)+HWY(6) in the Drones environment and ECBS(3) with ECBS(1.5)+HWY(6) in the Roundabout environment. Figure 4.19 shows the effectiveness of both MAPF solvers in all three environments. Surprisingly, in the Kiva environment, ECBS(2.2) is less effective than ECBS(1.5)+HWY(3) even though the ECBS(2.2) is 2.2-suboptimal and the ECBS(1.5)+HWY(3) is 4.5-suboptimal. For example, for 80 agents, the average lower bound on the solution cost is 4432.84, the average solution cost of ECBS(2.2) is 5847.34 (resulting in an average suboptimality bound of 1.31) and the average solution cost of ECBS(1.5)+HWY(3) is 4925.28 (resulting in an average suboptimality bound of 1.11). Once again, this exemplifies the strength of using $h_{\texttt{HWY}}$ in the low-level searches.

4.5.3 M* with Highways

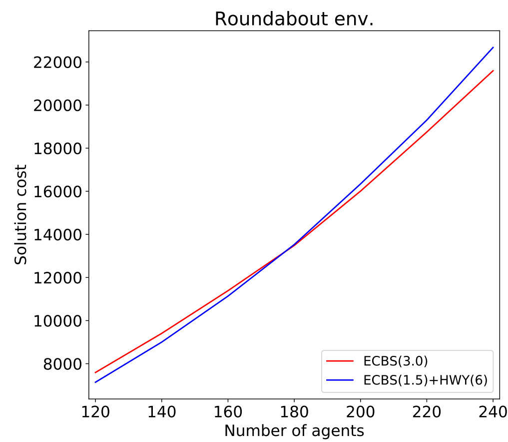
In this subsection, we provide anecdotal evidence that $h_{\texttt{HWY}}$ is also useful for M* [85], a search-based MAPF solver that is different from CBS. Like CBS, M* also tries to avoid operating in the joint state space. However, instead of a two-level search, it uses the idea of subdimensional expansion. M* starts by computing an individual policy for each agent separately. For every cell, the individual policy of agent j specifies an action to a successor cell that moves agent j along an optimal path to its goal cell. M* then uses A* to search a one-dimensional search space defined by the individual policies. If a collision is found during the search, the dimensionality of the search space is locally increased to allow A* to explore sucessor cells that diverge from individual policies of the colliding agents, which ensures that A* can find an alternative path that avoids the collision. Inflated-M* [85] is a bounded-suboptimal version of M* that uses inflated heuristic values similar to weighted-A*. Thus, using $h_{\texttt{HWY}}$ with M*, denoted by M*+HWY(w), is also bounded-suboptimal.



(a)

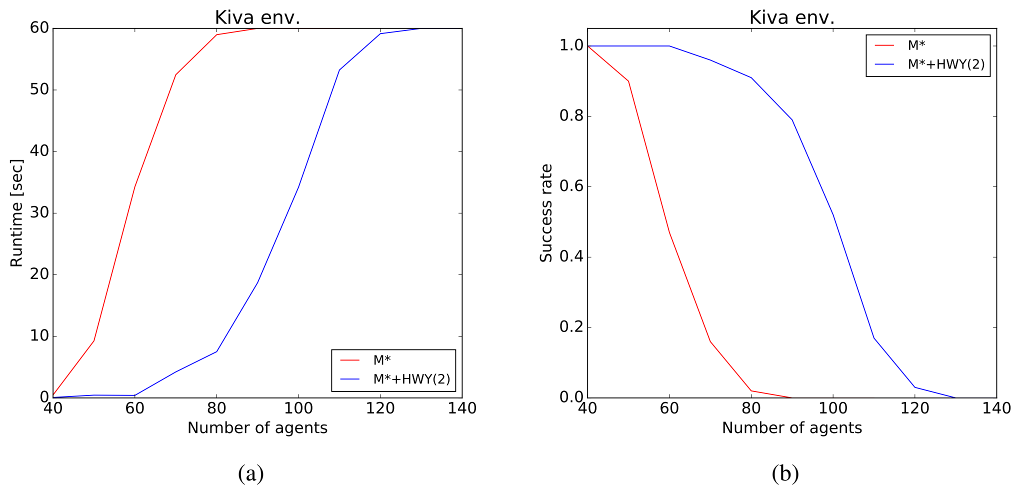


(b)



(c)

Figure 4.19: (a), (b) and (c) show the average solution cost of ECBS(2.2) and ECBS(1.5)+HWY(3) (ECBS(3) and ECBS(1.5)+HWY(6)) for the Kiva environmnet (Drones and Roundabout environments).



(a)

(b)

Figure 4.20: Shows the average runtimes and success rates M* and M*+HWY(2) in the Kiva environment.

We evaluate M* and M*+HWY(2) in the Kiva environment for the experimental setup described in Chapter 3.5.1 and the highways depicted in Figure 4.8(a). The experiments for M* and M*+HWY were done on a cluster of 38 Amazon EC2 c4.xlarge instances running on Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz with 4 vcpu (2 physical cores) and 7.5GB RAM per instance. Each MAPF solver used 1 worker per instance with a 1 minute time limit.[7] Figure 4.20 shows the average runtimes and success rates (in (a) and (b), respectively) of M* and M*+HWY(2). Using $h_{\texttt{HWY}}$ instead of $h_{\texttt{SP}}$ to compute the individual policies for each agent results in improved efficiency and, therefore, higher success rates for higher numbers of agents. For example, with 60 agents, M* has an average runtime of 34.287 seconds and a success rate of 47% while M*+HWY(2) has an average runtime of 0.412 seconds and a success rate of 100%.

4.6 Automatically Generating Highways

In this section, we present three approaches for automatically generating highways given a MAPF problem instance. The first approach, called crisscross (CC) highways, generates highways by using a simple regular pattern defined for two-dimensional four-neighbor grid environments. The second approach, called

[7]This is due to the high memory consumption of our M* implementation.

graphical model (GM)-based highways, generates highways by solving an optimization problem posed on a GM defined for two-dimensional four-neighbor grid environments. The GM is designed to capture the environment's structure and the collisions present in a given set of paths. Unlike the CC and GM-based highways, which are defined for two-dimensional four-neighbor grid environments[8], the third approach, called heat map (HM)-based highways, is defined for arbitrary graphs. HM-based highways are inspired by an oracular approach for solving multi-commodity flow problems [41].

The idea behind the GM- and HM-based approaches is simple: Optimally solving the MAPF problem is NP-hard, but computing the optimal paths for all agents independently is fast. The GM- and HM-based approaches use the information in these optimal paths to generate highways automatically. To solve a given MAPF problem instance, highways are first generated using one of the approaches, and then the MAPF problem instance is solved by ECBS(w_1)+HWY(w_2) with the generated highways. In the following subsections, we provide more details on each approach, explain why the CC highways are ill-suited for influencing $h_{\texttt{HWY}}$'s values, and evaluate the GM- and HM-based highways on the same MAPF problem instances used to evaluate human-designed highways. Finally, we discuss the limitations of the GM- and HM-based approaches.

4.6.1 Crisscross Highways

The CC highways use a simple fixed pattern for automatically generating highways, and are inspired by MAPF-related research done in the context of avoiding deadlocks [88]. For a two-dimensional four-neighbor grid environment, the CC highways are defined as follows:

1. For each grid cell in an even row, there is an edge pointing east is in the CC highways.

2. For each grid cell in an odd row, there is an edge pointing west is in the CC highways.

3. For each grid cell in an even column, there is an edge pointing north is in the CC highways.

4. For each grid cell in an odd column, there is an edge pointing south is in the CC highways.

[8] The CC and GM-based highways can be generalized to three-dimensional six-neighbor grid environments easily, but arbitrary graphs may require additional developments.

X_i	Y_i	$f(X_i, Y_i)$
0??	None	6
0??	East	1
0??	North	1
0??	West	1
0??	South	1
101 ; 111	None	3 ; 1
101 ; 111	East	6 ; 24
101 ; 111	North	1 ; 1
101 ; 111	West	1 ; 1
101 ; 111	South	1 ; 1
102 ; 112	None	3 ; 1
102 ; 112	East	1 ; 1
102 ; 112	North	6 ; 24
102 ; 112	West	1 ; 1
102 ; 112	South	1 ; 1
103 ; 113	None	3 ; 1
103 ; 113	East	1 ; 1
103 ; 113	North	1 ; 1
103 ; 113	West	6 ; 24
103 ; 113	South	1 ; 1
104 ; 114	None	3 ; 1
104 ; 114	East	1 ; 1
104 ; 114	North	1 ; 1
104 ; 114	West	1 ; 1
104 ; 114	South	6 ; 24

Y_i	Y_j	$f_H(Y_i, Y_j)$	$f_V(Y_i, Y_j)$
None	None	6	6
None	East	1	1
None	North	1	1
None	West	1	1
None	South	1	1
East	None	2.8	5
East	East	5	0.2
East	North	1	1
East	West	0.2	2.8
East	South	1	1
North	None	5	2.8
North	East	1	1
North	North	0.2	5
North	West	1	1
North	South	2.8	0.2
West	None	2.8	5
West	East	0.2	2.8
West	North	1	1
West	West	5	0.2
West	South	1	1
South	None	5	2.8
South	East	1	1
South	North	2.8	0.2
South	West	1	1
South	South	0.2	5

Table 4.1: Shows the different factors of our GM. The first digit in the value of X_i indicates whether ('1') or not ('0') there is a collision in the cell. The second digit indicates whether the magnitude of DV(i) is greater than 1/2 ('1') or not ('0'). The third digit indicates whether the direction of DV(i) is in the eastern quadrant $[315°, 45°)$ ('1'), northern quadrant $[45°, 135°)$ ('2'), western quadrant $[135°, 225°)$ ('3') or southern quadrant $[225°, 315°)$ ('4'). '?' represents a wildcard.

Figure 4.21 and Figure 4.22 show the CC highways for the Kiva environment and the $h_{\mathtt{HWY}}$ values for goal cell [18,51] for $w = 2$ and $w = 6$, respectively. Changing w from 2 to 6 has almost no effect on $h_{\mathtt{HWY}}$'s values. In general, the CC highways are ill-suited for influencing heuristic values because it is almost always the case that the optimal path between any two cells uses only a handful of edges that are not CC highways. Therefore, our ability to manipulate heuristic values with the CC highways is minimal. Since the heuristic values mostly do not change, ECBS with the CC highways is likely to be very similar (both in efficiency and effectiveness) to ECBS, and thus we do not evaluate them in experiments.

4.6.2 Graphical Model-Based Highways

The GM-based highways use a GM for automatically generating highways. A GM is a graph whose vertices represent variables and whose edges represent interactions between variables, resulting in a factored

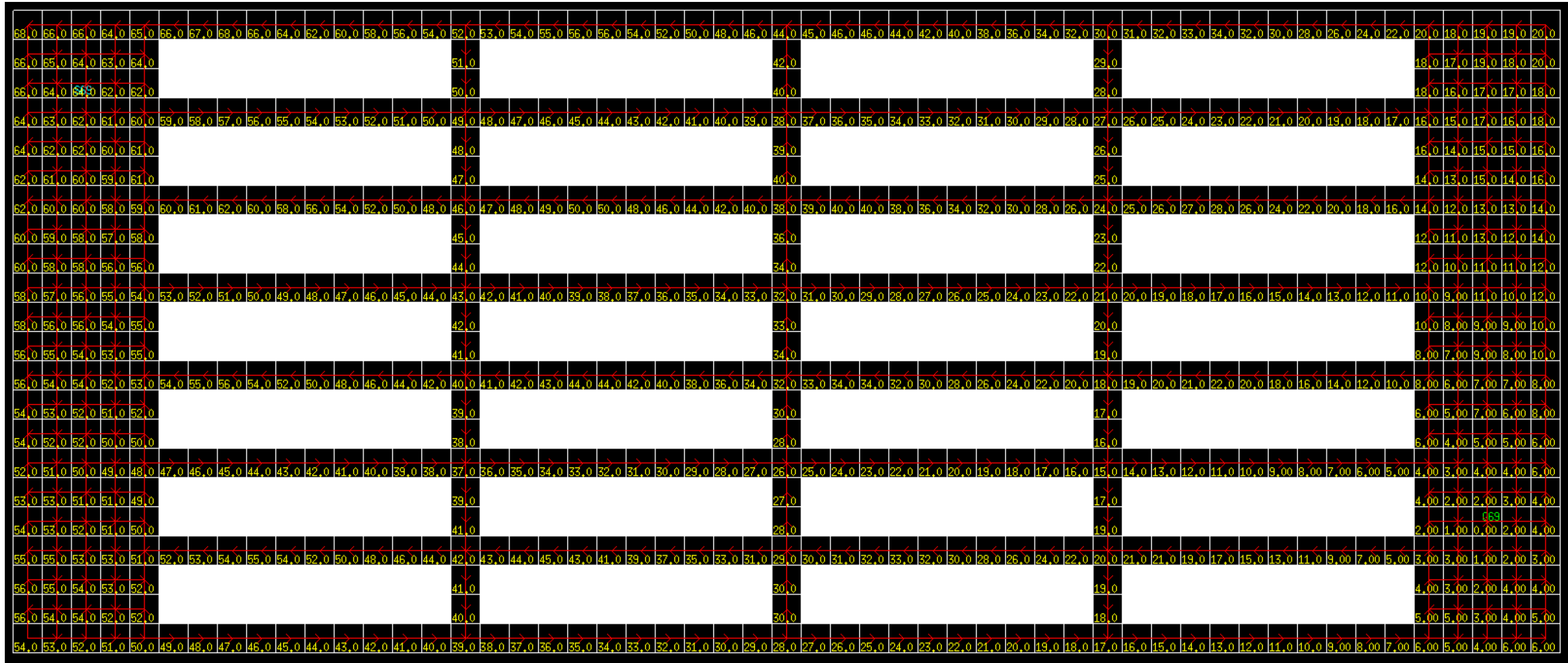


Figure 4.21: Shows the CC highways for the Kiva environment and $h_{\mathtt{HWY}}$ values for goal cell [18,51] with $w = 2$.

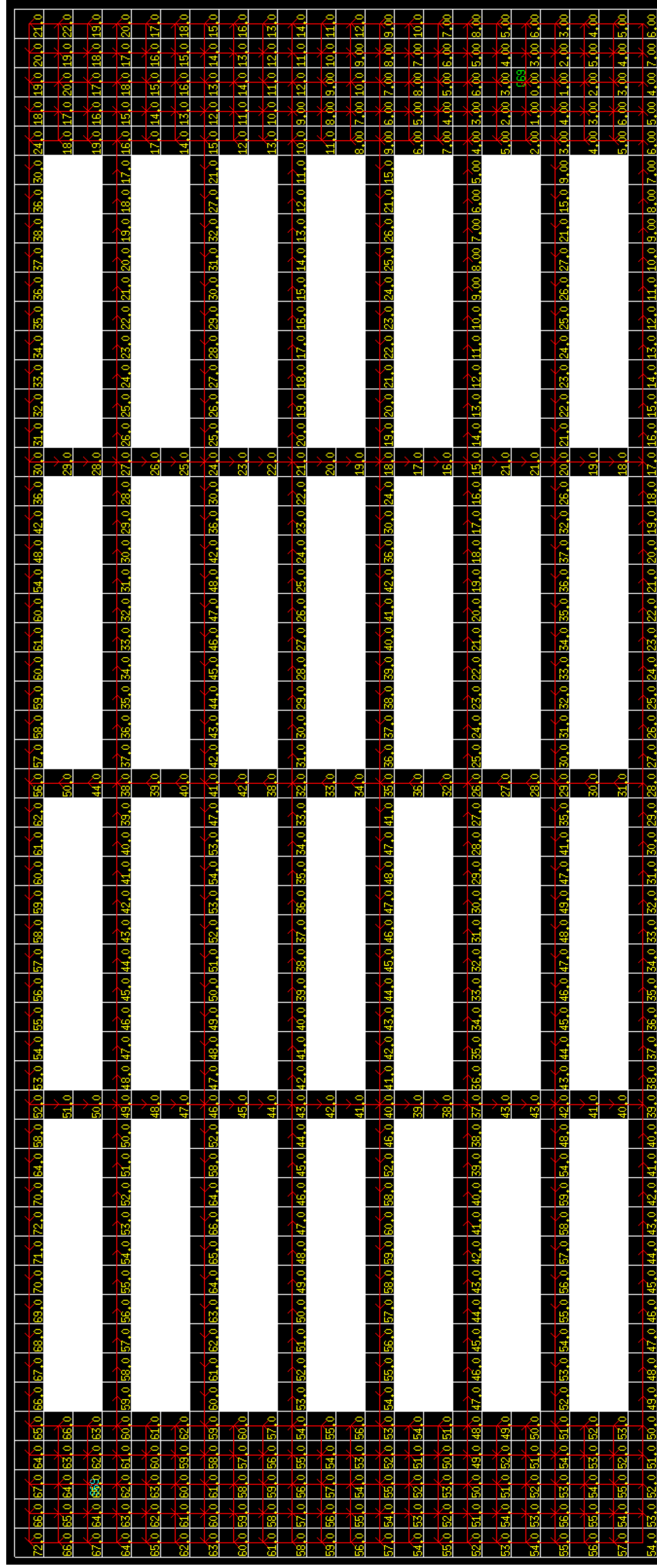


Figure 4.22: Shows the CC highways for the Kiva environment and h_{HWY} values for goal cell [18,51] with $w = 6$.

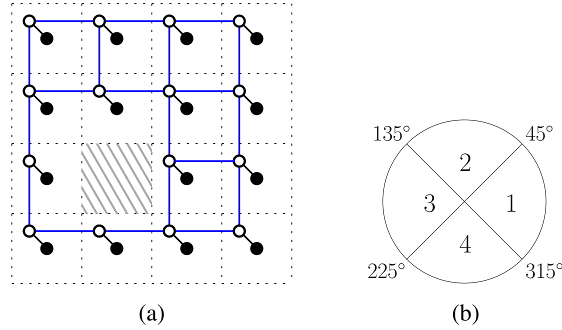


Figure 4.23: (a) shows a GM over a 4x4 grid environment. (b) shows the mapping from angles to discrete values.

representation of such interactions [36]. For a two-dimensional four-neighbor grid environment, the GM is defined as follows: Two variables, X_i and Y_i, are associated with each unblocked cell i. Figure 4.23(a) illustrates a GM. Solid circles represent the X_i variables, and hollow circles represent the Y_i variables.

The X_i variables are observable and correspond to the following statistics gathered for cell i when computing the optimal paths for all agents independently and assuming that the agents follow these paths:

1. Whether a collision occurs between any two agents in cell i (to be precise: whether a vertex collision occurs in the cell or an edge collision occurs on any adjacent edge).

2. Whether the magnitude of the direction vector DV(i) is greater than 0.5.

3. The direction of DV(i), discretized into the four compass directions.

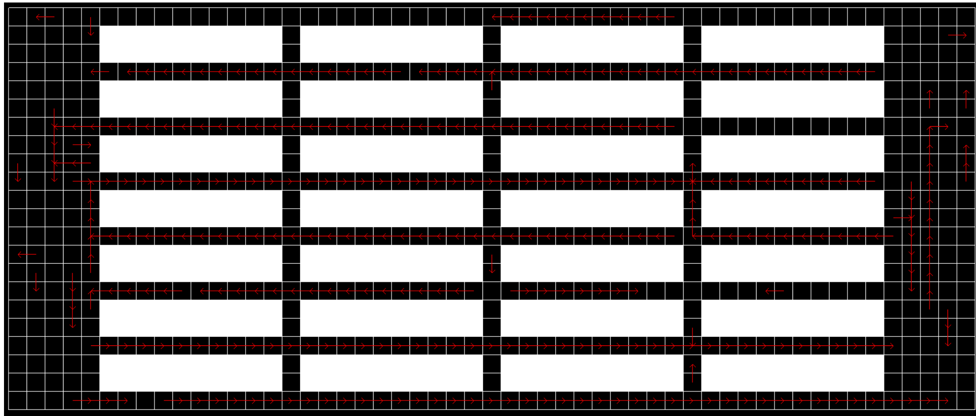
The two-dimensional direction vector DV(i), inspired by [33], is computed by generating two unit vectors whenever an agent traverses i, namely one in its entry direction and one in its exit direction. DV(i) is the average of all such unit vectors. The Y_i variables are hidden and indicate the compass direction to the adjacent cell, if any, to which a highway edge leaving i should be generated. If the value of Y_i is 'None' or the indicated adjacent cell is blocked, then no highway edge is generated.

The values of the Y_i variables are determined by solving the maximum-a-posteriori estimation problem for the GM with the factors shown in Figure 4.1. For example, $f(X_i = 111, Y_i = None) = 1$. The rationale behind the values of the factors is as follows: 1) Consider an unblocked cell i. If a collision occurs

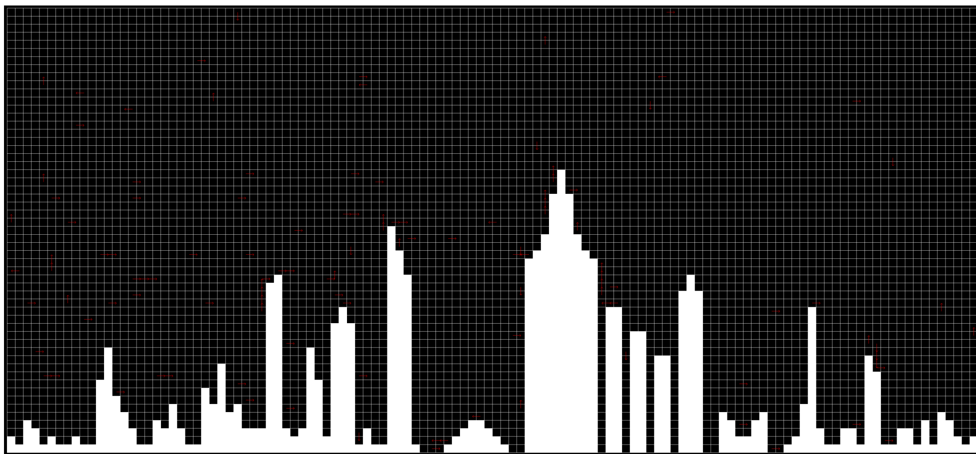
in i, there is a preference to have a highway edge leave i in the discretized direction of DV(i). A larger discretized magnitude of DV(i) indicates a stronger preference. These factors $f(X_i, Y_i)$ are represented by the black edges in Figure 4.23(a). 2) Consider two unblocked cells i and j that are horizontal [vertical] neighbors. There is a preference to have highway edges leave them in the same horizontal [vertical] direction (to construct a longer highway) or leave them in opposite vertical [horizontal] directions (to construct two highways in opposite directions next to each other to allow for two-way traffic). These factors $f_H(Y_i, Y_i)$ [$f_V(Y_i, Y_i)$] are represented by the blue edges in Figure 4.23(a). We do not completely optimize the values of the factors.

While solving the maximum-a-posteriori estimation problem is itself NP-hard even for the special case of two-dimensional grids, there are approaches that work well in practice. For example, our problem corresponds to a two-dimensional hidden Markov model, for which specialized approaches exist. Since we derive only heuristic values from the highways, we do not require an exact solution and thus use Gibbs sampling [36] with $5,000$ samples. The GM-based approach runs in about one second in our experiments and thus is fast compared to running the MAPF solvers on MAPF problems with many agents. Our implementation of the GM-based approach uses libDAI [51].

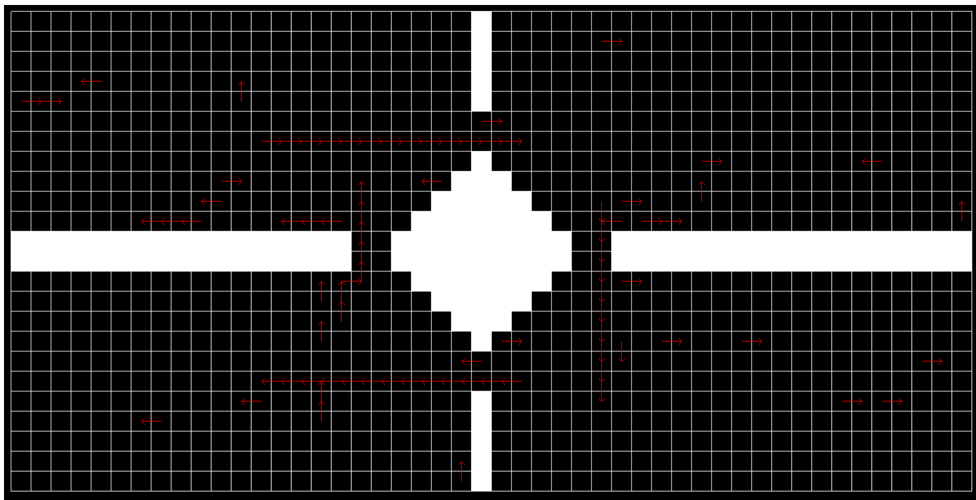
Figure 4.24 shows GM-based highways generated for some MAPF problem instances in the Kiva, Drones and Roundabout environments. These highways look somewhat intermittent compared to the human-designed highways in Figure 4.8. Still, the GM-based highways exploit some structure in each environment: In the Kiva environment, highways in the long narrow corridors are generally pointing in only one direction, which encourages agents to avoid head-on collisions when moving inside the corridors. In the Drones environment, highways in the two rows immediately above the city skyline are many times pointing in opposing directions horizontally (that is, $\rightleftarrows$ or $\leftrightarrows$), which encourages agents to avoid head-on collisions when moving past tall obstacles. Finally, in the Roundabout environment, highways inside the roundabout are generally pointing in a clockwise direction around the roundabout, which encourages agents to avoid head-on collisions when moving inside the roundabout.



(a)



(b)



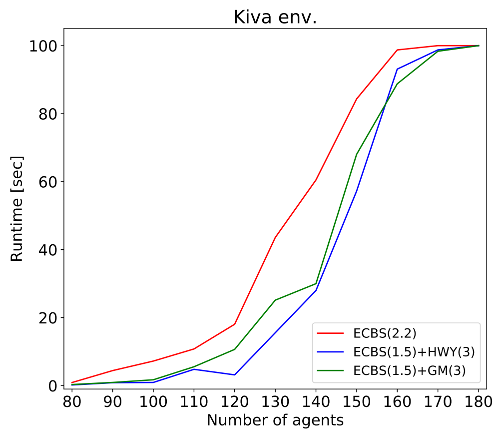
(c)

Figure 4.24: Shows the highways generated by the GM-based approach for MAPF problems with 150 agents in the Kiva environment (a), 120 agents in the Drones environment (b) and 150 agents in the Roundabout environment (c).

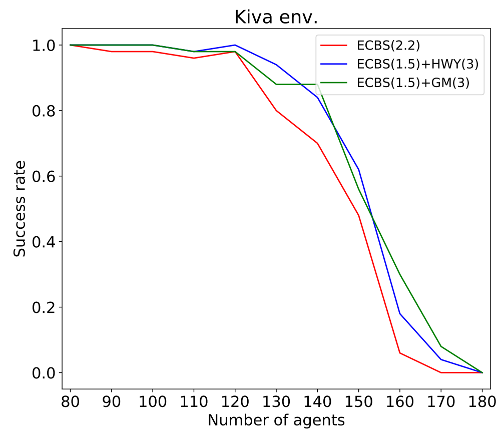
We now evaluate the efficiency of using the GM-based highways on the MAPF problem instances from the previous section. The term ECBS(w_1)+GM(w_2) refers to a MAPF solver that uses ECBS(w_1) with $h_{\texttt{HWY}}$ computed for the GM-based highways and an inflation factor of w_2. Similarly, the term ECBS(w_1)+HWY(w_2) refers to a MAPF solver uses ECBS(w_1) with $h_{\texttt{HWY}}$ computed for the human-designed highways (specified in Figure 4.8) and an inflation factor of w_2. Figure 4.25 shows the average runtimes (left column) and success rates (right column) of ECBS(w_1)+GM(w_2) (green curves), ECBS(w_1)+HWY(w_2) (blue curves) and ECBS(w) (red curves) for the Kiva (upper row), Drones (middle row) and Roundabout (bottom row) environments. The values of w_1, w_2 and w were set to the most efficient ones, as identified in Section 4.5.2, that is, for each number of agents, we compare the average runtimes of ECBS(2.2), ECBS(1.5)+HWY(3) and ECBS(1.5)+GM(3) in the Kiva environment, ECBS(3), ECBS(1.5)+HWY(6) and ECBS(1.5)+GM(6) in the Drones environment and ECBS(3), ECBS(1.5)+HWY(6) and ECBS(1.5)+GM(6) in the Roundabout environment. The same human-designed highways are used, that is, the ones in Figure 4.8.

Figure 4.25 shows that, for all three environments and numbers of agents, ECBS+GM is more efficient than ECBS. Table 4.2 shows the ASL values for the null hypothesis "ECBS+GM is not more efficient than ECBS". In all environments and numbers of agents except the Roundabout environment with 120 and 140 agents, the ASL is smaller than 10%, which means that there is evidence against the null hypothesis, that is, ECBS+GM is more efficient than ECBS.

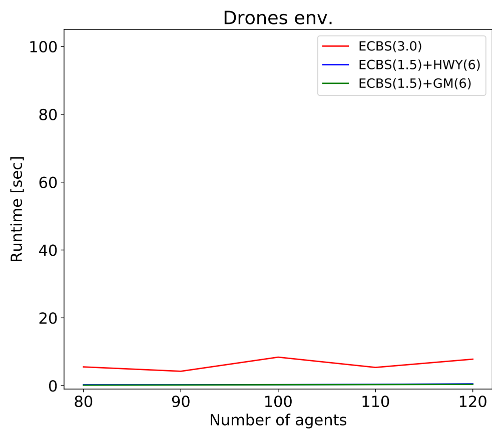
Figure 4.25 also shows that, for all three environments and numbers of agents except the Kiva environment with 160 agents, ECBS+GM is not more efficient than ECBS+HWY. Nevertheless, many times ECBS+GM is as efficient as ECBS+HWY. Table 4.3 shows the ASL values for the null hypothesis "ECBS+HWY is not more efficient than ECBS+GM". In all environments and numbers of agents except the Kiva environment with 80, 100, 120, 130 and 150 agents, the Drones environment with 120 agents and the Roundabout environment with 220 and 240 agents, the ASL value is larger than 10%, which means that we do not have strong evidence to reject the null hypothesis, that is, ECBS+HWY is not more efficient than ECBS+GM.



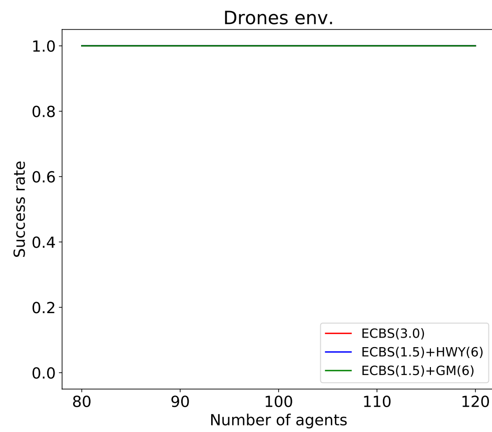
(a)



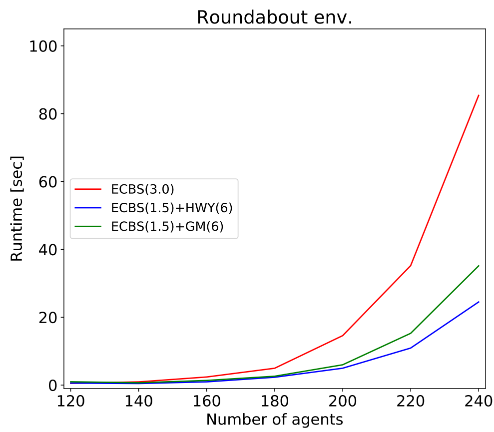
(b)



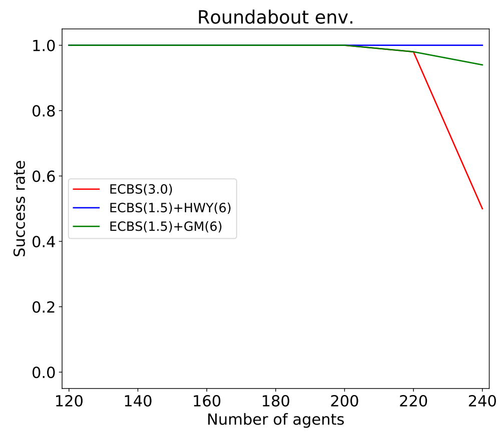
(c)



(d)



(e)



(f)

Figure 4.25: Shows the average runtimes and success rates of ECBS(2.2), ECBS(1.5)+HWY(3) and ECBS(1.5)+GM(3) (ECBS(3), ECBS(1.5)+HWY(6) and ECBS(1.5)+GM(6)) for the Kiva environmnet (Drones and Roundabout environments).

Kiva env.		Drones env.		Roundabout env.	
Number of agents	ASL	Number of agents	ASL	Number of agents	ASL
80	0%	80	0%	120	15.4%
90	0%	90	0%	140	15.5%
100	0%	100	0%	160	3.2%
110	7.8%	110	0%	180	0.3%
120	1.7%	120	0%	200	0%
130	0.1%			220	0%
140	0%			240	0%
150	0.3%				
160	0%				
170	0%				

Table 4.2: ASL values for the null hypothesis “ECBS+GM is not more efficient than ECBS” for each environment and number of agents.

Kiva env.		Drones env.		Roundabout env.	
Number of agents	ASL	Number of agents	ASL	Number of agents	ASL
80	0.6%	80	10.1%	120	22.4%
90	47.4%	90	21.8%	140	16.9%
100	0%	100	23.6%	160	30.6%
110	36.4%	110	14.3%	180	38.4%
120	0%	120	1.3%	200	21.7%
130	4.4%			220	9%
140	37.4%			240	1.6%
150	7.6%				
160	13.2%				
170	34.8%				

Table 4.3: ASL values for the null hypothesis “ECBS+HWY is not more efficient than ECBS+GM” for each environment and number of agents.

4.6.3 Heat Map-Based Highways

The HM-based highways are inspired by an oracular approach for solving multi-commodity flow problems [41]. An oracular approach uses a black-box that returns appropriate information on the problem at query points. In our case, the black-box computes optimal paths between pairs of cells. Different from the GM-based approach, the HM-based approach computes paths sequentially by repeatedly picking an agent randomly and computing its optimal path using the current HM costs of the edges. Each such path is influenced by the previously computed paths because agents are given an incentive to follow previously computed paths until the capacity constraints of edges (here: equal to one) are reached. The edges that are used frequently after $MaxIterations$ path computations (for a user-specified parameter $MaxIterations > 1$), namely those with small HM costs, become highway edges.

Algorithm 3 shows the HM-based approach, which first converts each undirected edge to two directed edges with HM costs one and then updates the following values for each directed edge $\langle u, v\rangle$ to update its HM cost:

- $n(u, v)$ (initialized to zero) is the number of times the directed edge is chosen in any optimal path.

- $p(u, v)$ is the follow preference of the edge, defined to be $\alpha n(u, v)/MaxIterations$ (for a user-specified parameter $\alpha \in (0, 1)$). The follow preferences encourage agents to traverse directed edges that appear in previously computed paths.

- $t(u, v)$ is the interference cost of the edge, defined to be $\beta n(v, u)/MaxIterations$ (for user-specified parameter $\beta > 1$). The interference costs discourage agents from traversing opposite directed edges (that correspond to the same undirected edge).

- $s(u, v)$ is the saturation cost of the edge, defined to be $\gamma^{(n(u,v)+n(v,u))/(2MaxIterations)}$ (for user-specified parameter $\gamma > 1$).

- $c(u, v)$ is the HM cost of the edge, defined to be $1 - p(u, v) + t(u, v) + s(u, v)$.

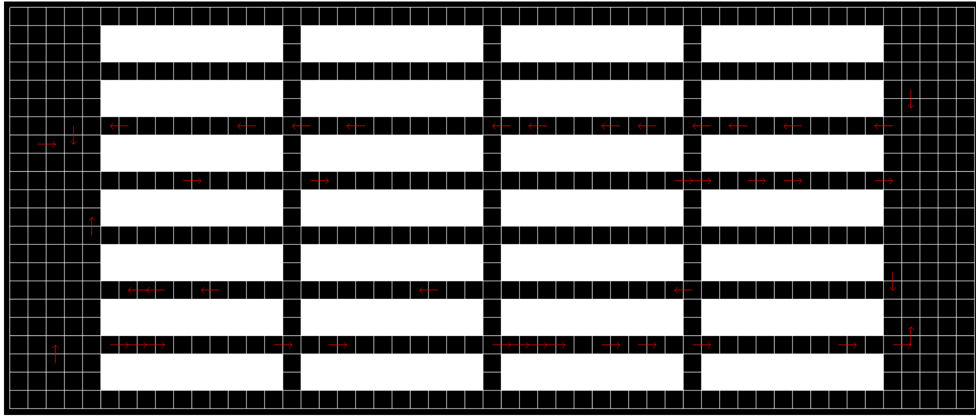
Algorithm 3: HM-Based Highways.

input : A MAPF problem (environment $G = (V, E)$ and start goal cells $(s^1, g^1), \ldots, (s^K, g^K)$).
output: Highways edges ($E_{\mathtt{hwy}}$).

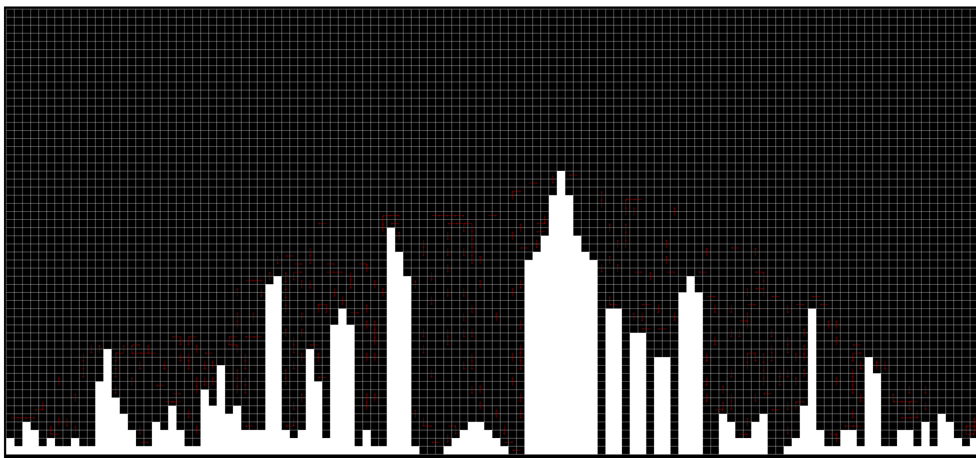
1 Convert each undirected edge $(u, v) \in E$ to two directed edges $\langle u, v\rangle$ and $\langle v, u\rangle$ with HM costs one.
2 **for** *iteration := 1 to MaxIterations* **do**
3   Pick a random (s^i, g^i) pair.
4   Compute an optimal path P from s^i to g^i in the directed graph using the current HM costs of the edges.
5   **for** *each edge $\langle u, v\rangle$ on this path P* **do**
6   Increase $n(u, v)$ by 1.
7   Update the follow preference $p(u, v)$.
8   Update the saturation cost $s(u, v)$.
9   Update the interference cost of the opposite edge $t(v, u)$.
10 Return some of the edges with small HM costs as highway edges.

Our implementation uses parameters $\alpha = 0.5$, $\beta = 1.2$, $\gamma = 1.3$ and $MaxIterations = 100,000$. It considers the $1/7^{th}$ of the edges with the lowest HM costs after $MaxIterations$ iterations and then returns $1/5^{th}$ of these edges randomly. We do not completely optimize the values of these parameters. The HM-based approach runs in about one second in our experiments and thus is fast compared to running the MAPF solvers.

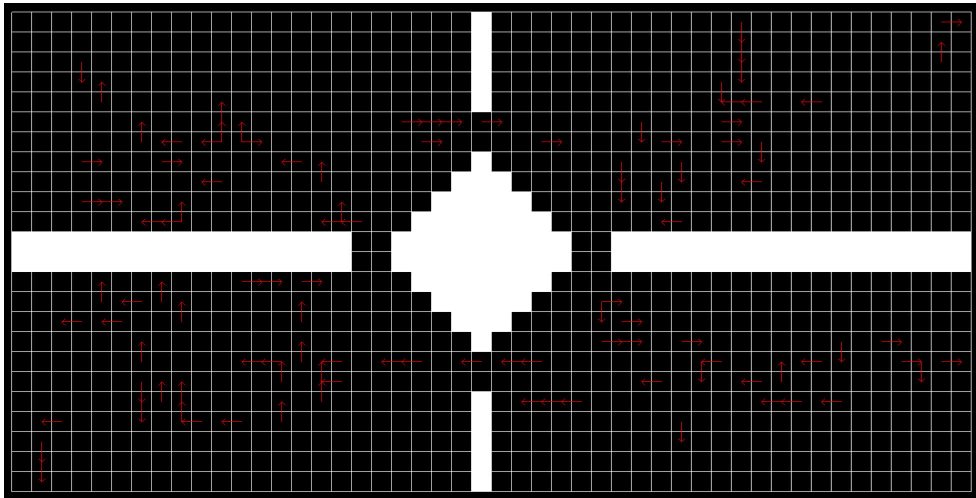
Figure 4.26 shows HM-based highways generated for some MAPF problem instances in the Kiva, Drones and Roundabout environments. These highways look even more intermittent than the GM-based highways in Figure 4.24. The term ECBS(w_1)+HM(w_2) refers to a MAPF solver that uses ECBS(w_1) with $h_{\mathtt{HWY}}$ computed for the HM-based highways and an inflation factor of w_2. Figure 4.27 shows the average runtimes (left column) and success rates (right column) of ECBS(w_1)+HM(w_2) (green curves), ECBS(w_1)+HWY(w_2) (blue curves) and ECBS(w) (red curves) for the Kiva (upper row), Drones (middle row) and Roundabout (bottom row) environments. The values of w_1, w_2 and w were set to the most efficient ones, as identified in Section 4.5.2, that is, for each number of agents, we compare the average runtimes of: ECBS(2.2), ECBS(1.5)+HWY(3) and ECBS(1.5)+HM(3) in the Kiva environment, ECBS(3), ECBS(1.5)+HWY(6) and ECBS(1.5)+HM(6) in the Drones environment and ECBS(3),



(a)



(b)



(c)

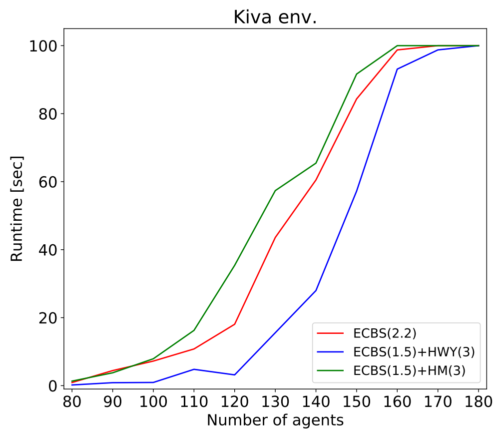
Figure 4.26: Shows the highways generated by the HM-based approach for MAPF problems with 150 agents in the Kiva environment (a), 120 agents in the Drones environment (b) and 150 agents in the Roundabout environment (c).

ECBS(1.5)+HWY(6) and ECBS(1.5)+HM(6) in the Roundabout environment. The same human-designed highways are used, that is, the ones in Figure 4.8.

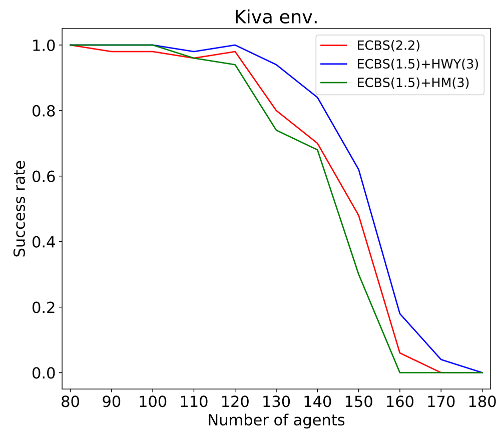
Figure 4.27 shows that ECBS+HM is less efficient than ECBS in the Kiva environment for all numbers of agents but the more efficient in the Drones and Roundabout environments as the number of agents increases. Table 4.4 shows the ASL values for the null hypothesis "ECBS is not more efficient than ECBS+HM" for the left column and "ECBS+HM is not more efficient than ECBS" for the right and middle columns.[9] According to the table, there is evidence that ECBS+HM is more efficient than ECBS only in the Drones environment with any number of agents and in the Roundabout environment with more than 180 agents.

Figure 4.27 also shows that, for all three environments and numbers of agents, ECBS+HM is not more efficient than ECBS+HWY. Table 4.5 shows the ASL values for the null hypothesis "ECBS+HWY is not more efficient than ECBS+HM". The ASL value is larger than 10% only in the Drones environment with any number of agents and the Roundabout environment with 120, 140, and 180 agents, which means that we have strong evidence to reject the null hypothesis in all other cases, that is, ECBS+HWY is more efficient than ECBS+HM in many cases. On the negative side, ECBS+HM does not perform as well as ECBS+GM, partially because it is defined for any graph (and thus more general). On the positive side, ECBS+HM is applicable to more environments.

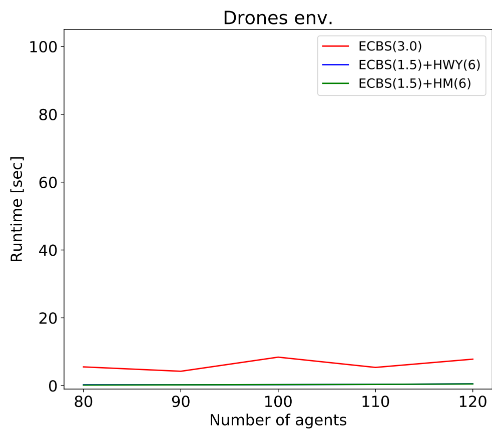
[9] In the Kiva environment, the null hypothesis is "ECBS is not more efficient than ECBS+HM" because, for all numbers of agents, ECBS is more efficient than ECBS+HM. Thus, if the ASL value is small, we conclude that ECBS is more efficient than ECBS+HM.



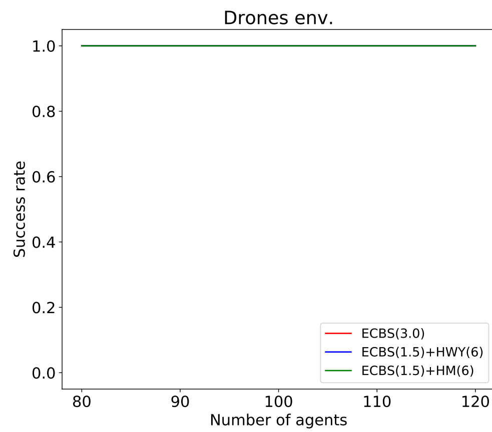
(a)



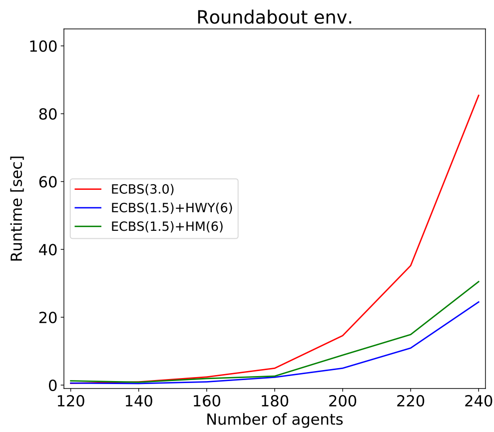
(b)



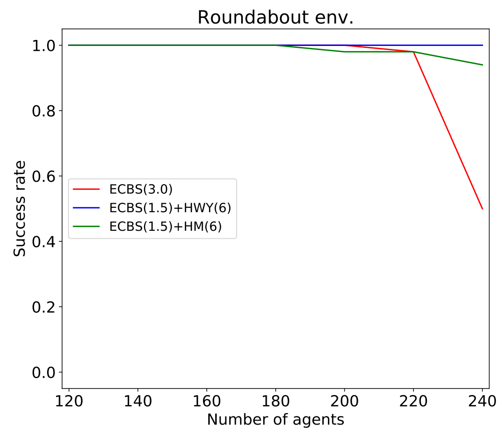
(c)



(d)



(e)



(f)

Figure 4.27: Shows the average runtimes and success rates of ECBS(2.2), ECBS(1.5)+HWY(3) and ECBS(1.5)+HM(3) (ECBS(3), ECBS(1.5)+HWY(6) and ECBS(1.5)+HM(6)) for the Kiva environmnet (Drones and Roundabout environments).

Kiva env.		Drones env.		Roundabout env.	
Number of agents	ASL	Number of agents	ASL	Number of agents	ASL
80	3.2%	80	0%	120	13%
90	43.9%	90	0%	140	41.8%
100	41.6%	100	0%	160	25%
110	8%	110	0%	180	0.3%
120	0%	120	0%	200	1.1%
130	2.1%			220	0%
140	20.3%			240	0%
150	2.6%				
160	–				
170	–				

Table 4.4: ASL values for the null hypothesis “ECBS is not more efficient than ECBS+HM” for the Kiva environment and “ECBS+HM is not more efficient than ECBS” for the Drones and Roundabout environments for each number of agents. “–” appears in all entries for which the success rate of ECBS+HM is 0%.

Kiva env.		Drones env.		Roundabout env.	
Number of agents	ASL	Number of agents	ASL	Number of agents	ASL
80	0%	80	19.2%	120	20.5%
90	0%	90	46.5%	140	18.8%
100	0%	100	49.9%	160	5.5%
110	0%	110	45.2%	180	36.7%
120	0%	120	38.1%	200	6.3%
130	0%			220	9.6%
140	0%			240	8.5%
150	0%				
160	0%				
170	0%				

Table 4.5: ASL values for the null hypothesis “ECBS+HWY is not more efficient than ECBS+HM” for each environment and number of agents.

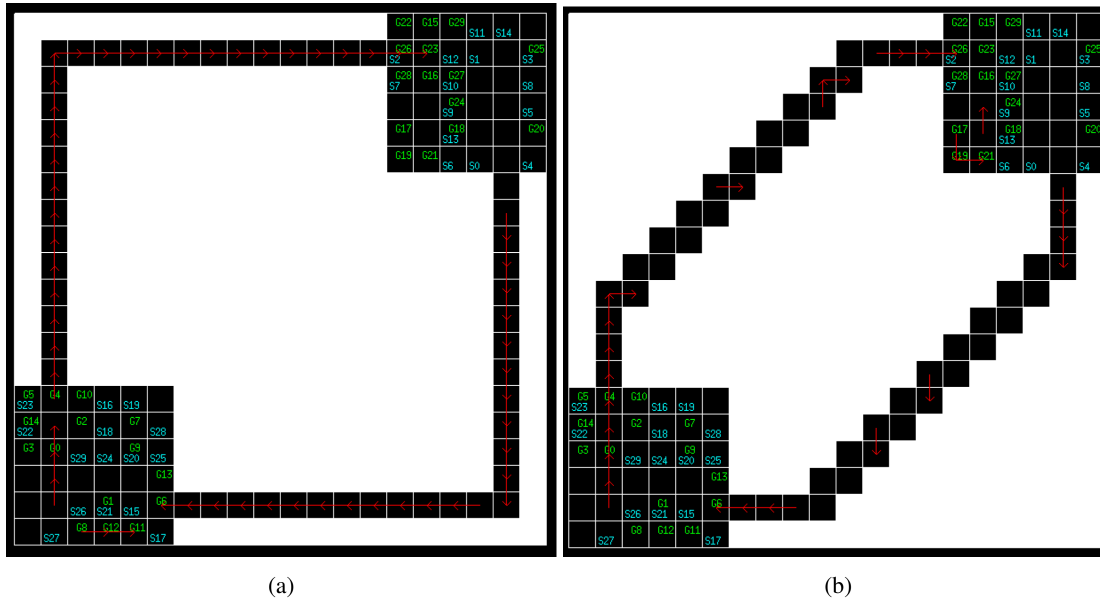
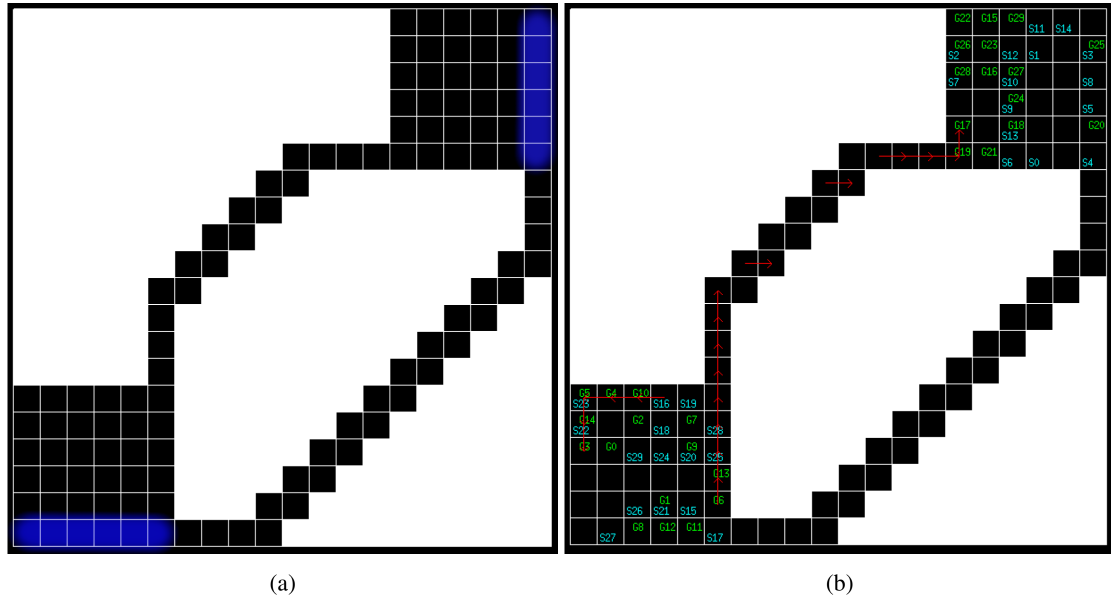


Figure 4.28: Shows two environments with the same start and goal cells of the agents (in cyan and green, respectively), and their GM-based highways (in red). In (a), straight corridors connect open spaces, and the GM-based highways utilize (almost) all of the edges in the corridors. (The only cell in the corridor without a highway edge is the one in which the corridor is not straight.) In (b), zig-zag corridors connect open spaces, and the GM-based highways do not utilize many edges in the corridors.

4.6.4 Limitations

We now discuss the limitations of the GM- and HM-based approaches. One limitation of both approaches is their difficulty in generating non-straight highways. Figure 4.28 illustrates this limitation. In the GM-based approach, the weight associated with any two consecutive highway edges that are orthogonal to each other (for example, $f_H(\text{East,North}) = 1$) is relatively small compared to the weight associated with two consecutive highway edges in the same direction (for example, $f_H(\text{East,East}) = 5$). Therefore, the first limitation is a consequence of the way the factors of the GM (Figure 4.1) are defined. In the HM-based approach, the first limitation is a consequence of the lack of incentives to choose consecutive edges (in any direction).

Another limitation of both approaches is their difficulty in generating highways that pass through areas in the environment that are not utilized by many agents. Figure 4.29 illustrates this limitation. In both approaches, this stems from the lack of abstract reasoning about the environment (for example, recognizing



(a) (b)

Figure 4.29: Shows an environment with a central corridor and a peripheral corridor. Assuming that all the start and goal cells are not inside corridors, only the optimal paths for the start and goal cells on the rightmost column or the bottommost row (corresponding to the blue colored cells in (a)) may utilize the peripheral corridor, and all other optimal paths necessarily utilize the central corridor. In (b), the start and goal cells (in cyan and green, respectively) from Figure 4.28 are specified. Only agent 8 may utilize the peripheral corridor. Red arrows show the GM-based highways, which, as expected, do not utilize the peripheral corridor.

open spaces and passageways connecting them). In other words, the second limitation is a consequence of looking only at the optimal paths.

4.7 Conclusions and Future Work

In this chapter, we observed that ECBS with $h_{\texttt{SP}}$ can be inefficient in environments with certain structures. We proposed to use the idea of highways from transportation, developed the $h_{\texttt{HWY}}$ heuristic and proved that ECBS with $h_{\texttt{HWY}}$ is bounded-suboptimal. In experiments, we showed that ECBS with $h_{\texttt{HWY}}$ is more efficient than ECBS with $h_{\texttt{SP}}$ in three environments, each having a different structure. We also developed two approaches for generating highways automatically and showed in experiments that they often produce useful highways.

The actual suboptimality of ECBS with $h_{\texttt{HWY}}$ is often much less than the bound $w_1 w_2$. This phenomenon has also been observed in the context of wA* [31]. In future work, it can be interesting to identify the causes of this difference and develop corrections for them. Given a suboptimality budget $w \geq$, it can also be useful to find ways to automatically choose w_1 and w_2 such that $w = w_1 w_2$ and ECBS with $h_{\texttt{HWY}}$ is expected to be efficient. In this chapter, we took the approach of heuristic inflation for using highways. However, one can instead inflate the costs of actions that do not traverse highway edges. We expect the efficiency of cost inflation to be quite similar to that of heuristic inflation, although a different proof is required to guarantee the bounded-suboptimality. Finally, it can be interesting to develop other approaches to automatically generate highways that address the limitations outlined in Section 4.6.4.

Chapter 5

Multi-Agent Motion Planning

Multi-Agent Motion Planning (MAMP) is the task of finding collision-free kinodynamically feasible plans for agents from start to goal configurations. While MAMP is of significant practical importance, existing solvers are either incomplete, inefficient or rely on simplifying assumptions. For example, MAPF solvers conventionally assume discrete timesteps and rectilinear movement of agents between neighboring vertices of a graph. In this chapter, we develop MAMP solvers that obviate these simplifying assumptions and yet generalize the core ideas of state-of-the-art MAPF solvers like ECBS. One of the characteristics of MAMP, which makes it combinatorially different from MAPF, is that different motions may take arbitrarily different durations. Thus, MAMP solvers need to reason efficiently about arbitrary wait durations. To do so, we adapt FS to reason about wait actions in bulk and develop an efficient implementation of this adaptation by using a contiguous set of timesteps (rather than a single timestep) in a state's description. On the theoretical side, we justify the completeness, optimality and bounded-suboptimality of our MAMP solvers. On the experimental side, we show that our MAMP solvers become more efficient as their suboptimality bounds increase.

This chapter is organized as follows: In Section 5.1, we discuss the limitations of the MAPF problem and outline our contributions. In Section 5.2, we provide relevant background on state lattices and safe interval path planning. In Section 5.3, we develop a variant of FS that reasons about wait actions in bulk and analyze it. In Section 5.4, we formally define the MAMP problem as a generalization of the MAPF

problem. In Section 5.5, we describe ECBS-CT, a bounded-suboptimal solver for the MAMP problem. Finally, in Section 5.6, we provide experimental results to evaluate the efficiency of ECBS-CT.

5.1 Introduction

Multi-Agent Motion Planning (MAMP) is the task of finding collision-free kinodynamically feasible paths for agents in a shared environment. Each agent has a unique start configuration and a unique goal configuration. Real-world applications of MAMP include autonomous aircraft towing vehicles [52], autonomous non-holonomic vehicles (such as forklifts) in industrial applications [10] and traffic management systems for unmanned drones [63].

While MAMP is of significant practical importance, existing formulations rely on simplifying assumptions. For example, the MAPF problem uses a formulation in which (1) the environment is captured by a graph with vertices representing cells and edges representing straight-line movements between vertices; and (2) time is discretized into synchronized timesteps. Although the MAPF problem is motivated by real-world applications, its solution may not be kinodynamically feasible for real agents, such as cars or drones, for two major reasons. First, rectilinear movements cannot be executed by agents with non-holonomic constraints. Second, different agents may have different motions of arbitrarily different durations and thus cannot be easily synchronized. To partially alleviate this problem, one can post-process a MAPF solution using simple temporal networks and continuous refinement of trajectories [28, 29]. However, these methods may produce arbitrarily suboptimal plans even if they post-process optimal MAPF solutions.

One way to address the limitations of the post-processing strategy is to work with an enriched formulation of the MAPF problem. A first step in this direction is the formulation of the MAPF$_R$ problem [87], which is identical to the MAPF problem but allows positive non-uniform edge costs to represent different action durations. However, in MAPF$_R$, vertices still represent cells in metric space and movements are still rectilinear with uniform velocities.

In this chapter, we formulate a richer MAMP problem in which vertices represent configurations and directed edges represent kinodynamically feasible motions. A configuration corresponds to a point in the configuration space of an agent. For example, it may include the (x, y, z)-coordinates, orientation, steering angle, velocity and other features that characterize it. An edge from one configuration to another configuration corresponds to a feasible motion between them. The cost of the edge represents the duration of the motion. The environment is discretized into cells. Each edge specifies a list of *swept* cells, where each cell has an associated contiguous set of timesteps (henceforth referred to as a timeinterval) during which the agent executing the motion occupies it. As a consequence, in our formulation, an agent is allowed to be of any geometric shape. Moreover, our formulation facilitates efficient collision detection regardless of the shape of the agent. Finally, our formulation naturally lends itself to reasoning over state lattices [60], probabilistic roadmaps (PRMs) [34] and rapidly exploring random trees (RRTs) [39].

Computationally, the richer formulation of the MAMP problem is more challenging, and all current solvers for it are either incomplete or inefficient [10, 67, 66]. In this chapter, we develop a significantly more efficient and provably effective MAMP solver by drawing inspiration from the success of CBS. While CBS is not directly applicable to the MAMP problem, we successfully extend its core ideas to the richer MAMP domain. To do so, (1) we adapt ECBS to reason about wait actions in bulk; (2) we introduce an efficient implementation of reservation tables using interval maps [15]; and (3) we develop Soft Collision Interval Path Planning (SCIPP), an efficient implementation of (1) which operates with timeintervals instead of timesteps. We also show that SCIPP can be thought of as a bounded-suboptimal version of Safe Interval Path Planning (SIPP) [59].

On the theoretical side, we justify the completeness, optimality and bounded-suboptimality of our MAMP solvers on state lattices. On the experimental side, we show that our MAMP solvers become more efficient as their suboptimality bounds increase.

5.2 Background

In this section, we provide relevant background on motion primitives, state lattices and SIPP.

5.2.1 Motion Primitives and State Lattices

A note about terminology: In this subsection, we maintain the commonly used term "state lattice". This term is not to be confused with our usage of the term "state", which includes the time dimension. The term "configuration lattice" is more consistent with our terminology, but we keep the commonly used term "state lattice" in this section in order to be consistent with its usage in the cited literature.

Kinodynamic motion planning (henceforth referred to as motion planning) is the task of moving an agent from an initial configuration to a goal configuration while avoiding obstacles present in the environment and obeying the kinematic and dynamic constraints[1] of the given agent. A kinodynamic motion planner explicitly considers the agent's kinodynamic constraints during the planning process. Such a planner produces trajectories that are both collision-free with respect to the environment and feasible with respect to the agent's model.

State lattices [60] are extensions of grids that are able to model kinodynamic constraints and are therefore well suited to planning for agents with limited maneuverability (such as non-holonomic mobile robots). A state lattice is constructed by discretizing the configuration space into a high-dimensional grid and connecting the cells of the grid with motion primitives. A motion primitive models kinodynamically feasible motions of the agent. A configuration in a state lattice is a tuple of the form $(x, y, z, \theta, v, \ldots)$, where x, y and z are the coordinates of the agent's center, θ is the agent's orientation, v is the agent's velocity, etc. An edge in a state lattice represents a motion primitive and is associated with a duration and a list of cells that are swept by the agent when the motion is executed. Motion primitives have successfully been used for autonomous cars in DARPA's urban challenge [19] and quadrotors [46]. A state lattice facilitates

[1] Kinematic constraints define the types of motions an agent can perform. For example, a car-like robot cannot move in a direction that is perpendicular to its wheels. Dynamic constraints define how motions develop over time. For example, a car-like robot cannot accelerate and decelerate faster than some bounds.

the application of heuristic search algorithms to finding optimal or bounded suboptimal trajectories[2] by concatenating motion primitives.

5.2.2 Safe Interval Path Planning

SIPP [59, 54] is a search-based single-agent path planner designed to handle dynamic obstacles efficiently. In SIPP, each configuration is associated with a fixed list of *safe timeintervals* during which an agent in that configuration does not collide with any dynamic obstacles. Timeintervals allow SIPP to reason about wait durations "in bulk," making it more efficient than A* in the presence of arbitrary wait durations. SIPP has already been successfully used for Multi-Agent Any-Angle Path Finding [92] and Multi-Agent Pickup and Delivery problems [48].

5.3 Focal Search with Reasoning About Wait Actions in Bulk

In this section, we present FS-B, an adaptation of FS that reasons about wait actions in bulk. FS-B can be more efficient than FS in environments with dynamic obstacles when the action costs (that is, action durations) are arbitrary, which is the case for the MAMP problem that we define in Section 5.4. We prove that FS-B is w-suboptimal and that ECBS-B, a version of ECBS that replaces the low-level FS with FS-B, is also w-suboptimal. The w-suboptimality of ECBS-B is key to arguing for the w-suboptimality of the multi-agent motion planner we present in Chapter 5.5.

We start by proving a helpful lemma for FS. Throughout this chapter, we use $g^*(s)$ to denote the optimal g-value of state s, that is, the optimal cost of a path from the start state to s. We assume that $g(s) = \infty$ before s is first generated and that FS uses a consistent heuristic.

Lemma 22. *When FS expands a state, if any state s (not necessarily the state that FS currently expands) is reachable from the start state and has not been expanded with an optimal g-value, then there exists a state $s' \in \texttt{OPEN}$ with optimal g-value that is on an optimal path from the start state to s.*

[2] optimally with respect to the state lattice discretization

Proof. Consider any state s reachable from the start state that has not been expanded with an optimal g-value. If $g(s) = g^*(s)$, it must be the case that $s \in \texttt{OPEN}$ because the premise asserts that s has not been expanded with an optimal g-value and, since s is on an optimal path from the start state to s, the lemma holds. Otherwise, $g(s) > g^*(s)$. WLOG, we show that there exists a state s' that is the deepest state on an optimal path from the start state to s that is in OPEN and has an optimal g-value. We prove the lemma by induction on the number of expansions. Before the first expansion, only the start state is in OPEN with an (optimal) g-value of zero. Since the start state is on an optimal path from the start state to s, the lemma holds. Assume that the lemma holds after i expansions. If the state chosen for the $(i+1)$st expansion is not s', the lemma still holds. If the state chosen for the $(i+1)$st expansion is $s' = s$, it is expanded with an optimal g-value and the premise no longer holds. If the state chosen for the $(i+1)$st expansion is $s' \neq s$, at least one of the successors of s', s'', is on an optimal path to s because the optimal path from the start state to s that goes through s' has to continue through at least one of the successors of s'. It also must be the case that $g(s'') = g(s') + c(s', s'') = g^*(s'')$ because, otherwise, there would be a path from the start state to s'' not through s' with a lower cost than the path from the start state to s'' through s'. This, in turn, entails that there would be a lower-cost path from the start state to s not through s', which contradicts our assumption. Finally, s'' is inserted into OPEN with an optimal g-value because s' is the deepest state on an optimal path from the start state to s that is in OPEN and has an optimal g-value. Therefore, $s'' \in \texttt{OPEN}$ with an optimal g-value and on an optimal path to s, and the lemma holds after $i + 1$ expansions as well.

□

The following corollaries are a consequence of Lemma 22.

Corollary 1. *When FS expands a state, $f_{\min} \leq$ OPT, where OPT is the cost of an optimal solution.*

Proof. If there is no path from the start state to any goal state, OPT $= \infty$ and the corollary holds. We now show that the corollary holds when there is a path from the start state to at least one goal state (that is, OPT $< \infty$). First, FS terminates as soon as any goal state is expanded. Thus, before each expansion that FS makes, it must be the case that no goal state has been expanded. Second, consider any optimal

goal state s_{opt} (that is, $g^*(s_{opt}) = \text{OPT}$). According to Lemma 22, there exists an $s' \in \texttt{OPEN}$ with an optimal g-value that is on an optimal path to s_{opt}. Since h is consistent, $f(s') = g^*(s') + h(s') \leq g^*(s_{opt}) + h(s_{opt}) = g^*(s_{opt}) = \text{OPT}$. Since $f_{\min} \leq f(s')$, $f_{\min} \leq \text{OPT}$ and the corollary holds. □

Corollary 2. *When FS expands a goal state, its g-value is bounded by wOPT, and therefore a w-suboptimal path is returned.*

Proof. FS only expands states having f-values of up to $wf_{\min}$. Since we assume that h is consistent (thus, $h = 0$ for any goal state) and $f_{\min} \leq \text{OPT}$, the corollary holds. □

We now discuss FS-B, which is similar to FS with one main difference: it aggregates decisions on wait actions. FS-B is presented for conceptual purposes only, and we will discuss an efficient method to implement it in Chapter 5.5.3. Nevertheless, analyzing FS-B is important for showing that ECBS with FS-B instead of FS as its low-level search is w-suboptimal, which is key to showing that the MAMP solver that we describe in Chapter 5.5 is w-suboptimal. Since FS-B is designed for pathfinding with dynamic obstacles, we use a specialized state notation, $s = (v, t)$ (that is, being in configuration v at timestep t). We also assume that costs equal durations, that is, $g(s) = t$. Thus, we simplify the pseudocode of Algorithm 4 and do not explicitly specify g-values. Algorithm 4 also avoids reopening states because all paths to s have cost t.[3]

FS-B assumes that every action has a duration that can be represented with some finite precision. More formally,

Assumption 1. *There exists an $\epsilon > 0$ such that any action (that is, motion or wait) duration is an integer multiple of ϵ.*

First, Assumption 1 is not very limiting. For example, a 64-bit unsigned integer can count up to 3481 years for $\epsilon = 1$ microsecond. Second, MAPF solvers often make this assumption implicitly because assuming that all actions have unit costs is equivalent to setting their costs to ϵ. However, existing MAPF

[3] It can still make sense to reopen states with similar t values if the reopened state has a better $h_{\texttt{FOCAL}}$, that is, represents a path with fewer collisions. This affects efficiency but not correctness, and we omit it for brevity.

solvers can be extremely inefficient when actions costs are arbitrarily different because they reason about wait actions "one ϵ at a time". Figure 5.1 illustrates this inefficiency. For the rest of this chapter, we simply use timesteps, where the duration of each timestep is ϵ.

FS-B also assumes that the dynamic obstacles are defined by `constraints`. The `constraints` data structure returns true when there is a collision with a dynamic obstacle (that is, when a constraint is violated). More specifically, $(v, t) \in$ `constraints` iff the agent being in configuration v at timestep t violates a constraint, and $(v, t; v', t') \in$ `constraints` iff the agent moving from configuration v at timestep t to configuration v' at timestep t' violates a constraint. Figure 5.2 shows three constraints as red dashed lines. A red dot in the middle of the dashed line represents a constraint on $(v, t; v', t')$, a red dot on top of the dashed line represents a constraint on (v', t'), and a red dot on the bottom of the dashed line represents a constraint on (v, t). In Section 5.5, we explain the rationale behind `constraints` and discuss an efficient data structure for it.

Algorithm 4 shows FS-B. It is similar to FS applied to path finding in environments with dynamic obstacles with the following additions:

(1) *Bulk generation* (lines 32-36 and lines 3-7): For any generated state (v, t), recursively generate $(v, t + 1)$ if it is possible to wait in v (that is, the agent's velocity is zero in this configuration, represented by `canwait(`v`)`) and being in v at timestep $t + 1$ does not violate any constraint.[4]

(2) *Bulk expansion* (lines 12-15): For any expanded state (v, t), recursively expand $(v, t + 1)$ if $(v, t + 1) \in$ `FOCAL`.

Theorem 23. *FS-B satisfies Lemma 22.*

Proof. We only need to show that the two additions that FS-B adds to FS, namely, bulk generation and bulk expansion, do not invalidate Lemma 22. Bulk generation does not expand any state and therefore does not remove any state from `OPEN`. Moreover, the g-value of any generated state is updated only when

[4] Note that lines 3-7 are also bulk generation (specifically, for the generated start state $(v_{start}, 0)$ in which we assume the velocity of the agent is zero and thus waiting is possible. This assumption is easy to change).

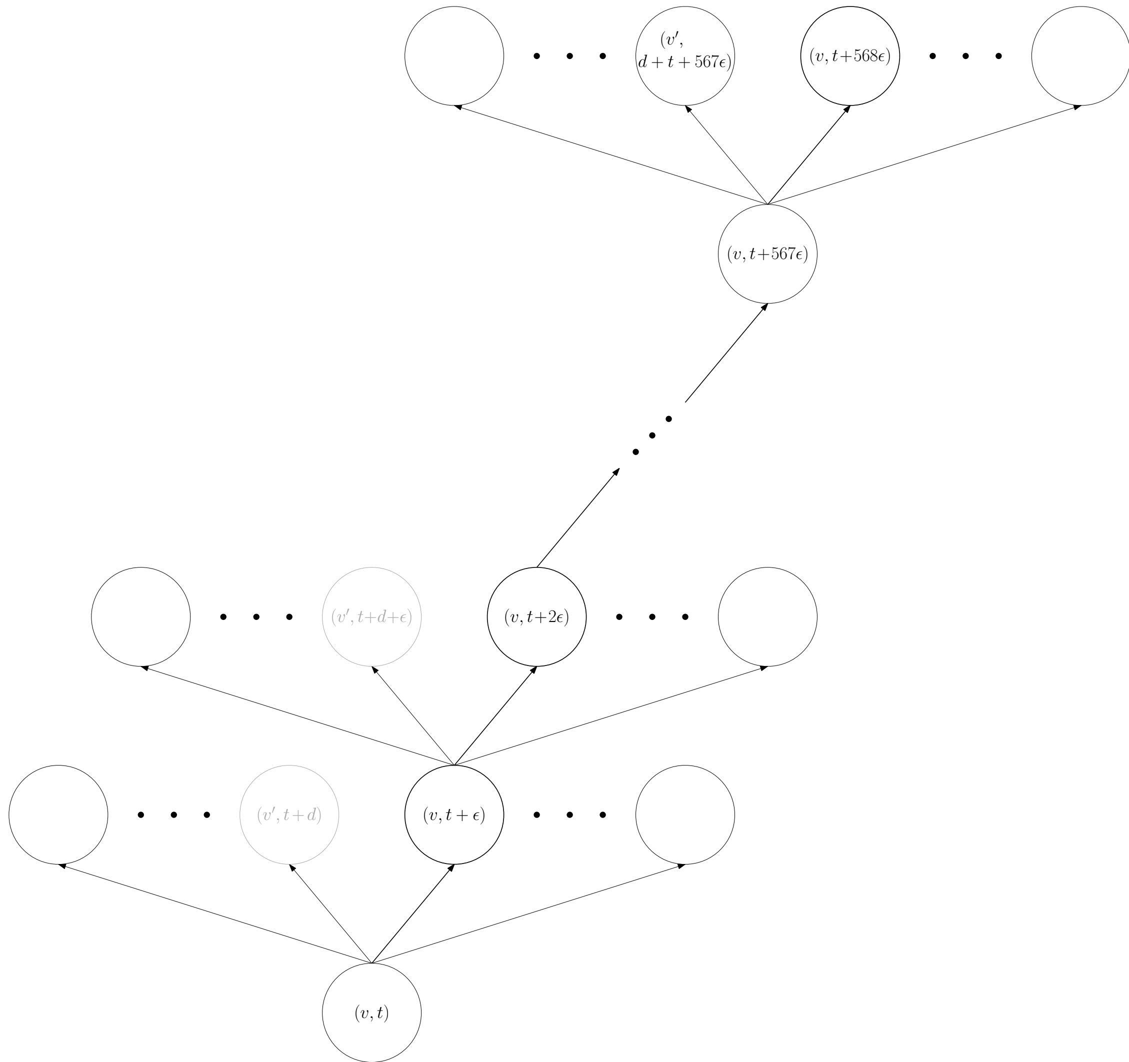


Figure 5.1: Illustrates the inefficiency of reasoning about wait actions "one ϵ at a time". Assume that any path to the goal configuration involves the action (v, v') and that $\{(v, t, v', t + d), (v, t + \epsilon, v', t + d + \epsilon), \ldots, (v, t + 566\epsilon, v', t + d + 566\epsilon)\} \in$ `constraints`. When FS expands the state (v, t), it can only generate $(v, t + \epsilon)$ for a (single) wait action. Thus, FS has to expand and generate 567 states (and possibly many other states) in order to find a path to state $(v, t + 567\epsilon)$, from which the actions (v, v') does not violate any constraint.

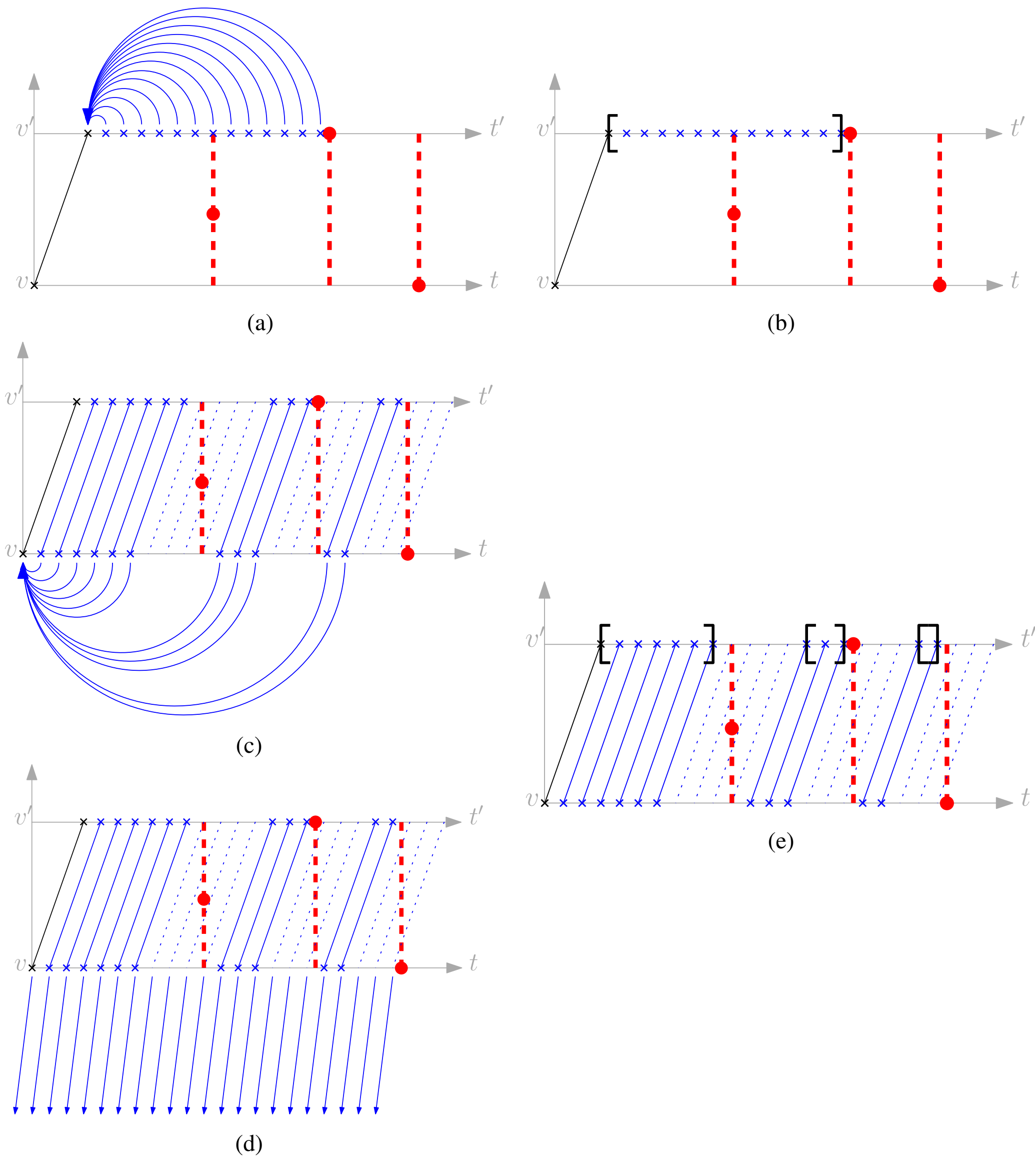


Figure 5.2: Illustrates bulk generation and bulk expansion. The x-axis represents time, and the y-axis represents configurations. Crosses represent states, and lines represent actions between states. Black color represents traditional expansions or generations, and blue color represents additional expansions or generations that are done in bulk. Red dotted lines represent constraints. A red dot on the horizontal line for v (v') represents a constraint (v, τ) ((v', τ)) at the time τ of its x-coordinate, and a red dot between v and v' represents a constraint (v, t, v', t') at the time $\tau \in [t, t']$ of its x-coordinate. (a) illustrates bulk generated states in blue crosses. Each state is generated using wait actions after reaching v', which the parent pointers in blue represent. (b) illustrates that it is possible to aggregate all states generated in (a) using timeintervals (see Chapter 5.5.2). (c) and (d) illustrate bulk expanded states in blue crosses. Each state is generated by either: waiting in v before executing the action (illustrated in (c)) or arriving at v later and executing the action immediately (illustrated in (d)). Any combination of arriving at v later and waiting in v is also possible (this amounts to different paths of getting to v' with the same g-value). (e) illustrates that it is possible to aggregate all states generated in (c) and (d) using timeintervals (see more in Chapter 5.5.2).

Algorithm 4: FS-B.

v_{start} is the start configuration; v_{goal} is the goal configuration; $\texttt{succ}(s)$ returns all successors of state s (more specifically, $(v', t') \in \texttt{succ}((v, t))$ iff $(v, v') \in E$, $t' = t + c(v, v')$ and $(v, t, v', t') \notin \texttt{constraints}$); $\texttt{constraints}$ is a set of constraints representing the dynamic obstacles in the environment; and w is the suboptimality bound. Blue text highlights the additions that FS-B adds to FS for reasoning about wait actions in bulk. The *updateLowerBound* procedure is defined in Algorithm 1 and omitted here for brevity.

Input: $v_{start}, v_{goal}, \texttt{succ}(s), \texttt{constraints}, w$.
Output: A w-suboptimal path.

```
1  OPEN = FOCAL = {(v_start, 0)}
2  CLOSED = {}
3  f_min ← f((v_start, 0))
4  t = 1
5  while (v_start, t) ∉ constraints do
6      generate(v_start, t)
7      t = t + 1
8  while FOCAL ≠ ∅ do
9      f_min ← f(head(OPEN))
10     (v, t) ← head(FOCAL)
11     expand(v, t)
12     t = t + 1
13     while (v, t) ∈ FOCAL do
14         expand(v, t)
15         t = t + 1
16     if OPEN ≠ ∅ and f_min < f(head(OPEN)) then
17         updateLowerBound(w f_min, w f(head(OPEN)))
18 return "no solution exists"
19 Procedure generate(v, t):
20     if (v, t) ∉ OPEN and (v, t) ∉ CLOSED then
21         OPEN ← OPEN ∪ {(v, t)}
22         if f((v, t)) ≤ w f_min then
23             FOCAL ← FOCAL ∪ {(v, t)}
24 Procedure expand(v, t):
25     FOCAL ← FOCAL \ {(v, t)}
26     OPEN ← OPEN \ {(v, t)}
27     CLOSED ← CLOSED ∪ {(v, t)}
28     if v = v_goal and t > latest constraint on v then
29         return path      // Terminates the algorithm, not just this procedure.
30     for each (v', t') ∈ succ((v, t)) do
31         generate(v', t')
32         if canwait(v') then
33             t' = t' + 1
34             while (v', t') ∉ constraints do
35                 generate(v', t')
36                 t' = t' + 1
```

it improves, and thus if a state in OPEN has an optimal g-value, it will not change. Therefore, the addition of bulk generation does not invalidate Lemma 22. An identical proof to the one given for Lemma 22 can be given for any state expanded by bulk expansion. □

Since FS-B satisfies Lemma 22, Corollaries 1 and 2 also hold for FS-B. We now show that ECBS-B, an algorithm similar to ECBS that uses FS-B instead of FS as its low-level search, is also w-suboptimal.

Theorem 24. *When Assumption 1 holds, ECBS returns a w-suboptimal solution if one exists.*

Proof. This is the original proof of ECBS [4]. This proof relies on two properties that the low-level search needs to satisfy:

1. When the low-level search expands a goal state, $f_{\min} \leq$ OPT (Corollary 1).

2. The low-level search terminates with a w-suboptimal path (Corollary 2).

□

Theorem 25. *ECBS-B, an algorithm similar to ECBS that uses FS-B instead of FS as its low-level search, returns a w-suboptimal solution if one exists.*

Proof. Since Corollaries 1 and 2 hold for FS-B, the proof in [4] is also applicable to ECBS-B. □

5.4 The Multi-Agent Motion-Planning Problem

In this section, we define the MAMP problem as a generalization of the MAPF problem. Thus, the objective of minimizing the total arrival time in the MAMP problem, as defined below, is also NP-hard. Unlike the MAPF problem, the MAMP problem is posed on configurations instead of cells. A configuration specifies discretized values of an agent's location, orientation, velocity, etc. An edge represents a kinodynamically feasible motion of a given arbitrary duration. The sequence of motions (henceforth referred to as actions) in a feasible plan (henceforth referred to as a feasible path) leads an agent from its

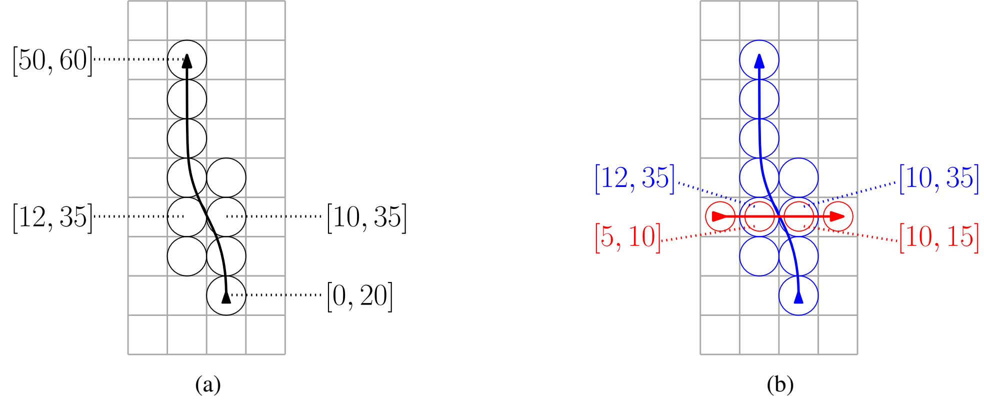


Figure 5.3: (a) Illustrates an action from v (depicted by the bottom triangle) to v' (depicted by the top triangle) with a duration of 60. Denote by $e = (v, v')$ the edge representing this action. Thus, $w(e) = 60$. Each swept cell c_i^e is depicted by a circle, and $[lb_i^e, ub_i^e]$ is shown for four swept cells. Note that $\min_i lb_i^e = 0$ and $\max_i ub_i^e = 60$. (b) Illustrates two actions, red and blue, colliding at one swept cell since $[10, 15] \cap [10, 35] \neq \emptyset$.

start configuration to its goal configuration. An agent is allowed to be of any geometric shape, implicitly specified by a set of occupied cells. We formally define the MAMP problem as follows.

Definition 26 (*The MAMP problem*)**.** *We are given an environment represented by a set of cells $\mathcal{C}$. We are given agents $1, \ldots, K$, each with an associated graph $G^j = (V^j, E^j)$ and start and goal configurations, $s^j, g^j \in V^j$. Each vertex $v \in V^j$ represents a configuration and is associated with a list of cells $\mathcal{C}^v = \{c_1^v, \ldots, c_{m(v)}^v\} \subseteq \mathcal{C}$ occupied by the agent while in v. Each edge $e \in E^j$ represents an action and has an associated cost $w(e) > 0$, that represents its duration. e is also associated with a multiset of cells $\mathcal{C}^e = \{c_1^e, \ldots, c_{m(e)}^e\} \subseteq \mathcal{C}$. Each cell c_i^e is associated with a timeinterval $[lb_i^e, ub_i^e]$ during which it is swept by agent j after the start of execution of the action represented by e. Thus, $\min_{1 \leq i \leq m(e)} lb_i^e = 0$ and $\max_{1 \leq i \leq m(e)} ub_i^e = w(e)$.*

Figure 5.3(a) illustrates an edge representing some action along with its swept cells.

Definition 27 (*A feasible path*)**.** *A sequence $\pi^j = \{\langle e_1^j, t_1^j \rangle, \ldots, \langle e_{T_j}^j, t_{T_j}^j \rangle\}$ is a path for agent j, where $e_i^j = (v_{i-1}^j, v_i^j) \in E^j$ and $t_i^j \geq 0$ is the beginning time of the execution of e_i^j. π^j is* feasible *iff $e_1^j \ldots, e_{T_j}^j$ is a path from s^j to g^j in G^j and, for all $i \in \{2, \ldots, T_j\}$, $t_i^j \geq t_{i-1}^j + w(e_{i-1}^j)$. This means that agent j waits at configuration v_{i-1}^j and therefore occupies cells $\{c_1^v, \ldots, c_{m(v)}^v\}$ for $v = v_{i-1}^j$ during the timeinterval*

$[t^j_{i-1} + w(e^j_{i-1}), t^j_i]$.[5] *We assume that agent j occupies $\{c^v_1, \ldots, c^v_{m(v)}\}$ for $v = g^j$ during the timeinterval $[t^j_{T_j} + w(e^j_{T_j}), \infty]$.*

Definition 28 (*A collision*)**.** *A* collision *between two agents occurs iff the timeintervals in which they sweep or occupy the same cell overlap.*

Figure 5.3(b) illustrates two actions that collide.

Definition 29 (*A solution to the MAMP problem*)**.** *A* solution *to the MAMP problem is a set of feasible paths so that no two agents collide.*

Definition 30 (*The cost of a solution*)**.** *The travel time of agent j with path π^j is given by $\tau(j) = t^j_{T_j} + w(e^j_{T_j})$. The cost of the MAMP solution is the total arrival time, that is, $\sum_{j=1}^{K} \tau(j)$.*

Note that Definitions 26 - 29 are generalizations of Definitions 1 - 4 and Definition 30 is a generalization of Definition 5(1). We omit the generalization of Definitions 5(2)-(4) because we focus on the total arrival time in this chapter.

In the rest of this chapter, we consider a state lattice representation of the MAMP problem, even though one could also consider a PRM or RRT representation. Finally, if safety considerations are an issue, one can either inflate the footprint of the agent (which is characterized by the swept cells) or prolong the duration in which swept cells are occupied by the agent (which is characterized by the timeintervals associated with the swept cells).

5.5 Adapting Conflict-Based Search to Multi-Agent Motion Planning

In this section, we present ECBS-CT,[6] a generalization of ECBS to the MAMP problem. Algorithm 5 shows the pseudocode for the high-level search of ECBS-CT. It takes as input a MAMP problem instance

[5] Waiting can be conditioned on the velocity being zero.

[6] CT stands for contiguous timesteps.

and a suboptimality bound $w \geq 1$. ECBS-CT generates a solution that has a cost of no more than w times the optimal cost. Thus, for $w = 1$, ECBS-CT is optimal and essentially generalizes CBS to the MAMP problem.

Algorithm 5: ECBS-CT (High-Level Search)

```
   Input: A MAMP problem instance, w ≥ 1.
   Output: A w-suboptimal MAMP solution.
 1 Initialize the root state, N_start, with a path for each agent using SCIPP
 2 OPEN = FOCAL = {N_start}
 3 while True do
 4     f_min ← f(head(OPEN))
 5     N ← head(FOCAL)
 6     FOCAL ← FOCAL \ {N}
 7     OPEN ← OPEN \ {N}
 8     if N is a solution then
 9         return N
10     Identify a collision (c, [lb, ub]) between agents j and k at cell c ∈ C during timeinterval
       [lb, ub].
11     Identify a timestep τ ∈ [lb, ub].
12     Generate two successor states, N^j (imposing the additional constraint (c, τ) on agent j) and
       N^k (imposing the additional constraint (c, τ) on agent k).
13     Replan using SCIPP for agents j and k in N^j and N^k.
14     Update the reservation tables in N^j and N^k.
15     if SCIPP did not return "no solution" when replanning for agent j then
16         OPEN ← OPEN ∪ {N^j}
17         if f(N^j) ≤ w f_min then
18             FOCAL ← FOCAL ∪ {N^j}
19     if SCIPP did not return "no solution" when replanning for agent k then
20         OPEN ← OPEN ∪ {N^k}
21         if f(N^k) ≤ w f_min then
22             FOCAL ← FOCAL ∪ {N^k}
23     if OPEN ≠ ∅ and f_min < f(head(OPEN)) then
24         updateLowerBound(w f_min, w f(head(OPEN)))
25 Procedure updateLowerBound(old_b, new_b):
26     for each n ∈ OPEN do
27         if (f(n) > old_b) ∧ (f(n) ≤ new_b) then
28             FOCAL ← FOCAL ∪ {n}
```

On lines 1-2, the high-level root state is initialized using a low-level search for each agent. In ECBS-CT, the low-level search uses SCIPP. The main loop on lines 3-24 performs a FS. On lines 23-24, FOCAL is appropriately updated to include all relevant states from OPEN if $f_{\min}$ increases. The secondary heuristic

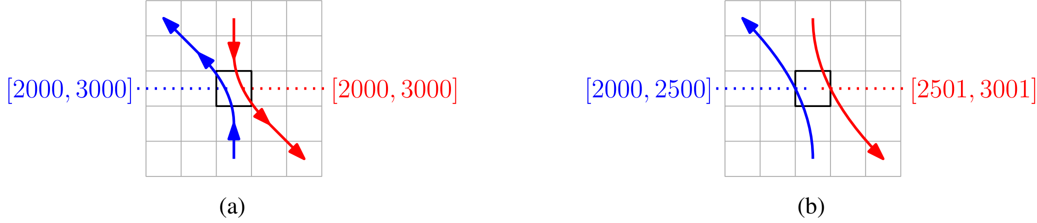


Figure 5.4: (a) Shows two agents, red and blue, colliding in a swept cell c during $[2{,}000, 3{,}000]$. (b) A constraint $(c, [2{,}000, 3{,}000]$ forbids an optimal path in which the blue agent sweeps c during $[2{,}000, 2{,}500]$ and the red agent sweeps c during $[2{,}501, 3{,}001]$.

value of a generated high-level state is defined to be the total duration in which two or more agents collide. As discussed in the next subsection, it can be efficiently computed while the reservation table is updated on line 14.

On line 10, a collision at cell c starting at timestep lb and ending at timestep ub, $(c, [lb, ub])$, between two agents j and k is identified. We resolve the earliest collision among all agents in high-level state N. As discussed in the next subsection, this earliest collision can be efficiently identified using the reservation table. Resolving the earliest collision is known to be beneficial in the CBS framework [69]. The collision is resolved on line 12 by posting a constraint on cell c at a timestep $\tau \in [lb, ub]$. This constraint (c, τ) prohibits the corresponding agent from being at cell c at timestep τ. Two questions that arise in this context are: (1) “Why is a timestep used instead of a timeinterval?” and (2) “What should the value of τ be?”

The answer to the first question relates to the requirements of CBS (ECBS) to guarantee optimality (w-suboptimality). The proof of optimality (w-suboptimality) relies on the property that any solution which obeys the constraints of a high-level state N also obeys the constraints of at least one of its high-level child states N^j or N^k. This is directly analogous to the proofs of Lemma 2 in [69] and Theorem 1 in [4]. However, if the constraint specifies a timeinterval $[lb', ub'] \subseteq [lb, ub]$ instead of a timestep, this property no longer holds. In particular, if an optimal solution includes agent j being at c during timeinterval $[lb', (lb' + ub')/2]$ and agent k being at c during timeinterval $[(lb' + ub')/2 + 1, ub']$, it is spuriously eliminated. Figure 5.4 illustrates this case.

The answer to the second question relates to Zeno behaviors [96]. Zeno behavior is a phenomenon in hybrid (discrete-continuous) systems, where the system undergoes an infinite number of discrete transitions in a finite duration of time. Specifically in our case, if $\tau < ub$, agents j and k can satisfy their respective constraints by waiting for $\tau - lb$ time units before colliding again at c during the timeinterval $[\tau + 1, ub]$. Similarly, in a future iteration, they may collide yet again during the timeinterval $[\tau' + 1, ub]$ for some $ub > \tau' > \tau$. Therefore, any strategy for choosing the value of τ other than setting it to the upper bound of the timeinterval (ub) results in a behavior that is similar to Zeno behavior, especially when the duration of a timestep (that is, ϵ) is small. Figure 5.5 illustrates different choices of τ.

Finally, although a constraint (c, τ) involves only a single timestep, it blocks a timeinterval of arrival times. More specifically, if an action $e = (v, v')$ of duration d sweeps c during $[t_1, t_2]$, it forbids the agent from getting to v' using e during $[\tau - t_2 + d, \tau - t_1 + d]$. Figure 5.6 illustrates the blocked timeinterval that corresponds to (c, τ).

5.5.1 Reservation Table

In ECBS, the reservation table is simply a set of discrete timesteps specified for each cell in the environment during which this cell is occupied by some agent. The low-level search queries the reservation table for the number of collisions in cell v at timestep t for every generated state (v, t). Since numerous states are generated[7], the reservation table has to support such queries efficiently. Thus, the reservation table is often implemented as an array of integers that stores the number of collisions for every cell v and timestep t.

In ECBS-CT, the discrete timesteps are replaced by timeintervals. Since the duration of a timestep (that is, ϵ) can be rather small, using an array is infeasible and, instead, we use timeintervals. A timeinterval $[lb, ub]$ in the reservation table for a cell c has an associated value $coll(c)$ that indicates the number of agents occupying c during $[lb, ub]$. Figure 5.7 illustrates the reservation table for a cell occupied by three agents during overlapping timeintervals.

[7]For example, the average number of generated states for all MAPF problem instances from the Kiva-like environment with 150 agents that are solved in 100 seconds by ECBS(3) is more than 26 million.

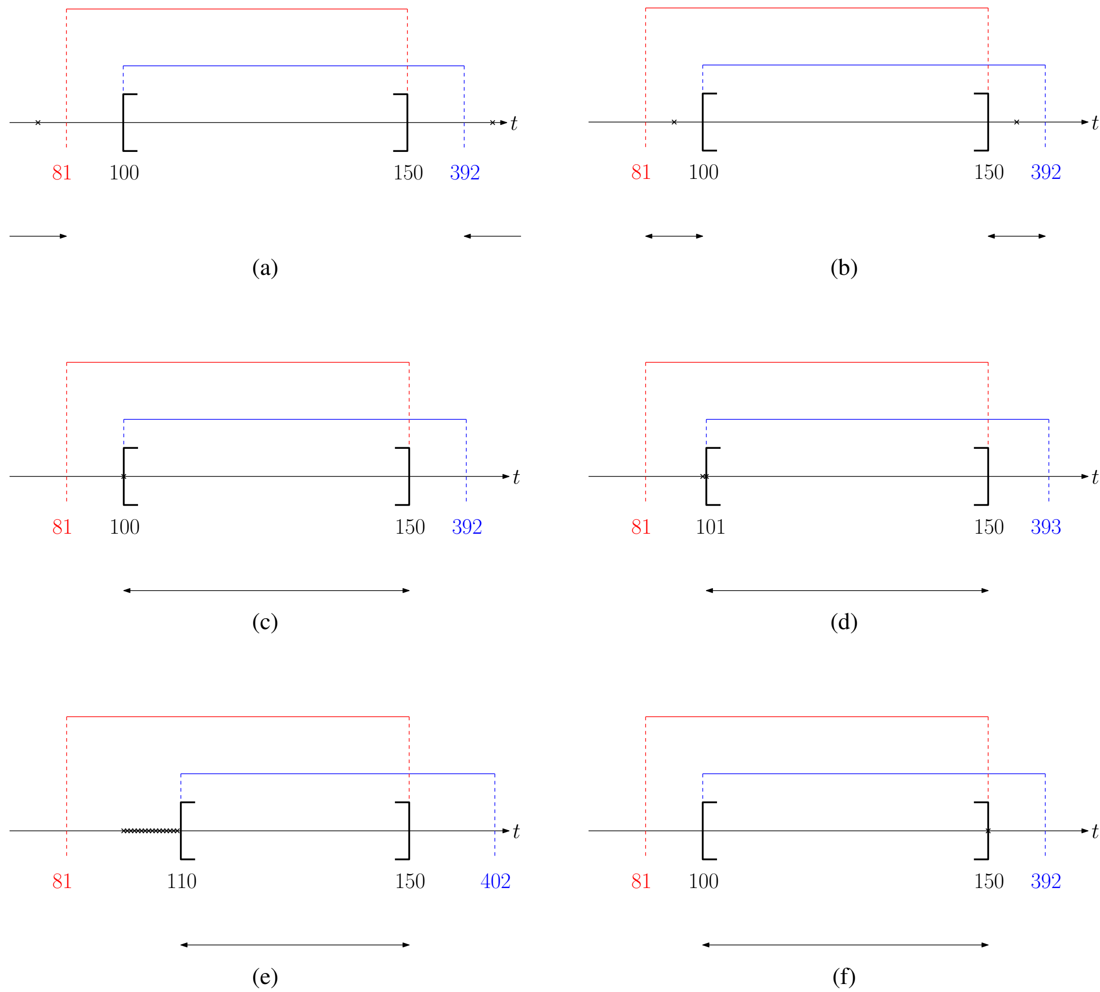


Figure 5.5: Illustrates different choices of a timestep τ for a constraint. Consider two agents, red and blue, that sweep a cell c during overlapping timeintervals. The blue agent sweeps c during $[100, 392]$, and the red agent sweeps c during $[81, 150]$. Therefore, the two agents collide in that swept cell during $[100, 150]$. (a)-(f) show the times during which the red and blue agents occupy the swept cell. (a) Choosing $\tau < 81$ or $\tau > 392$ will not enforce a change of path for any of the two agents and is thus not helpful. (b) Choosing $81 \leq \tau < 100$ or $150 < \tau \leq 392$ will enforce a change of path for just one of the two agents and is thus not helpful. (c) Choosing $\tau = 100$ will enforce a change of path for both agents. While this is helpful, the agents are likely to collide again during $[101, 150]$. (d) continues (c) with one additional constraint for $\tau = 101$. Once again, while this enforces a change of path for both agents, they are likely to collide again during $[102, 150]$. (e) Illustrates that continuing to choose $\tau = lb$ (for a given duration $[lb, ub]$ of a collision) is likely to lead to a (possibly very long) sequence of replanning operations, each of which results in the agent delaying the execution of the same action by only one timestep. (f) Illustrates our choice of $\tau = ub$ (in this example, $\tau = 150$). This choice guarantees that the replan operations for the red and blue agents cannot simply delay the execution of some actions in a way that causes them collide again in c during $[100, 150]$.

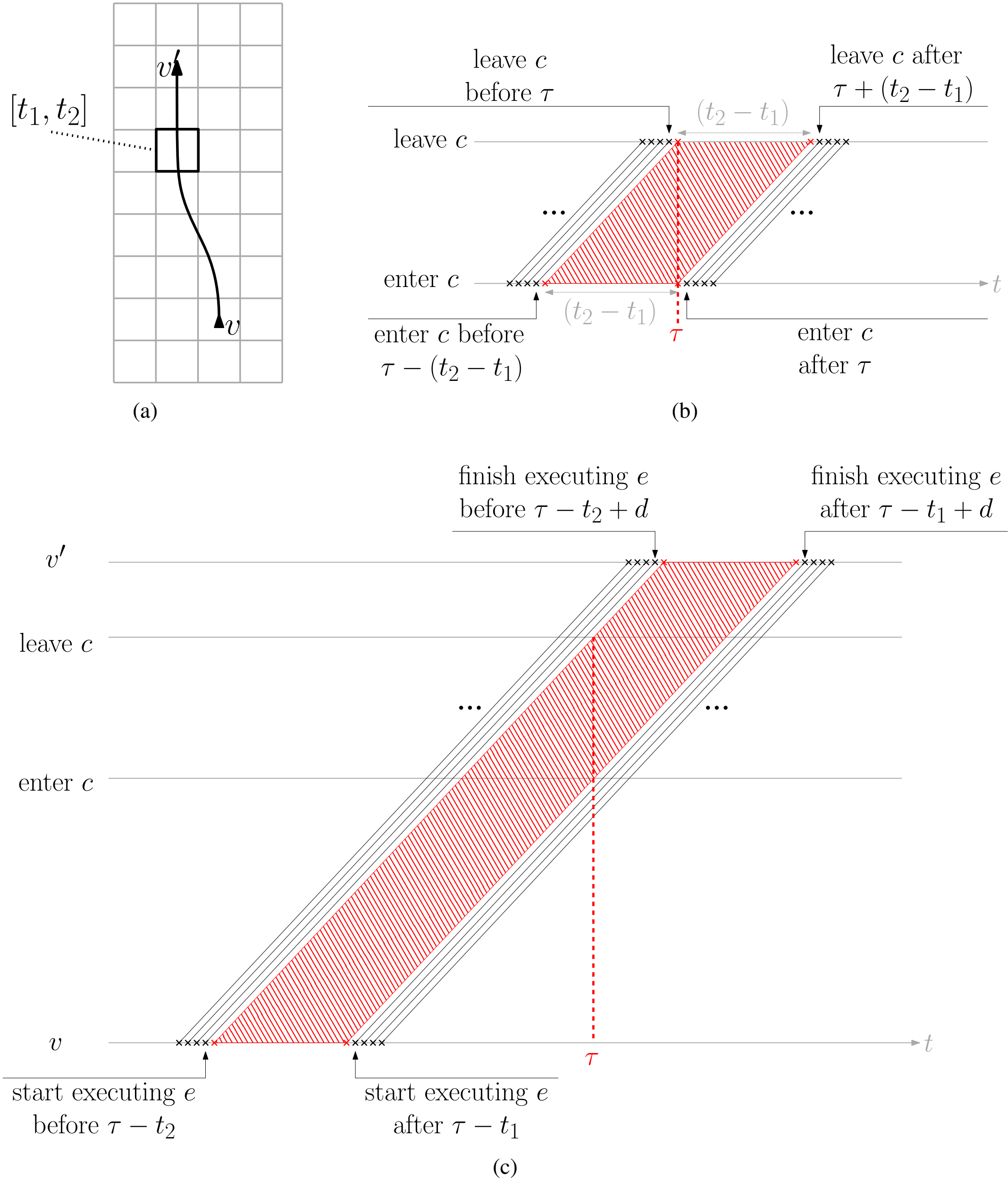


Figure 5.6: Illustrates the execution window that a constraint (c, τ) blocks for a given action. (a) shows an action $e = (v, v')$ that sweeps cell c (shown in black) during the timeinterval $[t_1, t_2]$. (b) illustrates a constraint that forbids an agent from occupying c at τ. Here, the x-axis represents time, and the y-axis represents the agent's progression through c (specifically, the bottom gray line signifies the agent entering c, and the top gray line signifies the agent leaving c). (c, τ) implies that the agent has to either: 1) leave c before τ (that is, it may occupy c at $\tau - 1$ at the latest); or 2) enter c after τ (that is, it may occupy c at $\tau + 1$ at the earliest). Thus, the agent has to start executing the action so that it either: 1) enters c before $\tau - (t_2 - t_1)$ (otherwise, it may not leave c before τ); or 2) leaves c after $\tau + (t_2 - t_1)$ (otherwise, it may enter c before τ). (c) is similar to (b) but shows the implication of (c, τ) on e, namely, starting to execute e during $[\tau - t_2, \tau - t_1]$ violates (c, τ). Thus, getting to v' using e during $[\tau - t_2 + d, \tau - t_1 + d]$, where d is the duration of e, is forbidden by (c, τ).

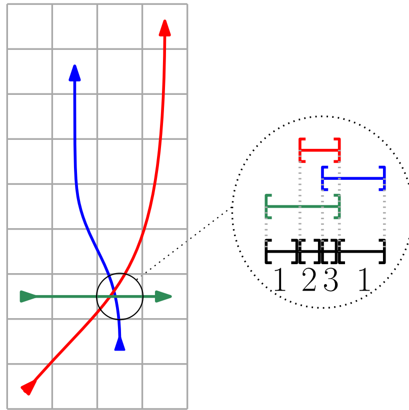


Figure 5.7: Shows three agents, red, blue and green, sweeping the encircled cell during overlapping timeintervals. The aggregate-on-overlap operation produces 4 timeintervals, each with an associated number of collisions, as indicated in black.

The reservation table we propose maintains an *interval map* [15] for each cell c in the environment. An interval map is a data structure that can efficiently manage intervals. It supports efficient insertion, deletion, search and aggregate-on-overlap operations. The insertion (deletion) operation can be done in logarithmic time unless the query interval overlaps with all existing intervals. Fortunately, this rarely happens in MAMP problems due to the locality of motion primitives. The aggregate-on-overlap operation combines (separates) the associated values of intersecting intervals on insertion (deletion) with the same complexity as the insertion (deletion) operation. Figure 5.7 also illustrates the aggregate-on-overlap operation.

In addition to insertion, deletion, search and aggregate-on-overlap, the usage of interval maps for the reservation table also facilitates the efficient detection of the earliest collision, which is part of every high-level state generation. Given the reservation table, one can simply iterate over all cells and extract the earliest collision (which is linear in the number of cells in the environment). This may be significantly faster than iterating over all pairs of paths (which is quadratic in the number of agents times the maximal number of swept cells in any path). Moreover, the usage of interval maps for the reservation table also facilitates the efficient computation of secondary heuristic values for the high-level FS. Generating a high-level state involves replanning for one agent. The reservation table of the generated child state can simply be copied from its parent state and updated by: 1) deleting the timeintervals of the relevant cells according to the agent's previous path; and 2) inserting the timeintervals of the relevant cells according to the agent's

new path. While updating the reservation table, we can also update the total duration in which two or more agents collide, which is used as the secondary heuristic value in the high-level search. Finally, the usage of interval maps for the reservation table also facilitates the efficient computation of secondary heuristic values for the low-level FS. Generating a low-level state involves looking at all swept cells associated with the motion primitive that generated it and checking the number of collisions during different timeintervals. This process is explained in more detail in the next section.

5.5.2 Reasoning About Wait Actions in Bulk with Timeintervals

In this section, we describe how to implement FS-B's bulk expansion and bulk generation efficiently using timeintervals. Assume that a given agent can wait in a configuration v during $[lb_v, ub_v]$ without violating any constraint and with a constant number of collisions. The number of collisions of the state $s = (v, [lb_v, ub_v])$ is defined to be the number of cells in which the path from the start state to s collides with any other agent's path, as specified in the reservation table[8]. Given an action $e = (v, v')$ of duration d, we would like to compute every successor $s' = (v', [lb_{v'}, ub_{v'}])$ such that the given agent can wait in v' during $[lb_{v'}, ub_{v'}]$ without violating any constraint and with a constant number of collisions. We first describe *GenerateIntervals* $(v, [lb_v, ub_v], e = (v, v'))$, a method that generates all such intervals $\{[lb^i_{v'}, ub^i_{v'}]\}_i$, and then show its similarity to FS-B's bulk expansion and bulk generation.

Definition 31 (A relevant constraint for an action during a timeinterval)**.** *A constraint (c, τ) is relevant for an action e during a timeinterval $[lb, ub]$ if: 1) $c \in \mathcal{C}^e$ (that is, c is a swept cell of e); and 2) $\tau \in [lb, ub]$.*

Definition 32 (A relevant collision for an action during a timeinterval)**.** *A collision $(c, [\tau_1, \tau_2])$ is relevant for an action e during a timeinterval $[lb, ub]$ if: 1) $c \in \mathcal{C}^e$ (that is, c is a swept cell of e); 2) $[lb_c, ub_c]$ (that is, the timeinterval during which c is swept) overlaps with $[\tau_1, \tau_2]$; and 3) $[\tau_1, \tau_2]$ overlaps with $[lb, ub]$.*

[8]Other measures, such as the total duration of collisions or the number of colliding agents across all swept cells, are also possible.

Definition 33 (A relevant constraint for a configuration during a timeinterval)**.** *A constraint* (c, τ) *is relevant for a configuration* v *during a timeinterval* $[lb, ub]$ *if: 1)* $c \in \mathcal{C}^v$ *(that is,* c *is an occupied cell of* v*); and 2)* $\tau \in [lb, ub]$.

Definition 34 (A relevant collision for a configuration during a timeinterval)**.** *A collision* $(c, [\tau_1, \tau_2])$ *is relevant for a configuration* v *during a timeinterval* $[lb, ub]$ *if: 1)* $c \in \mathcal{C}^v$ *(that is,* c *is an occupied cell of* v*); and 2)* $[\tau_1, \tau_2]$ *overlaps with* $[lb, ub]$.

GenerateIntervals $(v, [lb_v, ub_v], e = (v, v'))$ computes the (disjoint) timeintervals by first generating "arrival" timeintervals with respect to the constraints and collisions on the swept cells of the action, and then extending these arrival timeintervals with respect to the constraints and collisions on the cells associated with v' in order to reason about possibly waiting in v'. This can be achieved by performing the following steps:

1. Create a timeinterval $\mathcal{I}_e = [lb_v, ub_v + d]$. $\mathcal{I}_e$ includes all timesteps during which e can be in execution, that is, during which the agent's footprint may sweep some $c \in \mathcal{C}^e$.

2. Create timeintervals by removing from $[lb_v + d, ub_v + d]$ the blocked timeinterval $[\tau - ub_c + d, \tau - lb_c + d]$ (see Figure 5.6) for each relevant constraint (c, τ) for e during $\mathcal{I}_e$.

3. Annotate each timeinterval created in (2) with 0. For each relevant collision $(c, [\tau_1, \tau_2])$ for e during $\mathcal{I}_e$, aggregate-on-overlap the number of collisions for the colliding timeinterval $[\tau_1 - ub_c + d, \tau_2 - lb_c + d]$ (see Figure 5.8). Store the resulting annotated timeintervals (sorted in increasing order according to time) in a list, L_e. Note that the timeintervals in L_e do not overlap by construction.

4. Denote by $ub_{v'}$ the time of the earliest constraint among all relevant constraints for v' during $[ub_v + d, \infty]$. If no such constraint exists, set $ub_{v'}$ to ∞. Create a timeinterval $\mathcal{I}_w = [lb_v + d, ub_{v'}]$. $\mathcal{I}_w$ includes all timesteps during which the agent may choose to wait in v' after reaching it via e.

5. Create intervals by removing from $\mathcal{I}_w$ the timestep τ for each relevant constraint (c, τ) for v' during $\mathcal{I}_w$.

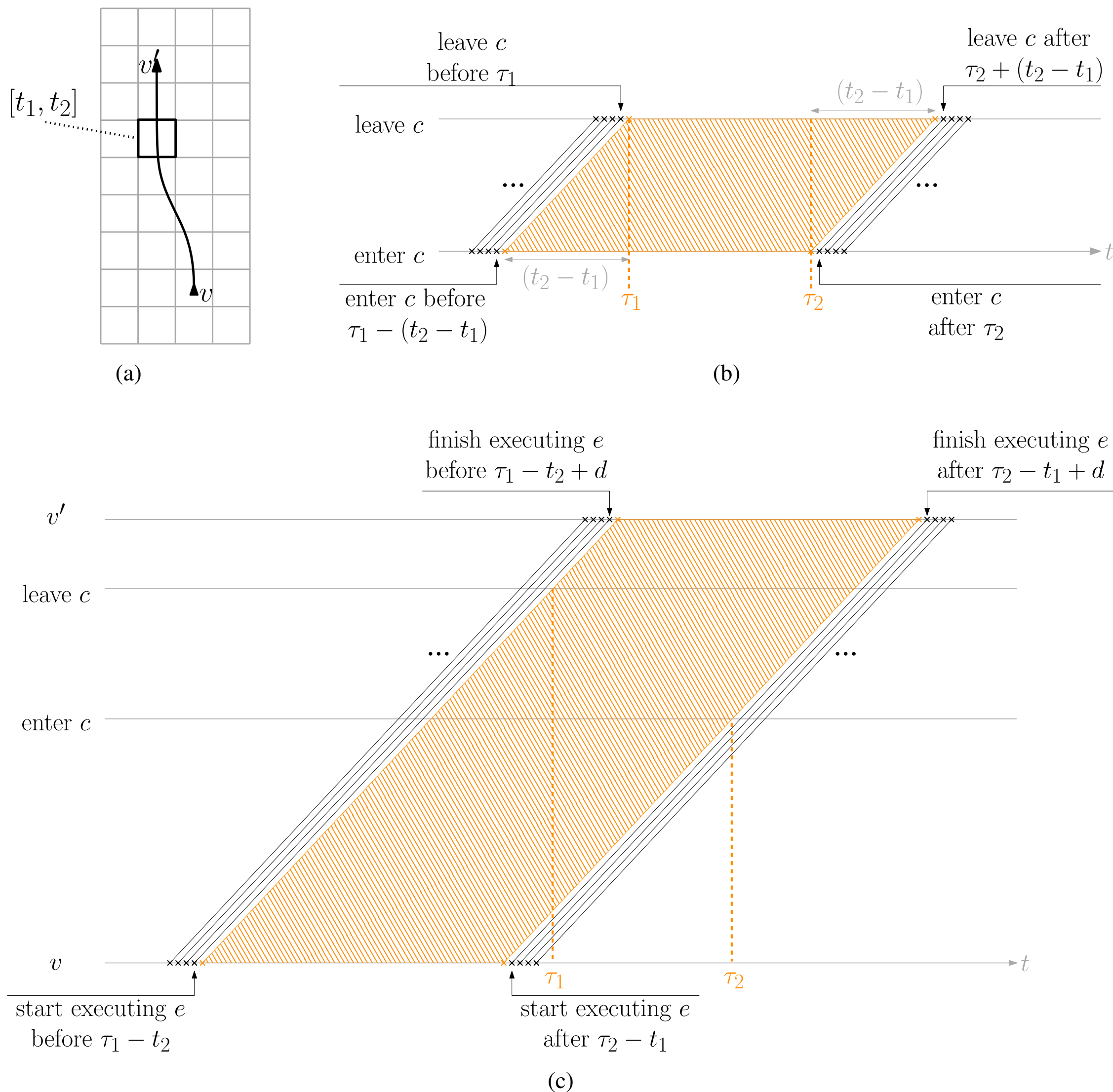


Figure 5.8: Illustrates the colliding timeinterval for a given action when another agent sweeps a cell c during the timeinterval $[\tau_1, \tau_2]$. (a) shows an action $e = (v, v')$ that sweeps c (shown in black) during the timeinterval $[t_1, t_2]$. (b) illustrates the timeinterval $[\tau_1, \tau_2]$ on c. Here, the x-axis represents time, and the y-axis represents the agent's progression through c (specifically, the bottom gray line signifies the agent entering c, and the top gray line signifies the agent leaveing c). Another agent sweeping c during $[\tau_1, \tau_2]$ implies that the agent executing e has to either: 1) leave c before τ_1 (that is, it may occupy c at $\tau_1 - 1$ at the latest); or 2) enter c after τ_2 (that is, it may occupy c at $\tau_2 + 1$ at the earliest). Thus, the agent has to start executing the action so that it either: 1) enters c before $\tau_1 - (t_2 - t_1)$ (otherwise, it may not leave c before τ_1); or 2) leaves c after $\tau_2 + (t_2 - t_1)$ (otherwise, it may enter c before τ_2). (c) is similar to (b) but shows the implication of another agent sweeping c during $[\tau_1, \tau_2]$ on e, namely, starting to execute e during $[\tau_1 - t_2, \tau_2 - t_1]$ results in a collision. Thus, getting to v' using e during $[\tau_1 - t_2 + d, \tau_2 - t_1 + d]$, where d is the duration of e, is associated with a collision.

6. Annotate each timeinterval created in (5) with 0. For each relevant collision $(c, [\tau_1, \tau_2])$ for v' during $\mathcal{I}_w$, aggregate-on-overlap the number of collisions for the timeinterval $[\tau_1, \tau_2]$. Store the resulting annotated timeintervals (sorted in increasing order according to time) in a list, L_w. Note that the timeintervals in L_w do not overlap by construction.

7. Reason about waiting in v' by using L_e and L_w as follows. For each $I_e \in L_e$, we denote the lower bound of I_e by lb_e, the upper bound of I_e by ub_e, the annotation (that is, the number of collisions) of I_e by $coll(I_e)$ and the successor annotated timeinterval of I_e in L_e by $succ(L_e, I_e)$.[9] Let $first(L_e)$ and $last(L_e)$ be the first and last annotated timeintervals in L_e, respectively. We use similar notation for $I_w \in L_w$ and $I_{ew} \in L_{ew}$. We also say that $I_1 = [lb_1, ub_1]$ is attached to $I_2 = [lb_2, ub_2]$ iff $ub_1 + 1 = lb_2$. Let $L_{ew} = L_e$, $I_{ew} = first(L_{ew})$ and $I_w = first(L_w)$. Iterate over the following until $I_{ew} = last(L_{ew})$ and $I_w = last(L_w)$:

   (a) Set I_w to the next relevant annotated timeinterval in L_w (that is, the earliest $I_w \in L_w$ during which it may be possible to wait after arriving to v' at ub_{ew} using e). Formally,

   while $(lb_w \leq ub_{ew})$ **do**

   Set I_w to $succ(L_w, I_w)$

   (b) If waiting in I_{ew} violates a constraint, then move to the next annotated timeinterval in L_{ew}. Formally,

   if $(ub_{ew} + 1 \notin I_w)$ **then**

   Set I_{ew} to $succ(L_{ew}, I_{ew})$

   Continue to the next iteration

   (c) *(If this iteration reaches (c) or further, then I_{ew} may be extended to the right with waiting.)*

   If I_{ew} is not attached to its successor and waiting does not incur additional collisions, then extend I_{ew} as much as possible in I_w up to I_{ew}'s successor. Formally,

[9] L_e is a list of disjoint annotated timeintervals sorted in increasing order according to time, thus $succ(I_e, L_e)$ is well defined.

if $(ub_{ew} + 1 \neq lb_{succ(L_{ew}, I_{ew})}$ and $coll(I_w) = 0)$ **then**

Set ub_{ew} to $\min\{lb_{succ(L_{ew}, I_{ew})} - 1, ub_w\}$

Continue to the next iteration

(d) If I_{ew} is not attached to its successor and waiting incurs additional collisions, then add a new timeinterval that is similar to the extension in 7(c) but annotated with the sum of collisions. Formally,

if $(ub_{ew} + 1 \neq lb_{succ(L_{ew}, I_{ew})}$ and $coll(I_w) > 0)$ **then**

Let $I = [ub_{ew} + 1, \min\{lb_{succ(L_{ew}, I_{ew})} - 1, ub_w\}]$ with $coll(I) = coll(I_{ew}) + coll(I_w)$

Add I to L_{ew} between I_{ew} and its successor

Continue to the next iteration

(e) *(If this iteration reaches (e) or further, then I_{ew} is attached to its successor.)*

If waiting in I_{ew} results in fewer collisions than executing e and waiting in $succ(L_{ew}, I_{ew})$, and waiting does not incur additional collisions, then extend I_{ew} to the right and delete (or adapt) its successor. Formally,

if $(coll(I_{ew}) + coll(I_w) \leq coll(succ(L_{ew}, I_{ew}))$ and $coll(I_w) = 0)$ **then**

Set ub_{ew} to $\min\{ub_{succ(L_{ew}, I_{ew})}, ub_w\}$

if $(ub_{ew} = ub_{succ(L_{ew}, I_{ew})})$ **then**

Remove $succ(L_{ew}, I_{ew})$ from L_{ew}

else

Set $lb_{succ(L_{ew}, I_{ew})}$ to $ub_{ew} + 1$

Continue to the next iteration

(f) If waiting in I_{ew} results in fewer collisions than executing e and waiting in $succ(L_{ew}, I_{ew})$, and waiting incurs additional collisions, add a new timeinterval that is similar to the extension in 7(e) but annotated with the sum of collisions and delete (or adapt) the successor. Formally,

if ($coll(I_{ew}) + coll(I_w) \leq coll(succ(L_{ew}, I_{ew}))$ and $coll(I_w) > 0$) **then**

Let $I = [ub_{ew} + 1, \min\{ub_{succ(L_{ew}, I_{ew})}, ub_w\}]$ with $coll(I) = coll(I_{ew}) + coll(I_w)$

Add I to L_{ew} between I_{ew} and its successor

if ($ub_I = ub_{succ(L_{ew}, I)}$) **then**

Remove $succ(L_{ew}, I)$ from L_{ew}

else

Set $lb_{succ(L_{ew}, I)}$ to $ub_I + 1$

Continue to the next iteration

(g) *(If this iteration reaches (g), then waiting in I_{ew} is not beneficial.)*

Set I_{ew} to $succ(L_{ew}, I_{ew})$ and continue to the next iteration.

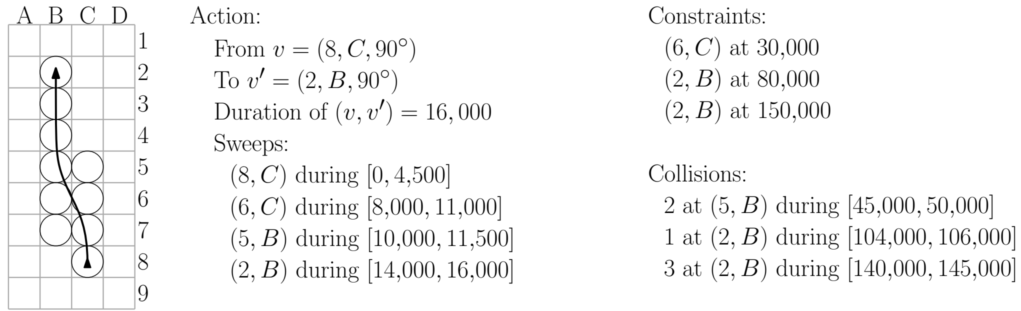
Return L_{ew}.

Figures 5.9(b)-(f) illustrate the steps of *GenerateIntervals* $(v, [lb_v = 10, ub_v = 120], e = (v, v'))$ for the action, constraints and collisions specified in Figure 5.9(a).

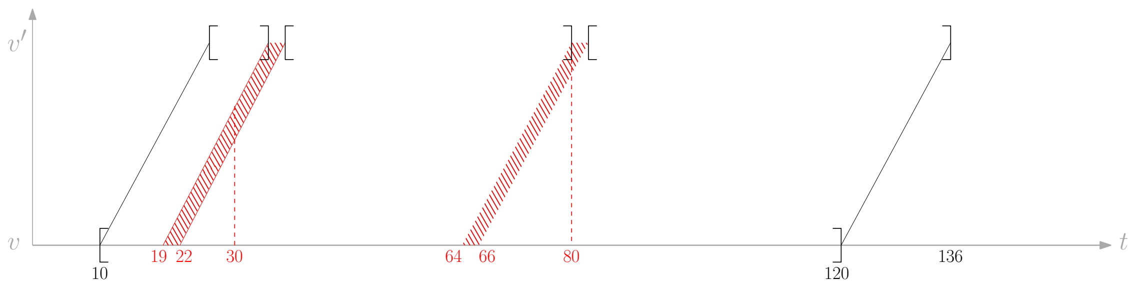
The following three lemmas are useful to prove that the algorithm we present in the next section is equivalent to FS-B. We define $t \in L$ (that is, a list of timeintervals L contains a timestep t) iff there exists an $I \in L$ such that $t \in I$. Figure 5.10 illustrates the following three lemmas.

Lemma 35. *L_e, the list of annotated timeintervals created in step (3) of GenerateIntervals $(v, [lb_v, ub_v], e = (v, v'))$ contains a timestep t' iff FS-B generates state (v', t') using action e when the states $(v, lb_v), \ldots, (v, ub_v)$ are bulk expanded (lines 11-15 in Algorithm 4, see Figure 5.2(c)-(e) for an illustration of these states).*

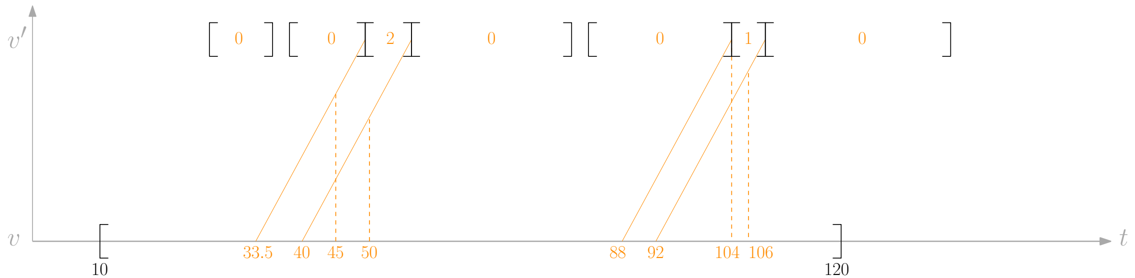
Proof. In step (1) of *GenerateIntervals*, the timeinterval $\mathcal{I}_e$ is created such that it contains all timesteps for which starting to execute e between timestep lb_v and timestep ub_v may violate a constraint. In step (2) of *GenerateIntervals*, the initial timeinterval $[lb_v + d, ub_v + d]$ contains all timesteps t' of arrival at v' that may be generated using e from the states $(v, lb_v), \ldots, (v, ub_v)$. The only timeintervals removed from $[lb_v + d, ub_v + d]$ in this step are those that contain arrival timesteps t' that violate a relevant constraint



(a)

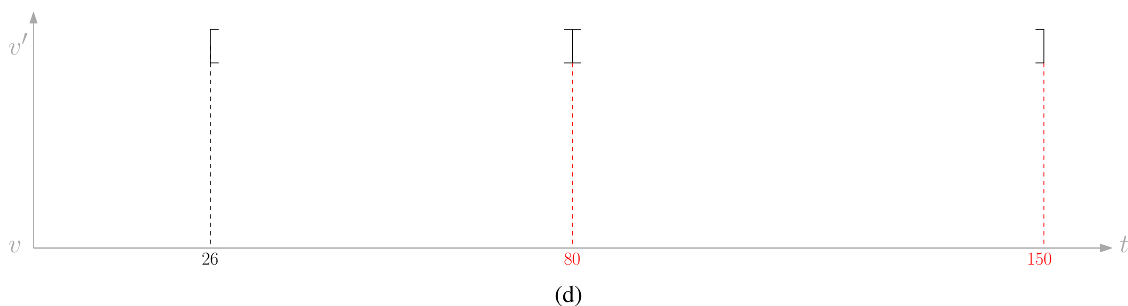


(b)

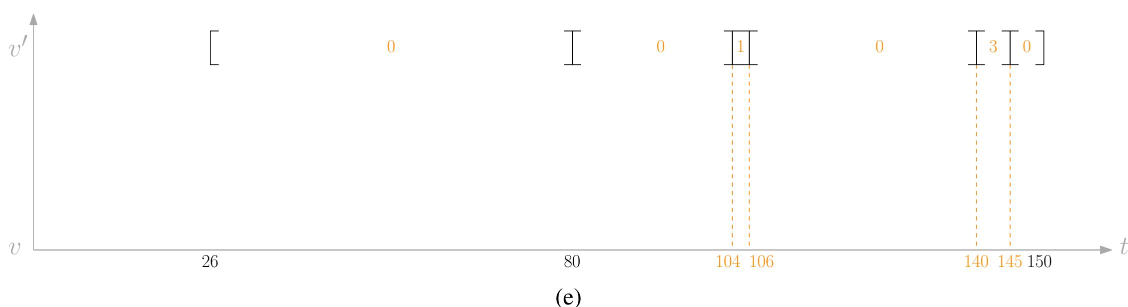


(c)

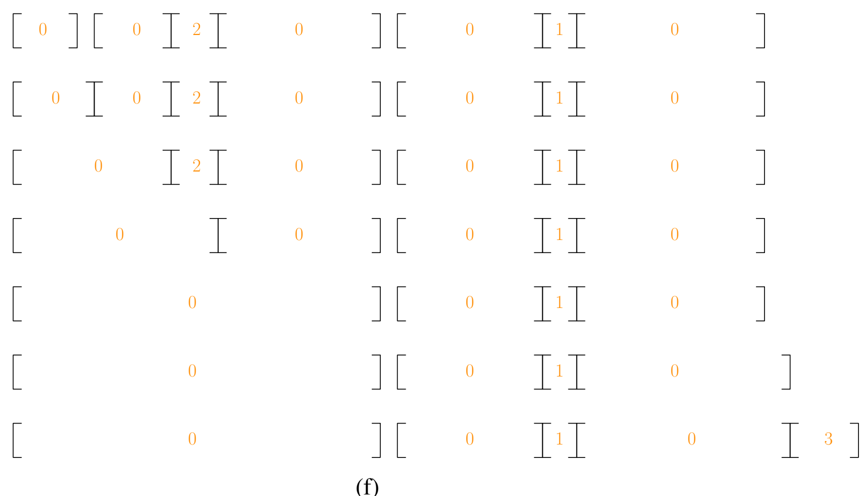
Figure 5.9: Illustrates how *GenerateIntervals* works. (a) describes an action $e = (v, v')$ of duration 16,000. The swept cells of e are shown in black circles. The timeintervals during which some cells are swept are also specified. A list of constraints and collisions is specified on the right side. Assume that the agent occupies cell $(8, C)$ while waiting in v and the cell $(2, B)$ while waiting in v'. (b) illustrates steps (1) and (2) of *GenerateIntervals*. v' is generated via e where waiting in v before execution is possible during $[10{,}000, 120{,}000]$. Thus, $I_e = [26{,}000, 136{,}000]$. The relevant constraints for (v, v') during I_e are shown in red dotted lines. Removing the blocked timeintervals results in the following intervals: $\{[26{,}000, 34{,}999], [38{,}001, 79{,}999], [82{,}000, 136{,}000]\}$. (c) illustrates step (3) of *GenerateIntervals*. The relevant collisions for (v, v') during I_e are shown in orange dotted lines. Aggregating-on-overlap the number of collisions results in the following annotated intervals: $\{([26{,}000, 34{,}999]; 0), ([38{,}001, 49{,}499]; 0), ([49{,}500, 56{,}000]; 2), ([56{,}001, 79{,}999]; 0),$
$([82{,}000, 103{,}999]; 0), ([104{,}000, 108{,}000]; 1)([108{,}001, 136{,}000]; 0)\}$.



(d)



(e)



(f)

Figure 5.9: Illustrates how *GenerateIntervals* works (continued). (d) illustrates steps (4) and (5) of *GenerateIntervals*. Since the earliest relevant constraint on v' during $[136, \infty]$ is 150, $I_w = [26{,}000, 150{,}000]$. The relevant constraints for v' during I_w are shown in red dotted lines. Removing them from I_w results in the following timeintervals: $\{[26{,}000, 79{,}999], [80{,}001, 149{,}999]\}$. (e) illustrates step (6) of *GenerateIntervals*. The relevant collisions for v' during I_w are shown in orange dotted lines. Aggregating-on-overlap the number of collisions results in the following timeintervals: $\{([26{,}000, 79{,}999]; 0), ([80{,}001, 103{,}999]; 0), ([104{,}000, 106{,}000]; 1), ([106{,}001, 139{,}999]; 0), ([140{,}000, 145{,}000]; 3), ([145{,}001, 149{,}999]; 0)\}$. (f) illustrates step (7). Each row shows one iteration, starting from L_e and ending at the returned annotated timeintervals: $\{([26{,}000, 79{,}999]; 0), ([80{,}001, 103{,}999]; 0), ([[104{,}000, 108{,}000]]; 1), ([108{,}001, 139{,}999]; 0), ([140{,}000, 150{,}000]; 3)$

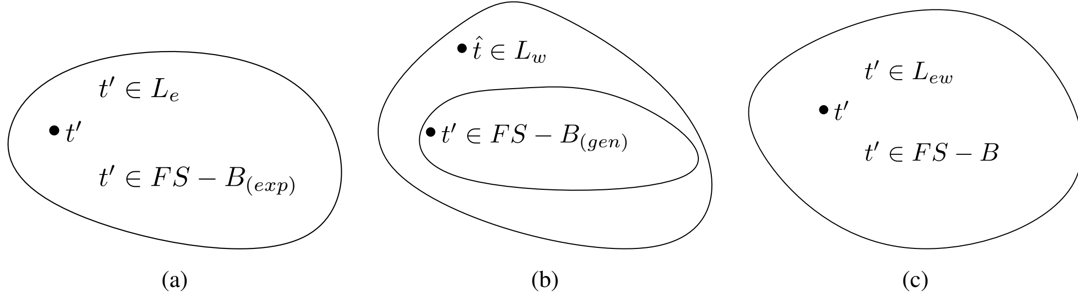


Figure 5.10: Illustrates lemmas 35, 36 and 37 in (a), (b) and (c), respectively. Lemma 35 shows that the set of timesteps in L_e is equal to the set of timesteps generated by the bulk expansion part of FS-B. Lemma 36, shows that the set of timesteps in L_w is a superset of the set of timesteps generated by the bulk generation part of FS-B. Lemma 37 shows that the set of timesteps in L_{ew} is equal to the set of timesteps generated by FS-B (both the bulk expansion and bulk generation parts).

for e during $\mathcal{I}_e$, which correspond exactly to the states (v', t') that FS-B does not generate (line 30 in Algorithm 4). In step (3) of *GenerateIntervals*, no timestep is removed or added since aggregate-on-overlap may only split or combine timeintervals according to their annotation. Thus, L_e corresponds exactly to the timeintervals defined by all timesteps t' of arrival at v' that FS-B generates using e when the states $(v, lb_v), \ldots, (v, ub_v)$ are expanded by bulk expansion. □

Lemma 36. *If FS-B generates (v', t') using wait actions when the states $(v', lb_v + d), \ldots, (v', ub_v + d)$ are bulk generated (lines 31-36 in Algorithm 4, see Figure 5.2(a)-(b) for an illustration of these states), L_w, the list of annotated timeintervals created in step (6) of GenerateIntervals $(v, [lb_v, ub_v], e = (v, v'))$, contains t'.*

Proof. In step (4) of *GenerateIntervals*, the timeinterval $\mathcal{I}_w$ is created such that it contains all timesteps t' of waiting in v' that may be generated using wait actions from the states $(v', lb_v + d), \ldots, (v', ub_v + d)$. In step (5) of *GenerateIntervals*, the only timesteps t' removed from $\mathcal{I}_w$ are those that violate a constraint when v' is occupied, and therefore such states (v', t') are not generated by FS-B (line 34 in Algorithm 4). In step (6) of *GenerateIntervals*, no timestep is removed or added since aggregate-on-overlap may only split or combine timeintervals according to their annotation. Thus, L_w contains all timesteps t' of waiting in v' that FS-B generates using wait actions when the states $(v', lb_v + d), \ldots, (v', ub_v + d)$ are bulk generated. □

Lemma 37. *L_{ew}, the list of annotated timeintervals created in step (7) of GenerateIntervals $(v, [lb_v, ub_v], e = (v, v'))$ contains a timestep t' iff FS-B generates the state (v', t') when the states $(v, lb_v), \ldots, (v, ub_v)$ are bulk expanded and generated.*

Proof. In step (7) of *GenerateIntervals*, L_{ew} is first initialized to L_e. Since no t' is removed from L_{ew} in step (7), if $t' \in L_e$ then $t' \in L_{ew}$. Thus, according to Lemma 35, L_{ew} contains t' iff FS-B generates (v', t') using e when $(v, lb_v), \ldots, (v, ub_v)$ are bulk expanded. Therefore, we only need to show that step (7) adds a $\hat{t} \in Ł_w$ to L_{ew} iff FS-B generates $\hat{t}$ using wait actions when $(v, lb_v), \ldots, (v, ub_v)$ are bulk generated.

Consider the states $(v', t' + 1), \ldots, (v', t' + x)$ that FS-B generates using wait actions during bulk generation for any (v', t') generated using e. We show that L_{ew} contains the timesteps $t' + 1, \ldots, t' + x$ when *GenerateIntervals* terminates. First, since (v', t') was generated using e (line 31 in Algorithm 4), $t' \in L_e$ and therefore $t' \in L_{ew}$. Second, according to Lemma 36, $t' + 1, \ldots, t' + x \in L_w$. Let $\hat{t_1} \in [t'+1, t'+x]$ be the earliest timestep such that $\hat{t_1} \notin L_{ew}$. Let $\hat{t_2} \in [\hat{t_1}, t'+x]$ be the latest timestep such that $\hat{t_1}, \ldots, \hat{t_2} \notin L_{ew}$. Since $\hat{t_1} - 1 \in L_{ew}$ and $\hat{t_1} \notin L_{ew}$, in step (7c) (or (7d)), $\hat{t_1}, \ldots, \hat{t_2}$ are added to L_{ew} by extending (or adding) a timeinterval. If $\hat{t_2} = t'+x$, L_{ew} contains the timesteps $t'+1, \ldots, t'+x$. Otherwise, $\hat{t_2} + 1 \in L_{ew}$ and the same arguments can be used recursively for the states $(v', \hat{t_2} + 1), \ldots, (v', t' + x)$ in the next iteration of *GenerateIntervals*.

Consider the timesteps t' such that $t' \in L_w$ and $t' \notin L_e$ that *GenerateIntervals* adds to L_{ew} in any iteration of step (7). We show that FS-B generates the states (v', t') using wait actions when generating in bulk. Only in step (7c) (or step (7d)), some timeinterval $I_{ew} \in L_{ew}$ can be extended (or a timeinterval I be added to L_{ew}) with such timesteps t'. Specifically, any $I_{ew} \in L_{ew}$ that does not have a successor attached to it (that is, $ub_{ew} + 1 \neq lb_{succ(L_{ew}, I_{ew})}$) is extended (or a new timeinterval is added) with timesteps $t' \in [ub_{ew} + 1, \hat{t}]$, where $\hat{t} = \min\{lb_{succ(L_{ew}, I_{ew})} - 1, ub_w\}$. First, due to step (7b), in step (7c) (or step (7d)), it is always the case that $ub_{ew} + 1 \in I_w$. Second, since $[ub_{ew} + 1, \hat{t}] \subseteq I_w$, no $t' \in [ub_{ew} + 1, \hat{t}]$ is a timestep of a relevant constraint on v'. Third, since $ub_{ew} \in L_{ew}$, FS-B generates state (v', ub_{ew}). Thus, FS-B also generates the states $(v', ub_{ew} + 1), \ldots, (v', \hat{t})$ using wait actions when generating in bulk (line 35 of Algorithm 4).

Thus, L_{ew} corresponds exactly to the timeintervals defined by all timesteps t' of the states (v', t') that FS-B generates when the states $(v, lb_v), \dots, (v, ub_v)$ are bulk expanded and generated.

□

5.5.3 Soft Collisions Interval Path Planning

In this section, we develop SCIPP for the low-level FS of ECBS-CT. SCIPP is similar in spirit to FS-B but, instead of bulk expansion and bulk generation of states with timesteps, it uses states with timeintervals. SCIPP efficiently reasons with timeintervals about constraints and collisions.

Algorithm 6 presents SCIPP. It takes as input a start configuration $s^j \in V^j$, a goal configuration $g^j \in V^j$, a list of constraints for each cell c and a reservation table, as specified in the high-level state. It outputs a w-suboptimal feasible path from s^j to g^j for the specified value of w without violating any constraint. Each state s represents a $(v, [lb, ub])$ pair. $\mathcal{L}$ is a hash table of all generated states. It maps a configuration v to a list $\mathcal{L}(v)$ of all generated states having v as their configuration. The timeintervals of all states in $\mathcal{L}(v)$ are maintained to be disjoint. For each timeinterval $[lb, ub]$ of a state $s \in \mathcal{L}(v)$, lb represents the earliest possible arrival time to s via s's predecessor without violating any constraint and with a constant number of collisions. Thus, $g(s) = lb$. $ub - lb$ represents the maximum wait duration in s while retaining the same number of collisions and without violating any constraint. The secondary heuristic value of s is $coll([lb, ub])$, the annotation of $[lb, ub]$ computed in *GenerateIntervals*.[10].

The main loop on lines 11-27 performs a FS after the initialization on lines 1-10. On line 3, $\mathcal{I}$, a list of (disjoint) timeintervals between 0 and the earliest constraint imposed on any cell associated with s^j, is created. $\mathcal{I}$ can be generated by executing steps (4) - (7) of *GenerateIntervals* for the start configuration s^j. This is equivalent to using *GenerateIntervals* $(s^j, [0,\text{earliest constraint on } s^j], e_0)$, where e_0 is a "dummy" identity action with duration 0 that has no constraints or collisions associated with it. Each timeinterval

[10] $coll([lb, ub])$ is computed to be the number of cells in which the path from the start state to s collides with any other agent's path, as specified in the reservation table. Other measures, such as the total duration of collisions or the number of colliding agents across all swept cells, are also possible.

Algorithm 6: Soft Collisions Interval Path Planning (SCIPP)

Input: Start and goal configurations (s^j and g^j), constraints, a reservation table and $w \geq 1$.
Output: A w-suboptimal feasible path from s^j to g^j.

```
1  OPEN = FOCAL = {∅}
2  ∀v ∈ V^j, L(v) = {∅}
3  I ← GenerateIntervals (s^j, [0,earliest constraint on s^j], e_0)
4  f_min = h(s^j)
5  for I ∈ I do
6      Create a state s = (s^j, I)
7      OPEN ← OPEN ∪ {s}
8      if f(s) ≤ w f_min then
9          FOCAL ← FOCAL ∪ {s}
10     L(s^j) ← L(s^j) ∪ {s}
11 while FOCAL ≠ ∅ do
12     f_min ← f(head(OPEN))
13     s = (v, [lb_v, ub_v]) ← head(FOCAL)
14     if w f_min < ub_v then
15         Create state s̃ = (v, [w f_min + 1, ub_v])
16         ub_v ← w f_min
17         OPEN ← OPEN ∪ {s̃}
18         L(v) ← L(v) ∪ {s̃}
19     FOCAL ← FOCAL \ {s}
20     OPEN ← OPEN \ {s}
21     if v = g^j and ub_v > latest constraint on v then
22         return π^j
23     for each (v, v') ∈ E^j do
24         I ← GenerateIntervals (v, [lb_v, ub_v], e = (v, v'))
25         for I ∈ I do
26             Create a state s' = (v', I)
27             Merge s' into L(v') and add/update the relevant states in OPEN and FOCAL
28     if OPEN ≠ ∅ and f_min < f(head(OPEN)) then
29         updateLowerBound(w f_min, w f(head(OPEN)))
30 return no solution.
```

in $\mathcal{I}$ has a constant number of collisions. On lines 5-10, a state is created for each timeinterval $I \in \mathcal{I}$, inserted into OPEN, conditionally inserted into FOCAL and added to $\mathcal{L}(s^j)$.

On lines 11-22, $f_{\min}$ is updated, the head of FOCAL, s, is removed from OPEN and FOCAL, and, if s is a goal state, a path is returned. This is similar to "vanilla" FS, with the only difference being on lines 12-15. In FS-B, bulk expansion is conditioned on $t \leq wf_{\min}$ (which is a consequence of $(v,t) \in$ FOCAL in line 13 in Algorithm 4). Since $s = (v, [lb_v, ub_v])$ in SCIPP corresponds to $(v, lb_v), \ldots, (v, ub_v)$ in FS-B, we truncate the upper bound of s if it is larger than the focal bound (on line 16). A new state corresponding to the truncated portion is created (on line 15), inserted into OPEN and $\mathcal{L}(v)$ (on lines 17-18). The new state may be expanded later. In this iteration, however, s is limited to the focal bound.

The for loop on lines 23-27 generates successor states for s. Unlike "vanilla" FS that generates one successor for every available action, SCIPP may generate more than one successor for every available action. More specifically, for every action $(v, v') \in E^j$, SCIPP generates all successor states s' using *GenerateIntervals*. The mechanism of duplicate detection is implemented using $\mathcal{L}$ on line 27. The subtlety for timeintervals is the possibility of a generated state $s' = (v', [lb', ub'])$ having the same configuration v' as a different state $s'' = (v', [lb'', ub''])$ already in $\mathcal{L}(v')$ such that their timeintervals overlap. Regardless of the different ways in which the two timeintervals may overlap, the *Merge* process restores the invariant that all states in $\mathcal{L}(v')$ are disjoint and each one has a constant number of collisions. The *Merge* process can: (1) add the newly created state s' to $\mathcal{L}(v')$ and insert it into OPEN and conditionally into FOCAL; or (2) update the timeinterval of an existing state in $\mathcal{L}(v')$ and thus also update its priorities in OPEN and FOCAL.

An example of the *Merge* process is illustrated in Figure 5.11. Suppose that s'' is already in $L(v')$ with timeinterval $[lb'', ub'']$ and a collisions (shown in Figure 5.11(a)), and v' is now generated with timeinterval $[lb', ub']$ and $b < a$ collisions (shown in Figure 5.11(b)). Suppose that, $lb'' \in [lb', ub']$ and $ub' \in [lb'', ub'']$ (shown in Figure 5.11(c)). After $Merge$, $\mathcal{L}(v')$ has states s_1 and s_2 with configuration v' and timeintervals $[lb', ub']$ and $[ub', ub'']$, respectively. The number of collisions for s_1 and s_2 is b and a, respectively (shown in Figure 5.11(d)). Finally, s_1 is a new state inserted into OPEN and conditionally into FOCAL, while s_2

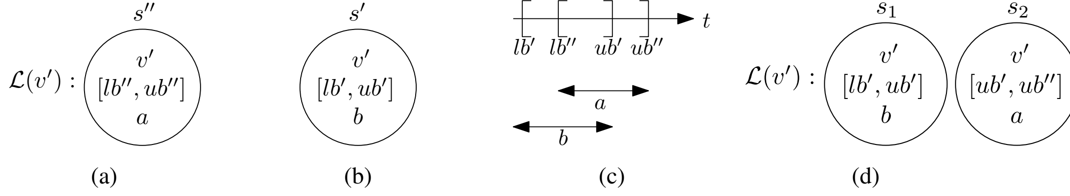


Figure 5.11: Illustrates duplicate detection using *Merge* for two states having the same configuration and overlapping timeintervals.

is the updated s'' with a larger g-value of ub'. If the timeintervals in $\mathcal{I}$ and $\mathcal{L}(v')$ are maintained sorted in increasing time, *Merge* can be performed in linear time in the size of both lists.

So far, we do not consider of a specific $h_{\texttt{FOCAL}}$ in the context of FS-B. In fact, all proofs in sections 5.3 and 5.5.2 hold for any $h_{\texttt{FOCAL}}$. This was done on purpose (see more in Section 5.7). SCIPP, however, operates with a concrete $h_{\texttt{FOCAL}}$, which is the number of collisions. Since SCIPP expands states with timeintervals that are being partially determined by $h_{\texttt{FOCAL}}$, we specialize FS-B to operate in the same way. First, the state $s = (v, [lb_v, ub_v])$ that SCIPP expands in each iteration of the while loop (line 11 in Algorithm 6) corresponds to the states $(v, lb_v), \ldots, (v, ub_v)$ in FS-B. Second, for any $h_{\texttt{FOCAL}}$, *GenerateInterval* creates timeintervals that, put together, contain the same timesteps that FS-B generates when expanding and generating in bulk the states $(v, lb_v), \ldots, (v, ub_v)$. However, *GenerateInterval* bunches these timesteps differently into timeintervals depending on $h_{\texttt{FOCAL}}$. For $h_{\texttt{FOCAL}}$ being the number of collisions, $[lb_v, ub_v]$ is generated by *GenerateInterval* such that $[lb_v, ub_v]$ is the longest possible timeinterval of being in v without violating any constraint and with a constant number of collisions. Thus, we can simply add "and $h_{\texttt{FOCAL}}((v, t)) = h_{\texttt{FOCAL}}((v, t-1))$" to the condition of the while loop on line 13 of Algorithm 4 to make FS-B equivalent to SCIPP.

Theorem 38. *SCIPP is equivalent to FS-B (with "and $h_{FOCAL}((v, t)) = h_{FOCAL}((v, t-1))$" added to the condition of the while loop on line 13 of Algorithm 4 and h_{FOCAL} being the number of collisions).*

Proof. The states that SCIPP generates for the start configuration (lines 3-10 in Algorithm 6) are equivalent to the states that FS-B generates for the start configuration (lines 3-7 in Algorithm 4). According to Lemma

37, every iteration of the main while loop of SCIPP (lines 11-29 in Algorithm 6) is identical to FS-B (lines 8-17 in Algorithm 4) when $h_{\texttt{FOCAL}}$ is the number of collisions. □

Theorem 39. *ECBS-CT is w-suboptimal*

Proof. Since SCIPP is equivalent to FS-B, ECBS-CT is w-suboptimal. □

5.5.3.1 Viewing SCIPP as a w-suboptimal Version of SIPP

SIPP is another single-agent heuristic search-based solver that efficiently reasons with timeintervals. Like SCIPP, SIPP can derive timeintervals from the constraints specified in a high-level state. Given a start configuration s^j, a goal configuration g^j and a list of constraints for each cell c, SIPP finds a feasible path from s^j to g^j with minimum arrival time at g^j. This path also guarantees that no constraint is violated because no cell is swept outside of its "safe" timeintervals. Here, the dynamic obstacles are simply the constraints, that is, the "safe" timeintervals associated with c are all timeintervals other than the specified timesteps of the constraints for c. In SIPP, a state s represents $(v, [lb_v, ub_v])$ pair, where v is a configuration and $[lb_v, ub_v]$ is a "safe" timeinterval for v. A successor s' of s represents a transition from v to v' via a "legal" action.[11]. $g(s)$ represents the earliest arrival time at v within $[lb_v, ub_v]$. Thus, when generating s', $g(s')$ is updated to $g(s) + d$ if $g(s') > g(s) + d$, where d is the duration of the action.

The example in Figure 5.12(a) shows an environment with 36 cells (grey squares) and an action $e = (v, v')$ in blue from v to v'. While waiting in v and v', the agent occupies cells $(8, C)$ and $(2, B)$, respectively. Black circles depict the swept cells of e. For the sake of simplicity, let us assume that the agent occupies all of the swept cells during the entire execution of e. The upper part of Figure 5.12(b) illustrates how SIPP works when generating "safe" timeintervals for e. The x-axis represents time, and the y-axis represents different configurations. A red dot between v and v' represents a constraint at the time of its x-coordinate at an intermediate swept cell of e, and a red dot on the horizontal line for v' represents a constraint at the time of its x-coordinate for being in v'. The red regions indicate execution times of e that violate a constraint. Since we assume that the agent occupies all of the swept cells during the entire

[11] An action is legal iff it sweeps each of its associated cells only during one of its "safe" timeintervals.

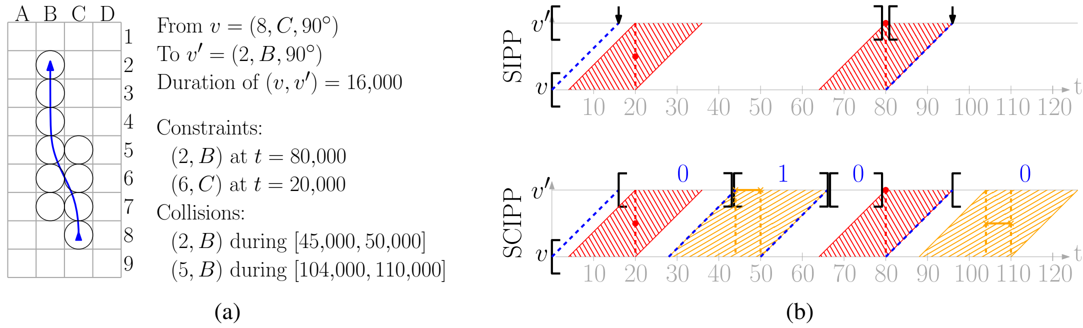


Figure 5.12: (a) shows an action (v, v') in blue. (b) illustrates the difference between SIPP and SCIPP for (a).

execution of e, the width of the parallelograms is equal to the duration of e. The "safe" timeinterval for v is $[0, \infty]$ and, due to the constraint on v' at timestep 80, the "safe" timeintervals for v' are $[0, 79]$ and $[81, \infty]$. The black arrows indicate the earliest arrival times when using e within these timeintervals, serving as g-values. The agent can arrive at v' as early as 16 and thus avoid violating the constraint on the intermediate cell at timestep 20.

Unlike SCIPP, SIPP does not guarantee that the number of collisions associated with each "safe" timeinterval is constant because SIPP sets safe timeintervals only according to the constraints. Thus, SIPP, unfortunately, cannot reason about the collisions specified in the reservation table of the high-level state, which is used to prioritize states in FOCAL. The lower part of Figure 5.12(b) illustrates how SCIPP works when generating disjoint timeintervals for e. Like in SIPP, constraints are represented in red. In addition, a yellow horizontal line between v and v' represents a collision during the timeinterval between its end-point x-coordinates at an intermediate swept cell of e, and a yellow horizontal line on the horizontal line for v' represents a collision during the timeinterval between its end-point x-coordinates when being in v'. Note that the disjoint timeintervals of v' have a constant number of collisions, depicted with blue labels.

Therefore, SIPP is unsuitable for FS. As we show in experiments in the next section, our w-suboptimal MAMP solvers with $w > 1$ are significantly better than the optimal MAMP solver, both in terms of runtime and success rate. Thus, SCIPP, which not only efficiently reasons about timeintervals and constraints but

also about collisions and derives secondary heuristic values from them, is more powerful than SIPP for solving MAMP tasks.

5.6 Experiments

We present experimental results for two environments, Arena and Den520d, and two motion primitives, Unicycle and PR2. Arena and Den520d are two benchmark environments from the Grid-Based Path Planning Competition (both environments can be found at https://movingai.com/GPPC/). Arena is shown in Figure 5.13, and Den520d is shown in Figure 5.14. Both are representative of obstacle-rich environments. The motion primitives are taken from the Search-based Planning Laboratory (both motion primitives can be found at http://sbpl.net/). The Unicycle and PR2 motion primitives, shown in Figures 5.15(a)&(b), are popularly studied. The Unicycle (PR2) motion primitives have 16 discrete orientations, each with 5 (13) primitives. Like in ECBS, we precompute the perfect single-agent heuristic in the environment. This is used for single-agent planning in SCIPP.

We generated MAMP problem instances for the Arena environment as follows. For each number of agents between 2 and 60 in increments of 2, we randomly generated 25 MAMP problem instances. We generated MAMP problem instances for the Den520d environment as follows. For each number of agents between 10 and 150 in increments of 10, we randomly generated 25 MAMP problem instances. Each MAMP problem instance is generated by choosing start and goal configurations for every agent at random. We ensure that no two agents share an occupied cell when being in their start configuration or their goal configuration. The experimental setup is identical to the one described in Section 3.5.1. Figures 5.16 and 5.17 report the runtime and success rate of ECBS-CT(w) for $w \in \{1, 1.2, 1.5, 2\}$ (averaged over the 25 randomly generated MAMP problem instances) for each number of agents.

For all combinations of environments and motion primitives, we observe that the w-suboptimal MAMP solvers with $w > 1$ are significantly better than the optimal MAMP solver, both in terms of runtime and success rate. This observation demonstrates the power of our bounded-suboptimality framework for

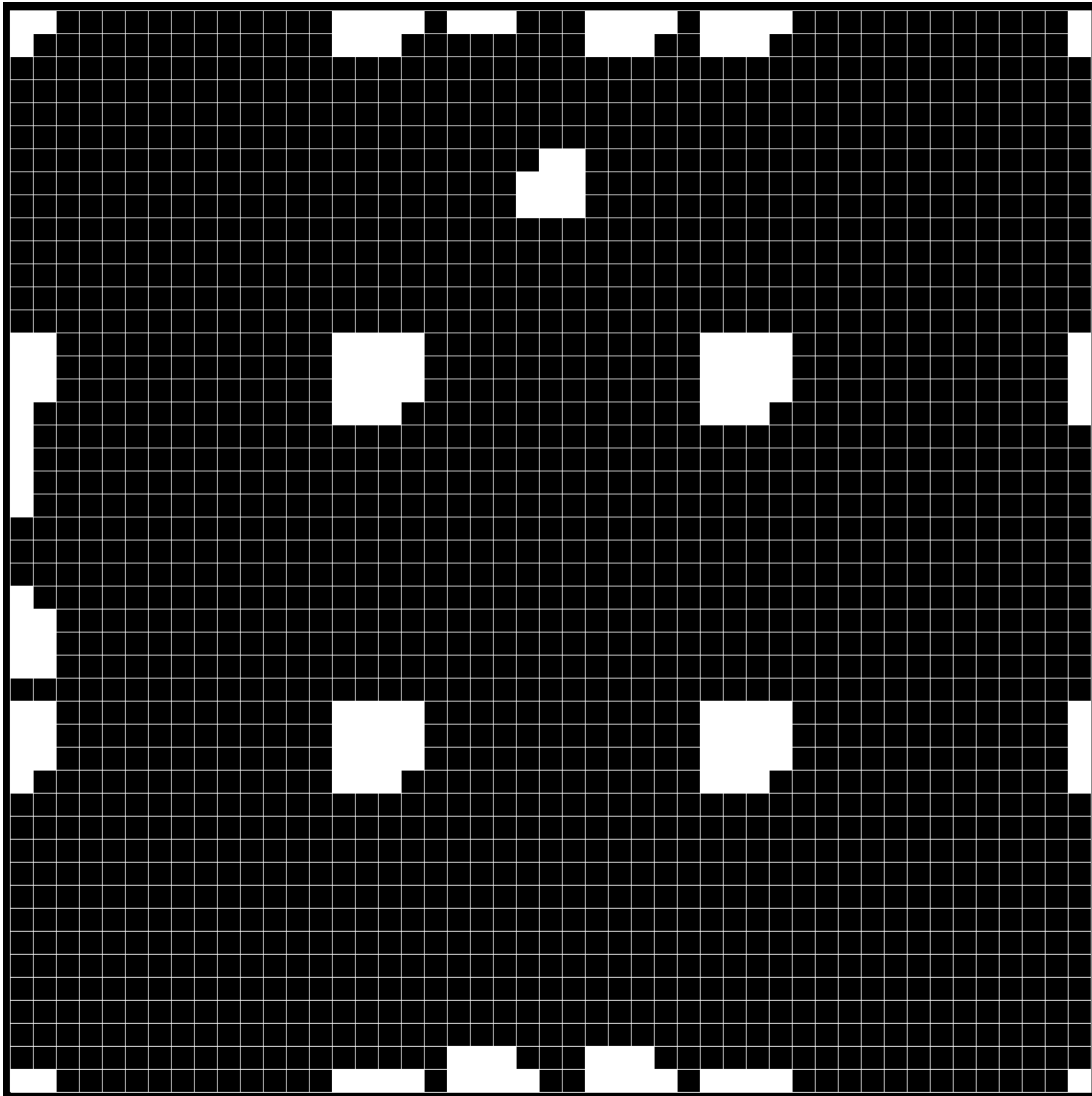


Figure 5.13: Shows the Arena environment with 49×49 cells. Unblocked cells are black, and blocked cells are white.

Figure 5.14: Shows the Den520d environment with 256×257 cells. Unblocked cells are black, and blocked cells are white.

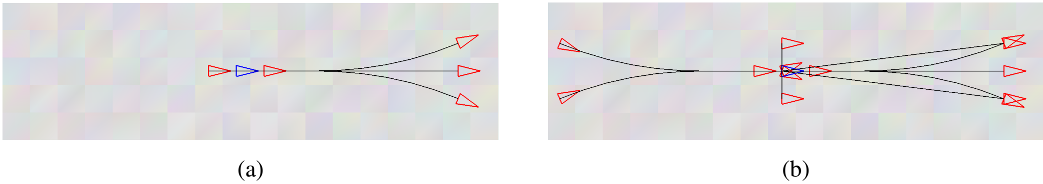


Figure 5.15: (a) and (b) show the Unicycle and the PR2 motion primitives, respectively, for (x, y, θ) in 5×18 free cells for the start state depicted in blue. For a motion primitive, a black line represents the trajectory of the center of the agent's footprint, and a red triangle at its end represents the orientation at the successor state.

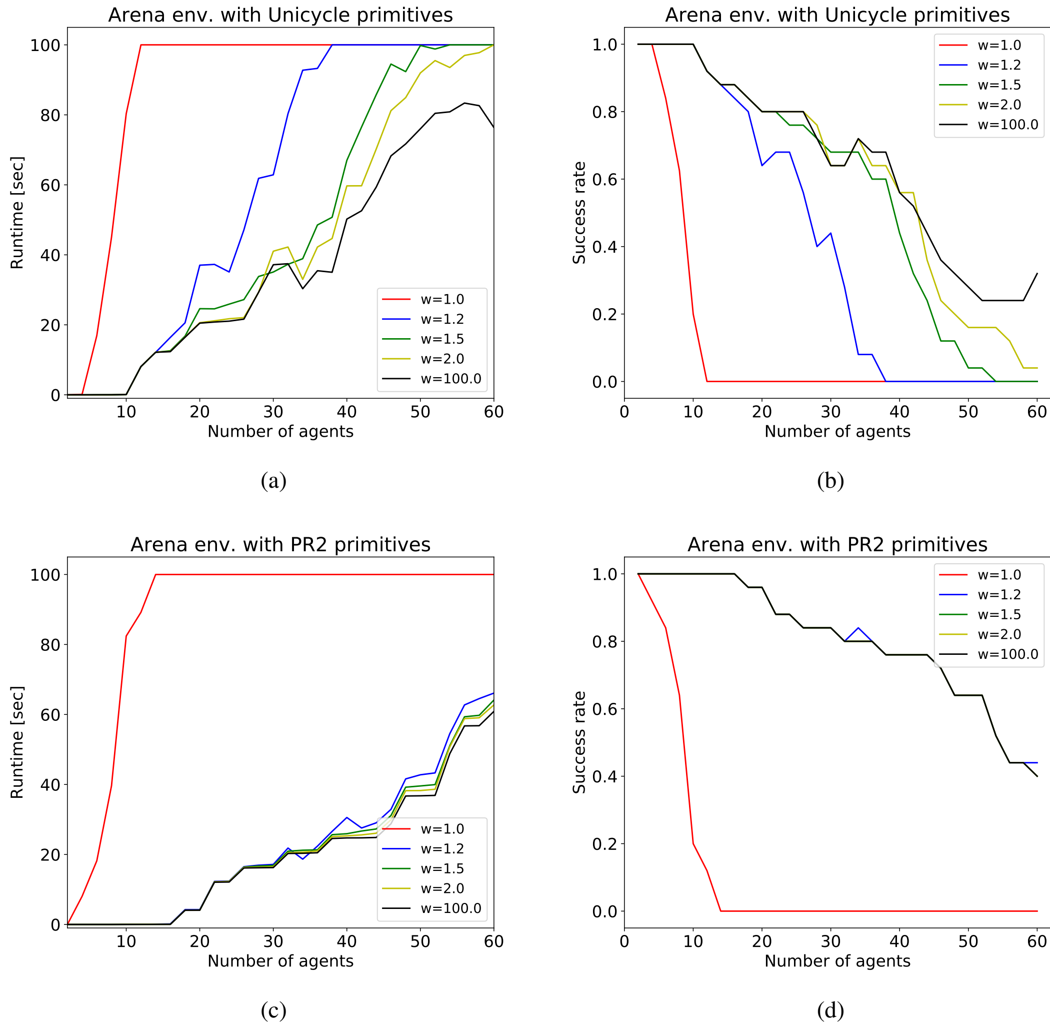
MAMP. The w-suboptimal MAMP solvers also exhibit the "diminishing returns" property with increasing w. This means that ECBS-CT with $w = 1.2$ or 1.5 is not only effective (since w is small) but also efficient in finding solutions. Moreover, for most MAMP problems, ECBS-CT with $w > 1$ produces a solution with an experimentally determined suboptimality[12] that is significantly smaller than the suboptimality bound w. For example, when $w = 2$, the success rate for 20 agents in the Arena environment with the Unicycle (PR2) motion primitives is 80% (96%) and the average experimentally determined suboptimality is 1.15 (1.07). When $w = 2$, the success rate for 50 agents in the Den520d environment with the Unicycle (PR2) motion primitives is 88% (84%) and the average experimentally determined suboptimality is 1.04 (1.02).

Before analyzing ECBS-CT further, we argue that the single-agent search effort[13] partially depends on the size of the environment and the number of motion primitives. First, the Arena environment is much smaller than the Den520d environment. Therefore, paths from start to goal locations tend to be shorter. Thus, the single-agent search effort tends to be smaller in the Arena environment compared to the Den520d environment. For example, with the Unicycle (PR2) motion primitives, $w = 2$ and 50 agents, for every state that the high-level search expands, the low-level search expands on average[14] 7,747 (18,022) states in the Arena environment compared to 24,367 (237,991) states in the Den520d environment, which is a factor of about $\times$3.1 ($\times$13.2) larger in the Den520d environment. Second, the PR2 motion primitives include 13 primitives for each discrete orientation while the Unicycle motion primitives include 5 primitives for each discrete orientation. Therefore, the number of successors each state has is larger when using the PR2 motion primitives compared to the Unicycle motion primitives. Thus, the single-agent search effort tends to be larger when using the PR2 motion primitives compared to the Unicycle motion primitives. For example, in the Arena (Den520d) environment with $w = 2$ and 50 agents, for every state that the high-level search expands, the low-level search expands on average 7,747 (24,367) states when using the Unicycle motion primitives and 18,022 (237,991) states when using the PR2 motion primitives, which is a factor of about $\times$2.3 ($\times$9.7) larger for the PR2 motion primitives.

[12] The experimentally determined suboptimality for each MAMP problem is the ratio of the solution cost and $f_{\min}$. This ratio is no larger than w by design.

[13] measured as the average number of states expanded during a low-level search, which highly correlates with runtime

[14] among the MAMP problem instances solved within 100 seconds



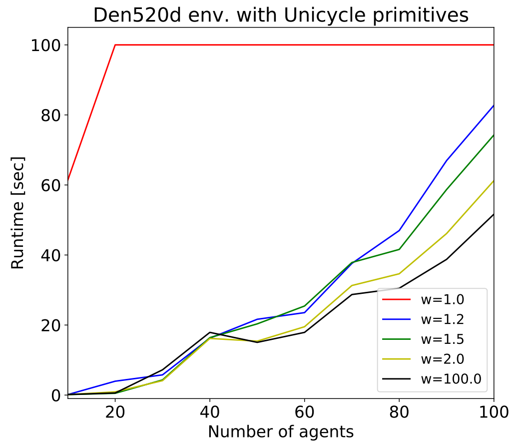
(a)

(b)

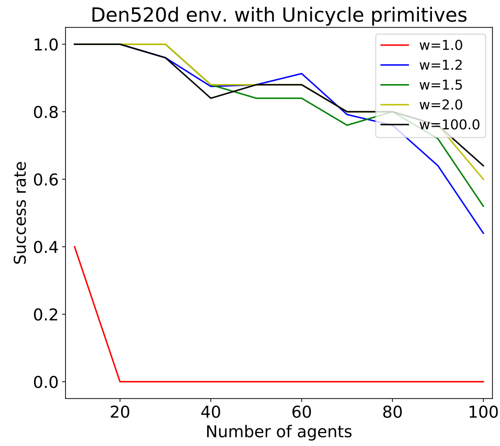
(c)

(d)

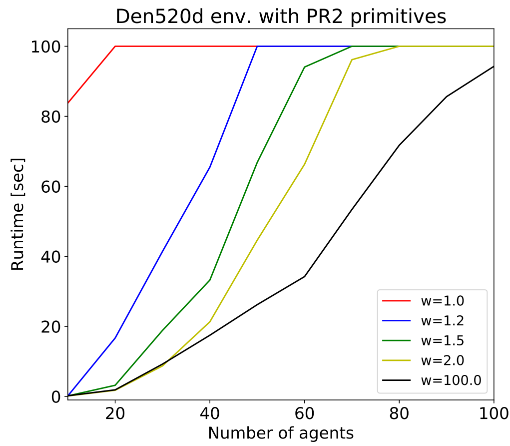
Figure 5.16: Shows the average runtimes (left column) and success rates (right column) of ECBS-CT with suboptimality bounds $w \in \{1, 1.2, 1.5, 2\}$ and the Unicycle motion primitives (top row) or the PR2 motion primitives (bottom row) in the Arena environment.



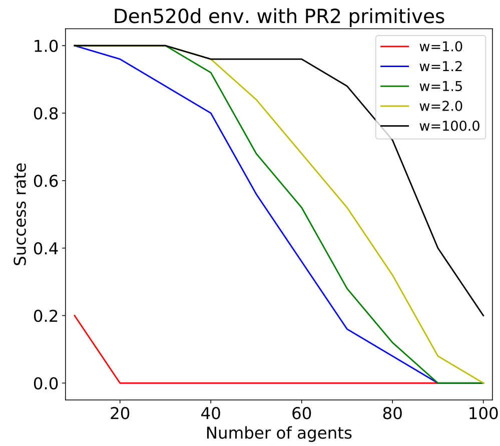
(a)



(b)



(c)



(d)

Figure 5.17: Shows the average runtimes (left column) and success rates (right column) of ECBS-CT with suboptimality bounds $w \in \{1, 1.2, 1.5, 2\}$ and the Unicycle motion primitives (top row) or the PR2 motion primitives (bottom row) in the Den520d environment.

We observe two ways in which the number of collisions that the high-level search needs to resolve is a dominant factor in ECBS-CT's runtime. First, as the number of agents increases, the number of collisions that the high-level search needs to resolve increases more rapidly in the smaller Arena environment compared to the larger Den520d environment due to a higher density of agents. For example, with the Unicycle motion primitives, $w = 2$ and 50 agents, the high-level search expands on average 258 states in the Arena environment and 17 states in the Den520d environment. When the high-level search needs to resolve more collisions, ECBS-CT's runtime tends to be much longer. For example, with the Unicycle motion primitives, $w = 2$ and 50 agents, the average runtime of ECBS-CT is 36.8 seconds in the Arena environment and 3.7 seconds in the Den520d environment. This explains why ECBS-CT has higher success rates and shorter average runtimes with the Unicycle motion primitives in the Den520d environment (Figures 5.17 (a) and (b)) compared to the Arena environment (Figures 5.16 (a) and (b)) for any given number of agents, even though Den520d is a much larger environment (which implies an increased single-agent search effort).

Second, as the flexibility of the single-agent search increases, it can find paths that collide less and thus reduce the number of collisions that the high-level search needs to resolve. We already observed that this is the case with increasing w. The flexibility of the single-agent search can also increase when using richer motion primitives. The PR2 motion primitives are richer than the Unicycle motion primitives since they include the possibility of turning in place and moving sideways. Thus, the number of collisions that the high-level search needs to resolve decreases with the richer PR2 motion primitives compared to the Unicycle motion primitives. For example, in the Arena environment with $w = 2$ and 50 agents, the high-level search expands on average 257.7 states with the Unicycle motion primitives and 6.3 states with the PR2 motion primitives. Once again, when the high-level search needs to resolve more collisions, ECBS-CT's runtime tends to be much longer. For example, in the Arena environment with $w = 2$ and 50 agents, the average runtime of ECBS-CT is 36.8 seconds with the Unicycle motion primitives and 3.2 seconds with the PR2 motion primitives. This explains why ECBS-CT has higher success rates and shorter average runtimes in the Arena environment with the PR2 motion primitives (Figures 5.16 (c) and (d)) compared to

the Unicycle motion primitives (Figures 5.16 (a) and (b)) for any given number of agents, even though the PR2 motion primitives are richer (which implies an increased single-agent search effort).

Finally, we observe that the combination of a larger environment with richer motion primitives can make the single-agent search effort prohibitively large (for the given time limit of 100 seconds) even though this combination reduces the number of collisions that the high-level search needs to resolve. For example, in the Den520d environment with $w = 2$ and 50 agents, the high-level search expands on average 16.45 states with the Unicycle motion primitives and 4.61 states with the PR2 motion primitives. However, the average runtime of ECBS-CT in this example is 3.7 seconds with the Unicycle motion primitives and 28 seconds with the PR2 motion primitives. This explains why ECBS-CT has higher success rates and shorter average runtimes in the Den520d environment with the Unicycle motion primitives (Figures 5.17 (a) and (b)) compared to the PR2 motion primitives (Figures 5.17 (c) and (d)) for any given number of agents, even though the Unicycle motion primitives are not as rich as the PR2 motion primitives (which implies more collisions that the high-level search needs to resolve).

To approximate HCA*, we use $w = 100$ with the intuition that a high suboptimality bound allows greater flexibility for agents to circumvent collisions with other agents when planning their path in the high-level root state. In the larger Den520d environment with the richer PR2 primitives, higher w results in increased efficiency. However, in the smaller Arena environment or with the Unicycle motion primitives, higher w does not increase efficiency much. Since in ECBS each agent is limited to w-suboptimal paths w.r.t. its optimal path, agents with relatively close start and goal configurations have less flexibility to deviate from their shortest paths compared to agents with relatively distant start and goal configurations. Thus, it can still be the case that there are collisions in the high-level root state, in which case $w = 100$ does not approximate HCA* well.

5.7 Conclusions and Future Work

In this chapter, we introduced the MAMP problem, a generalization of the MAPF problem to kinodynamically constrained agents. We developed FS-B, a variant of FS that reasons about wait actions in bulk and is particularly useful for computing bounded-suboptimal paths in environments with dynamic obstacles. We presented ECBS-CT, a generalization of ECBS that efficiently reasons with timeintervals instead of timesteps and is suitable for the MAMP problem. In the high-level search, this requires a proper consideration of completeness, w-suboptimality and Zeno behaviors. In the low-level search, this requires efficient data structures for the reservation table and an efficient implementation of FS-B, for which we developed SCIPP. We also showed that SCIPP can be thought of as a bounded-suboptimal generalization of SIPP that can account for collisions as a secondary heuristic. Experimental results with different motion primitives in different environments demonstrated the efficiency of our approach.

There are many avenues for future work. One of them is to explore the full implications of Theorem 23, which states that FS-B is bounded-suboptimal when the while loop on line 13 is conditioned only on $(v,t) \in \texttt{FOCAL}$ (however, not conditioned on $h_{\texttt{FOCAL}}((v,t)) = h_{\texttt{FOCAL}}((v,t-1))$). This opens the possibility of developing other bounded-suboptimal versions that, similar to SCIPP, expand states in bulk, that is, when expanding $s = (v, [lb_v, ub_v])$, expand more states from FOCAL with configuration v. Second, since the single-agent search space of the MAMP problem tends to be much larger than the single-agent search space of the MAPF problem, it might not always be viable to use $h_{\texttt{SP}}$. In such cases, it is necessary to develop heuristics that trade-off memory requirements for efficiency. Using abstractions of the search space[15] or other preprocessing techniques [12, 23] can be interesting, as well as finding ways of applying ideas from the highway heuristic to abstracted search spaces.

Other directions for future work include experimenting with different secondary heuristics for the high-level and low-level searches, different motion primitives (for example, ones with velocity considerations) and different environments. Moreover, since agents may occupy more than one cell at any time, posting constraints on more than one cell while maintaining bounded-suboptimality guarantees, such as explored

[15] for example, abstracting (x, y, θ) to (x, y)

in [43], could further improve the efficiency of our MAMP solvers. A detailed comparison between HCA* and ECBS-CT with a large w can reveal new insights on ECBS-CT. Finally, none of the heuristics developed for the high-level search, such as in [18, 42], apply to the MAMP problem. Thus, developing applicable heuristics for the high-level search may further increase the efficiency of our MAMP solvers.

Chapter 6

Conclusions and Future Work

In AI, the MAPF problem is a simplified model for cooperative autonomous agents navigating in different environments. Since cooperative autonomous agents navigating in different environments can be useful in many real-world application domains, it is important to find effective solutions to the MAPF problem efficiently. Unfortunately, the MAPF problem is NP-hard to solve optimally according to any of the common cost measures, such as minimizing the total arrival time. Since optimal solvers can have prohibitively long runtimes and computing effective solutions is important, bounded-suboptimal MAPF solvers are of interest.

In this dissertation, we first identified two shortcomings of bounded-suboptimal MAPF solvers that hinder their efficiency. The first shortcoming is the difficulty of determining a suboptimality bound such that an effective solution is found efficiently. To resolve this shortcoming, we developed Anytime BCBS, which finds a "good" solution quickly and refines it to better and better solutions if time allows. The second shortcoming is the inefficiency of bounded-suboptimal MAPF solvers in structured environments. To resolve this shortcoming, we developed the highway heuristic, which encourages a global coordination of agents in avoiding collisions with each other, and ECBS+HWY, a bounded-suboptimal MAPF solver that uses the highway heuristic. Finally, we argued that the MAPF problem is ill-suited when agents are kinodynamically constrained and formalized the MAMP problem, a generalization of the MAPF problem

suitable for kinodynamically constrained agents. To solve the MAMP problem, we developed ECBS-CT, a bounded-suboptimal MAMP solver that reasons efficiently about arbitrary wait durations.

Since our bounded-suboptimal MAPF solvers heavily rely on FS, all the developments above were based on improvements to FS. More specifically, Anytime BCBS is based on AFS, our framework for anytime FS, ECBS+HWY is based on FS with inflated heuristics and ECBS-CT, which uses SCIPP, is based on FS-B. Compared to other heuristic search methods, such as A* and wA*, FS is less restricted in the states it can choose to expand and can take advantage of a broader family of heuristics. Thus, our improvements to FS have a broader impact in domains other than MAPF. For example, in [72], inadmissible (but informative) heuristics are first learned with a deep neural network model, and later used by AFS. Notably, the authors (not affiliated with us) state that:

> "The results of our experiments show that directly applying learned heuristics does not perform well in complex scenarios. Using them as guiding heuristics improves performance considerably, though. Moreover, using (anytime) focal search has the benefit of getting first solutions fast that can be refined over time, always having a guaranteed upper bound on the suboptimality factor."

Possible future directions include further improving the efficiency of bounded-suboptimal MAPF solvers as well as further improving FS:

- *Incremental FS*. Incremental heuristic search methods are suitable for efficient search in dynamic environments. When the environment changes, such methods reuse previous search effort to correct the previously computed path. Instead of computing g-values from scratch every time the environment changes, LPA* [35] propagates corrected g-values from the states affected by the change in the environment. FS can benefit from a similar mechanism. Moreover, FS can potentially truncate the propagation of corrected g-values when the suboptimality bound is guaranteed (similar in spirit to truncated-LPA* [1], a bounded-suboptimal version of LPA*). Incremental heuristic search methods work particularly well when the changes in the environment are relatively small. In the context of

our bounded-suboptimal MAPF solvers, each high-level state adds only one constraint at a time. Thus, incremental FS has the potential to improve the efficiency of our bounded-suboptimal MAPF solvers. (Note, LPA* has recently been shown to increase the efficiency of a MAPF solver [8].)

- *Mechanical Generation of h_{FOCAL}.* Traditionally, the mechanical generation of heuristics is based on distances in a simplified state space of the problem at hand. Simplified state spaces can be mechanically generated by abstractions (that is, merging of states) or relaxations (that is, relaxing the necessary conditions to execute actions between states). For heuristic search methods like A* or wA*, these are the methods of choice because the generated heuristics are consistent. Unfortunately, A* and wA* can often spend a long time exploring many different paths with roughly equivalent costs. In FS, however, $h_{\texttt{FOCAL}}$ is not required to satisfy any property to guarantee the bounded-suboptimality of its solutions. Thus, one can try to generate an $h_{\texttt{FOCAL}}$ that prioritizes the exploration of a specific path, thus focusing FS on a small part of the search space. This, in turn, can significantly increase the efficiency of FS. One possible direction for the mechanical generation of such $h_{\texttt{FOCAL}}$ is to leverage the idea of random parity constraints [24].

- *Anytime Multi-Heuristic FS.* Multi-Heuristic A* (MHA*) [3] is a bounded-suboptimal heuristic search method that is often more efficient than wA*. MHA* uses a "traditional" OPEN list and additional OPEN lists, each of which prioritizes states according to a different heuristic. Focal-MHA* [53] is similar to MHA* but uses FOCAL lists instead of OPEN lists. Like FS, Focal-MHA* has an advantage when states are prioritized according to arbitrary $h_{\texttt{FOCAL}}$s (for example, $h_{\texttt{FOCAL}}$s that do not estimate the cost-to-go). Recently, an anytime version of MHA* was proposed [55]. Based on the ideas behind AFS, it seems possible to develop an anytime version of Focal-MHA* that maintains the bounded-suboptimality guarantees of each search iteration and the efficient reuse of previous search effort.

- *Bidirectional FS*. Bidirectional search methods interleave a forward search (from the start state to the goal state) and a backward search (from the goal state to the start state). This approach can potentially reduce runtime and memory requirements exponentially compared to a forward (or backward) search. Meet in the Middle (MM) [30] is a bidirectional heuristic search method that finds an optimal solution if one exists, and guarantees that its forward and backward heuristic searches meet in the middle. A bidirectional version of FS can potentially have the freedom to use $h_{\texttt{FOCAL}}$ to guide a forward FS and a backward FS to meet faster (but not necessarily in the middle) while guaranteeing bounded-suboptimality.

- *Real-Time FS*. Real-time heuristic search methods interleave planning (often in a small part of the search space that is locally close to the agent's current state) and execution. They are particularly suitable for agents that have to act within a given time limit that is not long enough to compute a complete plan from their current state to a goal state. Learning Real-Time A* (LRTA*) [37] is a real-time heuristic search method that updates heuristic values after each search and is guaranteed to converge (over several planning trails) to an optimal solution. Weighted Local Search Space LRTA* (wLSS-LRTA*) [65] is a variant of LRTA* that inflates heuristic values when they are updated, which allows it to tradeoff convergence speed with solution cost. Like wLSS-LRTA*, a real-time FS can potentially use the additional freedom of using $h_{\texttt{FOCAL}}$ to converge faster to a solution while guaranteeing bounded-suboptimality. Finally, real-time heuristic search was recently shown to be beneficial for MAPF [70], encouraging us to examine the usage of a real-time FS for our bounded-suboptimal MAPF solvers.

Other future directions for MAPF and MAMP solvers (regardless of improving FS) include:

- *Choosing constraints*. Exploring different strategies for choosing which collisions to resolve via constraints can be interesting since the choice of constraints dramatically influences the size and shape of the high-level search tree. Thus, the choice of constraints can be key for improving efficiency.

- *More experiments in MAPF*. Further experimenting with different environments and suboptimal MAPF solvers can help understand better when optimal and bounded-suboptimal MAPF solvers from the CBS family are preferable over suboptimal MAPF solvers such as HCA*. Furthermore, while we show that the bounded-suboptimal MAPF solver M* with $h_{\mathtt{HWY}}$ is more efficient than the optimal MAPF solver M* (see Chapter 4.5.3), no work has been done on using FS with M*. This can potentially allow for a subdimensional expansion that is guided by the number of collisions which is known to be very beneficial for increasing efficiency.

- *More experiments in MAMP*. Further experimenting with different state lattices (both different motion primitives and the number of motion primitives) and different sizes of environments (both different environments and different discretizations of a given environment).

Bibliography

[1] Sandip Aine and Maxim Likhachev. Truncated incremental search: Faster replanning by exploiting suboptimality. In *Proceedings of the AAAI Conference of Artificial Intelligence*, 2013.

[2] Sandip Aine and Maxim Likhachev. Search portfolio with sharing. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling*, 2016.

[3] Sandip Aine, Siddharth Swaminathan, Venkatraman Narayanan, Victor Hwang, and Maxim Likhachev. Multi-heuristic A*. In *Proceedings of the Robotics: Science and Systems Conference*, 2014.

[4] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the 7th Annual Symposium on Combinatorial Search*, 2014.

[5] Jur Den Van Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2008.

[6] Zahy Bnaya, Roni Stern, Ariel Felner, Roie Zivan, and Steven Okamoto. Multi-agent path finding for self interested agents. In *Proceedings of the 6th Annual Symposium on Combinatorial Search*, 2013.

[7] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[8] Eli Boyarski, Ariel Felner, Daniel Harabor, Peter Stuckey, Liron Cohen, Jiaoyang Li, and Sven Koenig. Depth-first conflict-based search with incremental search methods. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.

[9] Ethan Burns, Seth Lemons, Wheeler Ruml, and Rong Zhou. Suboptimal and anytime heuristic search on multi-core machines. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, 2009.

[10] Marcello Cirillo, Tansel Uras, and Sven Koenig. A lattice-based approach to multi-robot motion planning for non-holonomic vehicles. In *Proceedings of the International Conference on Intelligent Robots and Systems*, 2014.

[11] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernandez, Ariel Felner, Sven Koenig, and T.K.Satish Kumar. Anytime focal search with applications. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018.

[12] Liron Cohen, Tansel Uras, Shiva Jahangiri, Aliyah Arunasalam, Sven Koenig, and T.K.Satish Kumar. The fastmap algorithm for shortest path computations. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018.

[13] Liron Cohen, Glenn Wagner, David Chan, Howie Choset, Nathan Sturtevant, Sven Koenig, and T.K.Satish Kumar. Rapid randomized restarts for multi-agent path finding: Preliminary results.

In *Proceedings of the 17th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2018.

[14] Paul Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, 1995.

[15] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[16] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC, 1993.

[17] Ersa Erdem, Doga Kisa, Umut Oztok, and Peter Schueller. A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*, 2013.

[18] Ariel Felner, Jiaoyang Yang, Eli Boyarski, Hang Ma, Liron Cohen, T.K.Satish Kumar, and Sven Koenig. Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling*, 2018.

[19] Dave Ferguson, Thomas Howard, and Maxim Likhachev. Motion planning in urban environments. *Journal of Field Robotics*, 25(11-12):939–960, 2008.

[20] Daniel Gilon, Ariel Felner, and Roni Stern. Dynamic potential search - A new bounded suboptimal search. In *Proceedings of the 9th Annual Symposium on Combinatorial Search*, 2016.

[21] Julio Godoy, Ioannis Karamouzas, Stephen Guy, and Maria Gini. Implicit coordination in crowded multi-agent navigation. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, 2016.

[22] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan R. Sturtevant, Robert C. Holte, and Jonathan Schaeffer. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research*, 50:141–187, 2014.

[23] Meir Goldenberg, Nathan Sturtevant, Ariel Felner, and Jonathan Schaeffer. The compressed differential heuristic. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, 2011.

[24] Carla Gomes, Ashish Sabharwal, and Bart Selman. Near-uniform sampling of combinatorial spaces using xor constraints. In *Proceedings of the 21st International Conference on Neural Information Processing Systems*, 2007.

[25] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[26] Matthew Hatem and Wheeler Ruml. Simpler bounded suboptimal search. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, 2014.

[27] Malte Helmert and Gabriele Roger. How good is almost perfect. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, 2008.

[28] Wolfgang Hoenig, T.K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic contraints. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling*, 2016.

[29] Wolfgang Hoenig, James Preiss, T. K. Satish Kumar, Gaurav Sukhatme, and Nora Ayanian. Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics*, 34(4):856–869, 2018.

[30] Robert Holte, Ariel Felner, Guni Sharon, and Nathan Sturtevant. Bidirectional search that is guaranteed to meet in the middle. In *Proceedings of the 30rd AAAI Conference on Artificial Intelligence*, 2016.

[31] Robert Holte, Ruben Majadas, Alberto Pozanco, and Daniel Borrajo. Error analysis and correction for weighted A*'s suboptimality. In *Proceedings of the 12th International Symposium on Combinatorial Search*, 2019.

[32] Renee Jansen and Nathan Sturtevant. Direction maps for cooperative pathfinding. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008.

[33] Renee Jansen and Nathan Sturtevant. A new approach to cooperative pathfinding. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2008.

[34] Lydia Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[35] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1-2):93–146, 2004.

[36] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[37] Richard Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[38] Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.

[39] James Kuffner and Steven LaValle. RRT-Connect: An efficient approach to single-query path planning. In *Proceedings of the International Conference on Robotics and Automation*, 2000.

[40] Steven LaValle and Seth Hutchinson. Optimal motion planning for multiple robots having independent goals. *IEEE Transactions on Robotics and Automation*, 14(6):912–925, 1998.

[41] Tom Leighton, Clifford Stein, Fillia Makedon, Eva Tardos, Serge Plotkin, and Spyros Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50(2):228 – 243, 1995.

[42] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2019.

[43] Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding for large agents. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.

[44] Maxim Likhachev. *Search-based Planning for Large Dynamic Environments*. PhD thesis, Carnegie Mellon University, 2005.

[45] Maxim Likhachev, Gordon Geoffrey, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*, 2003.

[46] Sikang Liu, Kartik Mohta, Nikolay Atanasov, and Vijay Kumar. Search-based motion planning for aggressive flight in SE(3). *IEEE Robotics and Automation Letters*, 3(3):2439–2446, 2018.

[47] Ryan Luna and Kostas Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011.

[48] Hang Ma, Wolfgang Hoenig, T. K. Satish Kumar, Nora Ayanian, and Sven Koenig. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.

[49] Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems*, 2016.

[50] Hang Ma, Sven Koenig, Nora Ayanian, Liron Cohen, Wolfgang Honig, T. K. Satish Kumar, Tansel Uras, Hong Xu, Craig Tovey, and Guni Sharon. Overview: Generalizations of multi-agent path finding to real-world scenarios. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence Workshop on Multi-Agent Path Finding*, 2016.

[51] Joris M. Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research*, 11:2169–2173, August 2010.

[52] Robert Morris, Corina Pasareanu, Kasper Luckow, Waqar Malik, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Planning, scheduling and monitoring for airport surface operations. In *Proceedings of the Workshop on Planning for Hybrid Systems at the 30th AAAI Conference on Artificial Intelligence*, 2016.

[53] Venkatraman Narayanan, Sandip Aine, and Maxim Likhachev. Improved multi-heuristic A* for searching with uncalibrated heuristics. In *Proceedings of the 8th Annual Symposium on Combinatorial Search*, 2015.

[54] Venkatraman Narayanan, Mike Phillips, and Maxim Likhachev. Anytime safe interval path planning for dynamic environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[55] Ramkumar Natarajan, Muhammad Suhail Saleem, Sandip Aine, Maxim Likhachev, and Howie Choset. A-MHA*: Anytime multi-heuristic A*. In *Proceedings of the International Symposium on Combinatorial Search*, 2019.

[56] Judea Pearl and Jin Kim. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4:392 –399, 1982.

[57] Michael Phillips. *Experience Graphs: Leveraging Experience in Planning*. PhD thesis, Carnegie Mellon University, 2015.

[58] Michael Phillips, Benjamin J. Cohen, Sachin Chitta, and Maxim Likhachev. E-graphs: Bootstrapping planning with experience graphs. In *Proceedings of the Robotics: Science and Systems Conference*, 2012.

[59] Mike Phillips and Maxim Likhachev. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of the International Conference on Robotics and Automation*, 2011.

[60] Mikhail Pivtoraiko, Ross Alan Knepper, and Alonzo Kelly. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 26(3):308–333, 2009.

[61] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3):193–204, 1970.

[62] Ira Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 12–17, 1973.

[63] Thomas Prevot, Joseph Rios, Parimal Kopardekar, John Robinson III, Marcus Johnson, and Jaewoo Jung. UAS traffic management (UTM) concept of operations to safely enable low altitude flight operations. In *Proceedings of the AIAA Aviation Technology, Integration, and Operations Conference*, 2016.

[64] Craig Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, 1987.

[65] Nicolas Rivera, Jorge Baier, and Carlos Hernandez. Incorporating weights into real-time heuristic search. *Artificial Intelligence*, 225:1–23, 2015.

[66] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George Pappas, and Sanjit Seshia. Implan: Scalable incremental motion planning for multi-robot systems. In *Proceedings of the 7th International Conference on Cyber-Physical Systems*, 2016.

[67] Joao Salvado, Robert Krug, Masoumeh Mansouri, and Fedorico Pecora. Motion planning and goal assignment for robot fleets using trajectory optimization. In *Proceedings of the International Conference on Intelligent Robots and Systems*, 2018.

[68] Mohammad H. Shaelaie, Majid Salari, and Zahra Naji-Azimi. The generalized covering traveling salesman problem. *Applied Soft Computing*, 24:867 – 878, 2014.

[69] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[70] Devon Sigurdson, Vadim Bulitko, William Yeoh, Carlos Hernandez, and Sven Koenig. Multi-agent pathfinding with real-time heuristic search. In *IEEE Conference on Computational Intelligence and Games*, 2018.

[71] David Silver. Cooperative pathfinding. In *Proceedings of the 1st AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2005.

[72] Markus Spies, Marco Todescato, Hannes Becker, Patrick Kesper, Nicolai Waniek, and Meng Guo. Bounded suboptimal search with learned heuristics for multi-agent systems. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.

[73] Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, 2010.

[74] Roni Stern, Ariel Felner, Jur van den Berg, Rami Puzis, Rajat Shah, and Ken Goldberg. Potential-based bounded-cost search and anytime non-parametric A*. *Artificial Intelligence Journal*, 214:1–25, 2014.

[75] Roni Stern, Rami Puzis, and Ariel Felner. Potential search: A bounded-cost search algorithm. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling*, 2011.

[76] James Stowe. *Throughput Optimization of Multi-Agent Robotic Automated Warehouses*. PhD thesis, Massachusetts Institute of Technology, 2016.

[77] Pavel Surynek. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *Proceedings of the 12th Pacific Rim International Conference on Artificial Intelligence*, 2012.

[78] Pavel Surynek. A simple approach to solving cooperative path-finding as propositional satisfiability works well. In *Proceedings of the 13th Pacific Rim International Conference on Artificial Intelligence*, 2014.

[79] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *Proceedings of the 22nd European Conference on Artificial Intelligence*, 2016.

[80] Jordan Thayer, J. Benton, and Malte Helmert. Better parameter-free anytime search by minimizing time between solutions. In *Proceedings of the 5th Annual Symposium on Combinatorial Search*, 2012.

[81] Jordan Thayer and Wheeler Ruml. Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, 2008.

[82] Jordan Thayer and Wheeler Ruml. Anytime heuristic search: Frameworks and algorithms. In *Proceedings of the 2nd Annual Symposium on Combinatorial Search*, 2010.

[83] Jordan Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011.

[84] Jur van den Berg, Rajat Shah, Arthur Huang, and Kenneth Y. Goldberg. Anytime nonparametric A*. In *Proceedings of the 25th Conference on Artificial Intelligence*, 2011.

[85] Glenn Wagner. *Subdimensional expansion: a framework for computationally tractable multirobot path planning*. PhD thesis, Carnegie Mellon University, 2015.

[86] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24, 2015.

[87] Thayne Walker, Nathan R. Sturtevant, and Ariel Felner. Extended increasing cost tree search for non-unit cost domains. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018.

[88] Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, 2008.

[89] Boris Wilde, Adriaan Mors, and Cees Witteveen. Push and rotate: Cooperative multi-agent path planning. In *Proceedings of the 12th International Conference on Autonomous Agents and Multi-agent Systems*, 2013.

[90] Christopher Wilt and Wheeler Ruml. When does weighted A* fail? In *Proceedings of the 5th Annual Symposium on Combinatorial Search*, 2012.

[91] Christopher Wilt and Wheeler Ruml. Effective heuristics for suboptimal best-first search. *Journal of Artificial Intelligence Research*, 57:273–306, 2016.

[92] Konstantin Yakovlev and Anton Andreychuk. Any-angle pathfinding for multiple agents based on SIPP algorithm. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling*, 2017.

[93] Jingjin Yu and Steven LaValle. Multi-agent path planning and network flow. In *Algorithmic Foundations of Robotics X, Springer Tracts in Advanced Robotics*, volume 86, pages 157–173. Springer Berlin/Heidelberg, 2013.

[94] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multiple robots on graphs. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2013.

[95] Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*, 2013.

[96] Jun Zhang, Karl Henrik Johansson, John Lygeros, and Shankar Sastry. Zeno hybrid systems. *International Journal of Robust and Nonlinear Control*, 11(5):435–451, 2001.