2022

Multi-Agent Pathfinding in Mixed Discrete-Continuous Time and Space

Thayne T. Walker

Multi-Agent Pathfinding in Mixed Discrete-Continuous Time and Space

A Dissertation

Presented to

the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

Thayne T. Walker

June 2022

Advisor: Nathan R. Sturtevant

Author: Thayne T. Walker
Title: Multi-Agent Pathfinding in Mixed Discrete-Continuous Time and Space
Advisor: Nathan R. Sturtevant
Degree Date: June 2022

Abstract

In the multi-agent pathfinding (MAPF) problem, agents must move from their current locations to their individual destinations while avoiding collisions. Ideally, agents move to their destinations as quickly and efficiently as possible. MAPF has many real-world applications such as navigation, warehouse automation, package delivery and games. Coordination of agents is necessary in order to avoid conflicts, however, it can be very computationally expensive to find mutually conflict-free paths for multiple agents – especially as the number of agents is increased. Existing state-of-the-art algorithms have been focused on simplified problems on grids where agents have no shape or volume, and each action executed by the agents have the same duration, resulting in simplified collision detection and synchronous, timed execution. In the real world agents have a shape, and usually execute actions with variable duration. This thesis re-formulates the MAPF problem definition for continuous actions, designates specific techniques for continuous-time collision detection, re-formulates two popular algorithms for continuous actions and formulates a new algorithm called Conflict-Based Increasing Cost Search (CBICS) for continuous actions.

Acknowledgments

I would like to acknowledge my advisor Nathan Sturtevant who, in addition to a wealth of knowledge and experience, has kindness and patience in abundance. Additionally, Ariel Felner who was a second mentor to me. I am also grateful to Sven Koenig and my friends at USC: Tansil, Liron, Jiaoyang and Han. Certainly nothing would have worked out without Brittny, who provides meaning to my world, and keeps it turning at the same time. Finally, and most enthusiastically I thank God from whom all blessings flow and who gives "beauty for ashes".

. . . if we do not improve our time while in this life, then cometh the night of darkness wherein there can be no labor performed.

ALMA 34:33

Table of Contents

List of Tables ix

List of Figures x

List of Algorithms xiv

1. Introduction 1
- 1.1. Motivation . . . 1
  - 1.1.1. Example 1: Warehouse Automation . . . 2
  - 1.1.2. Example 2: Airport Surface Operations . . . 4
  - 1.1.3. Problem Breakdown of Airport Surface Operations . . . 5
  - 1.1.4. Summary of Motivation . . . 7
- 1.2. Thesis Statement . . . 7
- 1.3. Contributions . . . 7
  - 1.3.1. Background and Survey of MAPF . . . 8
  - 1.3.2. Introduction to General MAPF . . . 8
  - 1.3.3. Extended Increasing Cost Tree Search for General MAPF . . . 8
  - 1.3.4. Extensions to Conflict-Based Search for General MAPF . . . 9
  - 1.3.5. Conflict-Based Increasing Cost Search . . . 9
- 1.4. Summary . . . 10

2. Background 11
- 2.1. Definitions . . . 11
  - 2.1.1. Single-Agent Pathfinding . . . 11
- 2.2. Systematic Search Algorithms . . . 17
  - 2.2.1. The A* Algorithm . . . 18
  - 2.2.2. Heuristics . . . 19
  - 2.2.3. Time and Space Complexity . . . 19
  - 2.2.4. Free-space Search Algorithms . . . 20
- 2.3. The Multi-Agent Pathfinding Problem . . . 22
  - 2.3.1. Classic MAPF . . . 22
  - 2.3.2. General MAPF . . . 25
  - 2.3.3. Objective and Hardness of MAPF . . . 25
  - 2.3.4. Variants of MAPF . . . 27
  - 2.3.5. Agent Movement and Type Variants . . . 28

2.4. Prior Work in MAPF . . . 29
2.4.1. Centralized and Distributed Algorithms . . . 29
2.4.2. Coupled Algorithms . . . 30
2.4.3. Decoupled Algorithms . . . 33
2.4.4. Dynamically Coupled Algorithms . . . 35
2.4.5. Other Techniques Related to the MAPF Problem . . . 38
2.5. Summary . . . 39

3. General MAPF **40**
3.1. Definition of General MAPF . . . 40
3.2. Conflict Detection In General MAPF . . . 42
3.2.1. Geometric Containers . . . 43
3.2.2. Sampling-Based . . . 44
3.2.3. Algebraic . . . 44
3.2.4. Geometric . . . 45
3.2.5. Summary . . . 47
3.3. Completeness in General MAPF . . . 47
3.3.1. Temporally-Relative Duplicate Pruning . . . 50
3.3.2. Temporally-Relative Duplicate Pruning in General MAPF . . . 53
3.3.3. Temporally-Relative Duplicate Pruning for a Subset of Agents . . . 54
3.3.4. Summary . . . 55
3.4. Wait Times in General MAPF . . . 55
3.4.1. Durative Conflicts . . . 56
3.4.2. Arbitrary Wait Times . . . 56
3.4.3. Determining Arbitrary Wait Time . . . 58
3.4.4. Implementing Arbitrary Wait Times in MAPF . . . 60
3.5. The Conflict Avoidance Table . . . 60
3.6. Summary . . . 65

4. Extensions for Increasing-Cost Tree Search **66**
4.1. The Increasing-Cost Tree Search . . . 66
4.1.1. High Level . . . 66
4.1.2. Low Level . . . 67
4.1.3. Pruning Enhancements . . . 67
4.2. Sufficient Conditions for Completeness and Optimality . . . 68
4.3. Re-Formulation for General MAPF . . . 70
4.3.1. Reformulated High Level Search . . . 71
4.3.2. Sufficient Conditions for Optimality and Completeness: . . . 73
4.3.3. Reformulated Low Level Search . . . 74
4.3.4. Pruning Enhancements . . . 75
4.4. Sub-Optimal Variants . . . 75
4.4.1. ϵ-ICTS . . . 76
4.4.2. w-ICTS . . . 76

4.5. Theoretical Analysis . . . 76
4.5.1. MDD Size . . . 77
4.5.2. Low Level Search Complexity . . . 77
4.5.3. High Level Search Complexity . . . 78
4.5.4. Sub-Optimal Variants . . . 78
4.6. Experimental Results and Analysis . . . 78
4.6.1. OD-Style Versus Full Branching . . . 78
4.6.2. Best Setting for δ . . . 80
4.6.3. ICTS versus A* and CBS . . . 80
4.6.4. Sub-optimal variants . . . 81
4.7. Summary . . . 82

5. Extensions for Conflict-Based Search 84
5.1. Conflict-Based Search . . . 84
5.2. Conflicts . . . 86
5.2.1. Conflict Types . . . 86
5.2.2. Conflict Symmetries . . . 87
5.3. Constraints . . . 89
5.3.1. Basic Constraint Types . . . 89
5.3.2. Constraint Sets and Constraint Correctness . . . 91
5.3.3. Context-Specific Symmetry Breaking in Classic MAPF . . . 95
5.4. Time-Annotated Biclique Constraints . . . 96
5.4.1. Ensuring Completeness and Optimality . . . 97
5.4.2. Reduction to Bipartite Graphs . . . 97
5.4.3. Constraint Set Construction Using Bicliques . . . 98
5.4.4. Additional Variants . . . 100
5.4.5. Empirical Results . . . 101
5.5. Continuous-Time Mutex Propagation . . . 105
5.5.1. Background: Unit-Cost Mutex Propagation . . . 105
5.5.2. Continuous-Time Mutex Propagation . . . 107
5.6. Low Level Search . . . 109
5.7. High Level Search . . . 110
5.7.1. High-Level Heuristics . . . 110
5.7.2. Additional Enhancements of CBS . . . 112
5.7.3. Empirical Results . . . 113
5.8. Sub-Optimal Variants of CBS . . . 118
5.9. Constraint Layering . . . 118
5.9.1. Abstraction For Conflict Avoidance . . . 120
5.9.2. Constraint Layering for Conflict Avoidance . . . 121
5.9.3. Theoretical Analysis . . . 122
5.9.4. Airplane Domain Test Environment . . . 122
5.9.5. Experimental Results and Analysis . . . 125
5.9.6. Experimental Implementations . . . 126

5.9.7. Qualitative and Performance Results . . . 126
5.10. Sub-Optimal, Complete Constraints . . . 129
5.10.1. Conditional Constraints . . . 129
5.10.2. Theoretical Analysis . . . 132
5.10.3. The Conflicting Paths Strategy . . . 133
5.10.4. Empirical Results . . . 134
5.11. Theoretical Analysis of CBS for General MAPF . . . 137
5.11.1. Computational Complexity . . . 137
5.11.2. Sufficient Conditions for Completeness and Optimality . . . 139
5.12. Summary . . . 141

6. Conflict-Based Increasing Cost Search 143
6.1. Introduction . . . 143
6.2. CBICS High Level . . . 145
6.2.1. Motion Constraint Sets for Splitting . . . 149
6.2.2. Conjunctive Splitting . . . 150
6.2.3. Disjunctive Splitting . . . 150
6.2.4. Cost-Conditional Motion Constraints . . . 150
6.2.5. High-Level Search . . . 151
6.3. Pairwise Constraint Search . . . 153
6.4. Sufficient Conditions for Completeness and Optimality . . . 160
6.4.1. Additional Enhancements . . . 166
6.4.2. Analysis . . . 167
6.5. Empirical Results . . . 169
6.5.1. CBICS Enhancements . . . 169
6.5.2. Comparison with CBS . . . 173
6.6. Summary . . . 182

7. Revisiting Our Motivating Examples 183
7.1. The Warehouse Domain . . . 183
7.2. The Airport Surface Operations Domain . . . 189
7.3. Summary . . . 193

8. Summary 195
8.1. Summary of Contributions . . . 195
8.2. Open Questions and Future Work . . . 198
8.2.1. Open Questions . . . 198
8.2.2. Future Work . . . 199

Bibliography 200

Appendix A. Closed-Form Collision Detection for Circular Agents 222
A.0.1. Constant Velocity . . . 222

A.0.2. Initial Velocity with Constant Acceleration 224
A.1. Computing the Exact Conflict Interval For Circular Agents 225
A.1.1. Constant Velocity 226
A.1.2. Initial Velocity with Constant Acceleration 227
A.2. Determining Exact Minimum Delay or Velocity Adjustment for Conflict Avoidance for Circular Agents 227
A.2.1. Exact Delay for Constant Velocity 227
A.2.2. Exact Delay for Initial Velocity with Constant Acceleration 232
A.2.3. Minimum Velocity Change for Constant Velocity 233
A.2.4. Summary 234

Appendix B. Proofs for Temporally-Relative Duplicate Pruning 235

Appendix C. Proofs for PCS 239

List of Tables

3.1. Mean of *ratio* statistic for 1000 samples 64

4.1. Comparison of average and worst-case scenario for OD-style versus full branching for 10 agents on 8x8 grids. 79
4.2. ICTS and sub-optimal variant performance for various branching factors planning for 30 agents on a 64x64 grid. 82

5.1. A summary of conflict types and applicable constraints for Classic MAPF 90
5.2. A summary of conflict types and applicable constraints for General MAPF 92
5.3. Total problems solved in under 30 seconds on 8-neighbor grid MAPF benchmarks 102
5.4. Total problems solved in under 30 seconds on 16-neighbor grid MAPF benchmarks 103
5.5. Total problems solved in under 30 seconds on 32-neighbor grid MAPF benchmarks 104
5.6. Final size of CT on 16-connected grids 105
5.7. High and Low-Level Work by Configuration 128
5.8. Quality and Performance by Configuration 128
5.9. Total problems solved in under 30 seconds 135
5.10. Comparison of solution quality on 4- and 16-neighbor grids 136

6.1. Total problems solved in under 30 seconds on 4-neighbor grid MAPF benchmarks 178
6.2. Total problems solved in under 30 seconds on 8-neighbor grid MAPF benchmarks 179
6.3. Total problems solved in under 30 seconds on 16-neighbor grid MAPF benchmarks 180
6.4. Nodes expanded for small-map benchmarks with 10 agents. 181
6.5. Low-level runtime for benchmarks with 10 agents. 181

7.1. Average solution cost and runtime for warehouse benchmarks with varying grid connectivity. 187
7.2. Average optimal and sub-optimal solution costs for warehouse benchmarks with varying grid connectivity. 187
7.3. Average runtime for warehouse benchmarks with varying grid connectivity. 187

List of Figures

1.1. Warehouse Robots and Shelving Units [210] 2
1.2. An example of an automated warehouse layout with staging areas in the center in black, and with loading and picking stations on the sides in gray. 3
1.3. An example of (a) a path which uses unit duration actions and (b) a path which uses non-unit duration (diagonal) actions. 3
1.4. An airport taxiway intersection 4
1.5. Aircraft footprint on a taxiway 5
1.6. Daily Airport Volumes (a) and number of runways (b) at popular airports. 6

2.1. (a) A start configuration and (b) goal configuration for the fifteen puzzle. 12
2.2. (a) A pathfinding problem and (b) a path that solves the problem. 14
2.3. 2^k Neighborhood movement models for k = 2,3,4 and 5. 16
2.4. Set of optimal path edges with 4-neighbor (a), 8-neighbor (b), and 16-neighbor (c) grids. 16
2.5. A MAPF problem instance (a) and solution (b). 22
2.6. An example of the pareto-optimal cost structure of MAPF: (a) A MAPF problem instance, (b) a solution favoring the red agent and (c) a solution favoring the blue agent. 26
2.7. Taxonomy of MAPF Algorithms 37

3.1. Conflict types in General MAPF: (a) vertex conflict, (b) edge conflict, (c) intersecting-edge conflict, (d) non-intersecting edge conflict and (e) edge-vertex conflict 41
3.2. Collision Detection using geometric containers. A collision is correctly detected between agents (a) and (b), but erroneously detected between (a) and (c). 43
3.3. Sampling-Based Collision Detection. A collision is correctly detected between agents (a) and (b). 44
3.4. Collision times for agent trajectories with constant velocity (a) and initial velocity with constant acceleration (b) can be solved algebraically 45
3.5. Constructive Solid Geometry Collision Detection. Time is *extruded* into the model as an extra dimension, after which polygonal intersection detection is performed. 46
3.6. Velocity Obstacle (VO) construction based on (a) two agents with motion vectors. The trajectories and shapes of agents are interpreted to create (b) the velocity obstacle – labeled 'VO' 46

3.7. Illustration of (a) a MAPF instance with no feasible solution, (b) a corresponding breadth-first search tree and (c) a breadth-first search tree with wait actions of $1/2$ . . . 49
3.8. Pathological cases for General MAPF. . . . 57
3.9. (a) Agent trajectories, (b) squared distance plot and (c) ellipse showing collision intervals for varying *delay*. . . . 60
3.10. Set of optimal path edges with (a) 4-neighbor, (b) 8-neighbor, and (c) 24-connected grids. *area*, the number of grid cells in optimal paths, is 15, 9 and 3 respectively. . . . 62
3.11. Comparison of the reduction in mean number of total conflicts in CBS when using a conflict avoidance table planning on an unobstructed 64×64 grid. . . . 64

4.1. Unit-cost ICT with cost vectors . . . 67
4.2. (a) Problem instance and (b) associated MDD for all paths of cost between 2 and 3 for 4-Connected Grid (b) and (c) Fully Connected Grid . . 71
4.3. Reformulated ICT with cost interval vectors . . . 73
4.4. Performance of ICTS versus A* and CBS on 4, 8 and 16-neighbor grids. . 80
4.5. Performance of sub-optimal variants versus ICTS on 4, 8 and 16-neighbor grids. . . . 81

5.1. (a) An example Classic MAPF instance and (b) a *partial* CT for the instance. 84
5.2. Illustration of Classic MAPF instances with conflict symmetries: (a) a rectangle conflict, (b) a corridor conflict and (c) a swapping conflict. . . . 88
5.3. Example showing (a) a simple MAPF problem for circular agents i and j, (b) enumerated actions for each agent, (c) a possible CT for the problem and (d) a possible CT where multiple actions are blocked at a time. . . . 94
5.4. Illustration of rectangle reasoning and barrier constraints . . . 96
5.5. Illustration of (a) actions for two agents, (b) the corresponding bipartite conflict graph and (c) the corresponding time-annotated biclique. . . . 97
5.6. Example of (a) a TAB with (b) corresponding unsafe intervals plotted on a concurrent timeline. . . . 99
5.7. (a) Mutex propagation for agents x and z for the MAPF instance in Figure 5.1(a) with cost limits $\langle 2, -, 2 \rangle$ and (b) the same analysis as (a) but with cost limits $\langle 3, -, 2 \rangle$. . . . 106
5.8. An example of determining motion constraint sets using a bipartite conflict graph. (a) A General MAPF instance, (b) an enumeration of actions available to the agents with wait actions omitted, (c) a bipartite conflict graph for the problem in part (b); a biclique is shown with thick lines and (d) the same bipartite conflict graph with bi-complete nodes in bold and a biclique for them in thick lines. . . . 107

5.9. An example of determining motion constraint sets using mutex propagation. (a) Paths with a cost limit of 2.4, (b) an enumeration of actions in the respective MDDs, (c) steps in continuous-time mutex propagation. . 108
5.10. Success rates of CBS and CBS+MP for benchmark problems on 4-neighbor grids. 114
5.11. Success rates of CBS, CBS+BC and CBS+MP+BC for benchmark problems on 8-neighbor grids. 116
5.12. Success rates of CBS, CBS+BC and CBS+MP+BC for benchmark problems on 16-neighbor grids. 117
5.13. Continuum of constraint abstraction. 119
5.14. Comparison of grid-based search with differing movement constraints. In (a), movement for both agents is constrained to be on 4-neighbor grids, thus all optimal path combinations will conflict in any of the grey squares. In (b), agent S_2 moves on a 4-neighbor grid, but agent S_1 moves on an 8-neighbor grid. 121
5.15. Grid-based vertical and horizontal aircraft movement model for aircraft. 123
5.16. Comparison of performance of different environment configurations . . 127
5.17. Illustration of (a) sets of available actions for two agents and (b) the corresponding BCG. 130
5.18. Example showing (a) an example problem instance, (b) an optimal solution blocked by a conditional constraint and (c) a sub-optimal solution caused by the conditional constraint. 133
5.19. Illustration of (a) regular CBS constraint allocation and (b) allocation with the conflicting paths strategy. 134
5.20. Success rate of sub-optimal variants 136

6.1. (a) An example unit time MAPF instance, (b) a *partial* ICT with an implied δ of 1 for the MAPF instance and (c) a *partial* CT for the MAPF instance. 144
6.2. The *entire* CRCT for the MAPF instance in Figure 6.1(a). 145
6.3. (a) The combined cost range of agents y and z (a) for node A and (b) for nodes B, C and D of Figure 6.2. 146
6.4. An example of how PCS finds constraints. (a) Paths with a cost limit of 2.4 for agent x and 3.0 for agent y, (b) an enumeration of actions available to the agents including diagonal actions with wait actions omitted, (c) continuous-time mutex propagation. 154
6.5. Structure of a CBICS tree. 167
6.6. Success rates of CBICS, CBICS+PCST and CBICS+PCON of CBICS for benchmark problems on 4-neighbor grids. 170
6.7. Success rates of CBICS, CBICS+PCST and CBICS+PCON for benchmark problems on 8-neighbor grids. 171
6.8. Success rates of CBICS, CBICS+PCST and CBICS+PCON for benchmark problems on 16-neighbor grids. 172

6.9. Success rates of CBS+BC and two variants of CBICS for benchmark problems on 4-neighbor grids. 174
6.10. Success rates of CBS+BC, CBS+BC+MP and two variants of CBICS for benchmark problems on 8-neighbor grids. 175
6.11. Success rates of CBS+BC, CBS+BC+MP and two variants of CBICS for benchmark problems on 16-neighbor grids. 176

7.1. An example of (a) a path which uses actions on a 4-neighbor grid and (b) a path which uses actions on a 16-neighbor grid. 183
7.2. Success rates of CBS and CBICS for warehouse benchmark problems on 4-neighbor grids. 184
7.3. Success rates of CBS and CBICS for warehouse benchmark problems on 8-neighbor grids. 185
7.4. Success rates of CBS and CBICS for warehouse benchmark problems on 16-neighbor grids. 186
7.5. Edmonton International Airport: View from satellite 189
7.6. Edmonton International Airport: Closeup of tarmac lane lines with numbered intersections 190
7.7. Edmonton International Airport: View in simulation. 191
7.8. Edmonton International Airport: Closeup of tarmac graph in simulation. 192
7.9. Success rates of CBS variants and variants of CBICS for the airport surface operations domain. 194

A.1. Agents Trajectories and Corresponding Squared Distance Plot 226
A.2. (a) Agent trajectories, squared distance plot and ellipse showing collision intervals for Varying *delay* and (b) the same trajectories where the red agent is delayed delayed by 0.2 seconds 230
A.3. An example where the maximum delay time happens after the first agent arrives at its destination. 231
A.4. Velocity Obstacle (VO) construction based on (a) two agents moving on edges. (b) The minimum change for safe velocity is determined by the intersection points of the edge and the velocity obstacle. 233

List of Algorithms

2.1.A* Algorithm 19
2.2.Conflict-Aware Multi-Agent Expansion 32
2.3.Independence Detection 35

4.1.Reformulated ICTS High Level Search 72
4.2.reformulated ICTS Low Level Search 75

5.1.Conflict-Based Search 85
5.2.Compute Largest Time-Annotated Biclique 99
5.3.Expand-CT-Node 131

6.1.CBICS Algorithm 148
6.2.Pairwise Constraint Search Algorithm 157
6.3.Joint Expansion with Temporally-Relative Duplicate Pruning 158

A.1Unsafe Interval Computation for Segmented Motion 232

1. Introduction

In the multi-agent pathfinding (MAPF) problem, agents must move from their current locations to their individual destinations while avoiding collisions. Ideally, agents move to their destinations as quickly and efficiently as possible. MAPF has many real-world applications [122] such as navigation [176], warehouse automation [210, 120, 75, 109], package delivery [32], airport surface operations [187, 166, 124, 129, 204, 128], games [164, 23], firefighting [152], surveying [95], security patrolling [30, 105], search and rescue [157], intersection management [43] and farming [14]. Coordination of agents is necessary in order to avoid conflicts, however, finding optimized, mutually conflict-free paths for multiple agents is computationally difficult. The difficulty of the problem increases exponentially as the number of agents is increased [217].

Most state-of-the-art algorithms [52, 85] are tailored for a simplified model of the MAPF problem where agents move on regularly-spaced grids, have no shape or volume, and each action has the same duration. This results in simplified collision detection and synchronous, timed execution [97, 173]. However, in the real world, agents have a shape, and may execute actions with variable duration [78].

1.1. Motivation

The primary motivation for studying MAPF in continuous-time settings is to allow the practitioner to apply MAPF algorithms to any multi-agent scenario – whether it be controlled domains with a limited set of actions of equal duration and or domains with a large set of actions in continuous time.

We now present two motivating examples, however, we do not limit the scope of this thesis to these examples. This thesis focuses on general *one-shot planning*, that is, building a set of conflict-free paths for all agents from start to goal in virtually any type of MAPF domain.

1.1.1. Example 1: Warehouse Automation

Warehouse automation is critical for large modern retail and distribution companies. A popular approach to warehouse automation uses movable rectangular shelving units and autonomous robots as shown in Figure 1.1 [210]. Autonomous robots are used to transport the shelving units between loading and picking stations for adding and removing goods from the shelves and staging areas. The shelving units in staging areas are arranged in rows such that they can always be accessed from at least one side. Such an arrangement is shown in Figure 1.2.

A significant amount of work has studied approaches to MAPF in this domain [74, 119, 120, 114, 75, 133, 145, 109]. All of these works use unit-cost actions in cardinal directions (up, down, left or right) only. A path with cardinal direction ac-

Figure 1.1.: Warehouse Robots and Shelving Units [210]

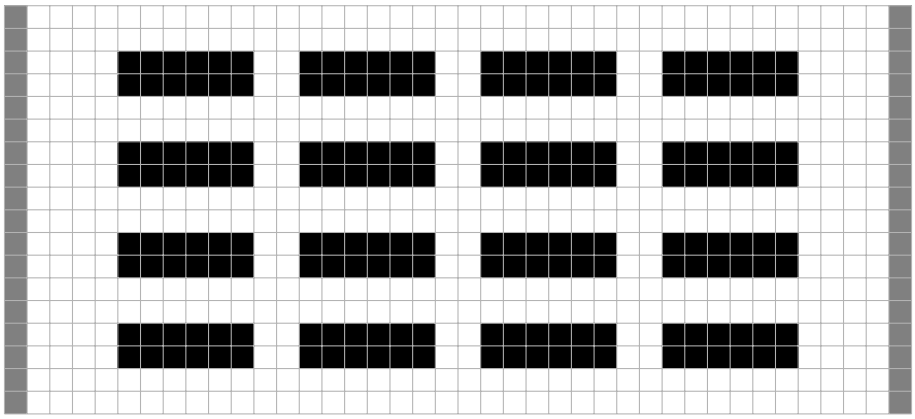
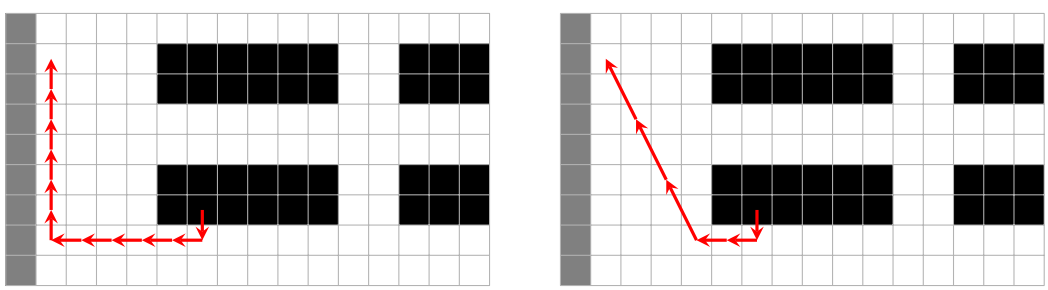


Figure 1.2.: An example of an automated warehouse layout with staging areas in the center in black, and with loading and picking stations on the sides in gray.

tions is shown in Figure 1.3(a). This figure only shows part of the warehouse. There are several paths that the robot could take, however, Figure 1.3(a) shows the one that requires the fewest turns. Keep in mind that other robots with different, concurrent paths would be executing in the same area.

Figure 1.3(b) shows a path which is not restricted to cardinal directions. This path is shorter than the one in part (a). Assuming that robots are capable of turning to arbitrary angles, such paths with arbitrary angles could result in significant cost savings. To put this in perspective, assuming the size of the grid squares in the figure



(a) (b)

Figure 1.3.: An example of (a) a path which uses unit duration actions and (b) a path which uses non-unit duration (diagonal) actions.

are 1m, then the length of the path in part (a) is 13m. The length of the path in part (b) is about 9.7m., a reduction of about 25%. Assuming 1 watt of energy consumption per meter, mean speeds of 1mps, 100 robots, and a use rate of 80% per robot in 24/7, 365 day operations. Such an update to path planning would result in up to 630 megawatts of savings per year. Not to mention up to 25% savings in maintenance and repair costs.

While such an update to path planning in warehouses would yield shorter paths, there are other trade-offs such as that it may introduce a higher risk of collisions, it may incur a requirement for more complex robotic control algorithms, and computing the plans becomes more complex. This thesis develops methods for computing the plans but we do not consider the broader trade-offs.

1.1.2. Example 2: Airport Surface Operations

The airport surface operations domain (ASO) [187, 166, 124, 129, 204, 128] is a familiar one for most modern travelers. Aircraft pick up passengers and cargo at a specific gate of an airport and deliver them to a specific destination. In order for aircraft to take off, they must do so via a restricted resource: the runway. Runways only admit one aircraft at a time for takeoff or landing. Airports may have more than one runway, and those runways may be reached from the gates via shared paved areas

Figure 1.4.: An airport taxiway intersection

called taxiways as shown in Figure 1.4. Taxiways and runways are used by both aircraft leaving the airport as well as those arriving at an airport.

ASO poses a particular challenge for current MAPF algorithms for several reasons. First, taxiways are mostly composed of long, narrow corridors which cause bottlenecks in planning. Second, path segments are of variable length, resulting in a need for continuous-time planning. Third, aircraft have a large footprint, often resulting in motion paths that may conflict even when aircraft are on taxiing on separate taxiway segments.

1.1.3. Problem Breakdown of Airport Surface Operations

The airport surface operations problem is very complex and involves the following factors:

- Aircraft have a large, variable-size footprint (See Figure 1.5) and kinematic constraints due to heterogeneous size, weight and power [74, 37, 110]. This requires continuous-time collision detection and planning [6, 198].

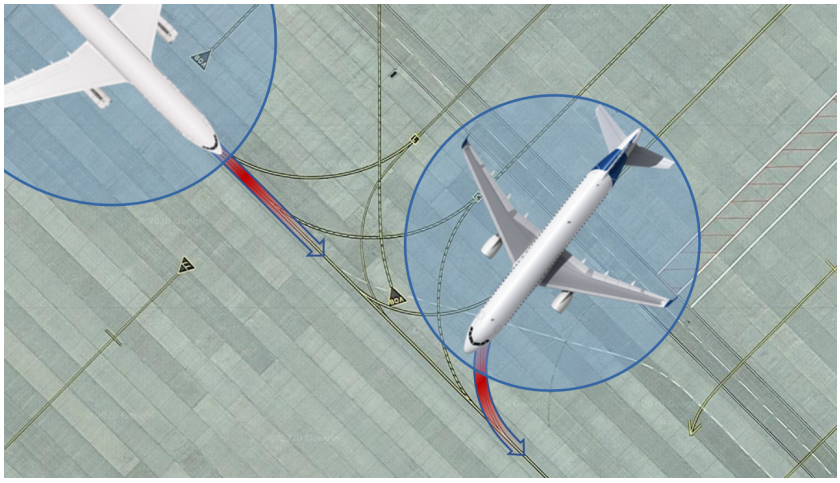


Figure 1.5.: Aircraft footprint on a taxiway

- Aircraft may be required to wait in place in runway queues or at gates and when yielding to other vehicles and aircraft. This means that variable wait times (as opposed to fixed-duration ones) are important [37, 6, 143].

- Due to different needs for motion planning, a solution may require *mixed-domain* planning. For example, a mixture of grids for tow cars and support vehicles and taxiway graphs for aircraft [196].

- Airport taxiway maps exist in various forms such as imagery and polygons. These maps can be processed into plannable representations. [104, 213]

- Although airports may experience very heavy volumes (see Figure 1.6 (a)) [1], they often have few runways (see Figure 1.6 (b)). This can cause delays, especially when movement on the ground is congested or obstructed.

- For safety reasons, the planner must be able to scale to perhaps dozens of simultaneous agents and thousands of flights per day. Can we scale to the level of YEG (400 flights per day)?, DEN (1,500+ flights per day)?, LAX (1,900+ flights per day)? or ATL (2,500+ flights per day [that's one flight every 28.8 seconds!])? Assuming an average of a 10-minute taxi from gate to runway, that is about 21

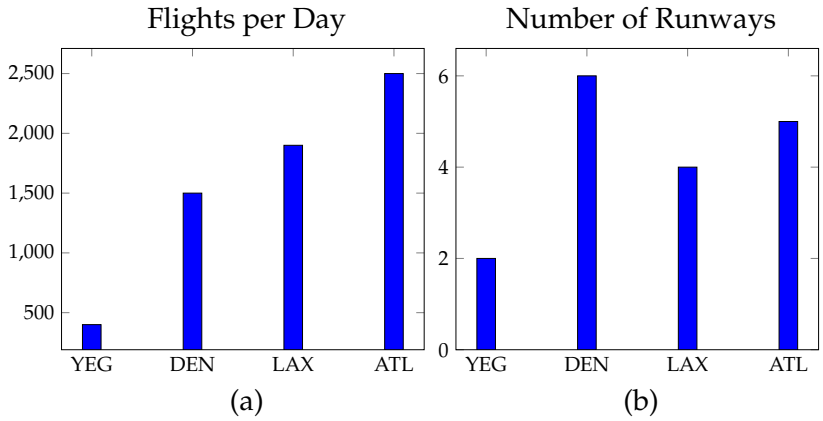


Figure 1.6.: Daily Airport Volumes (a) and number of runways (b) at popular airports.

aircraft actively taxiing at all times. Of course, these estimates do not account for peak hours of the day which may be much busier.

1.1.4. Summary of Motivation

These examples illustrate (1) potential savings can be achieved by introducing arbitrary-length actions into unit-length action spaces and (2) challenging domains exist that cannot be solved efficiently or with proper fidelity without the use of arbitrary-length actions. We believe there are many applications of which these examples are illustrative. Thus this thesis tackles algorithms for this broad class of problems.

1.2. Thesis Statement

Classic MAPF problems have two simplifying assumptions: Agents have no shape and their actions always have the same duration. Multi-agent coordination in continuous-time domains has been studied before [89, 103, 141], however, due to the complexity of the problem the approaches were not always optimal nor scalable to more than a handful of agents. On the other hand, state-of-the-art algorithms for Classic MAPF scale up to dozens or even hundreds of agents [52, 85]. This thesis studies MAPF without these simplifying assumptions.

Our thesis is that: **(1) We can relax the simplifying assumptions of Classic MAPF to allow continuous-time actions for agents with a shape** and **(2) Efficient algorithms can be formulated to exploit the properties of continuous-time actions.**

1.3. Contributions

Many contributions are contained in this dissertation. This section provides a brief overview of each. A more detailed comprehensive summary can also be found in Chapter 8. Additionally, a summary and relevant experimental results are found at the end of each chapter.

1.3.1. Background and Survey of MAPF

Chapter 2 begins with basic definitions for the single-agent pathfinding problem, and extends these definitions for multiple agents. It introduces the Classic MAPF problem; covering its components, objectives and and variants. This is followed by a comprehensive survey of prior work on MAPF, including a detailed taxonomy of MAPF algorithms.

1.3.2. Introduction to General MAPF

Chapter 3 formally defines General MAPF. General MAPF does not only mean continu-ous-time actions. It also introduces variable speeds and non-planar graphs. This induces the need for specialized conflict detection, conflict anticipation and wait-time calculations.

This chapter supplies a taxonomy of collision detection techniques suitable for segmented, continuous-time motion. It then elaborates on algebraic methods for collision detection for the cases of constant velocity and initial velocity with constant acceleration for circular and spherical agents. It also covers methods for collision avoidance, both for the cases where agents are allowed to wait and/or change velocity.

Search-based algorithms generate search nodes representing the state of agents. This chapter includes the algorithm for multi-agent search node generation which takes conficts and continuous-time into account. This algorithm paves the way for centralized search algorithms such as A* [69] and IDA* [96] to be easily built for non-unit time steps.

1.3.3. Extended Increasing Cost Tree Search for General MAPF

Chapter 4 contains a re-formulation of the Increasing Cost Tree Search (ICTS) algorithm [163] for General MAPF. The ICTS algorithm was originally formulated for classic MAPF [173] and works by exploring discrete cost vectors representing cost combinations for paths which make up a solution. For continuous-time actions, which may

have costs in a continuous range, ICTS cannot generate all possible cost vectors. Chapter 4 introduces a re-formulation of ICTS for General MAPF: Extended ICTS (E-ICTS) which changes the high level search to use cost *range* vectors and changes the vector verification step to solve an optimization problem instead of a feasibility problem. Additionally, Chapter 4 includes novel bounded sub-optimal variants of ICTS: w-ICTS and ϵ-ICTS.

1.3.4. Extensions to Conflict-Based Search for General MAPF

Chapter 5 describes new extensions for the Conflict-Based Search (CBS) algorithm [160] for General MAPF. Conflict-Based Search (CBS) [160] operates by systematically discovering and resolving conflicts between agents. Conflicts are resolved by adding constraints to agents. This dissertation presents two extensions of CBS for General MAPF, namely novel biclique constraints and a reformulation of mutex propagation [218] constraints for continuous-time. We additionally introduce two new unbounded sub-optimal variants of CBS.

1.3.5. Conflict-Based Increasing Cost Search

Chapter 6 describes the new Conflict-Based Increasing Cost Search (CBICS) algorithm. This algorithm is a hybrid of the newly extended versions of the CBS and ICTS algorithms. Conflict symmetries (see Section 5.2.2) are a key weakness of conventional MAPF algorithms, and the primary motivation for the creation of CBICS is to robustly resolve these symmetries. CBICS recognizes that ICTS is naturally robust to conflict symmetries that are extended over an area and that with the reformulation of mutex-propagation from Chapter 5, CBS is robust to conflict symmetries that are extended in time. CBICS combines both of these strengths.

CBICS differs from both ICTS and CBS because it allows non-uniform cost vector increases and uses a new technique called conjunctive splitting, which allows constraints to be applied to multiple sub-trees of the search simultaneously, often resulting

in a higher degree of pruning. We show that CBICS runs faster than CBS in a variety of General MAPF settings.

1.4. Summary

The research performed to date has advanced state of the art by introducing novel concepts and/or combining existing concepts in a novel way. This research allows MAPF algorithms to be used in more realistic real-world scenarios (e.g., with kinematic motion constraints, etc.) where previous algorithms could not. The rest of this thesis presents the work done so far to accomplish the objectives.

2. Background

2.1. Definitions

2.1.1. Single-Agent Pathfinding

In order to define the multi-agent pathfinding problem, it is helpful to first define the single-agent pathfinding problem.

Problem Definition

The objective of a *pathfinding problem* is to find a path (especially a shortest path) between two points. Formally, the pathfinding problem is defined as a tuple $\langle s_0, s_g, A \rangle$ where s_0 is an initial *state*, also called a *start* state or *source* state, s_g is a *goal* state or *target* state. For example, Figure 2.1(a) shows a start state and Figure 2.1(b) shows the goal state for the fifteen puzzle [165].

State: A representation of the physical or abstract attitude, location or configuration of an agent. A state can take on many representations such as a configuration of the tiles in the fifteen puzzle shown in Figure 2.1, GPS coordinates, a set of joint angles such as in a robotic arm, game piece positioning on a board, etc. A state is also known as a *configuration* [116] in some literature.

The goal can also be expressed as a set of states or even as a set of conditions. For example, when moving a robot across a warehouse from an initial location to a destination such as a docking station, there may be a specific docking station specified as the goal, or perhaps any docking station will do. For the purposes of this paper s_g is a single state. A is a set of *actions*.

13	10	11	6
5	7	4	8
1	12	14	9
3	15	2	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 2.1.: (a) A start configuration and (b) goal configuration for the fifteen puzzle.

Action: A rule for transition of an agent from one state to another, $s \rightarrow s'$. In the sliding tile puzzle shown in Figure 2.1, the set of possible actions (for the blank space) is A={"up","down","left","right"}. In some domains $|A|$ could be infinite. Not all actions are necessarily valid in all states. Actions may have preconditions and/or duration times. For example, a rook on a chess board may only move in a straight line, provided that there are no obstructions and that doing so will not put its king in check. The duration of an action for a rook is instantaneous and the resulting state for the rook is its new location. As another example, a robotic arm may perform a rotation at a wrist joint. There are no special prerequisites, but there is a specific time duration required. The new state for the wrist joint is its new orientation angle.

In this paper, we additionally define an action as a pair of start and stop states:

$$a = (s, s')$$

This definition differs from the traditional definition of an action as a transition function (e.g., as in reinforcement learning [181]). We use the two definitions interchangeably in this thesis and will specifically mention which definition is being used when it is important to do so. The state-pair definition is intuitive for explicit graphs where an action is equivalent to an edge, it is also useful for reasoning about multi-agent actions.

Typically an action has a cost. The cost could be almost anything such as a time duration, fuel cost, distance traveled, etc. We define the cost of an action a as a function:

$$c(a) = c(s, s')$$

Especially in the context of multi-agent systems, actions have a duration, the amount of time it takes to transition from s to s'. We also define this as a function:

$$d(a)$$

We may treat cost and duration either as equivalent or separate quantities depending on our objectives. The result of pathfinding is a *path*.

Path: π=$[s_0, ..., s_g]$, a sequence of n=$|\pi|$ states where the first state is s_0, and the last state is s_g, such that each pair of sequential states represents a valid action in A and for each pair of sequential actions a and a', $a.s'$=$a'.s$. Alternatively, a path can be expressed as a sequence of $n-1$ actions $\pi = [a_0, .., a_{n-1}]$. A path also has a cost and a duration function:

$$c(\pi) = \sum_{i=0}^{n-1} c(a_i) = \sum_{i=0}^{n-1} c(s_i, s_{i+1})$$

$$d(\pi) = \sum_{i=0}^{n-1} d(a_i) = \sum_{i=0}^{n-1} d(s_i, s_{i+1})$$

An example of a pathfinding problem is shown in Figure 2.2 (a). In this example, states are positions on the grid. s_0 is indicated in the bottom-left corner and s_g is indicated as the destination. The set of actions A is for the agent to transition from one grid cell to any unblocked adjacent grid cell (including diagonals). Blocked cells are filled in with black. The red dashed arrow indicates the objective, which is to find a path that leads from s_0 to s_g. Figure 2.2 (b) shows a sequence of actions (indicated by red arrows) that result in a path which solves the problem.

State Space: Also known as a *configuration space* or *C-space*, is the set of all possible states or configurations for an agent. In Figure 2.2, the state space is all cells

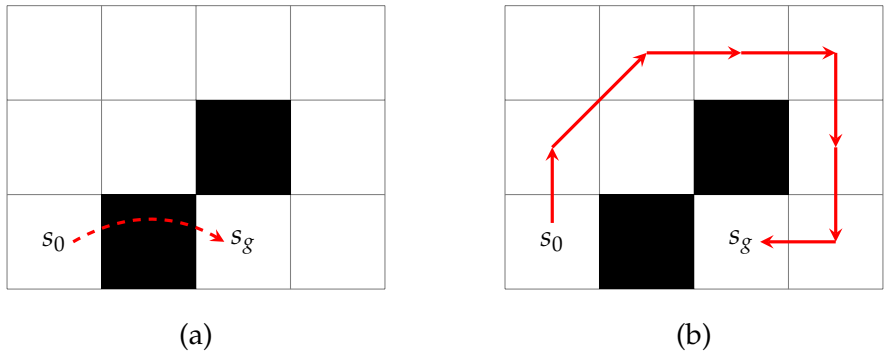


Figure 2.2.: (a) A pathfinding problem and (b) a path that solves the problem.

(including the blocked cells). For example, this could be the set of all possible combinations of joint angles for a robotic arm, all possible positions on a map, etc. A state space may be uncountably infinite, countably infinite, or finite. An unquantized orientation of a 360-degree robotic joint - any orientation in the continuous range of [0-360) is uncountably infinte. An example of a countably infinite state space is a Cartesian grid with no boundary where an agent is constrained to have integer coordinates $(x, y) \in \mathbb{I}$, but there is no upper or lower bound. An example of a finite state space is a grid with boundaries, all possible whole-number locations on the grid are countable in finite time. The example in Figure 2.2 represents a finite state space.

Search Space: A subset of the state space of all *reachable* states or configurations. In Figure 2.2, the search space is all unblocked cells since blocked cells are not reachable via any valid action.

Completeness: The guarantee that an algorithm will terminate [45] when given a finite search space (whether a path to the goal exists or not) and that it will find a path to the goal if one exists. These properties are called *termination complete* and *solution complete* respectively. An algorithm may also be considered *resolution complete*, meaning that a path is discoverable reliant on the resolution of the state space. It may be that at a coarse level of of discretization, no solution can be found, and at a finer level of discretization, a solution can be found.

Optimality: An important property of pathfinding is *optimality*. An algorithm may be able to guarantee a certain quality of the answer it produces with respect to the number of actions or total cost of actions in the path. Typically, our notion of path quality is minimization of $c(\pi)$. In other words, we seek π^* which has minimal cost among Π, the set of all valid paths:

$$\pi^* = \underset{\pi \in \Pi}{\text{ARGMIN}}\ c(\pi)$$

The path shown in Figure 2.2 (b) is the only shortest path.

The optimality property of pathfinding algorithms falls into three major categories:

- **Optimal**: The algorithm is guaranteed to return a lowest-cost path.
- **Bounded Sub-optimal**: The answer represents a path with cost that is no greater than optimal within a *bounding function* [188, 31]. For example, within a factor of w or no greater than optimal plus a constant margin ϵ.
- **Unbounded Sub-optimal**: Also known as *satisficing* [45], the objective is to find a any valid path, hence the answer has no bound on length or cost.

In the case that an algorithm is run on a discrete state space optimality can only be measured up to the degree of discretization. *Resolution optimal* means that no path of lower cost exists per the level of state space discretization. The same algorithm run on a state space with finer discretization or in continuous space may be able to achieve a lower cost path, although typically algorithms take longer to run with finer discretization.

An example of truly optimal pathfinding in continuous space is Polyanya [40] which finds shortest paths in two-dimensional spaces with polygonal obstacles. Related to the concept of optimality in continuous state spaces is *probabilistic optimality*. An example of this is the rapidly-exploring random trees (RRT) [102] algorithm where

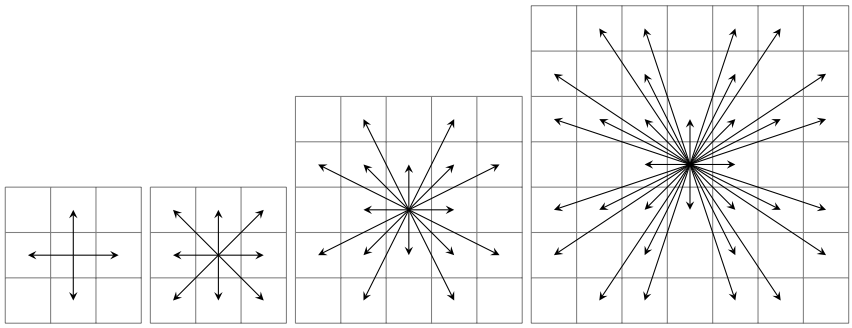


Figure 2.3.: 2^k Neighborhood movement models for k = 2,3,4 and 5.

state transitions are sampled from continuous space. Eventually, given enough samples, RRT will find a truly optimal solution.

A popular example of action-space discretization is 2^k neighborhoods [71, 151] (see Figure 2.3). In these configurations for grid maps, the action space is discretized by the number of cells an agent may move to. In these domains and more generally in domains with regularly spaced vertices, and/or symmetric-shaped obstacles, *symmetries* can occur. With respect to a path, a symmetry is an alternate path which is different, but has equivalent cost. For example, in Figure 2.4 (a) there are many equivalent-cost paths connecting s_0 and s_g. Symmetries are shown in (b) as well, but none exist in (c). Symmetries are the reason for multiple optimal paths existing for a problem instance.

It is also possible to formulate a search problem to simultaneously optimize multiple objectives. For example, to optimize for both distance and time elapsed. Generally, multi-objective search [66] can be formulated with a vector of action costs, mean-

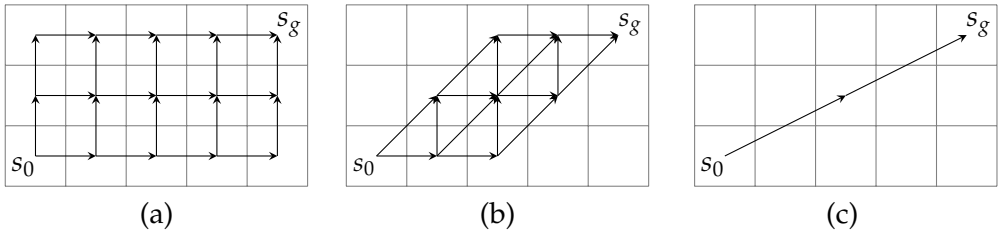


Figure 2.4.: Set of optimal path edges with 4-neighbor (a), 8-neighbor (b), and 16-neighbor (c) grids.

ing that the path cost function returns a vector of costs rather than a single scalar value. There are several variants of multi-objective search: *Lexicographic*, where objectives are strongly ordered and each cost objective is infinitely more important than another in order. *Weighted*, where each objective has a relative importance or weight. *Equally-weighted*, where each objective is as important as another.

Equally-weighted objectives introduce the notion of *pareto-optimality*, a situation where multiple differing optimal paths are possible. A set of paths is pareto-optimal when no other objective value can be increased without lowering another. Computing an entire pareto-optimal set may require an exponential number of node expansions [8]. The relevance of pareto-optimal sets to multi-agent pathfinding will be discussed in Section 2.3. In this work, however, we mainly focus on single-objective algorithms unless otherwise noted. Another property of pathfinding algorithms is *completeness*.

2.2. Systematic Search Algorithms

In the case of finite, or countably infinte state spaces, the search space can be represented as a graph $G = (V, E)$, where each vertex $v \in V$ represents a state and each edge $e \in E; e = (u, v) \in V$ represents an action. In the case of a finite state space, it may be possible to create a representation of the entire search space in memory; this is called an *explicit* search space. In the case of a very large or countably infinite search space, the search graph may not have a full representation in memory; this is an *implicit* search space.

All systematic search algorithms explore the search space by performing *node generations* and *node expansions*. A *node* is an in-memory representation of a state thus a node generation is the act of instantiating a state in the search space. Each state may have a set of neighboring states which it may transition to via actions. The average number of neighbors that can be reached from an arbitrary state in the search space is called the *branching factor*, denoted b. The act of generating the neighbors of a particular

node is called *node expansion* and usually means marking the state as visited in a data structure. A key characterization of a search algorithm is the order in which it performs node expansions.

There are many algorithms which are suitable for searching on graphs: breadth-first search (BFS) [125], depth-first search (DFS) [184], Dijkstra's Algorithm [42], A* [69] and bidirectional search [134]. There are a significant number of enhancements to these algorithms which cannot be discussed here.

2.2.1. The A* Algorithm

The A* algorithm is of central importance for much of the discussion in this thesis. The A* algorithm is a best-first search algorithm [45]. Best-first search algorithms use an OPEN list, which is a priority queue that orders search nodes with the "best" candidates first. The notion of "best" in A* is based on cost, which has two parts: *h-cost* and *g-cost*. To determine h-cost, A* employs a *heuristic*, which is a function that estimates the cost-to-go from the current state to the goal. A* combines *g-cost* which is the cost accumulated from the start node to the current node with the *h-cost* to get the *f-cost*, which is the estimated total cost of a path from the start node through the current node to the goal. Hence, the OPEN list is ordered by f-cost.

Pseudocode for the A* algorithm is shown in Algorithm 2.1. A* starts by initializing the OPEN list with the start state and the h-cost (lines 2, 3). Then A* systematically pops states from the OPEN list (lines 4, 5). It will terminate if it has found a goal (lines 6, 7), otherwise, it generates successors (line 8) and if the state does not yet exist in OPEN (i.e., its g-cost is ∞) (line 10 its g-cost is set (line 11) and it is added to OPEN (line 13) or if it is already in OPEN but its current g-cost is better, (line 10) its g-cost is updated (line 11). In the case no path to the goal exists, OPEN will eventually be empty (line 4) and a null solution is returned (line 14).

2.2.2. Heuristics

Whether A* will return an optimal result depends on the heuristic. If the heuristic never over-estimates the cost-to-go, it is called *admissible*. With an admissible heuristic, A* is guaranteed to return an optimal result [69]. Any heuristic that over-estimates the cost-to-go is *inadmissible*. An optimal result is not guaranteed with an inadmissible heuristic, but A* is complete whether it uses an admissible heuristic or not.

A* is also guaranteed have *optimal efficincy*, meaning it will do a minimal amount of work in terms of node expansions [41]. But this is only guaranteed if A* uses a *consistent* heuristic. A heuristic is consistent if it is admissible and $\{\forall(s,s'),(s.v,s'.v)\in E\}$, $h(s)\leq h(s')+c(s,s')$. That is, on a direct path to the goal, the h-cost will never increase.

2.2.3. Time and Space Complexity

In computer science, two properties are generally used to describe algorithms (including search algorithms): time complexity or *computational complexity* - the number of operations required to execute an algorithm and *space complexity* - the amount of memory that an algorithm uses.

Algorithm 2.1. A* Algorithm

```
Input: A search problem: s_0, s_g, h
OPEN ← {s_0, h(s_0)}
GCOST[s_start] ← 0
while OPEN ≠ ∅ do
    n ← OPEN.pop()                          ▷ Retrieve node with lowest f-cost
    if n = s_g then
        return RECONSTRUCTPATH(n)           ▷ Return solution
    for n' ∈SUCCESSORS(n) do                ▷ Generate successors
        g ← GCOST[n] + c(n, n')
        if g < (GCOST[n'] or ∞) then        ▷ Path to n' is better or not seen yet
            GCOST[n'] ← g
            if n' ∉ OPEN then
                OPEN ← OPEN ∪ n', g + h(n')
return ∅
```

For search, both computational complexity and space complexity are commonly defined in terms of b, the branching factor, d, the depth of a path from start to goal, and sometimes n, the total number of states in the search space - n is only concretely defined for finite search spaces.

The computational complexity, space complexity, optimality and completeness properties of algorithms may make one algorithm more suitable for a particular application than another. For example, both BFS and DFS have a computational complexity of $O(b^d)$, where d is the depth, or number of actions from the start to the goal. BFS has the property that it will terminate even on countably infinite search spaces provided that a goal is reachable, however its space complexity is $O(b^d)$, meaning it will take up to b^d memory. DFS on the other hand only uses $O(d)$ memory, but may run forever on countably infinite search spaces, even when a reachable goal is present.

There are many different variants of systematic search algorithms. One example is the iterative deepening A* (IDA*) algorithm [96] which is a hybrid of DFS and A*. It combines the strengths of DFS and A*. Another example is partial expansion A* (PEA*) [214] which defers some node generation operations until just before they are needed.

2.2.4. Free-space Search Algorithms

When dealing with uncountably infinite search spaces, it is impossible to explore every possible state in the state space without discretization. One approach to discretization is to build a graph with vertices at regular intervals in the search space and use systematic search algorithms. Another approach is to sample the continuous space dynamically during the search process.

Space Discretization

Several systematic graph building approaches have been proposed. One approach called visibility graphs [206] attempts to construct edges between all pairs of

obstacle vertices which have unobstructed line-of-sight. There are also several approaches which augment existing graphs to simplify them by converting some bidirectional edges to directed edges, or create hierarchical graphs that represent some subsets of the graph as a single node [73, 175].

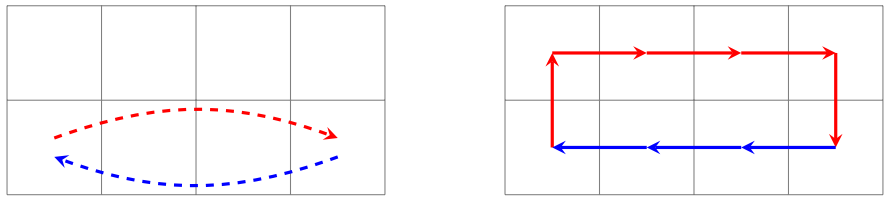
Workspace decomposition focuses on decomposing the free configuration space into a graph by discretizing or abstracting the free space into a smaller representative graph. Some methods for decomposition are: vertical cell decomposition [29], navigation meshes [190], generalized Voronoi diagrams [137] and reduced visibility maps [135, 101]. The probabilistic road map (PRM) approach [88] builds graphs using randomization. There are many approaches including, sampling the medial axis of open space [208] and visibility based sampling [136]. Typical space discretization approaches come with guarantees of connectedness and resolution completeness.

Dynamic Sampling

The rapidly-exploring random trees (RRT) algorithm [102] starts at the initial configuration and randomly creates branches outward by making a radius-constrained sample in the continuous space around the latest sample and then connecting the newest sample to the closest sample with an edge. If the edge is obstructed by an obstacle, the sample is discarded. The process is repeated until a sample is discovered within a certain radius of the goal. Several variants of this approach have been proposed: bidirectional sampling [99], optimal RRT (RRT*) [86], A*-RRT [28], Informed RRT* [56] and Theta*-RRT [138].

Other Algorithms

The single agent problem can also be formulated as a constraint satisfaction problem (CSP) [106, 115], boolean satisfiability problem (SAT) [4] or a integer linear programming problem (ILP) [144].



(a) (b)

Figure 2.5.: A MAPF problem instance (a) and solution (b).

2.3. The Multi-Agent Pathfinding Problem

The multi-agent pathfinding (MAPF) problem was originally defined in 1984 [97] and is a natural extension of the single-agent problem. We define two variants of the problem, *Classic* MAPF and *General* MAPF.

2.3.1. Classic MAPF

A classic MAPF problem instance is defined by a tuple (G, k, V_s, V_g). $G{=}(V, E)$ is an unweighted graph, and each $v{\in}V$ is associated with unique coordinates in a metric space. There is a set of k agents labeled 1 through k where agents occupy a point in space. $V_s{\subseteq}V{=}\{start_1, ..., start_k\}$ and $V_g{\subseteq}V{=}\{goal_1, ..., goal_k\}$ are sets of unique start and goal vertices for each agent where $start_i{\neq}start_j, goal_i{\neq}goal_j$ for all $i{\neq}j$.

The objective is to find non-conflicting (the definition of a *conflict* follows shortly) paths from start to goal for each agent. We use the following definitions:

- $s = (v, t)$ is a *single-agent state* composed of a vertex $v{\in}V$ and a time $t{\in}\mathbb{Z}^+$.
- $S = \{s_1, .., s_k\}$ is a *multi-agent state* or *joint state*.
- $S.V = \{s_1.v, .., s_k.v\}$ is the set of *vertices* of a multi-agent state.
- $a = (s, s')$ is a *single-agent action*, composed of two states such that $(s.v, s'.v){\in}E$. As described in Section 2.1.1 this definition of an action differs from the traditional reinforcement learning definition; it refers to traversing a specific edge at

a specific time. MAPF allows self-directed edges for single-agent *wait* actions, where $a.s'.v = a.s.v$ and $a.s'.t = a.s.t + 1$ meaning that an agent stays at the same vertex for one time step.

- $A = \{a_1, .., a_k\} = \{(s_1, s'_1), .., (s_k, s'_k)\} = (S, S')$ is a *multi-agent action* or *joint action* such that all actions overlap in time: $\bigcap_{a \in A} [a.s.t, a.s'.t] \neq \emptyset$.

- $\pi = [s^0, .., s^d]$ is a single-agent *path* composed of $d+1$ sequential single-agent states. Analogously, a path can be composed of d sequential actions: $\pi = [a^0, .., a^{d-1}] = [(s^0, s^1), (s^1, s^2), .., (s^{d-1}, s^d)]$.

- $\pi(s)$ is a single-agent path from the start state ending at s. In the context of search, $\pi(s)$ is the current single-agent path from the start state to s in the OPEN or CLOSED list.

- $\Pi = [\pi^1, .., \pi^k]$ is a *solution* which is a set of single-agent paths which terminate at the goal ($\pi(goal_i)$), one for each agent. Π is equivalently represented as $[S^0, .., S^d]$ or $[A^0, .., A^{d-1}]$. These alternative representations use joint states and joint actions.

- $\Pi(S)$ is a solution from the start joint-state ending at S. In a search context, $\Pi(S)$ is the current solution from the start joint-state to S in the OPEN or CLOSED list.

- $c(a)$ is the *cost* of a single-agent action. $c(a) = c(s, s') = w(e)$ where $e = (s.v, s'.v) \in E$.

- $d(a)$ is the *duration* of a single-agent action. For the purposes of this thesis cost and duration are analogous: $d(a) = c(a)$, but they can easily be treated as separate quantities.

- $c^*(s, s')$ denotes the cost of a shortest path between s and s' (e.g., when $s.v$ and $s'.v$ are not adjacent).

- $c(\pi) = \sum_{a \in \pi} c(a)$ is the cost of a path.

- $c(A) = \sum_{a \in A} c(a)$ is the cost of a multi-agent action.
- $g(s) = c(\pi(s))$ is the *g-cost* or cumulative cost of $\pi(s)$ for s in the OPEN or CLOSED list.
- $g(a) = g(a.s')$ is the g-cost or cumulative cost of $\pi(a.s')$.
- $g(S) = \sum_{s \in S} g(s)$ is the g-cost of a multi-agent state.
- $g(A) = g(A.S')$ is the g-cost of a multi-agent action.
- $h(s)$ is the *h-cost* or heuristic estimate from s to the goal.
- $h(a) = h(a.s')$ is the h-cost or heuristic estimate from $a.s'$ to the goal.
- $h(S)$ is the multi-agent h-cost or heuristic estimate from S to the goal.
- $h(A) = h(A.S')$ is the multi-agent h-cost or heuristic estimate from $A.S'$ to the goal.
- $f(s) = g(s) + h(s)$ is the *f-cost* of a shortest path from start to goal through s.
- $f(a) = g(a) + h(a)$ is the f-cost of a shortest path from start to goal that includes the action a.
- $f(S) = g(S) + h(S)$ is the estimate of summed-shortest paths from start to goal through S.
- $f(A) = g(A) + h(A)$ is the estimate of summed-shortest paths from start to goal that include the joint action A.
- $t_{min}(S) = \underset{s \in S}{\text{MIN}}\ s.t$ is the earliest time of all states in a joint state.
- $\boldsymbol{S}$ (bold 'S') is a set of joint states.
- s (sans bold 's') is a set of single-agent states for one agent.
- $\mathsf{S} = \{\mathsf{s}_1, .., \mathsf{s}_k\}$ (sans bold 'S') is a set of sets of single-agent states.

A solution is *feasible* if none of the k agents come into *conflict* at any time while traversing their respective paths. A conflict happens when two agents attempt to occupy the same vertex at the same time, or traverse an edge in opposite directions at the same time. A conflict is denoted as a tuple:

$$\langle a_i, a_j \rangle$$

, for a pair of actions in π_i and π_j respectively for agents i and j which conflict.

2.3.2. General MAPF

In Classic MAPF, because all actions have uniform duration, all actions happen in *lock-step*, meaning that all agents' actions start and end at synchronous times. General MAPF (also called MAPF_R [198]), the central focus of this dissertation, is an extension of Classic MAPF for weighted graphs with positive, real-valued weights, $\forall e \in E, w(e) \in \mathbb{R}^+$. Thus the cost and/or duration of single-agent actions are not uniform. This introduces the notion of continuous-time motion which is defined in Chapter 3. When we need to distinguish between *Classic MAPF* and *General MAPF*, we will refer to those terms specifically.

2.3.3. Objective and Hardness of MAPF

When all agents have equal priority, solutions to MAPF, like multi-objective optimization, have a pareto-optimal structure. This is illustrated in Figure 2.6 where part (a) shows the problem instance, part (b) shows a solution where the blue agent yields to the red agent, and (c) where red yields to blue. Arrows indicate actions. Looped arrows indicate wait actions. This illustration shows that there may be many feasible solutions to a problem instance where $c(\pi_i)$ cannot be reduced without increasing $c(\pi_j)$ and vice-versa. This pareto cost structure applies generally over any set of agents.

Computing and reasoning over the pareto cost structure for MAPF instances is expensive, since the number of pareto-optimal solutions is exponential in the number

of agents in the worst case [60]. However, in this thesis, we focus only on objectives that optimize a *combined* cost $c(\Pi)$. The most popular combined objective functions for MAPF are:

- **makespan**: minimizing the maximum cost path in a solution:

$$\text{MAKESPAN}(\Pi) = \textit{minimize}\left[\underset{\pi\in\Pi}{\text{MAX}}\left(c(\pi)\right)\right]$$

- **flowtime** or *sum-of-costs*: minimizing the total cost of all paths in a solution.

$$\text{FLOWTIME}(\Pi) = \textit{minimize}\left[\sum_{\pi\in\Pi} c(\pi)\right]$$

Most MAPF algorithms can be formulated for makespan or flowtime.

For satisficing search, Classic MAPF is solvable in polynomial time on undirected graphs [97, 90, 155, 207] and strongly biconnected graphs [22]. But Classic MAPF is NP-hard on general directed graphs [131]. Additionally, all known polynomial time algorithms for MAPF assume agents are *holonomic*, that is, they can turn in-place.

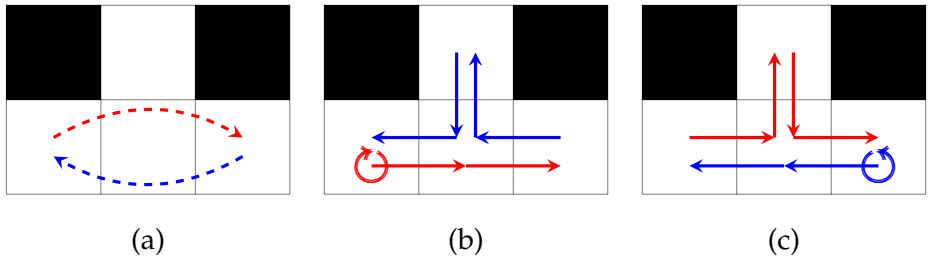


Figure 2.6.: An example of the pareto-optimal cost structure of MAPF: (a) A MAPF problem instance, (b) a solution favoring the red agent and (c) a solution favoring the blue agent.

Finding optimal and bounded sub-optimal solutions to MAPF is NP-hard for both flowtime [217] and makespan [178] – furthermore, makespan and flowtime cannot be optimized simultaneously [217]. As the Classic version is a special case of the General version, the optimization of General MAPF instances is also NP-hard.

2.3.4. Variants of MAPF

Our definition so far covers the Classic MAPF problem, however there are many variants of the MAPF problem in terms of its objectives and rules of movement [173].

- **Multi-Objective MAPF** – Agents simultaneously optimize multiple individual objectives [149].
- **Anonymous MAPF** – Goals are not assigned to any specific agent [92, 215].
- **K-Color MAPF** – Teams of agents and goals are assigned a color. Any agent may satisfy a goal of the same color. In other words, goals are interchangeable within agents belonging to the same team [118, 168].
- **Multi-Agent Meeting** – Agents goal is to meet at a dynamically-determined location. This variant may be with or without timing constraints [20, 10].
- **Coverage optimization** – Agents must maintain distributed coverage of an area [39].
- **Convoy planning and Trains** – Agents may span a chain of multiple vertices and edges in the graph at one time [186, 9].
- **Network/Capacity/Flow planning** – Agents may move along edges simultaneously, but edges have capacity limits. This is the classic maximum flow problem [68, 17].
- **MAPF with Deadlines** – Agents must arrive at their goals inside certain deadlines [121].

- **Online MAPF** – A variant of MAPF where agents may dynamically enter or leave the joint planning space [182, 127, 117].
- **Lifelong MAPF** – Continuous planning where agents receive new goals once a goal is reached [120, 109].
- **Robust MAPF** – Agents must account for possible delays in the execution of other agents' paths before execution of the plans [12, 13, 11, 158].

2.3.5. Agent Movement and Type Variants

Variants in the rules for MAPF and nuances for agent movement can make a significant difference in the performance of an algorithm [173].

- **Sequential Movement**: This variant is known as the pebble motion on graphs (PMG) [97] problem and is essentially the same as the 15-puzzle game [165] where only one agent (tile) is allowed to move into a blank space at a time. Solutions can be found in polynomial time, however, finding an optimal (minimum number of moves) solution is NP-hard [64, 147]. Most modern variants of MAPF allow **parallel movement**, where agents perform actions simultaneously.
- **Lock-step Movement**: In this variant, all actions have unit duration and actions are begun simultaneously, hence all agents movements start and end synchronously. Lock-step assumptions greatly simplify conflict detection. This is always the case in Classic MAPF. With **Non Lock-step Movement** actions [198, 6, 179], where agent actions are not all at the same time nor of identical duration introduce the need for continuous-time conflict detection in all algorithms and partial time overlap (PTO) expansion routines for coupled algorithms 2.4.2.
- **Goal blocking**: In this variant, agents remain at their respective goals after arriving at their destinations. This variant may require agents to move out of the way. **Disappearing** agents [196] do not remain in the configuration space once

they reach their goal. Some problems naturally have disappearing agents, for example in airport surface operations aircraft that have parked on an apron can no longer conflict with agents on taxiways.

- **Following**: When the workspace is decomposed into vertices or grids, some formulations of MAPF allow one agent to occupy a vertex in a time step directly after another agent has left it.
- **Rotations**: Some formulations allow agents to swap adjacent vertices via an edge without incurring a collision [172]. This variant makes solution sizes smaller in general, but does not change the hardness of the problem [217].
- **Agent body size**: Can have a significant impact, especially when agent bodies are larger than the workspace discretization – this becomes similar to the convoy planning problem in that an agent may occupy multiple edges and vertices simultaneously [110].

2.4. Prior Work in MAPF

A wide variety of solutions to MAPF have been published since the problem was formally defined in 1984 [97].

2.4.1. Centralized and Distributed Algorithms

MAPF algorithms broadly fall into two categories: *Centralized* and *Decentralized* or *Distributed*. Centralized algorithms assume that perfect information about the location and movement of all agents is known and being managed from a central algorithm. Distributed algorithms assume that agents do not have perfect information about the state of other agents. Therefore, they cannot plan collision avoidance without some form of message passing or organically observed information about the state of other agents.

All distributed algorithms are decoupled, while centralized algorithms may use coupled, decoupled or dynamically coupled approaches. The definitions for coupled, decoupled and dynamically coupled are discussed in the following sections.

2.4.2. Coupled Algorithms

Coupled algorithms conceptually combine the states of k agents into a joint state space. Coupled algorithms fall into several categories: Search Based, Rule Based, Reduction Based and Auction Based.

Search Based Algorithms

Well-known search-based, single-agent algorithms such as A*[69], IDA*[96], EPEA*[63], and others may be used for the multi-agent problem. A difficulty with coupled, search-based algorithms is that the number of combinations of ways in which the set of agents can move is exponential. If each agent has a branching factor of b_{base}, the multi-agent branching factor is ${b_{base}}^k$. Meaning that if the number of agents is fixed, the branching factor is polynomial in the number of agents – exponential if we think of k as not being fixed.

Multi-Agent Expansion: In addition to dealing with exponential branching factors, additional difficulty is that multi-agent expansions must account for action combinations that result in conflict between agents. Such combinations are typically eliminated during the expansion phase. Also, in the case of General MAPF, arbitrary offsets in time must be accounted for. Algorithm 2.2 illustrates a multi-agent, conflict-aware expansion routine.

The Conflict-Aware Multi-Agent Expansion routine takes a joint state S for k agents as input and determines which of the k single-agent states can be expanded and which must be repeated in order to maintain *time overlap*. Time overlap means that for all $s' \in S'$ the respective actions $A = \{a_1, .., a_k\} = \{(s_1, s'_1), .., (s_k, s'_k)\} = (S, S')$ where S is the predecessor of S', the actions overlap in time: $\bigcap_{a \in A} [a.s.t, a.s'.t] \neq \emptyset$. This is is done

on lines 3-9. Then, given the k sets of single-agent states whose generating actions have time overlap, the subroutine GENJOINTSTATES is called (line 10) which recursively computes the Cartesian product of the sets, removing conflicting ones in the process (lines 22-24). Resulting in a set of non-conflicting joint-state successors with time-overlap.

One way to mitigate the daunting size of the branching factor is to use a technique called operator decomposition (OD) [171]. With OD, during the expansion process, we fully expand one agent to generate b_{base} successors (see Algorithm 2.2 line 6) and pick only one *intermediate* state to generate for the other $k-1$ agents. In this manner, the branching factor at each step of the search is reduced to only b_{base}. However, OD increases the search depth to the goal by a factor of k. Hence, when using blind search algorithms such as Dijkstra's algorithm, there is no savings in the number of operations required to find the goal, however, when using heuristically-guided algorithms such as A*, a significant amount of savings can be realized.

Multi-Agent Heuristics: A straightforward approach to computing a multi-agent heuristic is to sum the single-agent h-costs, $\sum_{s \in S} h(s)$. This is an admissible multi-agent heuristic if all single-agent heuristics are admissible. However, more accurate heuristics can be computed by finding solution costs for subsets of agents [98] or analyzing conflicts [51, 108, 112].

Enhanced partial expansion A* (EPEA*) [63] uses heuristic information to limit the nodes generated during the expansion process (its predecessor PEA*[214] generated all nodes, but discarded heuristically sub-optimal ones.) In this manner EPEA* saves a significant amount of time and memory to perform a multi-agent search. However, the function for selecting which nodes to generate can be very difficult to formulate - especially for spaces with large branching factors or non-trivial cost functions.

M* is a framework for A* called subdimensional expansion [194] which initially plans paths using an optimal policy and then selectively *backpropagates* extra edges when agents come into conflict, effectively allowing them to circumvent each other. M* has been shown to be especially effective in mazes [52, 85].

Algorithm 2.2. Conflict-Aware Multi-Agent Expansion

JOINTEXPANSION

```
1:  Input: S — A joint-state
2:  Output: S' — A set of joint state successors
3:  S ← {∅_1, ∅_2, ..., ∅_k}                    ▷ Sets of single-agent successor states
4:  minTime ← t_min(S)
5:  for s_i ∈ S do
6:      if s_i.time = minTime then   ▷ Expand only the earliest. (One state only for OD)
7:          S_i ← SUCC(s_i)
8:      else
9:          S_i ← {s_i}
10: S' ← GENJOINTSTATES(S, 1, ∅)                ▷ Generate Cartesian product
11: return S'
```

GENJOINTSTATES

```
12: Input: S — A set of k sets of single-states
13:        agent — Current agent under consideration
14:        S — Partially accumulated joint-state
15: Output: S — A set of joint-states; initially empty, modified in-place
16: if agent = k then
17:     S ← S ∪ S
18: for s ∈ S_agent do
19:     S' ← S                                  ▷ Make a copy
20:     conflicting ← false
21:     for q ∈ S' do
22:         if CONFLICTTEST(s,q) then           ▷ Check for conflict
23:             conflicting ← true
24:             break
25:     if conflicting then        ▷ Skip this state – it has a conflict with a state in S'
26:         Back to line 18
27:     S ← S' ∪ GENJOINTSTATES(S, agent + 1, S')
```

The **increasing cost tree search (ICTS)** algorithm [163] is a two-level search algorithm. ICTS is based on the observation that a solution to a MAPF problem contains a set of paths with unique cost. ICTS searches for a cost vector that represents a lowest-cost feasible solution by combinatorically generating cost vectors and testing for feasibility. ICTS has been shown to be especially effective in spaces with large rooms or wide corridors [52, 85]. ICTS is covered in detail in Chapter 4.

Rule Based Algorithms

There are several polynomial-time algorithms that achieve constant-bounded quality solutions for holonomic agents: Pebble Motion on Graphs [97], Tree-based Agent Swapping Strategy, [90], Parallel Push-and-Swap [155] and Push-and-Rotate [207].

Additionally time interleaving strategies [141], where the only resolution option that agents are given for resolving conflicts is to wait are polynomial in time complexity.

Reduction Based Algorithms

Similar to single agent reduction based algorithms, there are several reduction based algorithms for MAPF: Constraint Satisfaction (CSP) [153, 18, 17, 202], Satisfiability (SAT) [177, 180], ILP [216] and Answer Set Programming (ASP) [46].

Auction Based Algorithms

Combinatorial auction-based algorithms implement a mechanism where agents bid on sets of preferred paths for themselves (e.g., shortest paths). The auctioneer then determines whether the sets of preferred paths are disjoint (non-conflicting) and will award some or all agents their bids for non-conflicting paths. The auction continues, raising the price until all agents are able to win non-conflicting bids [5].

2.4.3. Decoupled Algorithms

This family of algorithms does not plan in the joint state space. All decentralized algorithms are also decoupled.

Reactive Planning

Reactive planners do not coordinate their plans with other agents. Each agent is autonomous and reacts to the world in the immediate vicinity, or a limited version of

the world which it can sense. Reactive planning is unbounded-sub-optimal in all cases and from a multi-agent global perspective completeness is not always guaranteed.

Velocity Obstacles [53] were first defined as a method of collision prediction, and have been adapted for use with reactive planners [54, 19, 167]. Velocity obstacles define a temporospatial region in which two bodies in motion will collide assuming their trajectories are straight, constant velocity motion vectors. Velocity Obstacles are discussed in more detail in Chapter 3. Reactive planners use the velocity obstacle to select alternate motion vectors which do not put the agent on a collision course.

Potential Fields [89] represent attractive and repulsive electromagnetic forces. Potential field methods are used for robotic planning to attract agents toward their individual goals and repel them away from obstacles and other agents. Once the fields are defined, a path from start to goal can be found via gradient descent. For the multi-agent case, each agent's repulsive potential fields are updated dynamically as agents move through the configuration space until agents reach their respective goals. It is possible for agents to get trapped in local minima when using potential fields. This problem can be mitigated however by the use of harmonic functions [2] or local search methods. Potential fields have been used ubiquitously in robotic planning not only for goal-oriented planning, but for formations [154], swarming [16], flocking [183], and joint manipulation behaviors [169].

Prioritized Planning

Prioritized planning [47] is a scheme of planning the paths for agents sequentially where any agent earlier in the planning process gets its actions reserved, precluding agents later on in the planning sequence from coming into conflict with earlier agents. This approach is sub-optimal but fast.

Some incomplete prioritized planning algorithms are Windowed-Hierarchical Cooperative A* (WHCA*) [164], Conflict-oriented WHCA* (C-WHCA*) [21], Distributed Multi-Agent Path Planning (DMAPP) [34], Safe Interval Path Planning (SIPP) [143],

Any-angle SIPP (AA-SIPP) [212]. The first known approach to MAPF_R [174] is also based on prioritized planning.

An extension of DMAPP that has completeness guarantees (DiMPP) [33], systematically tests independent sub-permuatations of agent priority orderings in order to achieve completeness.

2.4.4. Dynamically Coupled Algorithms

These algorithms solve MAPF problems by breaking them up into smaller sub-problems. Sub-problems are joined together dynamically, when the algorithm determines that it needs to do so.

Independence Detection

It is often the case that a MAPF instance contains multiple independent sub-problems. An independent sub-problem in MAPF consists of a subset of agents and their start and goal states. We partition sets of agents into independent sub-problems based on whether the agents' paths conflict with each other. The Independence Detection (ID) algorithm [171] (Algorithm 2.3) starts by assigning each agent to its own sub-problem (line 3) and planning a path for each agent independently (line 4). The paths of the agents are then checked for conflicts (line 5). Any agents which have con-

Algorithm 2.3. Independence Detection

```
1: Input: V_s, V_g   ▷ Start and goal states for each agent P ← ∅   ▷ Set of sub-problems
   Π ← ∅                                                            ▷ Solution
2: for i ∈ [1, .., k] do
3:     P ← P ∪ {i}                          ▷ Create a problem group for each agent
4:     Π_i ← FINDSHORTESTPATH(v_s^i ∈ V_s, v_g^i ∈ V_g)   ▷ Find shortest path for each agent
5: while Π has conflicts do
6:     Merge conflicting sub-problems in P
7:     for p ∈ P do
8:         Run MAPF algorithm on agents in p
9:         Update Π with new paths for agents in p
10: return Π
```

flicts are combined into a new sub-problem (line 6) and re-planned (line 8) repeatedly until no conflicts exist. This simple approach can be very effective, but in the worst case, the entire problem must still be solved.

Conflict-Based Search (CBS) [160] takes the notion of independence detection further by recognizing that merging conflicting agents together to plan their entire paths is a larger granularity of coupling than needed. CBS only couples agents together at the point of their conflicting actions and solves their paths separately, using *motion constraints* to help avoid conflicts. The details of how this is done is addressed in Chapter 5. CBS is a very popular algorithm with many enhancements [24, 196, 25, 15, 35, 118, 51, 110, 113, 111, 218, 112, 26].

Several enhancements to CBS for General MAPF have also been proposed. In general, these enhancements focus on continuous-time collision detection and constraint representation [6, 37, 199, 192]

Conflict-Based Increasing Cost Search (CBICS) [200] combines the strengths of CBS and ICTS in the form of *motion constraints* and *cost constraints*. CBICS is discussed in Chapter 6.

Expanding A* (X*) [191] is an anytime framework which performs local *repairs* for conflicting sets of agents by defining a *window*, which includes a bounded state space region. The repair operation attempts to find non-conflicting paths for the subset of agents from where they enter and exit the window region. If unsuccessful, or with remaining time, the window region is expanded systematically. As time allows, the window regions are expanded to include the entire search space, at which full optimality is guaranteed. X* works well for problems with sparse, geographically dispersed conflicts because it reuses much of the planned paths for each agent.

Techniques for Dynamically Centralized Algorithms

The **Conflict Avoidance Table** (CAT)[172] is used with dynamically coupled algorithms. The idea is to store the set of paths for each agent in a data structure

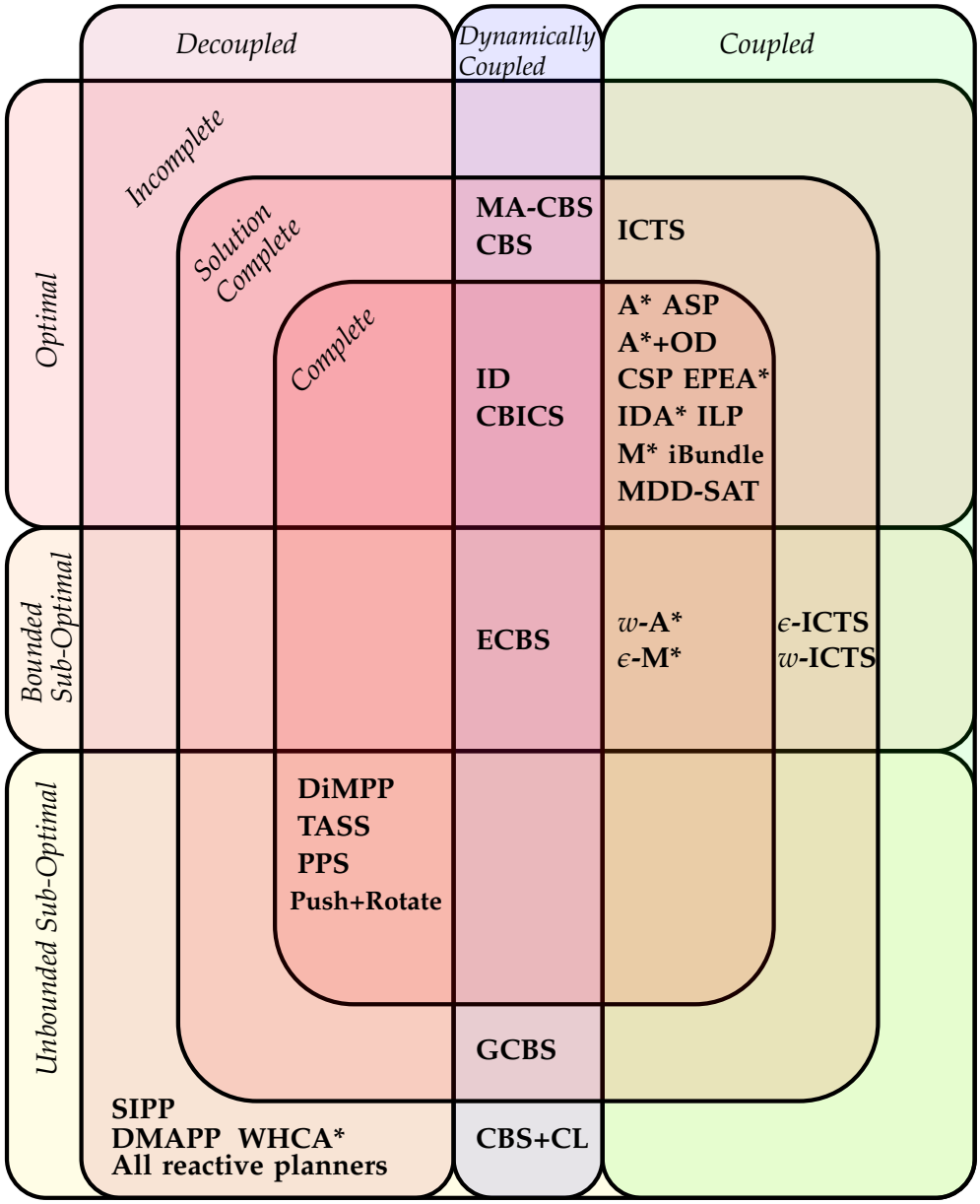


Figure 2.7.: Taxonomy of MAPF Algorithms

where the agents' actions can be looked up quickly. When a subset of agents is re-planned, actions which conflict with the paths of the other agents which are not being re-planned are updated with the number of conflicts. Then paths with fewer conflicts can be prioritized over paths which conflict – either by tie-breaking in the OPEN list (for optimal algorithms) or as the primary priority (for sub-optimal algorithms). The CAT has proven to be effective at helping MAPF algorithms avoid unnecessary work.

The **Conflict Count Table** (CCT) is an undocumented technique for dynamically coupled algorithms. It is used with ID, CBS and CBICS. The CCT stores a list of the number of conflicts between pairs of agents which have conflicts in their paths. This bookkeeping technique helps avoid performing unnecessary conflict checks. The CCT is initialized by checking all pairs of paths (that's $O(k^2)$) path checks. Then whenever any agent's path is re-planned, the re-planned path is checked versus all other agents' paths (only $O(k)$ path checks) and the CCT is updated. If the CCT is empty, the current solution is conflict free. The CCT helps reduce the overhead of conflict checking by remembering which pairs do not need to be re-checked.

2.4.5. Other Techniques Related to the MAPF Problem

There are several techniques that also apply to MAPF and may be used with various algorithms. They generally fall into two categories: workspace decomposition and heuristic guidance. All of these methods are sub-optimal but have been shown to be effective.

Workspace Decomposition Techniques

The physical characteristics of graphs can be augmented to promote collision avoidance [203, 153, 201]. Methods such as probabilistic road maps (PRM) [195] sample a continuous state space to produce a discrete state space of smaller size. Another approach is to dynamically change agent branching factors to avoid collisions or speed up the search [196, 62].

Heuristic Guidance Techniques

Some techniques encourage collision avoidance via heuristics, e.g., to promote circular movement in open spaces [80] or to promote the use of predetermined, or dynamically learned directional highways [35]. Reinforcement Learning has been used to train heuristic models which help agents avoid collisions [156, 77].

Algorithm Selection Techniques

There is no MAPF algorithm that dominates all others in every type of problem instance [52, 85]. An algorithim selector [148, 84] can be used to select the best algorithm based on physical characteristics of maps or other factors in a specific problem instance.

2.5. Summary

We have provided a formal problem definition for MAPF, building on definitions from single-agent pathfinding. We have provided common formulations of cost and definitions of optimality and completeness. We have provided definitions, complexity analysis and background for heuristic search. We have provided extensive background information on MAPF, including a list of MAPF problem variants and variations on cost and movement for Classic MAPF.

We reviewed prior work on the MAPF problem and provided a *taxonomy* of the various algorithms. We have provided pseudocode for important sub-routines of MAPF and reviewed other techniques that may be applied to various types of algorithms. All of this information now sets the stage for the next chapter, where we introduce the primary focus of this thesis: General MAPF.

3. General MAPF

3.1. Definition of General MAPF

The definition of General MAPF is necessitated by the fact that the simplifying assumptions of unit-duration actions and unit costs in the Classic MAPF definition is insufficient for many real-world and real-world-inspired domains such as the ones mentioned in the introduction. The definition that follows is more general in that it allows for non-unit costs, non-uniform action durations, shaped agents, variable speeds, and/or non-holonomic or kinodynamic movement constraints. Numerical methods for planning in General MAPF domains were being considered in robotics as early as 1986 [89]. The earliest known search-based algorithm for General MAPF was run on 8-neighbor grids [174].

General MAPF (also called MAPF_R [198]) is a natural extension of Classic MAPF where G is a positive-weighted graph: $\forall e \in E\ w(e) \in \mathbb{R}^+$, and each $v \in V$ is associated with unique coordinates in a metric space. Also, agents have a physical shape such as circles, spheres, polygons or polygonal meshes which are situated in metric space relative to a *reference point* [110].

The form of a solution in General MAPF is similar to a solution in Classic MAPF; a solution is a set of k paths, one for each agent. However, a solution in General MAPF is conceptually different. In General MAPF, actions represent continuous motion, hence each path represents *segmented motion*. Segmented motion is defined as a sequence of actions (or motions) that have a discrete duration. A state s in its simplest form is $s{=}(v,t)$ where $v{\in}V$ and $t{\in}\mathbb{R}^+$. However, a state may be represented differently and contain other information such as Cartesian coordinates $s = \langle x, y, t\rangle$,

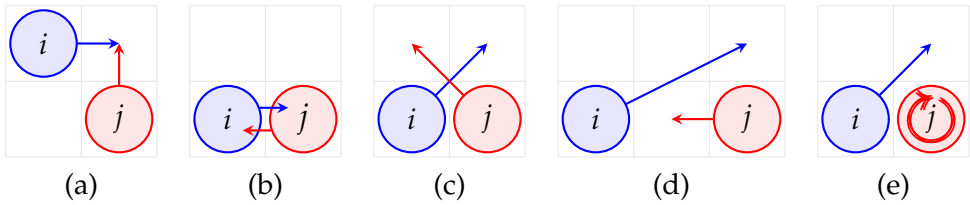


Figure 3.1.: Conflict types in General MAPF: (a) vertex conflict, (b) edge conflict, (c) intersecting-edge conflict, (d) non-intersecting edge conflict and (e) edge-vertex conflict

velocity $s = \langle x, y, \dot{x}, \dot{y}, t \rangle$, acceleration $s = \langle x, y, \dot{x}, \dot{y}, \ddot{x}, \ddot{y}, t \rangle$ or other state information. The definition of a state, including the number of dimensions will depend upon the application. For simplicity in this thesis, we use $s{=}(v, t)$ unless otherwise stated. An action (or motion segment) $a = (s, s')$ is continuous between s and s', thus the transition between them must be kinematically feasible.

An action a_i begins with the agent's reference point at $a_i.s.v$ at time $a_i.s.t$ and its reference point moves in continuous metric space, ending at $a_i.s'.v$ at time $a_i.s'.t$. In this thesis, we assume all agents move with the same constant velocity. But motion could be described differently, such as parameterized motion (e.g., with velocity and/or acceleration [197]) or discretized motion (e.g., using a reservation table of grid squares and time interval of the reservations [37]).

A conflict is the condition when the area or volume of two or more agents overlap at the same instant in time. Because agents have a shape, conflicts not only occur when two agents arrive at the same vertex at the same time (Figure 3.1(a)) or traverse an edge in opposite directions (Figure 3.1(b)). But, conflicts can also occur when agents traverse separate edges (Figure 3.1(c)), even when the edges don't intersect (Figure 3.1(d)). Furthermore, an agent waiting in place can come into conflict with another agent traversing an edge (Figure 3.1(e)). Agents are able to traverse the same edge simultaneously, provided they are both moving in the same direction and have sufficient spatial separation. In the case of arbitrary-length action durations, agents may start and end their actions at asynchronous times, which requires more sophisticated conflict detection mechanisms than Classic MAPF.

The remainder of this thesis focuses on General MAPF, but we simply refer to it as MAPF for notational simplicity. When we need to distinguish between Classic MAPF and General MAPF, we will refer to those terms specifically.

3.2. Conflict Detection In General MAPF

Conflict detection is important for many problems with multiple moving agents. In the case of navigation and routing problems for multiple agents, feasible solutions cannot be found or verified without proper conflict detection. In a general sense, a conflict is a simultaneous attempt to access a joint resource. Depending on the target domain a conflict may have different meanings. Consider, for example, when states have non-spatial attributes such as time scheduling problems or resource allocation problems (e.g., allocating time slots for classrooms [130] or coordinating memory and cpu allocation for processing jobs [83]) or when abstract states are used such as in dimensionally-reduced spaces [146]. Typically, when considering only temporospatial aspects, conflict detection is referred to as *collision detection*.

Collision detection has been extensively studied in the fields of computational geometry, robotics, and computer graphics [81, 93]. When selecting a method for checking conflicts one must be cognizant of *type I* and *type II* errors [132], that is, false positives (reporting a conflict that does not actually occur) and false negatives (not reporting a conflict that actually does occur). A method that exhibits type II errors should never be used because type II errors can lead to infeasible solutions. A method that exhibits type I errors may be used, but may make an overall algorithm incomplete or lead to sub-optimal solutions. In this section we provide a brief taxonomy of collision detection techniques for multiple moving agents.

In order to detect conflicts accurately in the General MAPF problem, agents' respective motion must be considered in continuous time and space. These conflicts are denoted $\langle a_i, a_j \rangle$ where a_i and a_j are the conflicting actions for agents i and j respectively. Continuous time and space conflict detection algorithms fall into four basic categories:

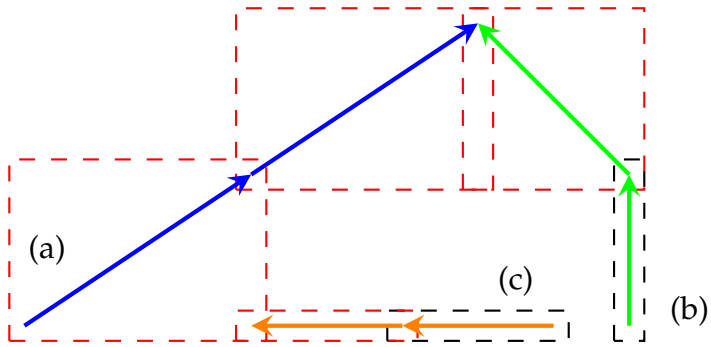


Figure 3.2.: Collision Detection using geometric containers. A collision is correctly detected between agents (a) and (b), but erroneously detected between (a) and (c).

geometric containers, sampling based, algebraic and geometric. In the following subsections, we provide an overview of each.

3.2.1. Geometric Containers

Geometric containers encapsulate portions of segmented motion in time and space using polygons, polytopes or spheres [193]. Then intersection detection is detected between the geometric containers of differing agents to determine if a collision has occurred. There are various approaches to intersection detection for stationary objects [82, 94].

In Figure 3.2 an example of this approach is shown which uses axis-aligned bounding boxes as geometric containers. The temporal dimensions are not shown, but each bounding box also has a temporal component. Agent (a), (b) and (c) take actions (represented as directed edges) to arrive at their goal. Axis-aligned bounding boxes are reserved for each of these edges, then an intersection check is carried out. Although a collision is correctly detected between (a) and (b), an erroneous collision is detected between (b) and (c). Although this approach is computationally fast, it will reserve more temporospatial area than necessary, especially when long edges are present in a path, resulting in the possibility of type I errors.

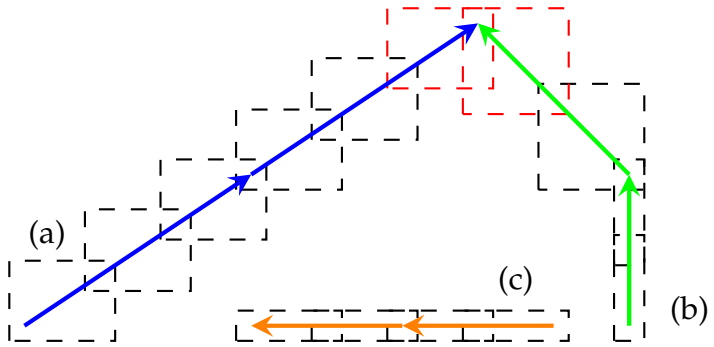


Figure 3.3.: Sampling-Based Collision Detection. A collision is correctly detected between agents (a) and (b).

3.2.2. Sampling-Based

This approach involves translating objects along their trajectories incrementally and using static spatial collision detection methods to detect overlaps at each increment. Figure 3.3 shows an example of this approach. Axis-aligned bounding boxes are created for each agent at regular intervals along their trajectories. In contrast to the example in Figure 3.2, there is no erroneous collision detected between agent (a) and agent (c). The sampling approach is very important, samples too far apart may miss a collision, but samples too close together is computationally costly. Adaptive sampling approaches can help improve the computational cost [61].

In grid worlds, a coarse form of collision detection, Brezenham's line algorithm [27], can be used for selecting a specific set of grid-squares covered by a trajectory and then checking whether agents are in the same grid square at intersecting times. A more robust approach based on Wu's antialiased line algorithm [209] can be used for agents with a shape [212].

3.2.3. Algebraic

When dealing with discrete-length motion segments, algebraic methods can be used to determine whether a collision will occur during the segment of motion. By parameterizing the trajectory, a closed-form solution to continuous-time conflict

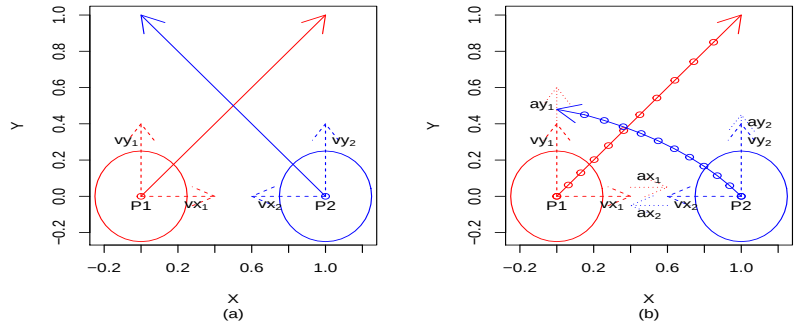


Figure 3.4.: Collision times for agent trajectories with constant velocity (a) and initial velocity with constant acceleration (b) can be solved algebraically

detection for circular, spherical, [48] and trianglular [126] shaped agents have been formulated. An example for circular agents is shown in Figure 3.4.

Algebraic methods calculate the exact time and duration of collision between two moving obstacles using closed-form equations. A detailed derivation of these equations for circular agents are presented in Appendix A.

3.2.4. Geometric

Geometric solutions are the most computationally expensive collision detection approaches, however they are formulated for many different obstacle shapes - typically primitive shapes, polygons or meshes. Two of the most popular approaches are constructive solid geometry (CSG) [150], and velocity obstacles (VO) [55].

CSG approaches treat the time domain as an additional polygonal dimension, *extruding* polygons into the time dimension, after which a static polygonal intersection check is applied. Computation of the extruded volumes can be very expensive and formulating ways to enhance CSG has been a subject of ongoing research [44, 91].

Velocity obstacles have been formulated for infinite length vector collision detection for arbitrary-shaped agents [55]. A velocity obstacle is depicted in Figure 3.6. A VO is created for two agents A and B, located with center points A and B as shown

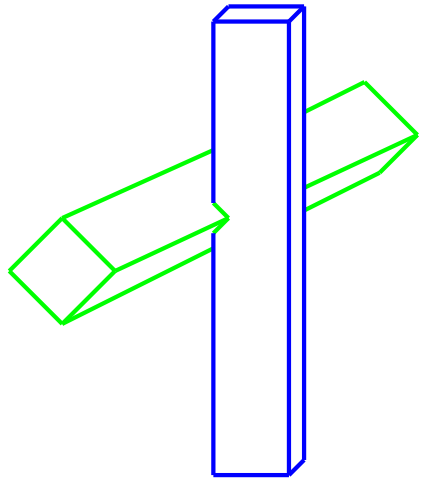


Figure 3.5.: Constructive Solid Geometry Collision Detection. Time is *extruded* into the model as an extra dimension, after which polygonal intersection detection is performed.

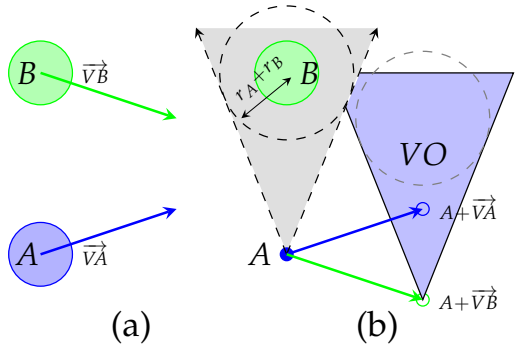


Figure 3.6.: Velocity Obstacle (VO) construction based on (a) two agents with motion vectors. The trajectories and shapes of agents are interpreted to create (b) the velocity obstacle – labeled 'VO'

in diagram (a). The agents have shapes – here shown as circles with radius r_A and r_B. The agents' motion follows velocity vectors VA and VB shown as arrows.

In order to construct the VO, first, the shape of agent B is inflated by computing the Minkowski sum $A \oplus B$ of the two agent's shapes. Next, two tangent lines from point A to the sides of $A \oplus B$ are calculated to form a polygon. Finally, the polygon is translated so that its apex is at $A + VB$. The area between the translated tangent lines is the velocity obstacle (labeled VO in the diagram). The VO represents the unsafe region of velocity for agent A, assuming agent B does not change it's trajectory. If the point $A + VA$ lies inside the VO, agent A will collide with agent B some time in the infinite future.

In the case of segmented motion, VOs can still be used for collision detection with some adaptations [6]. In addition, collision avoidance can be achieved by choosing a velocity for A such that $A + VA$ lies outside the VO. One approach is to set VA so that $A + VA$ lies on the intersection point of either of the VO tangent lines $\pm\epsilon$.

3.2.5. Summary

Depending on the application, any of the above methods may meet the problem constraints. Geometric containers is the approach of choice for domains with discretized-time movement models as it is the cheapest and (depending on the movement model) may yield no loss in accuracy. In continuous-time domains, one of the latter choices is usually preferable, with sampling often being the cheapest approach, followed by algebraic and geometric approaches. There is a trade-off with respect to accuracy and computational cost. The latter approaches provide the most flexibility when high accuracy and complex agent shapes are necessary.

3.3. Completeness in General MAPF

Recall that *completeness* means that an algorithm is guaranteed to terminate in a finite amount of time. There are two parts to completeness:

1. Termination with a solution *if one exists*. We call this *solution complete*.

2. Termination if a solution *does not exist*. We call this *termination complete*.

The proof for case 1 in MAPF has been shown extensively for A* [171] and CBS [160, 110]. However, a proof for case 2 has so far remained elusive. In fact, CBS and other search-based algorithms will run forever given a MAPF instance with no solution. Even a simple instance like the one shown in Figure 3.7(a) will cause CBS to run forever.

For CBS with Classic MAPF instances, it has been suggested [160] that practitioners run a sub-optimal, polynomial-time, complete algorithm [97, 155, 90] in parallel with CBS. In the case that no solution exists, the polynomial-time algorithm will report this fact and then CBS can be terminated. Complete, sub-optimal algorithms have been formulated for General MAPF [33]. However, polynomial-time algorithms for General MAPF remain elusive.

Figure 3.7(b) shows a breadth-first search tree for the instance in part (a). In the figure, each node depicts a joint-state, showing the action used to get to the state from its parent. Circles containing loops mean wait actions and arrows mean motion to the right or left. Some nodes lower in the tree are abbreviated with "..." for brevity. An "x" means the node was pruned due to a conflicting combination of actions. Because both agents can take actions such as waiting in place or moving back and forth without coming into conflict, the search tree grows infinitely.

The reason for this infinite growth is related to the inclusion of time in the state space. Adding the time dimension to a finite map like the one in Figure 3.7(a) makes the state space infinite. This results in an infinite search space.

In spite of this infinite search space, it has been shown that the upper bound on the number of joint-states in a Classic MAPF solution is $O(|V|^3)$ [97]. Therefore any Classic MAPF algorithm can safely return $NO\ SOLUTION$ once the length of agents' plans exceeds $|V|^3$. However, a proof for a similar polynomial bound for General MAPF remains elusive.

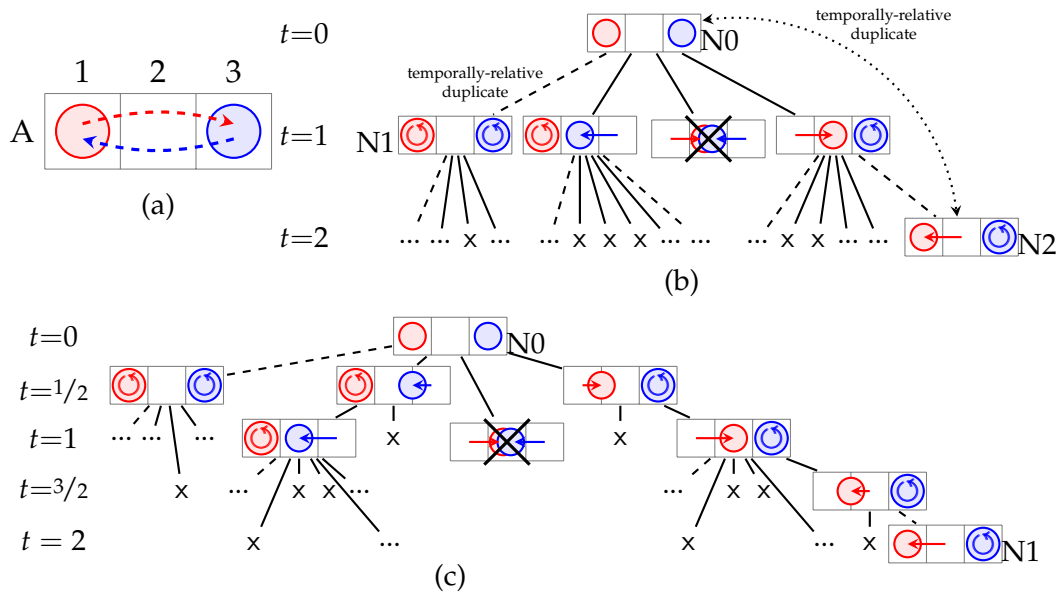


Figure 3.7.: Illustration of (a) a MAPF instance with no feasible solution, (b) a corresponding breadth-first search tree and (c) a breadth-first search tree with wait actions of $1/2$.

Temporally-relative duplicate pruning is a novel technique to ensure completeness for search-based MAPF algorithms including A*, CBS and CBICS. We first formally define it, then show its applicability to both Classic MAPF and General MAPF.

3.3.1. Temporally-Relative Duplicate Pruning

Duplicate pruning is a technique commonly used with search algorithms in graphs with cycles in order to eliminate exploring sub-optimal paths with loops [185]. A *loop* in a path visits the same state twice anywhere in the sequence of states. In MAPF, loops are not recognized as duplicates because of the time dimension. In fact, a feasible solution may correctly contain many, or even all agents visiting the same vertex more than once (for example when one agent waits for another).

In Figure 3.7(b), each level in the tree represents a unique time step which distinguishes a joint-state at one level from the nodes above and below it. However, if the way we distinguish between joint-states at different time steps is relaxed, duplicates can be found. Two joint-states from different time steps that are identical when the comparison of the time dimension is relaxed are called *temporally-relative duplicates*. In Figure 3.7(b), a dashed line connecting a parent to child node means the child is a temporally-relative duplicate of its parent or one of the parent's ancestors. For example, the leftmost child (node *N1*) of the root node (node *N0*) is a temporally-relative duplicate of its parent because the location of agents are exactly the same in both the parent and child nodes and the temporally-relative time is the same for each agent. The node labeled *N2* is a temporally-relative duplicate of *N0*.

Nodes with solutions containing temporally-relative duplicates have sub-optimal solutions and expanding these nodes is unnecessary for MAPF algorithms. Furthermore, temporally-relative duplicates always occur with unsolvable instances, and pruning them ensures completeness.

Temporally-relative duplicate pruning removes successors during expansion which have been visited before in a temporally-relative sense. In this discussion, and as

shown in Figure 3.7(b), we assume the Classic MAPF setting, using the breadth-first search (BFS) [38] algorithm. Within the BFS, we include all k agents in the state space. This is an important distinction from decoupled algorithms like CBS and CBICS which plan subsets of agents at the low level. In Sections 3.3.2 and 3.3.3, we address the General MAPF setting, CBS and CBICS.

BFS starts with an initial state, and then performs an *expansion*, which means generating all successors of a node. For this, we use the conflict-aware successor generation algorithm, Algorithm 2.2 which generates $\mathbf{S}'$, the Cartesian product of all single-state successors which are non-conflicting. Figure 3.7(b) shows conflicting joint-states with an "x" over them.

We define $\Pi(S)$ to be the predecessor path of S such that $\Pi(S)$ is a sequence of joint-states reaching from the joint start state to S. We define a joint-state temporal adjustment function which adjusts the time component of all single-agent states to be relative to the earliest single-agent state:

$$\Delta_t(S) = \{\forall s_i \in S; (s_i.v, s_i.t - t_{min}(S))\}$$

where

$$t_{min}(S) = \underset{s_i \in S}{\text{MIN}} s_i.t$$

We determine the set of temporally-relative duplicates $\mathbf{S'_{dup}} \subseteq \mathbf{S}'$ by performing a set-intersection of temporally-adjusted joint-states:

$$\mathbf{S'_{dup}} = \Delta_t(\Pi(S)) \cap \Delta_t(\mathbf{S}')$$

Then all *corresponding* $S' \in \mathbf{S'_{dup}}$ are removed from $\mathbf{S}'$:

$$\mathbf{S}' \leftarrow \mathbf{S}' \setminus corr(\mathbf{S'_{dup}})$$

where $corr()$ fetches original states corresponding to temporally-relative ones.

This procedure is easily added to the JOINTEXPANSION routine shown in Algorithm 2.2. It will prevent temporally-relative duplicated states from being added to the OPEN list. For example, temporally-relative duplicate pruning will remove the left-most child of the root $\{(A1, 1), (A3, 1)\}$ in Figure 3.7(b) because when adjusted to be temporally-relative $\{(A1, 0), (A3, 0)\}$, it is identical to the start state which is a member of $\Pi(S)$.

Temporally-relative duplicate pruning has two effects:

1. It eliminates sub-optimal paths from consideration in the search.

2. It renders the search-space finite.

With temporally-relative duplicates removed (e.g., eliminating child nodes linked to parents with dashed lines in Figure 3.7(b)), the search tree is much smaller, saving computation. Additionally, once all combinations of feasible locations have been explored (two locations for the red agent and two locations for the blue agents) no further child nodes can be visited. Equivalently, the original infinite search space becomes finite. Hence, instead of growing forever, the tree is truncated allowing the search to terminate and BFS recognizes that no solution exists. Detailed proofs are given in Appendix B.

It has been shown that if a solution exists to a "classic" MAPF problem, the lowest-cost solution can have no more than $O(|V|^3)$ time steps [97]. Thus we can conclude that one can terminate any MAPF algorithm once candidate solutions reach a cost of $|V|^3$. By this logic, for the instance in Figure 3.7(a) with $|V|$=3, one can terminate the algorithm once the BFS tree reaches a depth of $|V|^3$=9. This would render BFS complete, however, because this problem has a max branching factor of b=6 it would perform $O(6^9)$ expansions to do so. On the other hand, with temporally-relative duplicate pruning, the search space is exhausted after only 7 expansions.

3.3.2. Temporally-Relative Duplicate Pruning in General MAPF

Now, we further generalize temporally-relative duplicate pruning to General MAPF. The theory for temporally-relative duplicate pruning ultimately relies on modular arithmetic. That is, some time t is a duplicate of $t' \texttt{ mod } \Delta$, where $t < t'$ and Δ is a differential in action start times. If we make the assumption that all action durations used by all agents are in $\mathbb{Q}+$, the set of non-negative rational numbers, then Δ is also a rational number. Also, if the set of action durations is finite, then Δ is guaranteed to repeat eventually, resulting in a temporally-relative duplicate. For this reason, we define MAPF_Q as MAPF with rational-valued action durations. Considering that in modern computing, numeric representations are limited to discrete values and that many problems can be represented with a finite set of action durations, MAPF_Q is a reasonable surrogate for MAPF_R (MAPF with real-valued action durations).

In order to transform the example problem in Figure 3.7 into a MAPF_Q instance, we change wait actions to be 0.5 duration. Doing so changes the search tree so that levels in the tree are now staggered as shown in Figure 3.7(c). This also means that some of the longer-duration actions may be duplicated in $\Pi(S)$. The following shows $\Pi(S)$ for N1.

$$
\begin{aligned}
\Pi(N1) = [& \\
& S^0 = \{(A1, 0),\ (A3, 0)\}, \\
& S^1 = \{(A1, 0),\ (A3, .5)\}, \\
& S^2 = \{(A2, 1),\ (A3, 1)\}, \\
& S^3 = \{(A2, 1),\ (A3, 1.5)\}, \\
& S^4 = \{(A1, 2),\ (A3, 2)\} \\
&]
\end{aligned}
$$

the following shows $\Delta_t(\Pi(S))$ for N1:

$$\begin{aligned}
\Delta_t(\Pi(N1)) = [& \\
& S^0_\Delta = \{(A1, 0),\ (A3, 0)\}, \\
& S^1_\Delta = \{(A1, 0),\ (A3, .5)\}, \\
& S^2_\Delta = \{(A2, \mathbf{0}),\ (A3, \mathbf{0})\}, \\
& S^3_\Delta = \{(A2, \mathbf{0}),\ (A3, \mathbf{.5})\}, \\
& S^4_\Delta = \{(A1, \mathbf{0}),\ (A3, \mathbf{0})\} \\
&]
\end{aligned}$$

Because $S^0_\Delta = S^4_\Delta$, S^4 would be pruned during the expansion of S^3. The temporal adjustment correctly distinguishes between duplicates. For example, even though the *vertices* are identical between S^0 and S^1, the blue agent is allowed to wait in place for two time steps from S^0 to S^1 while the red agent is taking a longer action. Temporally-relative duplicate pruning is guaranteed to make the search space finite for MAPF_Q. When no solution exists, enough temporally-relative states will be visited so that eventually no non-duplicated joint states can be generated, and the search will halt with no solution. Detailed proofs of these claims are included in Appendix B.

3.3.3. Temporally-Relative Duplicate Pruning for a Subset of Agents

So far, we have seen how temporally-relative duplicate pruning works for centralized algorithms such as BFS or A*. For dynamically centralized algorithms such as CBS or CBICS, agents are planned separately in the low level search. Duplicate pruning cannot be applied in the same way for subsets of agents because an agent or subset of agents may need to repeat actions multiple times due to the presence of other agents. We would not want to prune such states in the low level search. However, the influence of other agents is applied via motion constraints at the low-level. Thus we can apply temporally-relative duplicate pruning in the local context to any generated

states which occur after the end of the last motion constraint. If re-planning for agent i given a set of motion constraints M_i, temporally-relative duplicate pruning can be applied to states that occur after $t_{max}(M_i)$. Other approaches are also possible, but are outside the scope of this thesis. This guarantees completeness at both the low and the high level and proofs for this claim is in Appendix B.

3.3.4. Summary

We have shown that temporally-relative duplicate pruning has desirable properties for MAPF algorithms, namely, increased efficiency and completeness guarantees. It can be applied to search-based algorithms for both Classic MAPF and MAPF_Q, a reasonable surrogate for General MAPF. We also discussed an application for dynamically centralized MAPF algorithms.

As described at the beginning of this section; no Classic MAPF instance can take more than $|V|^3$ time steps [97]. Hence, instead of eliminating temporally-relative duplicates we could prune nodes with $\text{MIN}(S.t) \geq |V|^3$. Although this approach would allow us to definitively claim completeness for Classic MAPF, the $O(|V|^3)$ bound (or any bound) has yet to be proven for General MAPF.

Experimentally, we found that explicitly pruning temporally-relative duplicates for small unsolvable Classic MAPF instances allowed termination significantly before $|V|^3$ steps were reached. For solvable instances we found that pruning temporally-relative duplicates had no statistically significant impact on runtime.

3.4. Wait Times in General MAPF

The algorithms in this thesis focus on finite graphs. However, some formulations of General MAPF allow for *arbitrary wait times* [6, 37], meaning agents may wait at a vertex for an arbitrary time duration, $d(a) \in \mathbb{R}^+$. Another natural extension would allow arbitrary velocities.

3.4.1. Durative Conflicts

In contrast to Classic MAPF in which conflicts are instantaneous, conflicts in General MAPF are typically *durative*, meaning, the period of overlap for two conflicting agents' shapes may be longer than one instant in time. An example of a durative conflict is shown in Figure 3.9. Given a pair of conflicting actions for a pair of agents, the amount of time necessary for one agent to wait in order to avoid conflict with another agents is called *wait time*. See Appendix Figure A.2 for an illustration of wait time.

The wait time for avoiding conflict is also known as an *unsafe interval* [6, 199, 197] because it is the interval in which an agent cannot start traversing an edge without coming into conflict. An unsafe interval $[t, t']$, may vary depending on the length and relative angles of the edges being traversed, as well as the size and shape of the agents. Details surrounding this phenomenon, including calculation of the duration of overlap and are covered in Section 3.4.3. Unsafe intervals are an important component of developing constraints for General MAPF as discussed in the next section.

3.4.2. Arbitrary Wait Times

Allowing arbitrary wait times can have several benefits: (1) path and solution costs may be reduced in many cases, (2) the amount of work may be reduced in many cases (due to both shorter paths and simpler conflict resolution) and (3) an unsolvable problem may be made solvable. Figure 3.8(a) illustrates a scenario where the optimal solution is for agent a to arrive at vertex x at exactly t=2. However, assuming fixed wait times of 1 (shown by self loops), this is impossible because edge (a_s, x) has a duration cost of 1.4. Again, with fixed wait times of 1, the next best alternative is for agents b, c and d to move through node x first, followed by agent a. This will incur a total cost of 23.8. Waiting for agent a to move first will also incur a total cost of 23.8.

Allowing a wait time of 0.6 for agent a (moving from a_s to a_g will allow all agents to proceed directly to their respective goals, without incurring extra cost on any

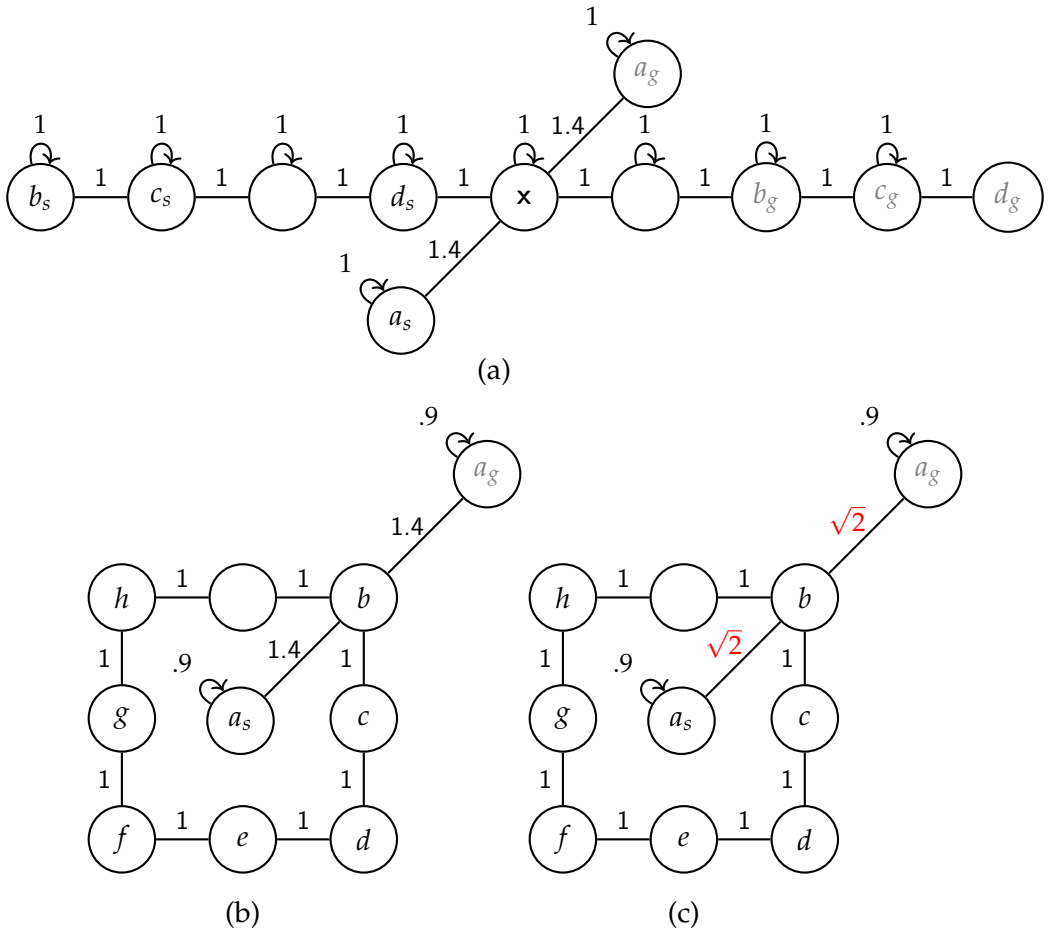


Figure 3.8.: Pathological cases for General MAPF.

other agent. Thus the total cost is reduced to 20.4. Alternatively, agent a could begin its motion immediately, but at a reduced velocity of 0.7.

Figure 3.8(b) shows a solvable instance and Figure 3.8(c) shows an unsolvable instance. In these examples, each agent is already at its goal except for agent a which needs to move from a_s to a_g. Additionally, waiting is not allowed in any node except the two nodes with self loops. In instance (b), an optimal solution will require agents b-h to move back and forth, clockwise or counter-clockwise until the blank space lines up such that agent a can arrive at the open space between its start and goal positions

at the exact time that it is unoccupied. for example, if agents *b*-*h* move in the clockwise direction (1.0 duration) and then back in the counter-clockwise direction (for a total of t=2.0) and perform the same two actions again to arrive back at the start position (for a total of t=4.0). Also, at the same time agent *a* performs four wait actions for a total time of t=3.6. Then agents *b*-*h* move clockwise one step, which will leave the space between agent *a*'s start and goal open at precisely t=5.0, while agent *a* traverses to the open space to arrive at precisely $t = 5.0$. Finally, one more step puts agent *a* at its goal, and one step clockwise returns all other agents to their goal position.

We observe from instance (b) that the ability to arrive at the goal depends on the action durations to be rational (action durations need to have a common denominator). Thus instance (c) is unsolvable. No matter how many combinations of moves the agents make, the gap will never line up for agent *a* to pass through without conflict. Thus solvability for fixed wait times in general for General MAPF instances is reliant upon rational action durations. Furthermore, the amount of work can increase exponentially as the *resolution* of the action durations increases. If example (b) were to change the cost of the diagonal edges to 1.42 instead of 1.4. The length of the solution d would increase linearly, causing the base complexity of the problem $O(b_{base}^{k^d})$ to increase exponentially.

The unsolvability and extraneous work that fixed wait times incurs can be alleviated by allowing arbitrary wait times. In instance (b), a wait action of duration 1.6 for agent *a* could be used to solve the problem in only 3 steps. In instance (c), a wait action of $3-\sqrt{2}$ would make the problem solvable.

3.4.3. Determining Arbitrary Wait Time

In order to determine the arbitrary wait time needed to avoid conflict, a binary search algorithm can be used for actions a_i, a_j for two conflicting agents. The approach is to test different time delays for agent a_i until no conflict occurs. We start with a delay value of $\Delta t = a_i.s'.t - a_i.s.t$. If a conflict occurs with that delay, Δ_t is doubled and

a conflict check is performed iteratively until no conflict occurs. After a value of Δt for which no conflict occurs is found, binary search is performed between the previously conflicting delay time and Δt until the difference between non-conflicting and conflicting values for Δt is within some accuracy threshold ϵ.

The delay time is not always the same for both agents in the conflict, but the procedure can either be performed for agent j separately, or binary search can be applied in the negative direction for agent i to find the delay for agent j.

Although this approach is generic, it is limited in accuracy by the parameter ϵ, and limits our usage to problem instances with rational action durations. In some cases, a more accurate and computationally efficient solution is possible via a closed-form equation to find the exact delay required. Figure 3.9 illustrates the use of a conic equation to find the *exact* amount of delay required for collision avoidance for circular agents.

Figure 3.9(a) shows the agents position and motion vectors. The agents start their motion at the same time and their radii are .25 units. Figure 3.9(b) shows a squared distance plot between the edges of the circles as a function of time. The portion of the graph below zero indicates overlap of the circles.

Figure 3.9(c) shows a bivariate conic section (ellipse) which represents Δt as a function of time. Any combination of Δt and time that falls inside the ellipse will result in collision. Meaning, a delay of Δt will cause a conflict at time t. If delay is increased or decreased sufficiently, no conflict will occur. Thus the minimum time of delay for agent i occurs at the top of the ellipse, and the minimum time of delay for agent j occurs at the bottom of the ellipse. The derivation of equations for these values is detailed in Appendix A.2.1.

Determining the velocity change needed to avoid conflict can be done in a similar fashion, using binary search. An analytic method for minimum velocity change for circular agents is detailed in Appendix A.2.3.

Note that the example in Figure 3.8(b) uses rational numbers for edge weights. If the value 1.4 is changed to $\sqrt{2}$ (an irrational number) as in 3.8(c), the binary search

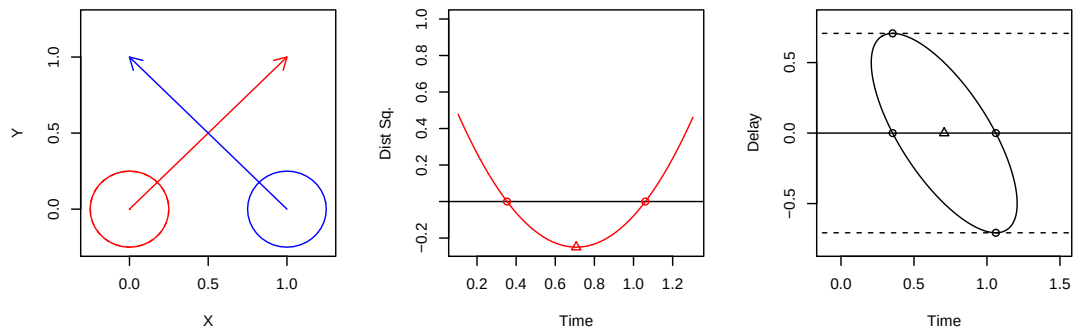


Figure 3.9.: (a) Agent trajectories, (b) squared distance plot and (c) ellipse showing collision intervals for varying *delay*.

method will not suffice, since an acceptable accuracy threshold can theoretically never be reached. Hence, the binary search method is only valid for MAPF_Q.

3.4.4. Implementing Arbitrary Wait Times in MAPF

For planning with arbitrary-duration wait actions the Safe Interval Path Planning (SIPP) [143] algorithm can be used. SIPP is an A*-based algorithm for finding a safe and optimal path for an agent among moving obstacles with known trajectories. In SIPP, the state space is augmented such that states are tuples $(v, [t, t'])$ where $v \in V$ is a vertex in G and $[t, t']$ is a *safe interval*. A safe interval is a contiguous time range in which an agent may occupy vertex v without coming into conflict with any obstacles.

In environments applicable to canonical orderings such as the 2^k neighborhoods, Jump Point Search with Temporal Obstacles (JPST) [76] has also been successfully used for MAPF.

3.5. The Conflict Avoidance Table

The conflict avoidance table (CAT) is used with dynamically coupled algorithms. As described in Section 2.4.4, the CAT stores the actions of all agents' paths

in a way that they can be looked up quickly. It functions as a tie-breaking mechanism in the OPEN list. When there are multiple paths of equal cost, preference is given to paths containing fewer conflicts with other agents. It has been shown to be an effective enhancement on 4-neighbor grid domains for selecting candidate solutions in the independence detection (ID) framework [171], and has been shown to be useful with CBS [160] and CBICS [200]. Outside of this thesis, the utility of the conflict avoidance table has not been studied for General MAPF.

Tie-breaking in the OPEN list using the CAT can be done as follows:

1. During a node generation, determine the number of actions n_c in the CAT that the action $a{=}(s, s')$ conflicts with. Where s is the parent node and s' is the newly generated child node.

2. Store n_c as an attribute of the newly generated node.

3. For the OPEN list's comparison operator, prioritize states with equal cost (as a secondary criterion) by lower n_c.

In step 1, determining n_c for a requires a lookup of other agents' actions which have time overlap with a in the CAT, followed by performing a conflict check. We will refer to this process as a lookup+conflict check. With this approach, a conflict check operation will occur at most once per node in the OPEN list.

The data structure for a CAT in Classic MAPF can be constructed to allow random access for fast lookup and insertion times. However, more complex data structures are required for General MAPF. With arbitrary length intervals, one action in the path of agent i may have full or partial temporal overlap with many actions by the other $k{-}1$ agents. In the worst case one must check an action for agent i against *all* other actions in the paths of *all* other agents. That is $O(kd)$ operations where d is the maximum path length of all paths in the solution. Some possible implementations of this lookup mechanism are an interval tree or a hash table with direct addressing quantized to the smallest possible action duration.

An interval tree approach will incur $O(kd\log(kd))$ operations per node in OPEN (compared to $O(k)$ for classic MAPF). A hash table approach will incur an expected $O(k\ell)$ operations per OPEN node where ℓ is the average time length of actions divided by the bucket size of the hash table. Because most actions are not likely to land exactly on the hash boundary, the time interval will usually span at least two buckets in the hash table.

The key question is: *Considering that lookups and collision checks are much more expensive in General MAPF and a lookup+conflict check must occur at least once for each node in the OPEN list, at what point does the computational cost of the CAT outweigh the benefits?*

There are often many symmetric, optimal paths in grids and (more generically) graphs where the problem has certain start/goal configurations. The simplest example is that of cardinal grids (see Figure 3.10(a)), when the start and goal are separated by Δx and Δy, the total number of points, *area*, that fall in optimal paths is of size $\Delta x \Delta y$. In grid domains, *area* is computable directly, however, it can be computed in a domain agnostic manner as will be explained later. Though *area* is polynomial in the dimensional space, the number of combinations of possible optimal paths through the optimal points is exponential [151]. For 2^k neighborhoods, the number of optimal paths is $O(4^{\frac{d}{2(k-1)}})$ based on the central binomial coefficient where $d{=}\Delta x{+}\Delta y$, (i.e., Manhattan distance), but is also highly dependent on the ratio $\Delta x/\Delta y$. Notably, this asymptotic bound decreases with k – meaning the number of optimal paths decreases exponentially as k increases.

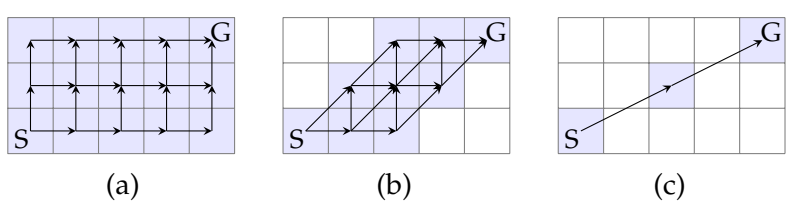


Figure 3.10.: Set of optimal path edges with (a) 4-neighbor, (b) 8-neighbor, and (c) 24-connected grids. *area*, the number of grid cells in optimal paths, is 15, 9 and 3 respectively.

This phenomenon is illustrated in Figure 3.10 where *area* is shown for domains with 4, 8, and 16-neighbor grids. Comparing *area* in the examples, it is easy to see that when actions allow direct, or near-direct movement toward the goal, the number of optimal edges from start to goal decreases, hence *area* decreases. This means that when performing tie-breaking which the CAT facilitates, there are fewer options to tie-break between. Thus we hypothesize that the CAT will be less effective as *area* decreases. Because this type of analysis is domain-specific, we propose a generic sampling-based approach to computing a "directness" statistic *ratio* – a measure which we hypothesize is correlated to the effectiveness of the CAT.

1. Randomly create a set of problem instances **I** for the target environment.

2. Compute *area*, the number of vertices in all optimal paths from start to goal for each problem instance. This can be done using a depth-first or breadth-first search or by using an A* search from start to goal, and then counting nodes on the OPEN list for which $f = C*$.

3. Compute *length*, the length of any optimal path - that is, the number of actions needed to traverse from start to goal.

4. Compute *ratio*, the mean of $\frac{area/length}{d}$ over all instances in **I**. Where d is the dimensionality.

Empirically, when *ratio* is much greater than 1, the conflict avoidance table will be of benefit. When *ratio* is close to 1, the impact of a conflict avoidance table will be marginal or negative.

We tested this hypothesis on a 2-dimensional environment with varying connectivity. First, we computed *ratio* for 4, 8, 16, and 32-neighbor grids (see Table 3.1). Next, we ran CBS on 4, 8, 16 and 32-neighbor grids. Figure 3.5 shows the reduction in CBS high-level conflicts when the CAT was used in each of the environments. The experiments for Figure 3.5 consist of 100 random start/goal configurations for varying

Table 3.1.: Mean of *ratio* statistic for 1000 samples

Configuration	*area*	*length*	*ratio*
4-neighbor	56.1444	14.0602	1.9966
8-neighbor	27.9296	10.2445	1.4265
16-neighbor	16.9658	5.94673	1.3632
32-neighbor	11.4157	4.59839	1.2413

numbers of agents. Instances that took more than 5 minutes to find a solution were terminated early and their results are included in the averages.

We found that the reduction in conflicts was directly correlated to *ratio*, and that in the case of the 32-neighbor environment, the CAT was of little benefit. In addition, the overhead of maintaining the CAT negatively impacted the runtime. In the case of the 32-neighbor environment, almost all of the runtimes were lower when the CAT was not used.

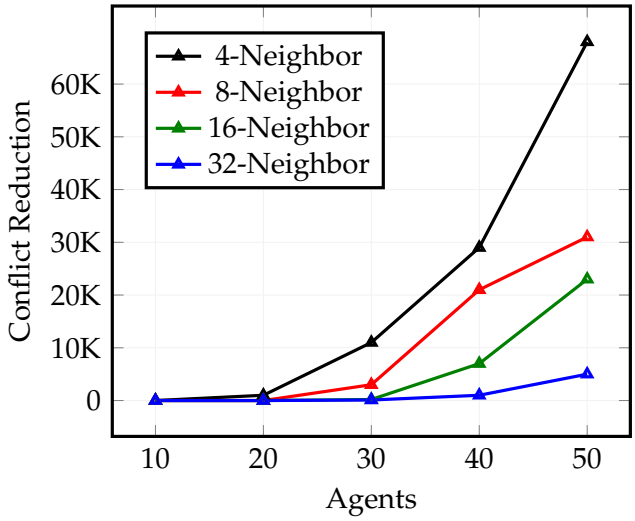


Figure 3.11.: Comparison of the reduction in mean number of total conflicts in CBS when using a conflict avoidance table planning on an unobstructed 64 × 64 grid.

3.6. Summary

We provided a new formal definition of *General MAPF* which allows for a wider range of applications by allowing actions of arbitrary duration and velocity; and agents of arbitrary size and shape. Because of this, conflict detection takes a considerable amount of extra attention. We provided a short survey of conflict detection techniques. One must decide on a conflict detection mechanism that is accurate and fast, and preferably does not return false positives and never returns false negatives.

We explained the concept of durative conflicts. We provided a new *analysis of completeness* for General MAPF and a new approach for guaranteeing completeness for MAPF in general called *temporally-relative duplicate pruning (TRDP)*. We proved that completeness can be guaranteed in General MAPF, assuming action durations are rational and temporally-relative duplicate pruning is performed. Finally, allowing arbitrary wait times has multiple benefits, including better cost quality for solutions, exponentially less work, and may allow solvability when fixed wait actions will not.

Finally, we provided a *new statistic* for describing the ratio of equivalent-cost alternate paths in a General MAPF domain which is useful in determining whether to use a conflict avoidance table with dynamically centralized algorithms.

4. Extensions for Increasing-Cost Tree Search

4.1. The Increasing-Cost Tree Search

Increasing-Cost Tree Search (ICTS) [163] is an optimal search algorithm for MAPF. ICTS is complete, but only if a solution exists. In Classic MAPF, it was shown to be state-of-the-art for certain types of problems such as grid maps with wide open areas [52, 85]. ICTS is a two-level algorithm; we describe each level next.

4.1.1. High Level

At its high level, ICTS searches the increasing cost tree (ICT). Every node in the ICT consists of a k-ary vector $\langle C_1, ..., C_k \rangle$ which represents the question: *Is there a feasible solution where the path cost of each agent a_i is exactly C_i?* The total cost of an ICT node **I** is $C{=}C_1 + ... + C_k$. The objective is to find an ICT goal node of minimal C. The ICT root contains the k-ary vector of the shortest path cost from $start_i$ to $goal_i$ in G for each agent, ignoring other agents. A child in the ICT is generated by increasing the cost for one of the agents by an increment value $\delta{=}1$. Figure 4.1.1 depicts an ICT for 3 agents. The root node contains the optimal path costs for individual agents ignoring conflicts: $\langle 10, 10, 10 \rangle$. The leftmost child is created by incrementing the first element to yield $\langle 11, 10, 10 \rangle$, the next child increments the second element to yield $\langle 10, 11, 10 \rangle$ and so on. Dashed lines indicate duplicate children which are pruned.

An ICT node containing $\langle C_1, ..., C_k \rangle$ is a goal if there is a feasible solution such that the individual path cost for each agent a_i is exactly C_i. We use Δ to denote the

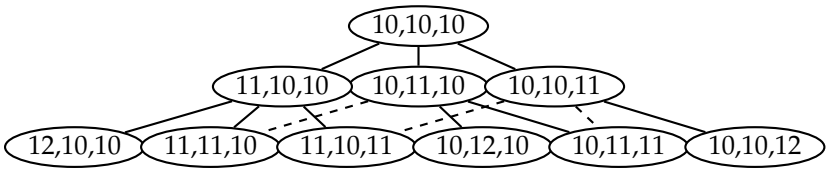


Figure 4.1.: Unit-cost ICT with cost vectors

depth of the lowest cost ICT goal node. Since all nodes at the same height have the same total cost, a breadth-first search of the ICT will find the optimal solution.

4.1.2. Low Level

The low-level acts as a goal test oracle for the high-level. For each ICT node generated by the high-level, the low-level is invoked. Its task is to find a feasible solution such that the cost of the individual path of agent a_i is exactly C_i. For each agent a_i, ICTS stores all single-agent paths of cost C_i as a directed acyclic graph with no duplicated edges or vertices. This compact representation is also known as a multi-value decision diagram (MDD) [170]. The low-level searches the Cartesian product of the MDDs in order to find a set of k feasible paths. If a feasible set of paths exists, the low-level returns true and the high-level halts. Otherwise, false is returned and the high-level resumes its search. In Figure 4.1.1(a) the low-level returned false for $\langle 10, 10, 10 \rangle$ (the root of the ICT) so 3 successors are generated. The next node visited by the high-level is $\langle 11, 10, 10 \rangle$. Assuming the low-level returns true at this node the high-level would then halt.

4.1.3. Pruning Enhancements

Several pruning enhancements have been introduced for ICTS [161]. These techniques search for a solution at the low level for $m < k$ agents. If there exists a subset of m agents for which no valid solution exists, there cannot exist a valid solution for k agents. Thus, the low-level can immediately terminate with false. Since building a Cartesian product for a subset of agents takes exponentially fewer steps,

it is often worth testing the Cartesian product of subsets of agents for feasibility before committing to a testing a full k-agent Cartesian product. Experimentally, testing m-agent subsets for m=2 and m=3 was shown to yield a significant improvement in performance. In general, settings of m=2 performed the best. This enhancement is called Simple Pairwise Pruning (SPP).

Additionally, analysis of conflicts of the Cartesian product between $m < k$ agents' MDDs allows for eliminating mutually-conflicting nodes from the individual MDDs. This makes the individual MDDs sparser, potentially allowing for elimination of even more MDD nodes versus the Cartesian product of other agents. Additionally, this speeds up the process of computing the cross-product of all k MDDs. Again, settings of m=2 performed the best. This enhancement is called Enhanced Pairwise Pruning (EPP).

4.2. Sufficient Conditions for Completeness and Optimality

Although the comptational complexity of ICTS is discussed in the literature, to our knowledge, no formal proof of optimality and completeness for ICTS was ever published [162, 161, 163]. We offer a proof here as follows: We first show that all possible cost combinations are explored in an increasing manner until a goal is found. Next, we show that the low-level is optimal and complete. Finally, we show that if a solution exists, ICTS is guaranteed to find it.

Lemma 4.2.1. *In the ICTS High-Level, all possible cost combinations for all agents are explored in order of increasing sum-of-costs until a goal is found.*

Proof. Each level in the ICT contains all possible cost combinations for a fixed sum-of-costs for all agents and the sum-of-costs is strictly increasing as the level increases. Assuming the low-level is complete, a breadth-first search of the ICT guarantees an optimal solution will be found if one exists because each level is explored in its entirety before moving on to the next level. By contradiction, assume that a sub-optimal

solution was found by ICTS on level ℓ of the ICT. This would mean that no solution was found on any level $<\ell$. But this is impossible since an optimal solution must lie on some level $<\ell$. Thus, either the low-level has returned an invalid result for some node in a previous level, or some cost combination has been skipped in a previous level. This contradicts the known properties of breadth-first search, or else it contradicts our assumptions about the low-level. □

Now that we have shown that the high-level will explore all possible lowest costs until a goal is found, we show that the low-level is complete.

Lemma 4.2.2. *The low-level is complete.*

Proof. The low-level operates on a set of k finite-sized MDDs. Therefore, the search space is finite. We assume that the low-level uses the successor generation scheme from Algorithm 2.2 and depth-first search. Because depth-first search is complete for finite state spaces [38], then a joint state where all agents are in their goal configuration is guaranteed to be found if one is reachable. Hence, the low-level is guaranteed to return the correct result, whether a goal is reachable or not. □

We now combine the proofs to show that ICTS is optimal and complete.

Theorem 4.2.3. *ICTS is optimal and complete if a solution to the problem instance exists.*

Proof. Per Lemma 4.2.1, the ICTS high-level is guaranteed to be optimal if the low-level is guaranteed to be complete. The low-level is guaranteed to be complete per Lemma 4.2.2. Therefore, ICTS is guaranteed to be optimal and complete (if a solution to the problem instance exists). □

Observation 4.2.4. *ICTS is incomplete if no solution to the problem instance exists. This is because the search space of the high-level is infinite. We conjecture that temporally-relative duplicate pruning can be used to allow ICTS to terminate in the case no solution exists.*

Recall from Section 3.3.1 that temporally-relative duplicate pruning prunes duplicate states when a state is re-visited in a temporally-relative sense. We conjecture

that ICTS can be made to terminate in the case that a solution does not exist as follows: First, perform temporally relative duplicate pruning at the low-level. Second, compare the size of the low-level tree for an ICT node with that of its parent.

Because temporally-relative duplicate pruning makes the search space finite per Lemma B.0.8, if the parent and child explored the exact same number of nodes, it means their finite search space is identical in spite of the child node having a cost limit increase. Therefore, we hypothesize that further exploration of the child branch is guaranteed not to lead to a solution and can be pruned. If the instance is unsolvable, eventually all branches at the high level will be pruned, allowing ICTS to terminate. Further clarification and proofs of this hypothesis are beyond the scope of this thesis.

4.3. Re-Formulation for General MAPF

Several changes to the original ICTS algorithm are necessary for General MAPF. The formulation of the new high level algorithm is dependent on the structure of the MDDs built by the low-level. Figure 4.2 depicts a single-agent pathfinding problem on a grid where the agent must move from the start coordinate B1 to the goal coordinate A3. Figures 2(b) and 2(c) depict the MDD for optimal paths when the grid is 4-connected, and fully connected – where every grid square is directly connected to all other grid squares. The x-axis shows cost which increases as the agent moves from the start toward the goal with Euclidean costs. Because the 4-connected MDD has unit costs, it results in a single sink node at the goal. However, when fully-connected, the resulting MDD has multiple sink nodes in the highlighted interval $(\sqrt{5}, \sqrt{5}+1]$. This leads to the simple observation that with non-unit costs, for each ICT node multiple goals may be found in the *interval* $(C, C+\delta]$.

Setting $\delta{=}\epsilon$, the smallest possible increment, will ensure optimal results, but may cause the ICT depth, Δ, to be extremely large. On the other hand, if we set δ to a large value and change the low-level search to solve an optimization problem (instead of a satisfaction problem), the result will also be optimal. While this would

incur a smaller ICT, our new choice of δ might push the value of C significantly past the optimal solution cost C^*, causing a large computational cost at the low-level. Since the value of C^* is usually unknown, we recommend setting δ to be a moderate value in order to mitigate the size of Δ and reduce the risk of drastically overshooting C^*.

4.3.1. Reformulated High Level Search

Algorithm 4.1 shows pseudo code for the reformulated high level search. In order to find a solution with cost C^* in the interval $(C, C+\delta]$, we generalize ICT nodes to have a vector of cost *intervals*: $\langle (lb_1, ub_1], ..., (lb_k, ub_k] \rangle$. The root node now consists of the vector $\langle (opt_1, opt_1], ..., (opt_k, opt_k] \rangle$, where opt_i is the cost of the optimal path for agent i, ignoring other agents (line 4). A child ICT node $\mathbf{I}'$ is generated from its parent $\mathbf{I}$

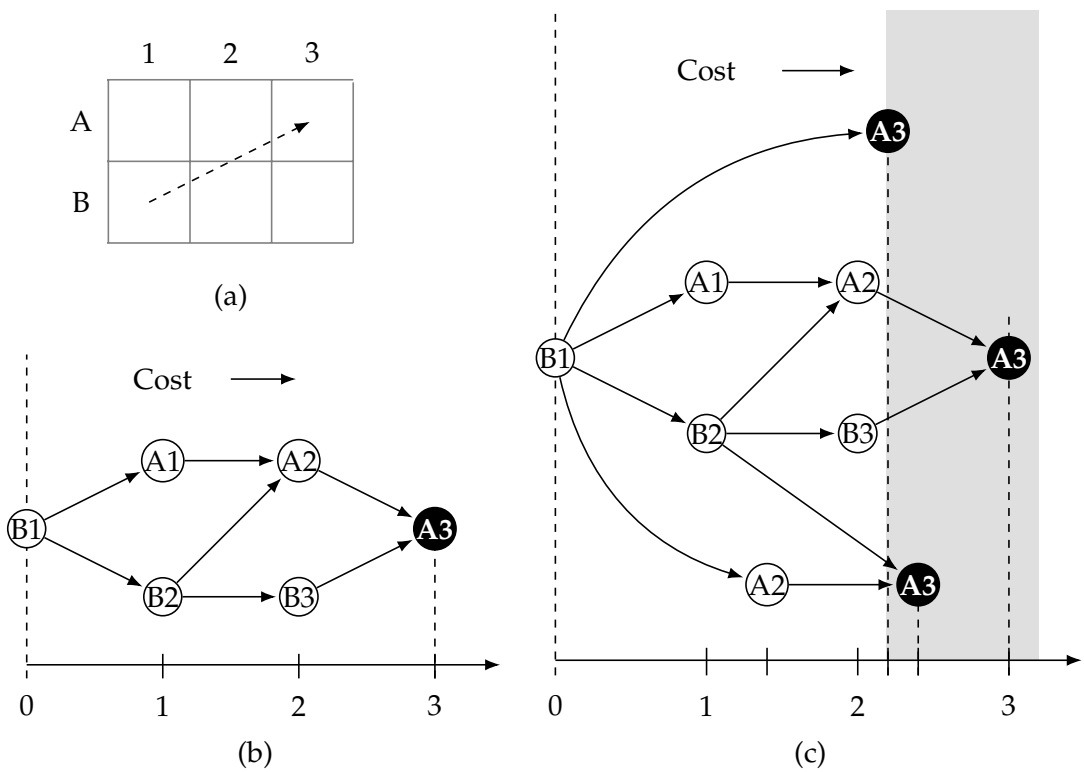


Figure 4.2.: (a) Problem instance and (b) associated MDD for all paths of cost between 2 and 3 for 4-Connected Grid (b) and (c) Fully Connected Grid

Algorithm 4.1. Reformulated ICTS High Level Search

```
ICTS
1: Input: A MAPF instance, δ: Increment value
2: incumbent ← ∞: Best solution cost so far
3: best ← ∅: Best solution so far
4: Build and push the root ICT node onto OPEN
5: while OPEN not empty do
6:     I ← OPEN.pop()
7:     if h(I) ≥ incumbent then
8:         return best
9:     C ←LOW-LEVEL(I,incumbent)
10:    if C = h(I) then                                  ▷ Was goal found?
11:        best ← I
12:        return best                                   ▷ This is the optimal solution
13:    else if C < incumbent then                        ▷ New incumbent?
14:        incumbent ← C
15:        best ← I
16:    else
17:        for i in 1 to k do                            ▷ Generate successors
18:            I′ ← I
19:            I′.lb_i ← I.ub_i
20:            I′.ub_i ← I.ub_i + δ
21:            Compute h(I′) by building ub_i-limited MDD_i
22:            OPEN.push(I′)
```

by setting $\mathbf{I}'.lb_i$ to $\mathbf{I}.ub_i$ and incrementing $\mathbf{I}'.ub_i$ by δ (lines 19, 20). Figure 4.3.1 shows an example of an ICT with δ=1. The root node of the tree contains vectors where both lb_i and ub_i contain optimal costs of 10 (with the abuse of notation $(10, 10]$) for each agent. Now, instead of reporting the *existence* of a solution in the ICT, the low-level will detect and report the *best cost* C, in the summed-interval $(lb, ub] = (lb_1 + ... + lb_k, ub_1 + ... + ub_k]$ if a feasible solution exists, ∞ otherwise (line 10). With cost-interval ICT nodes, it is now most efficient to search the ICT in a *best-first* manner. We define the minimum-cost single agent solution in the interval $(lb_i, ub_i]$ from each MDD_i as $best_i$ and use this for a lower-bound heuristic for an ICT node: $h(\mathbf{I}) = best_1 + ... + best_k$ (line 21).

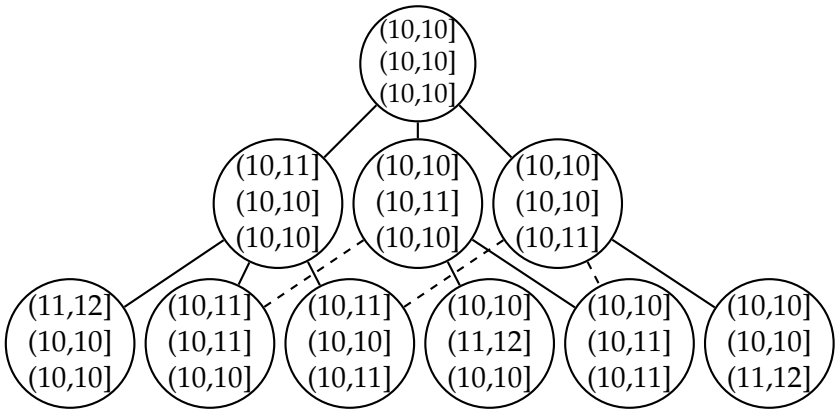


Figure 4.3.: Reformulated ICT with cost interval vectors

4.3.2. Sufficient Conditions for Optimality and Completeness:

In unit-cost domains, the first feasible solution found in the high-level is guaranteed to be optimal. This is not necessarily the case in non-unit cost domains. If the low-level reports that $\mathbf{I}$ in level ℓ of the ICT has a feasible solution of cost C, there are two possibilities:

1. $C{=}h(\mathbf{I})$: $\mathbf{I}$ is a goal (line 11). Because OPEN is ordered by $h(\mathbf{I})$, there can be no other node in the OPEN list that contains a better solution. Therefore, in this case optimality is guaranteed.

2. $C > h(\mathbf{I})$: $\mathbf{I}$ may not contain an optimal solution. We set $incumbent \leftarrow C$ (line 14), and then continue the search until a new ICT node with a lower cost is found, in which case *incumbent* is updated again, or $h(\mathbf{I}) \geq incumbent$ (line 7), at which point we are guaranteed that *incumbent* is the best cost.

Continuing the high level search until $C{=}h(\mathbf{I})$, or $h(\mathbf{I}) \geq incumbent$, may cause up to k-1 additional levels past ℓ to be searched to ensure optimality in the worst case. For instance, if $\delta{=}1{+}\epsilon$ and the difference between C^* and C is 1, theoretically, the cost of one agent could be decreased by 1 and then each agent's cost could be increased by $1/k$, to result in a cost equal to C^*. This situation would require an additional $k{-}1$

additional levels to be generated before the upper cost bound of each of the $k-1$ agents could be increased to allow the $1/k$ additional cost for each agent.

This reformulated algorithm is guaranteed to be complete if a solution to the problem exists. However, observation 4.2.4 also applies to the subset of General MAPF, MAPF_Q. Therefore, we conjecture that ICTS can be made complete with temporally-relative duplicate pruning in the case no solution exists as well.

4.3.3. Reformulated Low Level Search

The low-level determines the best cost of a feasible solution for $\mathbf{I}$ if one exists. First, the low-level builds MDD_i for each agent from $start_i$ to $goal_i$ respectively. This can be done using a depth-first or breadth-first search. In this process, the lowest cost path in the interval $(lb_i, ub_i]$ is saved as $best_i$ for use in the heuristic function $h(\mathbf{I})$.

In order to find the lowest cost solution, a search of the joint k-MDD space (Algorithm 4.2) is performed. The root node of the low-level, $\mathbf{I}_{root}=\{MDD_1^{root}, ..., MDD_k^{root}\}$, is a joint state containing the root nodes from MDD_1,...,MDD_k. $\mathbf{S}$, the set of joint state successors of S (line 9) are generated using joint branching as defined in Algorithm 2.2. A feasible solution is found when a joint state is visited such that all $s_i \in S{=}goal_i$ (line 5).

The low-level continues until one of the following occurs: (1) the search is exhausted or (2) a solution that is optimal in the joint-MDD space is found based on $h(\mathbf{I})$ (line 17). If no solution is found, ∞ is returned. If a feasible solution is found with $C > h(\mathbf{I})$ (line 6) it is saved as the new *incumbent*, but it is not necessarily optimal and the low-level must continue. If the low-level only finds a feasible solution with $C > h(\mathbf{I})$, more ICT nodes may need to be explored at the high-level to ensure optimality.

Algorithm 4.2. reformulated ICTS Low Level Search

```
JOINTDFS
Input: S: A joint state, incumbent: Best cost so far
C ← Σ_{i=1}^{k} c(s_i ∈ S)                          ▷ Get sum of costs
if incumbent < C then                                ▷ Not a better solution
    return incumbent
if All agents are at their goal then
    incumbent ← C                                    ▷ Mark as best so far
    return incumbent
S ← JOINTEXPANSION(S)                                ▷ Expand S
for S' ∈ S do
    C ← JOINTDFS(S', incumbent)
    if C < incumbent then
        if satisficing then                          ▷ Return first solution...
            return C
        if C = h(I) then                             ▷ Return optimal solution
            return C
        incumbent ← C
return incumbent
```

4.3.4. Pruning Enhancements

The SPP and EPP enhancements are still valid for General MAPF with no significant changes. Our empirical results include the SPP enhancement.

4.4. Sub-Optimal Variants

Previous attempts in formulating a bounded sub-optimal variant of ICTS for Classic MAPF yielded mixed results [3]. These variants increase the individual cost of *all* agents at each level in the ICT, effectively increasing the sum-of-costs by k at each level. The optimal formulation of ICTS was faster on problem instances with low agent density (e.g., a large map with few agents), but the sub-optimal variant was faster on problem instances with high agent density. For General MAPF, it is possible to obtain a much tighter cost bound, and the effectiveness is not adversely affected by the agent density of the problem instance.

4.4.1. ϵ-ICTS

Instead of searching the k-MDD space for an optimal solution, one can treat the low-level as a satisficing search and exit upon finding the first feasible solution (See algorithm 4.2 line 12). Assuming a solution with $C \geq h(\mathbf{I})$ is found at the low-level, an immediate exit may result in a significant time savings in the low-level as well as a significant pruning of ICT nodes in the high-level (up to $k-1$ *levels* of search may be avoided in the best case).

For flowtime, the cost of each single-agent path is at most δ greater than optimal, thus the overall bound on sub-optimality is guaranteed to be no greater than $\epsilon = k\delta$. Therefore a specific ϵ can be achieved by setting δ, though very large or small values may negatively impact performance. Additionally, a more precise bound on sub-optimality is returnable as $C - h(\mathbf{I})$, the difference between the actual cost and the lower bound.

4.4.2. w-ICTS

In order to obtain a weighted bound on sub-optimality, as the sub-optimality bound for ϵ-ICTS is $k\delta$, δ can be adjusted on the fly for each ICT node to guarantee a weighted bound $w > 1$. We initialize the root in the same way as in the optimal algorithm, then for the generation of each ICT node $\mathbf{I}'$ thereafter, set $\delta = (w-1)h(\mathbf{I})/k$.

4.5. Theoretical Analysis

The time complexity of the high level search is a combination of three factors: (1) The size of the MDDs used at the low-level and the search space required to build the MDDs; (2) The computational complexity of the low level search; (3) The computational complexity of the high level search.

4.5.1. MDD Size

When building an MDD for agent i at the low-level, a *cost-limit* parameter ub_i is supplied, yielding $MDD_i{=}(V_i, E_i)$, a time-extended directed acyclic graph for all paths from $start_i$ to $goal_i$ with cost $\leq ub_i$. ub_i is incremented by δ as the high level search proceeds, causing the size of MDD_i to increase. Eventually, V_i will span every vertex in V and after that point, $|V_i|$ will only increase by $|V|$ with each increase of ub_i. Hence in unit-cost domains, the change in $|V_i|$ between ub_i and $ub_i + \delta$ is bounded by $|V|$. For example, on a 5x5 grid, the change in $|V_i|$ as ub_i increases by 1 will never be greater than 25.

In non-unit time step domains, assuming that there are at least two discrete action durations allowed (e.g. 1 and $\sqrt{2}$ as in 8-connected grids), and at least one of the action durations shares no common denominator with the others, the increase in $|V_i|$ is upper-bounded by $|V|/r$ where r is the resolution of cost. Continuing our example with the 5x5 grid, with $r{=}10^{-3}$, the change in $|V_i|$ can be no greater than 25,000. Although the rate of MDD growth is still linearly bounded, there is a much steeper growth. The number of operations required to build the MDD is, in the worst case, linear in $|V|$, d and r. If a very fine resolution is supplied for r, e.g., IEEE floating-point precision, optimal ICTS may spend a lot of time to save a very small amount of cost. Fortunately, a coarse setting of r may be feasible for many applications.

4.5.2. Low Level Search Complexity

Because MDDs contain only ub_i-bounded paths the average branching factor for MDDs, b_{mdd}, is typically much smaller than b_{base}. Our experiments showed an average OD-style branching factor of only 1.54 at the low-level on 8x8, 8-connected grids with 10 agents. Recall from Section 2.4.2 and Algorithm 2.2 that operator decomposition repeats the states of some agents in a joint-state expansion to achieve a lower branching factor. With OD-style branching, the depth of the low level search is $O(dk)$ where d is the max depth of all MDDs, resulting in a complexity of $O((b_{mdd})^{dk})$.

4.5.3. High Level Search Complexity

The number of nodes at level ℓ of the ICT (with duplicates removed) is the same as the number of terms in a multinomial coefficient [123], the number of ways of adding k positive integers that add up to ℓ. Hence the size of the ICT is: $\sum_{\ell=0}^{\Delta} \binom{\ell+k-1}{k-1}$ $= \frac{(k+\Delta)!}{k!\Delta!} = O(\text{MIN}(\Delta^k, k^\Delta))$. Assuming the number of agents is fixed, this is Δ^k. When a candidate solution is found at depth Δ, an additional $k-1$ levels of the ICT may need to be explored in the worst case to prove optimality. Therefore the overall complexity of ICTS is: $O((b_{mdd})^{dk}(\Delta+k-1)^k)$.

4.5.4. Sub-Optimal Variants

Let $\ell \leq \Delta$ be the shallowest level of a feasible solution in the ICT. In the best case, the k deepest levels in the ICT tree may be pruned, including all remaining nodes in level ℓ plus $k-1$ levels past ℓ. Let the average MDD depth at $\ell-1$ be d_{-1} and at $\ell+k-1$ be d_{k-1}. With OD-style branching, the amount of savings could be up to $(\ell + k-1)^k (b_{mdd})^{kd_{k-1}} - (\ell-1)^k (b_{mdd})^{kd_{-1}}$ which is $O((\ell+k)^k (b_{mdd})^{kd_{k-1}})$.

4.6. Experimental Results and Analysis

All empirical tests were conducted on a machine with 64 Intel Xeon (r) cores at 2.2GHz with 128 GB of RAM. Test sets consist of 100 random instances of the MAPF problem with varying numbers of agents on 4, 8, 16, and 32-connected grid domains also known as 2^k neighborhoods [151] with wait actions allowed. Any instances taking longer than 300 seconds to complete were terminated and marked as a failure.

4.6.1. OD-Style Versus Full Branching

In order to quantify the differences between OD-style and full branching, we configured the planner in two ways: 1) Worst-case simulation, where ID is turned off and the low-level is set not to exit immediately when finding a solution, but to search

the entire joint-MDD space; and 2) average case simulation, where ID is turned on and the low level is allowed to exit as soon as an optimal solution is found. We ran 100, 10-agent tests in 8x8 grids with both branching styles. Table 4.1 displays the mean statistics for various grid connectivity settings. In the worst-case simulation, we see a tradeoff between the number of node generations and collision checks. OD-style branching generates more nodes in non-unit time step domains, but incurs fewer overall collision checks, especially with higher branching factors. This tradeoff suggests that in general, when collision checks are expensive, OD-style branching is preferred and when node generations are expensive, full branching is preferred.

Let $\mathcal{O}$ and $\mathcal{F}$ be the sets of nodes of the search trees created at the low-level by OD-style branching and full branching respectively in the worst-case scenario. OD-style branching only calls $succ(s_i)$ for one $s_i \in S$ (with $s_i.time{=}t_{min}$) in the Cartesian product. Hence $t_{min} \leq t'_{min}$ for all S and S' in $\mathcal{O}$. Because full branching calls $succ(s_i)$ for *all* s_i having minimum time, $t_{min} < t'_{min}$ for all S and S' in $\mathcal{F}$. Hence $\mathcal{F} \subseteq \mathcal{O}$, therefore $|\mathcal{F}| \leq |\mathcal{O}|$. This explains the difference in the number of search nodes in the worst-case as shown in Table 4.1. However, as evidenced by the statistics for the average-case simulation, OD-style branching appears to save a small amount of work.

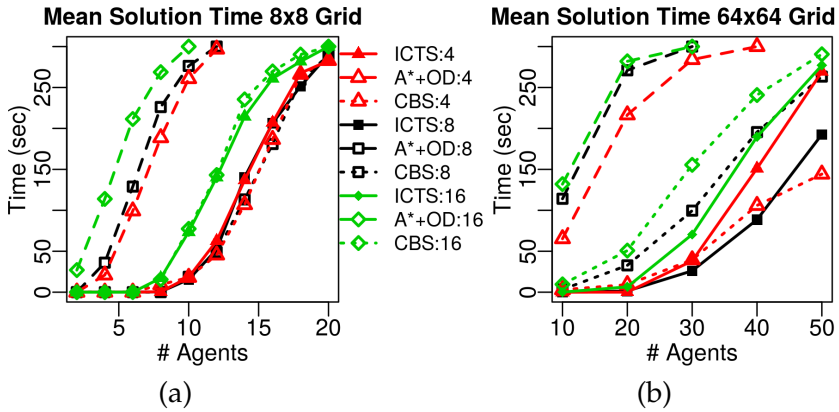
Worst-Case Simulation

	Search Nodes (in thousands)				Collision Checks (in millions)			
Conn.	4	8	16	32	4	8	16	32
OD	**21.6**	3.7	166.3	311.4	**1.18**	**.50**	**80.27**	**198.30**
Full	22.7	**1.6**	**102.3**	**183.5**	1.25	.51	115.68	666.31

Average-Case Simulation

	Search Nodes (in thousands)				Collision Checks (in thousands)			
OD	**.16**	**.91**	9.00	**17.16**	**4.33**	**81.05**	231.15	**371.47**
Full	.24	.96	**7.45**	17.53	4.49	91.31	**223.07**	384.90

Table 4.1.: Comparison of average and worst-case scenario for OD-style versus full branching for 10 agents on 8x8 grids.



(a) (b)

Figure 4.4.: Performance of ICTS versus A* and CBS on 4, 8 and 16-neighbor grids.

4.6.2. Best Setting for δ

We ran tests for 10 agents on 8-neighbor, 8x8 grids for different settings of δ with action costs of 1 and 1.4 for cardinal and diagonal actions respectively. Run times were best with a value of 1 and very similar for all values of δ in the range .4$\sim$1.4 and performance degraded outside of that range. Considering that for 8-neighbor grids, 4 actions are diagonals with a cost of 1.4 and 5 actions have a cost of 1 (4 cardinal actions and 1 wait action), actions with a cost of 1 are slightly more likely. This gives a rule-of-thumb that the best setting for δ is the most commonly occurring action cost. Additionally, with a set M of unique action costs, the "sweet spot" for δ lies in the range $[max(M) - min(M), max(M)]$.

4.6.3. ICTS versus A* and CBS

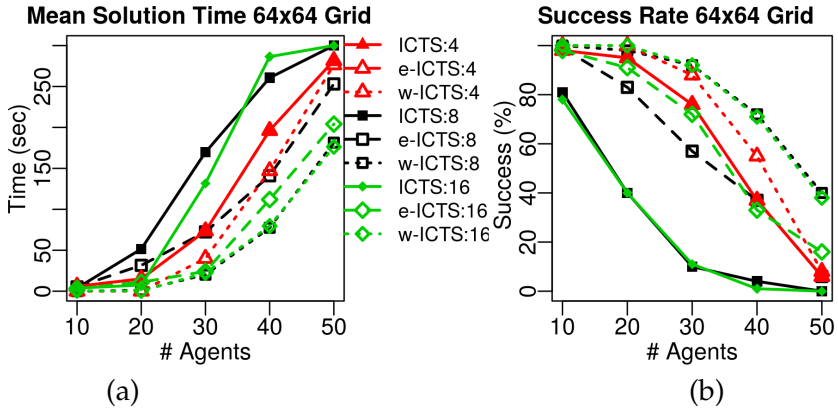
As an initial test, we compared A* and CBS against the reformulated ICTS algorithm. Both A* and ICTS use OD-style branching and the ID framework, solving only conflicting agents jointly. The CBS algorithm is using continuous-time collision detection and the conflict prioritization (PC) enhancement from ICBS [25]. Our initial analysis shows that the conflict avoidance table (CAT) is ineffective and often detri-

mental for the high branching factor domains of General MAPF, hence is not used. All algorithms use a time resolution of r=10^{-3}. The ICTS algorithm is using an increment of δ=1 and the simple pairwise pruning enhancement.

Figure 4.4 shows the mean time to solution (with failure times of 300 sec averaged in) of 100 trials (y-axis) and the number of agents (x-axis). ICTS and CBS clearly dominate A* and have nearly identical run-times in 8x8 grids, but in 64x64 grids ICTS clearly dominates CBS for 8 and 16-connected domains (non-unit costs), but not in the 4-neighbor domain (unit costs). These results may indicate a strength in ICTS for non-unit cost domains, however, this is an area that needs further research.

4.6.4. Sub-optimal variants

We ran both ICTS and the sub-optimal variants on 64x64 grids with δ=1.0, w=1.5, and both r=10^{-3} and r=10^{-6}. The results for the latter setting of r are shown in Figure 4.5. The results for the former setting are not as dramatic, but show the same trend. Figure 4.5(a) displays the mean time to solution with failure times of 300 seconds averaged in (y-axis) and number of agents (x-axis) and Figure 4.5(b) shows the percentage of problems solved in under 300 seconds. Run times for both ϵ-ICTS



(a) (b)

Figure 4.5.: Performance of sub-optimal variants versus ICTS on 4, 8 and 16-neighbor grids.

Performance on 2^k Neighborhoods								
	Cost				Time (sec)			
Conn.	4	8	16	32	4	8	16	32
ICTS	1283	1013	1012	1005	**73**	170	132	181
ϵ-ICTS	1283	1041	1013	1010	73	72	**24**	54
w-ICTS	1284	1051	1051	1020	40	**20**	22	71
Ratio Versus Optimal ICTS, 4-neighbor								
	Cost ratio				Time ratio			
ICTS	1.0	.79	.79	.78	1.0	2.33	1.81	2.48
ϵ-ICTS	1.0	.81	.79	.79	1.0	.99	.33	.74
w-ICTS	1.0	.82	.82	.80	.55	.27	.30	.97

Table 4.2.: ICTS and sub-optimal variant performance for various branching factors planning for 30 agents on a 64x64 grid.

and w-ICTS are better for higher branching factor domains compared to run times in 4-neighbor domains.

Table 4.2 displays partial results from Figure 4.5 for 30 agents. Note that not only is the time to solution in the sub-optimal algorithms lower than for 4-neighbor grids, but the solution quality is better. For example ϵ-ICTS on 16-neighbor grids yields a mean 21% improvement on solution quality *and* a 3× improvement on solution time versus ϵ-ICTS on 4-neighbor grids. This surprising result suggests that higher quality paths can be achieved in less time by using finer discretization in agent actions and using a sub-optimal solver.

4.7. Summary

In this Chapter, we explained the Increasing-Cost Tree Search Algorithm (ICTS). We presented the first proofs for completeness and optimality of ICTS. We introduced *novel extensions for General MAPF* domains. These extensions are general, and allow ICTS to be used with virtually any General MAPF instance. We showed that it has

better performance than multi-agent A* and comparable performance to CBS in many settings.

We introduced two new *bounded sub-optimal variants* of ICTS: ϵ-ICTS and w-ICTS. We showed that these enhancements allow significant performance improvements over optimal ICTS without a significant impact on solution quality.

5. Extensions for Conflict-Based Search

5.1. Conflict-Based Search

Conflict-Based Search (CBS) [160], shown in Algorithm 5.1, is an optimal state-of-the-art algorithm for MAPF. CBS can be decomposed into three abstract pieces:

1. A conflict detection mechanism. E.g., collision detection (See Section 3.2).
2. A way to represent constraints. For example, a constraint for an agent to avoid an action at a specific time (See Section 5.3).
3. A low-level single-agent (or multi-agent) solver capable of planning with constraints. For example, A* [69], Safe Interval Path planning (SIPP) [143] or Jump Point Search with Temporal Obstacles (JPST) [76] (See Section 5.6).

CBS uses a two-level search. The *high level* searches the conflict tree (CT). An example of a CT is shown in Figure 5.1 (b). Each node N in the CT contains a set of k

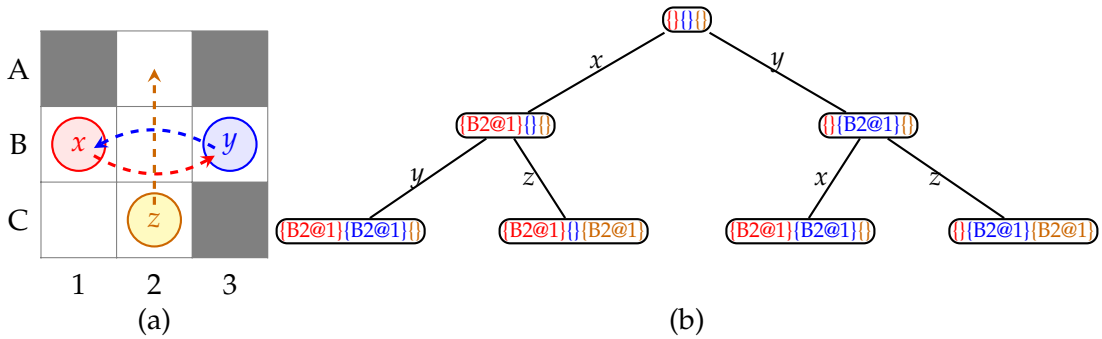


Figure 5.1.: (a) An example Classic MAPF instance and (b) a *partial* CT for the instance.

Algorithm 5.1. Conflict-Based Search

```
1: Input: A MAPF instance
2: Plan N.Π ignoring conflicts, insert N into OPEN
3: while OPEN not empty do
4:     N ← best node from OPEN
5:     if No conflict between all π_i, π_j ∈ N.Π then
6:         return N.Π
7:     else                                  ▷ Perform a split by creating two child nodes
8:         N_1 ← N
9:         N_2 ← N
10:        Add a constraint to N_1 for agent i to avoid conflict
11:        N_1.Π.π_i ←Replan(π_i)
12:        Add a constraint to N_2 for agent j to avoid conflict
13:        N_2.Π.π_j ←Replan(π_j)
14:        Add N_1, N_2 to OPEN
15: return NO SOLUTION
```

paths $N.\Pi = \{\pi_1, ...\pi_k\}$, representing a possible solution. ($N.\Pi$ is not shown in Figure 5.1 (b).) $N.\Pi$ for the root node is constructed using a *low level* search without taking other agents into account (line 2). $N.\Pi$ is checked for conflicts between paths (line 5). A *conflict* is defined minimally as a tuple $\langle a_i, a_j \rangle$ where a_i and a_j are the conflicting actions from π_i and π_j. In Section 5.2 we will provide more details.

If no conflict is found, $N.\Pi$ is a feasible solution and the algorithm terminates (line 6). If a conflict is found, a *split* is performed, meaning two child nodes N_1 and N_2 are generated with constraints c_1 and c_2 respectively. These constraints cause the low-level to avoid the conflict (lines 10, 12). This is shown in Figure 5.1 (b). At the root node, a conflict between agent x and y is detected and two child nodes are generated. Constraint sets are shown in curly brackets with the location and time step. For example, $B2@1$ means the agent cannot occupy position $B2$ in the example instance from Figure 5.1 (a) at time step 1. In Section 5.3 we discuss several choices for representing constraints.

Next, the conflicting agents are re-planned to respect the new constraints and their paths are updated in the child nodes (lines 11, 13). Constraints are added cumulatively as the CT deepens, where child nodes append to the constraint sets of their

parent nodes as shown in Figure 5.1 (b). Eventually, enough constraints are added in order to allow a feasible solution. These CT nodes are placed into an OPEN list (line 14) which is prioritized by flowtime. The search terminates when a feasible solution is found or when the OPEN list is empty.

In the next sections, we discuss details of the three pieces of CBS – the low-level, conflicts and conflict detection, and constraint representation with novel techniques for constraint generation. Then we discuss areas of research regarding the high-level search itself. This is followed by novel contributions on sub-optimal search and theoretical analysis.

5.2. Conflicts

The notion of a conflict was explained in detail in Section 3.2. Essentially, a conflict occurs when two agents' shapes or volumes overlap in the same time time interval (also known as a collision). It is possible to change or extend the notion of conflicts for CBS. For the scope of this thesis, we do not explore alternative definitions of conflict other than collisions, except to note that it has been done successfully. For example, in k-robust MAPF [12], the notion of a conflict for CBS is extended in time, so that agents' shapes must not occupy the same space for at least k actions.

5.2.1. Conflict Types

The concept of a conflict has meaning at the individual *action level* in the sense that two actions may conflict. It also has meaning at the *path level* in the sense that two paths may have a series of action-level conflicts based on local obstacles and agents' relative path orientation.

Section 3.1 enumerated the possible types of action-level conflicts, namely: vertex conflict, edge conflict, intersecting-edge conflict, non-intersecting edge conflict and edge-vertex conflict. The former two are only applicable in Classic MAPF. Figure 3.1 shows examples of these basic conflict types.

A *Cardinal conflict* is a path-level conflict in which neither agent can reach their goal unless at least one agent increases its path cost (e.g., by adding a wait action or by taking a different, longer route to the goal). For example, in the instance in Figure 5.1(a), *either* agent x must increase its cost to avoid conflicting with agent z in cell B2 at time step 1 or vice-versa in order for both agents to arrive at their respective goals without conflict.

A *semi-Cardinal conflict* is one in which, to avoid a conflict, a specific agent must increase its path cost, but the other agent in the conflict must not. For example, in Figure 5.1(a), if agent x increases its cost by 2 (by moving through cell C1), agent y may proceed to its goal with no cost increase. The converse is not true.

A *non-Cardinal conflict* is one in which neither agent must increase its path cost in order to avoid conflict.

An improved version of CBS (ICBS) [25] recognizes the distinction between path-level conflicts. ICBS prioritizes Cardinal conflicts to be resolved first, and in the case of non- or semi-Cardinal conflicts, it attempts to "fix up" $N.\Pi$ to avoid the conflict by computing a *bypass* path for one or possibly both agents. If the total number of conflicts in the new fixed up $N.\Pi$ is less than the original, the conflict is successfully bypassed and the fixed up node is re-inserted into OPEN. These two improvements, namely *conflict prioritization* and *bypass* resulted in significant performance gains [25].

5.2.2. Conflict Symmetries

Cardinal conflicts may involve symmetries. In Classic MAPF, these symmetries are classified into at least 3 types: Rectangle conflicts [113], corridor conflicts [111] and swapping conflicts [111], however other types of conflict symmetries may also exist. An example of each is shown in Figure 5.2.

A conflict symmetry occurs when all paths for all infeasible cost combinations (for example, the individual lowest-cost paths) for two agents conflict when either of them move through a region called a *region of conflict*. Figure 5.2 (a) illustrates a *rectan-*

gle conflict [113] in a 4-neighbor grid map where the red and blue agent (shown as filled circles) try to move to their respective goals (shown with dashed lines). The shaded region in the center is the region of conflict. No matter which lowest-cost path the agents take through the region, they will always conflict. Figure 5.2(b) illustrates a *corridor conflict* in a 4-neighbor grid. Figure 5.2(c) shows a swapping conflict. The shaded region shows the region of conflict - not only can the agents not move through these regions with their lowest-cost individual paths, but also multiple higher cost paths.

Conflict symmetries in General MAPF have not yet been classified into specific types, but can be broadly placed into two categories: *spatial conflict symmetries* and *temporal conflict symmetries*. Spatial conflict symmetries are similar to the rectangle conflict in the sense that all paths of lowest cost in a spatial region between two agents will conflict. Temporal conflict symmetries are similar to corridor and swapping conflicts in the sense that two agents will incur the same conflict or set of conflicts over and over at increasing time steps until one of the agents increases its cost sufficiently (for example, by waiting) and allowing the other agent to pass. While spatial symmetries require agents to explore many alternate equal-cost paths, temporal conflict symmetries require the agents to increase costs multiple times.

It has been shown that conflict symmetries in general require an exponential amount of work to resolve in CBS [113]. Detection and resolution of conflict symme-

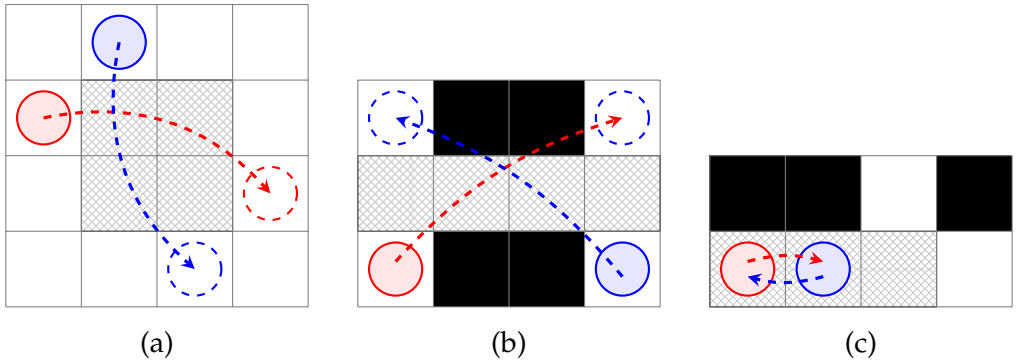


Figure 5.2.: Illustration of Classic MAPF instances with conflict symmetries: (a) a rectangle conflict, (b) a corridor conflict and (c) a swapping conflict.

tries in both Classic and General MAPF settings has been a subject of study [113, 111, 112, 218, 199, 200]. In the subsequent sections, the details of these studies are explained and expanded upon.

5.3. Constraints

In CBS, the detection of a conflict triggers the creation of constraints and a split in the CT (see Algorithm 5.1 lines 10-13). Constraints are then used at the low-level to constrain agents to avoid conflicts. Minimally, a constraint must include enough information about the conflicting action so that the low level search can avoid reproducing the conflicting action during re-planning; for example, by restricting constrained actions or states from being generated during expansion in the low-level. Various approaches to constraint formulation and constraint generation have been studied.

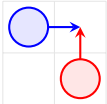
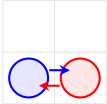
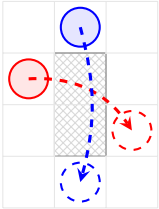
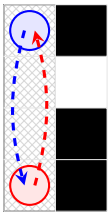
5.3.1. Basic Constraint Types

In Classic MAPF settings, two constraint types are used, namely *vertex constraints* and *edge constraints*. A vertex constraint is a tuple $\langle i, s \rangle$ where i is the agent number and s is a state (vertex and time tuple). An edge constraint is a tuple $\langle i, a \rangle$ where i is the agent number and a is an action (a tuple of two states).

Note that vertex constraints are not strictly necessary because an edge constraint can also be used to prevent an agent from entering a vertex at a specific time. However, vertex constraints are more powerful, because they may block agents from moving to the blocked state via any adjacent edge. Hence, vertex constraints are typically used in response to vertex conflicts and edge constraints are used in response to edge conflicts. We also note that using a vertex constraint in response to an edge conflict can result in blocking optimal solutions, or even in incompleteness of the overall algorithm. See Table 5.1 for an overview of Classic MAPF conflicts and applicable constraints.

[1]See section 5.3.3

Table 5.1.: A summary of conflict types and applicable constraints for Classic MAPF

Non-Cardinal Conflicts		Vertex Constraint	Edge Constraint		
Vertex Conflict		✓ ✓	✓ ✗	correct efficient	
Edge Conflict		✗	✓ ✓	correct efficient	
Cardinal Conflicts		**Vertex & Edge Constraints**	**Context-Specific Constraint[1]**	**Mutex-Propagation Constraint**	
Spatial Conflict Symmetry		✓ ✗	✓ ✓	✓ ✓✓	correct efficient
Temporal Conflict Symmetry		✓ ✗	✓ ✓✓	✓ ✓	correct efficient

In General MAPF settings, both vertex and edge constraints are valid, however, due to the durative nature of conflicts, *time range constraints* [12] are more effective [37, 6]. A time range constraint is a tuple $\langle i, e, [t, t'] \rangle$ where i is the agent number, $e \in E$ is an edge and $[t, t']$ is the unsafe interval for the durative conflict (see Section 3.4.1). A time range constraint blocks agent i from traversing e at any time during the unsafe interval. In contrast to an edge constraint which blocks an agent from edge traversal at a single point in time, time range constraints are more powerful because they can potentially block multiple actions and prevent more conflicts from occurring in the sub-tree of the high-level search. A time range constraint can also be applied to a vertex instead of an edge. See Table 5.2 for an overview of General MAPF conflicts and applicable constraints.

These constraint types are the basic building blocks for building sophisticated constraint sets for CBS in Classic and General MAPF.

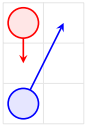
5.3.2. Constraint Sets and Constraint Correctness

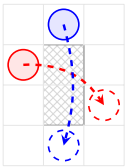
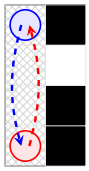
In order to ensure completeness and optimality, constraints used in a split must be *mutually disjunctive* [12, 110]. That is, no pair of conflict-free paths for agents i and j involved in a split can violate both c_i and c_j simultaneously. A simple way to check whether c_i and c_j are mutually disjunctive, is to compute A_i and A_j, the sets of actions blocked by c_i and c_j respectively, and check that the sets are *mutually conflicting*. Two sets of actions A_i , A_j are mutually conflicting if for all pairs of actions (a_i, a_j) in the Cartesian product $A_i \times A_j$, a_i conflicts with a_j.

Lemma 5.3.1. *CBS is complete if constraints added to child nodes N_i, N_j created from a CBS split only block action sets A_i, A_j which are mutually conflicting.*

Proof. Let N_i and N_j be the child nodes created during a CBS split. Also let A_i and A_j be the set of actions which are blocked by the constraints from N_i and N_j respectively. Additionally, assume there is only one feasible solution for the problem instance, Π^* which contains paths $\pi_i^* \in \Pi^*$ and $\pi_j^* \in \Pi^*$. By contradiction, if $\exists (a_i, a_j) \in A_i \times A_j$ such

Table 5.2.: A summary of conflict types and applicable constraints for General MAPF

Non-Cardinal Conflicts		Vertex Constraint	Edge Constraint	Time Range Constraint	Biclique Constraint	Time-Annotated Biclique Constraint	
Vertex Conflict		✓ ✓	✓ ✗	✓ ✓✓	✓ ✓✓	✓ ✓✓✓	correct efficient
Edge Conflict		✗	✓ ✗	✓ ✓✓	✓ ✓✓	✓ ✓✓✓	correct efficient
Intersecting Edge Conflict		✗	✓ ✗	✓ ✓	✓ ✓✓	✓ ✓✓✓	correct efficient
Non-Intersecting Edge Conflict		✗	✓ ✗	✓ ✓	✓ ✓✓	✓ ✓✓✓	correct efficient
Edge-Vertex Conflict		✗	✓ ✗	✓ ✓	✓ ✓✓	✓ ✓✓✓	correct efficient

Cardinal Conflicts		Vertex & Edge Constraints	Time Range Constraint	Biclique Constraint	Time-Annotated Biclique Constraint	Mutex-Propagation Constraint	
Spatial Conflict Symmetry		✓ ✗	✓ ✗	✓ ✓	✓ ✓✓	✓ ✓✓	correct efficient
Temporal Conflict Symmetry		✓ ✗	✓ ✗	✓ ✗	✓ ✗	✓ ✓	correct efficient

that a_i does not conflict with a_j and $a_i \in \pi_i^*$ or $a_j \in \pi_j^*$, then CBS cannot find a goal solution of Π^* in N_i or N_j nor their respective sub-trees. This is because in N_i and N_j and their sub-trees a_i or a_j are blocked by constraints and no solution exists that does not contain a_i or a_j. Thus CBS will either terminate with no solution or fail to terminate with the solution. Hence, unless constraint sets block action sets A_i, A_j which are mutually conflicting, we cannot guarantee completeness.

Conversely, if constraint sets used in child nodes of a split are mutually disjunctive, meaning that their blocked action sets are mutually conflicting, then CBS will never block actions in both N_i and N_j simultaneously which are part of a feasible solution. This is because, assuming A_i and A_j are mutually conflicting, no pair of actions in their Cartesian product can ever exist simultaneously in a feasible solution. Hence, a feasible solution must lie in one of N_i or N_j or their respective sub-trees and CBS will eventually find it. □

Following the mutually disjunctive property, it is evident that using a pair of vertex constraints in response to an edge conflict may block other valid paths through the vertex in question (though, in some cases it might not), hence the pair of constraints is not necessarily mutually disjunctive in this case. This is indicated in Table 5.1.

Instead of using exactly one constraint per child node in a split, it is also valid to use sets of constraints, C_i, C_j, provided that the sets of actions blocked by the constraints are mutually conflicting. This variant is called multi-constraint CBS (MC-CBS) [110]. The proof of optimality for MC-CBS is the same as for the original CBS [160]. In short, optimality is guaranteed if no optimal solutions are blocked by constraints (i.e., constraints are mutually disjunctive) and both the high and low level OPEN lists are prioritized by flowtime.

The motivation for blocking multiple actions with each split is clear – by blocking multiple actions in one node, we resolve potentially multiple conflicts and avoid additional splits in the sub-tree. An example problem is shown in Figure 5.3 (a) where the goal location for agent i is the start location for agent j and vice versa. The move-

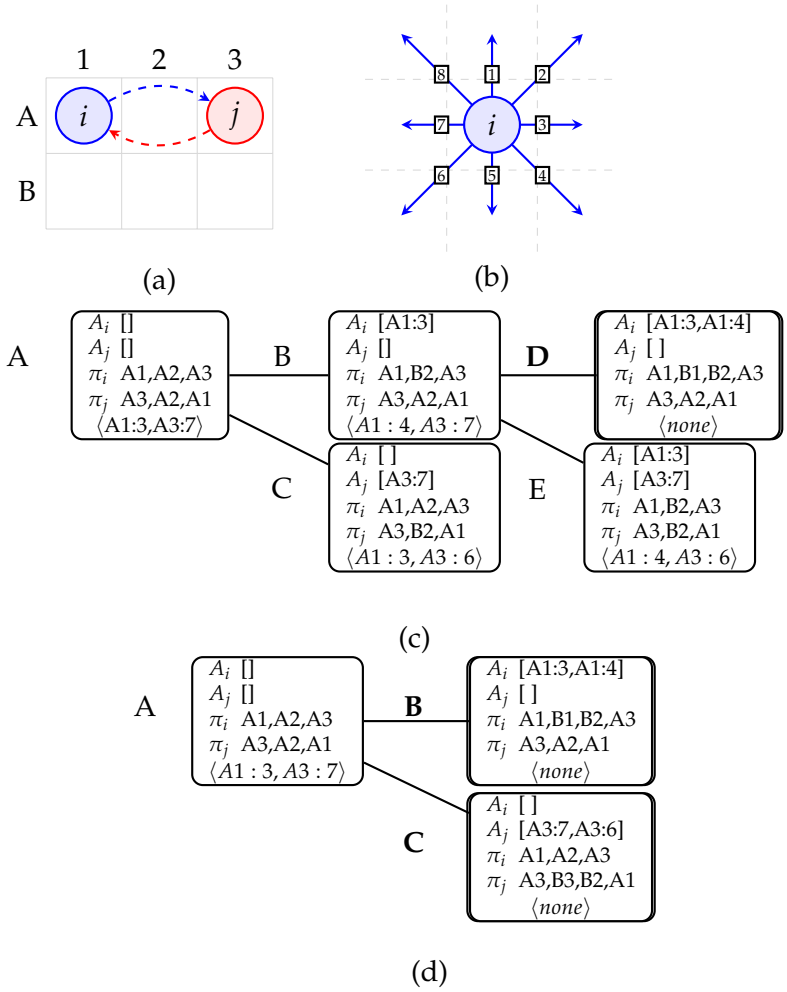


Figure 5.3.: Example showing (a) a simple MAPF problem for circular agents i and j, (b) enumerated actions for each agent, (c) a possible CT for the problem and (d) a possible CT where multiple actions are blocked at a time.

ment model is based on an 8-connected grid as shown in (b) where each action is enumerated for reference. A CT for the problem is shown in (c). Each node contains:

- A_i, A_j, sets of blocked actions. For example, node **B** shows A_i $[A1:3]$ meaning that action 3 is blocked for agent i at location A1
- A potential solution, which is a pair of paths π_i and π_j
- A conflict $\langle a_i, a_j\rangle$, which is a pair of actions that conflict.

For example, node **A** shows the conflict $\langle A1 : 3, A3 : 7\rangle$ because agent i, performing action 3 at A1 and agent j, performing action 7 at A3 results in a conflict. When a conflict is detected, a split operation is performed. For example, the conflict $\langle A1 : 3, A3 : 7\rangle$ in node **A** causes nodes **B** and **C** to be generated with constraints that block action A1:3 and A3:7 respectively. This process continues until a non-conflicting set of paths is found in **G**.

Diagram (d) shows a possible CT that can occur if multiple actions are blocked per CT node. Observe that when multiple actions are blocked ($A_i : [A1 : 3, A1 : 4]$) as shown in node **B**, after the first split a goal node is found immediately instead of at depth 3 as in part (c). This observation leads us to a few simple corollaries: (1) blocking multiple actions during a split can result helpful in pruning of the CT, often reducing Δ, the depth of the goal. (2) The size of the CT is $O(2^{\Delta})$; hence the number of nodes pruned can be exponential in the best case, and (3) the pruning potential is directly proportional to the cardinality of the sets A_i, A_j; hence as $|A_i|$ and $|A_j|$ increase, Δ is likely to decrease.

Based on these observations, we wish to maximize the number of blocked actions in a split, while maintaining the mutually disjunctive property. We now review some prior and novel approaches to generating large mutually disjunctive constraint sets.

5.3.3. Context-Specific Symmetry Breaking in Classic MAPF

The characteristics of 4-connected grids allow some context-specific approaches to conflict symmetry breaking [112]. In response to a rectangle conflict, shown in Figure 5.2(a), a technique called *rectangle reasoning* [113] can be applied to recognize rectangle conflicts and generate mutually disjunctive constraint sets called *barrier constraints* which help resolve the conflict immediately. An example of a rectangle conflict and barrier constraints is shown in Figure 5.4. Rectangle reasoning discovers the region of conflict (shaded area with hatched lines) and generates the barrier constraint which is

a set of constraints that occupies the row or column of the region of conflict opposite the agent. These individual constraints block the agent from entering the cell at the specific time necessary to resolve the conflict.

Corridor and swapping (or target) conflicts are also solved in a context-specific way by identifying the conflict type and the region of conflict, then applying constraint sets to the conflicting agents in order to resolve the conflict [111].

These techniques are very effective in resolving specific types of conflict symmetries, reducing the amount of work required by CBS exponentially. However, they are not always applicable to General MAPF instances.

5.4. Time-Annotated Biclique Constraints

Biclique constraints are a primary contribution of this thesis. Biclique constraints, which are really sets of constraints C_i and C_j, are created via a process called bipartite reduction (BR) for use with multi-constraint CBS (MC-CBS) [110]. Because $|A|$, the size of the set of actions blocked by C is positively correlated with pruning at the high-level, it is beneficial to maximize $|A|$. BR heuristically maximizes $|A|$ while maintaining the mutually disjunctive property. In contrast to context-specific symme-

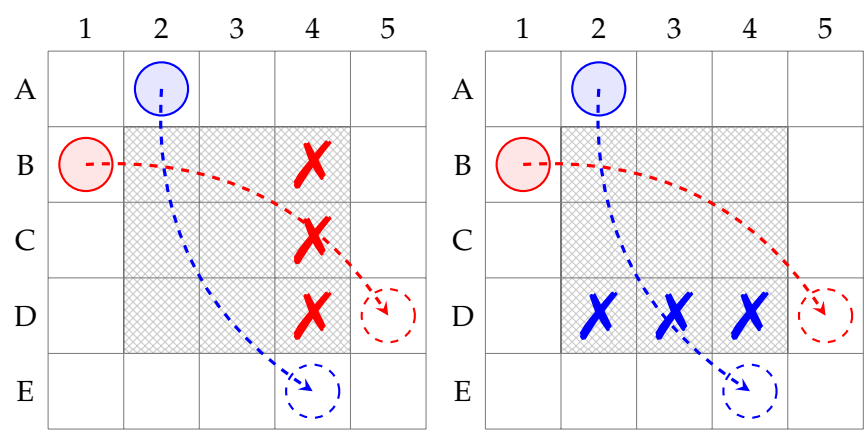
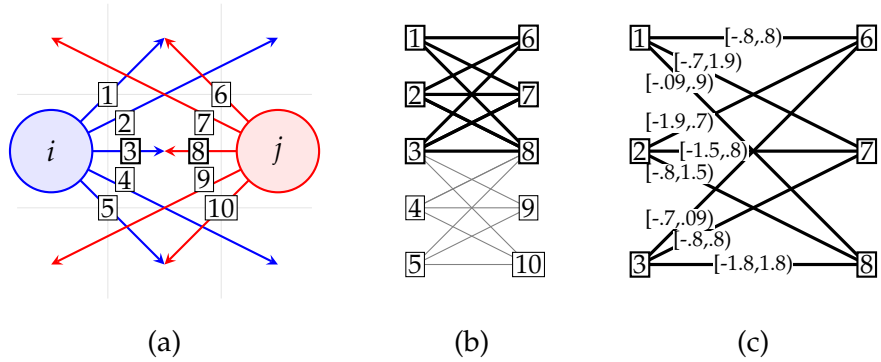


Figure 5.4.: Illustration of rectangle reasoning and barrier constraints



(a) (b) (c)

Figure 5.5.: Illustration of (a) actions for two agents, (b) the corresponding bipartite conflict graph and (c) the corresponding time-annotated biclique.

try breaking approaches, BR is general and may be used without any special analysis of the planning graph nor relative motion of agents. It is also applicable to both discretized and continuous environments.

5.4.1. Ensuring Completeness and Optimality

As stated earlier, for completeness, constraint sets C_i and C_j which are created during a split, must be *mutually disjunctive*, which means the sets of actions blocked by C_i and C_j must be mutually conflicting.

For example, in Figure 5.5(a) action 1 conflicts with 6, 7 and 8; action 2 conflicts with 6, 7 and 8 and action 3 conflicts with 6, 7, 8, 9 and 10. Thus the action sets $\{1,2,3\}, \{6,7,8\}$ are mutually conflicting. Although actions 9 and 10 also conflict with action 3, they cannot be included because they do not conflict with actions 1 and 2. BR reduces the problem of constructing mutually conflicting constraint sets to finding a biclique in a bipartite graph.

5.4.2. Reduction to Bipartite Graphs

The conflicts between a pair of action sets A_i and A_j, (shown as arrows in Figure 5.5(a)), can be represented as a *bipartite conflict graph* (BCG), shown in Figure 5.5(b). A BCG, $G = (U, V, E)$, has two sets of vertices U and V such that each $u \in U$ represents

an action $a_i \in A_i$ and each $v \in V$ represents an action $a_j \in A_j$. E consists of the subset of vertex pairs $(u, v) \in U \times V$ for which the corresponding actions $(a_i, a_j) \in A_i \times A_j$ conflict.

For CBS, it is sufficient to construct a BCG only for the subset of actions which conflict with the *core action pair* which is the actions from the conflict $\langle a_i, a_j \rangle$ that caused a split. In Figure 5.5(a), the core action pair is $\langle 3, 8 \rangle$, hence only actions which conflict with 3 or 8 are depicted. In this setting, each vertex is guaranteed to be connected to the opposing agent's core action in the BCG.

Although Figure 5.5 shows a BCG construction based only on actions from the start states of the core action pair, in practice, a BCG can include all actions from all states that conflict with an opposing agent's core action. However, it may not be computationally efficient to do so.

5.4.3. Constraint Set Construction Using Bicliques

A *biclique* $G' = (U', V', E') \subseteq G$ is a fully bi-connected bipartite graph, that is, $E' = U' \times V'$, meaning all $u \in U'$ are connected via an edge to all $v \in V'$. A BCG may have many bicliques. In order to maximize pruning in the CT, we find a *max-vertex biclique* (MVB) in G which is a biclique with a maximal number of vertices. This can be done in polynomial time [58]. Algorithm 5.2, lines 1-5 shows pseudocode for computing a MVB. Because G' is fully bi-connected, U' and V' represent the mutually conflicting action sets suitable for a split, and edge constraints could be used to block these actions. In CBS, edge constraints are only for a single time t. However, given a MVB, unsafe intervals can be computed and time-range constraints can be used (see Section 5.3.1).

After extracting G' from G, G'_t, a *time-annotated biclique* (TAB) is constructed (Algorithm 5.2, line 6) by annotating each edge $e' \in E'$ with its unsafe interval (see Section 3.4.1). An example of a TAB is shown in Figure 5.5(c). Finally each vertex in U'_t, V'_t is annotated with an interval that is *fully included* by the annotated intervals for each $e \in E'_t$ incident to it (line 7). An interval $tr_i = [t_i^{start}, t_i^{end})$ *fully includes* another interval $tr_j = [t_j^{start}, t_j^{end})$ if $tr_i^{start} \leq tr_j^{start}$ and $tr_i^{end} \geq tr_j^{end}$. In set notation, this is $tr_j \subseteq tr_i$.

Algorithm 5.2. Compute Largest Time-Annotated Biclique

1. INPUT: A bipartite graph $G = (U, V, E)$
2. Construct $\overline{G}$, the bipartite complement of G
3. Find M, a maximal matching in $\overline{G}$
4. Construct K, a minimum vertex cover of $\overline{G}$ from M
5. Take the bipartite complement of K to get $G' \subseteq G$, a max-vertex biclique
6. Annotate all edges $e \in E' \in G'$ with computed unsafe intervals to create a time-annotated biclique G'_t:

   For each $e \in E'$, $E'_t \leftarrow E'_t \cup (e, \text{UNSAFEINTERVAL}(e.u, e.v))$
7. Annotate all vertices U'_t, V'_t with the intersection of all unsafe intervals of incident edges:

   For each $u \in U'_t$, $u \leftarrow (u, \bigcap_{e \in \text{INCIDENT}(u)} e.intvl)$; analogously for V'_t
8. return (U'_t, V'_t)

A time interval tr_i is fully included by a set of time ranges T if $tr_i \subseteq \bigcap_{tr_j \in T} tr_j$. This relation is illustrated in Figure 5.6 (b): the interval in blue is fully included by all other intervals. Figure 5.6(a) illustrates the annotation of a vertex in a TAB. The blue time interval annotation on vertex 1 is the intersection of all intervals annotated on its adjacent edges as shown by the blue interval in part (b). Thus, the result of Algorithm 5.2 is a TAB where each vertex is annotated with an unsafe interval which is fully included by the intervals of its incident edges.

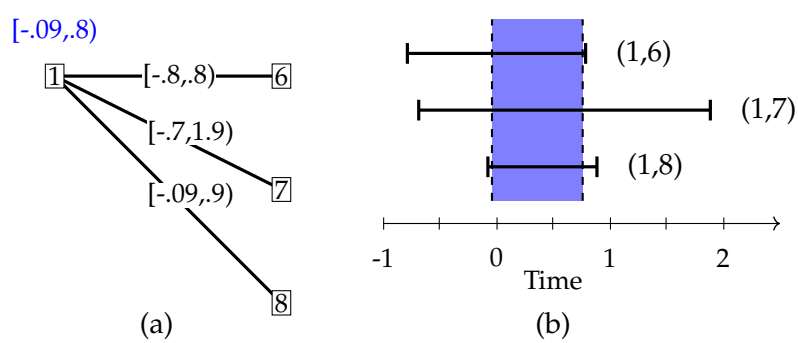


Figure 5.6.: Example of (a) a TAB with (b) corresponding unsafe intervals plotted on a concurrent timeline.

Thus, for a split we first build the relevant TAB. Then for the left node we add the set of time-range constraints C_i that includes $\langle u, tr = [t_{start}, t_{end})\rangle$ for each $u \in U'_t$ where $[t_{start}, t_{end})$ is the unsafe interval associated with it. This is then done analogously for the right node using V'_t. This approach yields constraint sets that guarantee completeness. We call CBS with time annotated biclique constraints CBS+TAB.

Theorem 5.4.1. *CBS+TAB is complete.*

Proof. First, the action sets $U', V' \in G'$ are guaranteed to be mutually conflicting because G' is a biclique. Second, since the annotated unsafe interval for each vertex $u_t \in U'_t$ and $v_t \in V'_t$ is the intersection of all unsafe intervals of incident edges $\in E'_t$, all time range constraints $c_i \in C_i$ and $c_j \in C_j$ constructed from those intervals are guaranteed to block only actions that conflict. Hence, C_i, C_j are mutually disjunctive. Thus, per Lemma 5.3.1 completeness is guaranteed. □

5.4.4. Additional Variants

We define two additional variants which utilize BR: **CBS+MVB** omits the time annotation step and uses U' and V' from the MVB to create edge constraints instead of time range constraints. This variant may be required for some domains in which computing unsafe intervals is not possible or too expensive. **CBS+TMA** (for time-annotated max-biclique approximation), approximates a TAB by assuming that the MVB is a $1 \times N$ biclique, that is, $|U'| = 1$ and $|V'| = N$. For example, using the sets {3}, {6,7,8,9,10} from Figure 5.5. However, instead of explicitly blocking each action in U'_t and V'_t, the TAB is represented *implicitly*, using only two constraints, one time range constraint c_i for agent i that blocks a_i (this is U'_t), and another constraint c_j for agent j that blocks all actions that conflict with a_i (this is V'_t). c_j in this case is implemented such that it performs a collision check versus a_i during low-level expansions. With this representation, c_i and c_j can be created without explicitly constructing a BCG.

5.4.5. Empirical Results

We experiment with CBS+TAB, CBS+MVB and CBS+TMA. CBS+TAB and CBS+MVB use TABs that were computed a-priori and saved in a lookup table. We also experiment with Extended-ICTS [198] (denoted *ICTS*), CBS with edge and vertex constraints [160] (denoted *Classic*) and CBS with time-range constraints [12] – based on CCBS [6] and ECBS-CT [37] (denoted *Time*). Our implementation uses A* with a fixed duration of 1 for wait actions at the low level instead of SIPP. Hence, we do not run CCBS and ECBS-CT, but perform a direct comparison of the effectiveness of the time-range constraints which they use.

Tables 5.3, 5.4 and 5.5 show results for 8-, 16- and 32-neighbor grids on the MAPF benchmarks [173] which consists of 25 tests on each of 28 grid-based maps of various types. Each test consists of up to 1,000 problem instances with increasing numbers of agents. Tests were run by incrementally adding one agent at a time until it becomes unsolvable within the allotted time limit of 30 seconds. The results for each experiment are the sum of the max number of agents solvable per each of the 25 trials. Top scores are in bold.

With the exception of some DAO maps where ICTS is faster, CBS+TAB is the strongest overall algorithm in 8-connected grids, and about equally as strong as CBS+TMA in 16-connected grids. CBS+TMA is consistently stronger in 32-connected settings.

Table 5.6 shows the size of the CT from sample problems from each category in Table 5.3. The results are for a number of agents that were solvable by all algorithms in under 30 seconds. CBS+TMA and CBS+TAB show a significant reduction over prior approaches. When comparing the amount of node reduction to the values in Table 5.3, the improvement is generally less significant – this is due to the low-level performing extra work evaluating constraints. In the case of CBS+TAB and CBS+MVB, a large number of constraints are usually added per CT node. In the case of edge, vertex and time-range constraints, there is only one constraint added per CT node and these

Table 5.3.: Total problems solved in under 30 seconds on 8-neighbor grid MAPF benchmarks

Type	Map	ICTS	Classic	Time	MVB	TMA	TAB
City	Berlin_1_256	628	1,626	1,595	1,624	1,785	**1,790**
	Boston_0_256	623	1,406	1,312	1,446	1,563	**1,574**
	Paris_1_256	615	1,545	1,480	1,607	1,653	**1,695**
DAO	brc202d	363	627	585	**658**	637	**658**
	den312d	**777**	550	456	558	545	549
	den520d	941	856	879	880	911	**954**
	lak303d	520	586	575	594	583	**600**
	orz900d	227	707	706	736	739	**780**
	ost003d	**924**	571	615	589	669	687
Dragon Age 2	ht_chantry	484	638	640	649	**711**	705
	ht_mansion_n	393	843	774	854	843	**871**
	lt_gallowstemplar	461	634	633	676	661	**699**
	w_woundedcoast	322	795	825	865	899	**935**
Open	empty-8-8	442	451	237	461	485	**493**
	empty-16-16	429	567	96	592	595	**599**
	empty-32-32	674	986	70	1,001	1,019	**1,027**
	empty-48-48	899	1,297	40	1,314	**1,393**	1,307
Open+ obstacles	random-32-32-10	487	880	938	910	903	**925**
	random-32-32-20	305	686	773	699	757	**773**
	random-64-64-10	656	**1,539**	1,521	1,383	1,483	1,415
	random-64-64-20	535	1,068	1,013	1,101	**1,152**	**1,152**
Maze	maze-32-32-2	239	306	373	308	315	**344**
	maze-32-32-4	223	297	269	299	291	**304**
	maze-128-128-10	252	356	309	356	399	**422**
	maze-128-128-2	232	237	243	250	241	**278**
Room	room-32-32-4	278	440	347	441	457	**480**
	room-64-64-16	355	516	426	529	555	**575**
	room-64-64-8	310	346	299	**399**	371	383

Table 5.4.: Total problems solved in under 30 seconds on 16-neighbor grid MAPF benchmarks

Type	Map	ICTS	Classic	Time	MVB	TMA	TAB
City	Berlin_1_256	601	1,356	1,001	1,431	**1,629**	1,570
	Boston_0_256	559	1,226	858	1,351	**1,463**	1,442
	Paris_1_256	645	1,207	1,075	1,236	**1,489**	1,393
DAO	brc202d	324	460	401	485	485	**505**
	den312d	450	476	347	477	**507**	502
	den520d	805	695	670	686	**821**	748
	lak303d	**542**	369	308	393	449	435
	orz900d	197	330	320	345	**369**	361
	ost003d	**622**	468	414	490	541	539
Dragon Age 2	ht_chantry	318	499	426	529	**585**	559
	ht_mansion_n	344	561	468	607	**645**	609
	lt_gallowstemplar	412	568	495	599	607	**625**
	w_woundedcoast	292	461	411	479	**519**	503
Open	empty-8-8	384	375	254	387	**445**	386
	empty-16-16	354	486	259	518	521	**527**
	empty-32-32	422	832	490	808	**891**	827
	empty-48-48	510	1,196	727	1,204	**1,299**	1,214
Open+ obstacles	random-32-32-10	387	624	358	662	**761**	724
	random-32-32-20	305	586	354	607	627	**645**
	random-64-64-10	519	1,032	632	1,078	**1,203**	1,112
	random-64-64-20	448	732	495	792	853	**882**
Maze	maze-32-32-2	229	271	251	**291**	271	282
	maze-32-32-4	170	173	173	184	251	**264**
	maze-128-128-10	176	244	211	291	**315**	306
	maze-128-128-2	190	236	190	184	197	**213**
Room	room-32-32-4	259	382	274	393	395	**426**
	room-64-64-16	326	405	300	435	485	**513**
	room-64-64-8	294	291	241	302	315	**345**

Table 5.5.: Total problems solved in under 30 seconds on 32-neighbor grid MAPF benchmarks

Type	Map	ICTS	Classic	Time	MVB	TMA	TAB
City	Berlin_1_256	565	1,104	715	1,212	**1,521**	1,310
	Boston_0_256	600	855	579	927	**1,205**	974
	Paris_1_256	631	1,076	745	1,124	**1,375**	1,280
DAO	brc202d	329	373	291	386	**473**	413
	den312d	522	426	273	452	**479**	470
	den520d	**802**	511	373	521	627	552
	lak303d	**534**	307	211	326	363	346
	orz900d	157	260	223	276	**297**	281
	ost003d	**701**	391	244	421	471	423
Dragon Age 2	ht_chantry	433	396	282	448	**541**	477
	ht_mansion_n	377	429	273	511	**561**	513
	lt_gallowstemplar	364	520	331	549	**571**	553
	w_woundedcoast	253	326	261	362	**453**	384
Open	empty-8-8	329	337	134	361	**423**	333
	empty-16-16	314	412	210	414	**471**	457
	empty-32-32	438	709	407	735	**841**	762
	empty-48-48	484	1,015	506	1,030	**1,205**	1,091
Open+ obstacles	random-32-32-10	395	575	316	601	**713**	624
	random-32-32-20	313	518	303	577	605	**615**
	random-64-64-10	463	857	438	904	**1,085**	953
	random-64-64-20	480	644	399	662	**811**	774
Maze	maze-32-32-2	208	232	195	250	259	**260**
	maze-32-32-4	156	158	152	179	**233**	230
	maze-128-128-10	180	220	162	250	267	**283**
	maze-128-128-2	192	203	134	179	**193**	187
Room	room-32-32-4	267	373	246	378	397	**415**
	room-64-64-16	281	333	218	367	**443**	424
	room-64-64-8	267	255	191	263	**309**	306

Table 5.6.: Final size of CT on 16-connected grids

Configuration	Classic	Time	MVB	TMA	TAB
City: Boston	1,422	360	304	180	**174**
DAO: ost003d	2,043	1,197	575	485	**473**
DA2: ht_chantry	2,762	669	607	164	**150**
Open: 16x16	27	25	23	24	**19**
Obstacles: 64x64-20	896	240	602	201	**199**
Maze: maze-32-32-4	13,237	9,332	10,326	6,261	**5,752**
Room: room-64-64-8	1,462	459	1,451	468	**321**

constraints are inexpensive to evaluate. In the case of CBS+TMA, there is only one constraint per CT node, however, because the implicit constraints perform a collision check when evaluated, they are more costly in terms of runtime. It is often the case that an MVB is a $1 \times N$ biclique (about 56% in 16-connected grids), thus, the set of blocked actions in CBS+TMA constraints is identical to CBS+TAB in many cases.

5.5. Continuous-Time Mutex Propagation

5.5.1. Background: Unit-Cost Mutex Propagation

Mutex propagation (MP), a technique for finding unreachable states in planning graphs [205], has been integrated into CBS for symmetry breaking [218]. MP helps to determine sets of mutually exclusive states for conflicting agents, often allowing for an immediate resolution of conflict symmetries.

Unlike biclique reduction which only applies to spatial conflict symmetries, MP is general and applies to both spatial and temporal conflict symmetries. MP for unit time is carried out in four steps:

1. Build an MDD for each agent.

2. Discover initial mutexes between MDDs.

3. Propagate the mutexes.

4. Extract motion constraints for CBS.

In Step 1, MDDs are built for two conflicting agents as shown in Figure 5.7(a), where the nodes in red correspond to agent x and the nodes in yellow correspond to agent y in Figure 5.1(a).

In Step 2, the MDDs are traversed in parallel from start to goal. At each time step, the Cartesian product of the two MDDs is checked for conflicts. For example, in Figure 5.7(a), the combination {B1}×{C2}={(B1,C2)} is checked at time step t=0, and so on, for each time step. A mutex $\langle s_i^t, s_j^t \rangle$ is created for any pair $s_i^t \in MDD_i$ and $s_j^t \in MDD_j$ of MDD nodes (or edges) which conflict (e.g., (B2,B2) at time step 1). These initial mutexes are depicted as blue dashed lines in Figure 5.7(a).

In Step 3, the initial mutexes are propagated, meaning that whenever all parent MDD nodes of s_i^t are mutex with all parent MDD nodes of s_j^t, a new mutex $\langle s_i^t, s_j^t \rangle$ is created. A propagated mutexes represent a combination of actions that can never be reached due to conflicting predecessors. These propagated mutexes are depicted as red dotted lines in Figure 5.7(a). For example, at time step 2, B3 and A2 get a propagated mutex because their only parents (B2 and B2) respectively, at time step 1 are mutex.

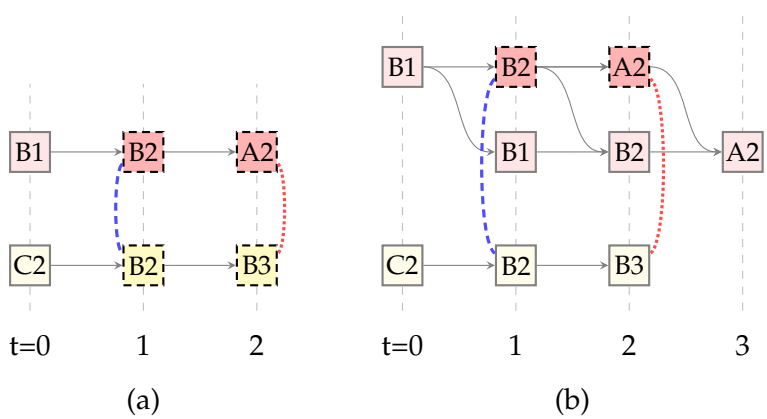


Figure 5.7.: (a) Mutex propagation for agents x and z for the MAPF instance in Figure 5.1(a) with cost limits $\langle 2, -, 2 \rangle$ and (b) the same analysis as (a) but with cost limits $\langle 3, -, 2 \rangle$.

In Step 4, we analyze the MDDs and mutexes. For any s_i^t which is mutex with all s_j^t at the same time step, vertex constraints are created for s_i^t. These are shown in Figure 5.7(a) as dashed nodes. For example, B2 at time step 1 in the upper MDD is used for a constraint for agent x because it is mutex with all nodes at the same time step in the lower MDD. In Figure 5.7(b), the cost limit for agent x has been increased to 3, resulting in additional MDD nodes being added to the upper MDD. Because of these additional MDD nodes, B2 in the lower MDD is no longer mutex with all MDD nodes in the upper MDD at t=1. Hence it can no longer be used for a constraint in CBS.

5.5.2. Continuous-Time Mutex Propagation

To perform mutex propagation in General MAPF domains, we present a novel algorithm: Pairwise Constraint Search (PCS). In contrast to classic mutex propagation which uses explicitly constructed MDDs, PCS plans two conflicting agents in their joint state space to discover motion constraints. Like biclique constraints, the approach to discovering valid sets of motion constraints in PCS involves analysis of bipartite conflict graphs [199]. This is illustrated in Figure 5.8.

Note that it is not necessary to use vertex constraints – edge constraints capture enough information to make CBS complete. We can replace a vertex motion constraint

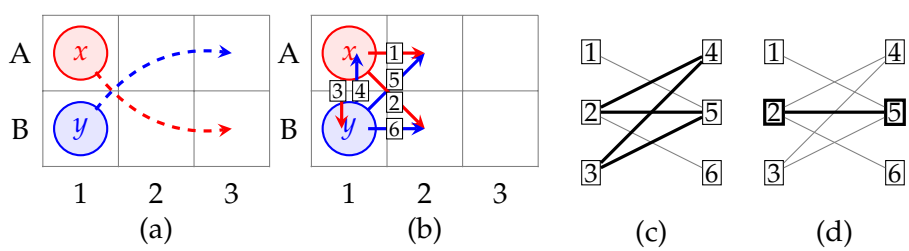


Figure 5.8.: An example of determining motion constraint sets using a bipartite conflict graph. (a) A General MAPF instance, (b) an enumeration of actions available to the agents with wait actions omitted, (c) a bipartite conflict graph for the problem in part (b); a biclique is shown with thick lines and (d) the same bipartite conflict graph with bi-complete nodes in bold and a biclique for them in thick lines.

with a set of edge constraints for all edges that are connected to a vertex. We additionally note that, unlike vertex constraints, edge constraints are valid for all types of conflicts in General MAPF (see Figure 5.2). For this reason, analysis for vertex constraints are omitted from PCS.

Figure 5.8(a) shows an example problem where two agents must cross paths. Figure 5.8(b) shows an enumeration of all actions available to two agents at overlapping time frames with wait actions omitted. Figure 5.8(c) and (d) show the bipartite conflict graph for the enumerated actions. Bicliques are shown with thick lines.

A vertex $v \in V$ of a bipartite graph $B = (U, V)$ is said to be *bi-complete* if v is connected to all vertices in U. Figure 5.8(d) shows a *bi-complete biclique* (BBC), where each vertex in the biclique is *bi-complete*. Bi-complete vertices are shown with bold borders in Figure 5.8(d). For example, node 2 is connected to all nodes on the right side of the full BCG, hence is bi-complete. Vertex 5 is also bi-complete, hence the biclique shown in thick lines (connecting nodes 2 and 5) is a BBC.

While biclique constraints as discussed in Section 5.4 are limited to a single time frame, we show that with the notion of BBCs they can be extended to future time frames as well, using mutex propagation. Figure 5.9(a) shows all actions for paths which arrive at the goal within a cost of 2.4, where we assume diagonal actions cost 1.4 and cardinal actions cost 1. In other words, these are MDDs. The actions in these MDDs are enumerated in Figure 5.9(b).

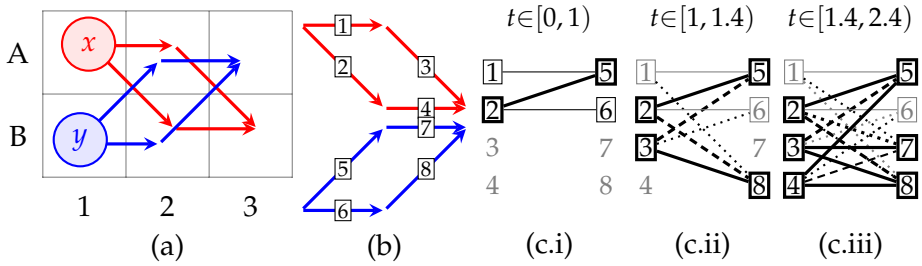


Figure 5.9.: An example of determining motion constraint sets using mutex propagation. (a) Paths with a cost limit of 2.4, (b) an enumeration of actions in the respective MDDs, (c) steps in continuous-time mutex propagation.

Figures 5.9(c.i-c.iii) show the steps of continuous-time mutex propagation. In Figure 5.9(c.i) the time window of $[0, 1)$ is considered and a BBC containing nodes 2 and 5 is found – actions 2 and 5 conflict with all other actions in this time frame. In Figure 5.9(c.ii) the time window of $[1, 1.4)$ is considered, and the BBC is increased to include nodes 3 and 8. Note that although nodes 3 and 5 do not directly conflict, the parent of node 3 (node 1) conflicts with node 5, hence nodes 3 and 5 get a propagated mutex, shown with a dashed line. The same situation applies for nodes 2 and 8.

Furthermore, we introduce a new notion called *inherited mutexes*. An inherited mutex is applied to all parents of mutexed nodes. These are shown as dotted lines in Figures 5.9(c.ii), (c.iii). For example, in Figure 5.9(c.ii), node 1 is connected with a dotted line to node 8 because node 1 is the parent of node 3 and node 3 conflicts with node 8. This process is continued in Figure 5.9(c.iii), resulting in a BBC with 6 nodes. Using only bi-complete nodes in the BBC guarantees that they are mutually conflicting across space and the inclusion of inherited mutexes guarantees that the actions are mutually conflicting across time.

Thus we use Algorithm 5.2 to find a max-vertex BBC which covers both the spatial and temporal domains. CBS with mutex propagation would use the BBC vertex sets $\{2, 3, 4\}$ and $\{5, 7, 8\}$ in a disjunctive split in the high-level tree.

The final details, pseudocode and empirical results of continuous-time mutex propagation are concluded in Section 6.3.

5.6. Low Level Search

In this section, we discuss prior work which is not a new contribution of this thesis. It is included here for completeness. A key factor for the choice of algorithm for the low-level depends upon whether fixed or arbitrary-duration wait actions are allowed. In some domains, only discrete wait actions are allowed, for example in Classic MAPF, where in others, continuous-time/arbitrary duration wait actions may be preferred, for example, in robotic motion planning.

For fixed-duration wait actions, A* is sufficient because the expansion routine is well-defined. For planning with arbitrary-duration wait actions the Safe Interval Path Planning (SIPP) [143] algorithm can be used (See Section 3.4.4).

For the purposes of MAPF, SIPP is modified to respect CBS constraints during successor generation [6, 37]. All vertices are initialized with safe intervals of $[0, \infty]$. Then, as conflicts are detected, the safe intervals are split up such that they omit the unsafe interval. For example, with an unsafe interval $[2, 3]$, the safe interval for v becomes $[0, 2), (3, \infty]$. The safe intervals are then used to plan wait actions. For example, during planning, if the shortest path for agent i requires it to enter vertex v at time t, and no safe interval at time t exists, a wait action is generated such that agent i will wait a sufficient amount of time to move to v at a safe interval. It has been shown that CCBS with SIPP can not only find solutions faster in many cases due to resolving conflicts by waiting appropriate durations, but often the wait times are shorter resulting in lower-cost solutions.

In environments applicable to canonical orderings such as the 2^k neighborhoods, Jump Point Search with Temporal Obstacles (JPST) [76] has also been successfully used with CBS for arbitrary-duration wait actions.

5.7. High Level Search

In Section 5.2.1 we mentioned the Improved CBS algorithm [25] which does conflict prioritization and uses a bypass whenever possible. In this section, we discuss further improvements for the high-level search algorithm.

5.7.1. High-Level Heuristics

Classic CBS prioritizes nodes in the OPEN list by flowtime. This is analogous to g-cost in A* (see Section 2.2.1). Heuristics can also be applied to CBS to guide the search. Recall from Section 2.2.2 that a heuristic is an estimate of cost-to go. In CBS, the cost-to-go is analogous to determining the number of conflicts left to resolve and

estimating the expected cost increase required to resolve them. In order to do this, the concept of a conflict graph was introduced [51].

A conflict graph (CG) is a graph with k vertices, one for each agent. Edges connect two vertices if there is a Cardinal conflict between the two agents' paths. It is possible that resolving a conflict between two agents will also resolve a conflict between two other agents when they are re-planned. In order to keep the heuristic admissible, we cannot count all of the conflicts, instead we must count the minimal number of conflicts that require resolution. Computing this is analogous to finding the size of a minimum vertex cover (MVC) [50]. The MVC problem is NP-hard [211], thus may be prohibitive to compute in large graphs, however a slightly weaker heuristic can be computed by finding the maximal matching (MM) [142] which can be done in polynomial time. Use of the CG heuristic was shown to reduce the total number of expanded nodes significantly, with consistent improvement in overall runtime.

The conflict graph was extended to the notion of a dependency graph (DG) [108] which incorporates conflict symmetry analysis to add edges for cases when symmetries are detected. Furthermore, a DG is extended to use weighted edges, known as a weighted dependency graph (WDG) [108]. It captures the required cost increase to resolve conflicts as edge weights. With the WDG, instead of solving the MVC problem, the edge-weighted minimum vertex cover (EWMVC) problem or max-weight matching (MWM) problem is solved instead. The EWMVC problem is formulated as an integer linear program [139]:

$$
\begin{aligned}
&\text{minimize} \\
&\sum_{x \in X} (x) \\
&\text{subject to} \quad x_i + x_j \geq= w_{ij} \; \forall (x_i, x_j) \in X \times X; w_{ij} \in w(E); i \neq j \\
&\qquad\qquad\qquad x_i \geq 0 \; \forall x_i \in X \\
&\qquad\qquad\qquad w_{ij} \geq 0 \; \forall w_i j \in w(E)
\end{aligned}
\tag{5.1}
$$

where X is the set of all k path costs and w_{ij} is the weight of the edge between vertices i and j in the WDG.

In the case of General MAPF, because costs are not integers, Formula 5.1 is solved using a (non-integer) linear solver, such as the revised simplex algorithm [49] or the internal point method [87]. The former has exponential complexity in the worst case, and the latter has polynomial complexity. But the revised simplex method tends to outperform the internal point method for this particular LP formulation.

5.7.2. Additional Enhancements of CBS

The conflict avoidance table (CAT) as discussed in Sections 2.4.4 and 3.5 has been shown to be effective with CBS for Classic MAPF, but may have limited effectiveness in domains with few path symmetries. The conflict count table (CCT) as discussed in Section 2.4.4 is an effective memoization technique for CBS.

Meta-Agent CBS (MA-CBS) [159] is an extension of CBS which is optimal. It was invented in order to resolve pathological cases such as conflict symmetries using an alternate solver. In MA-CBS, initially all meta-agents are initialized as groups consisting of a single agent. When a pathological case is detected between two or more meta-agents, the meta-agents are merged and solved using a separate multi-agent solver at the low-level.

Disjoint splitting [107] re-formulates a split to use a *positive* constraint which forces one agent a_i to be at vertex $v \in V$ at time t and all other agents $a_j; x \neq i$ to avoid v at time t using *negative constraints*, and re-plans any agents that are in violation of these negative constraints. This approach resolves a fundamental inefficiency of CBS which is that it may resolve the same conflict multiple times in different sub-trees of the CT. Disjoint splitting was shown to significantly reduce the overall runtime in most cases when compared to non-disjoint splitting.

Iterative Deepening CBS (IDCBS) [26] re-formulates the high-level to perform a depth-first [96] approach instead of a best-first approach. IDCBS mitigates a shortfall

of CBS which is that it must retain the entire CT, or at least the leaf nodes of the CT in memory, and may quickly exhaust available memory when solving difficult problems. IDCBS alleviates this problem by using an approach analogous to depth-first iterative deepening [96] which only holds $O(\Delta)$ nodes in memory at one time where Δ is the depth of the goal. In addition to this, IDCBS capitalizes on minimizing memory use for the CAT, CCT and even the open list for the low-level search. Although IDCBS may ultimately generate more nodes than CBS, the overhead of multiple operations such as sorting the OPEN list, re-planning agents, etc. is smaller, hence IDCBS outperforms CBS in most cases.

5.7.3. Empirical Results

We now evaluate the effectiveness of mutex propagation. All tests in this section use an implementation of mutex propagation (MP) that follows Algorithm 6.2, which is for General MAPF domains based on A* with bookkeeping and does not build the MDDs explicitly. Figure 5.10 shows the performance of CBS+MP versus CBS for various benchmark problems on 4-neighbor grid maps (Classic MAPF instances). Both variants use the WDG heuristic. The x-axis shows the number of agents and the y-axis shows the percentage of problems successfully solved in under 30 seconds for a specific number of agents. As the number of agents increases the problem gets exponentially harder, so solving for even one more agent in the allotted time frame represents a significant gain. We see in Figure 5.10 that the addition of mutex propagation is the most helpful in maps with relatively large empty areas. It is less helpful in maps with many small obstacles.

In our implementation, MP is only run when CBS encounters a Cardinal conflict. Depending on the specific configuration, MP may or may not find useful constraint sets. In the case where a Cardinal conflict is a conflict symmetry, MP will find constraint sets of size larger than one. However, when a Cardinal conflict is not a conflict symmetry, MP finds only one constraint for each agent. Because MP is relatively

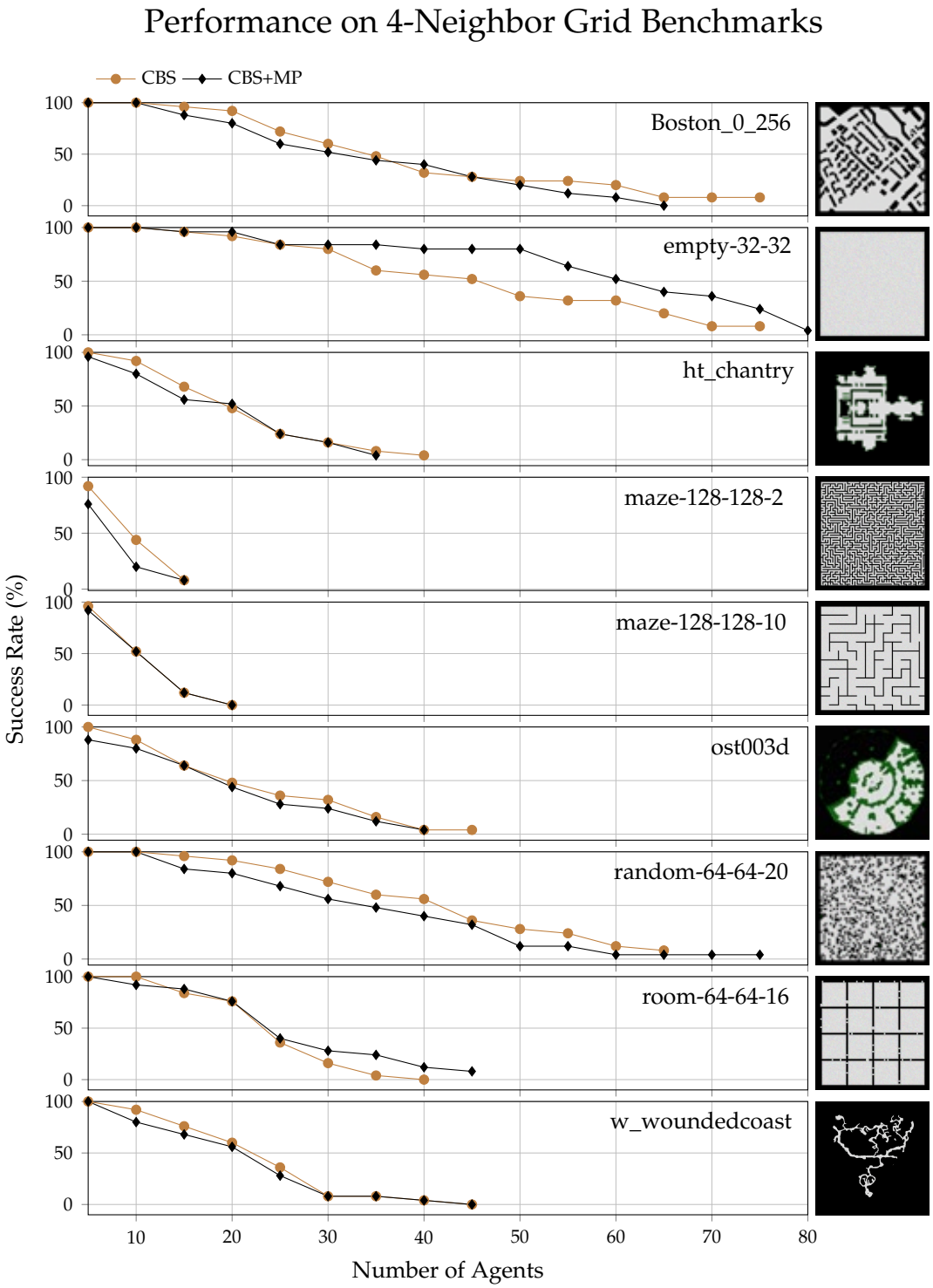


Figure 5.10.: Success rates of CBS and CBS+MP for benchmark problems on 4-neighbor grids.

computationally expensive, running it in maps without a significant number of conflict symmetries can be detrimental.

Figures 5.11 and 5.12 show performance of CBS, CBS with biclique constraints (CBS+BC) and CBS+BC+MP. CBS+BC uses the time-annotated biclique approximation (referred to as TMA in Section 5.4) in 8- and 16-neighbor grids respectively (General MAPF domains). CBS+BC+MP uses mutex propagation on Cardinal conflicts and biclique constraints on non-Cardinal conflicts. Note that Figure 5.10 does not contain results for an implementation of biclique constraints because bicliques are always equivalent to edge or vertex constraints in Classic MAPF. All implementations use the WDG heuristic.

In 8-neighbor grids, CBS+BC dominates CBS+MP+BC on all shown benchmarks in Figure 5.11 except for empty grids. The reason for this is the BC algorithm is significantly less computationally expensive, and also has properties of spatial conflict symmetry breaking. CBS+BC+MP is stronger in empty maps because empty maps typically contain a large amount of spatial conflict symmetries and MP is able to resolve these conflict symmetries with one split operation. CBS+BC dominates CBS+MP+BC in all benchmarks shown in Figure 5.12, including the empty grid maps because 16-neighbor grids have fewer symmetries in general (see Figure 5.2). Therefore, The more computationally-efficient biclique constraints are more effective in this case.

We also note that mutex propagation is affected significantly by the length of the paths being planned. The complexity of MP (which plans for two agents) is $O({b_{base}^2}^d)$ where b_{base} is the mean single-agent branching factor and d is the depth or length of the solution. Hence, large maps which tend to have longer paths will cause exponentially more overhead than paths in small maps. A strong heuristic for MP however, can significantly reduce the complexity.

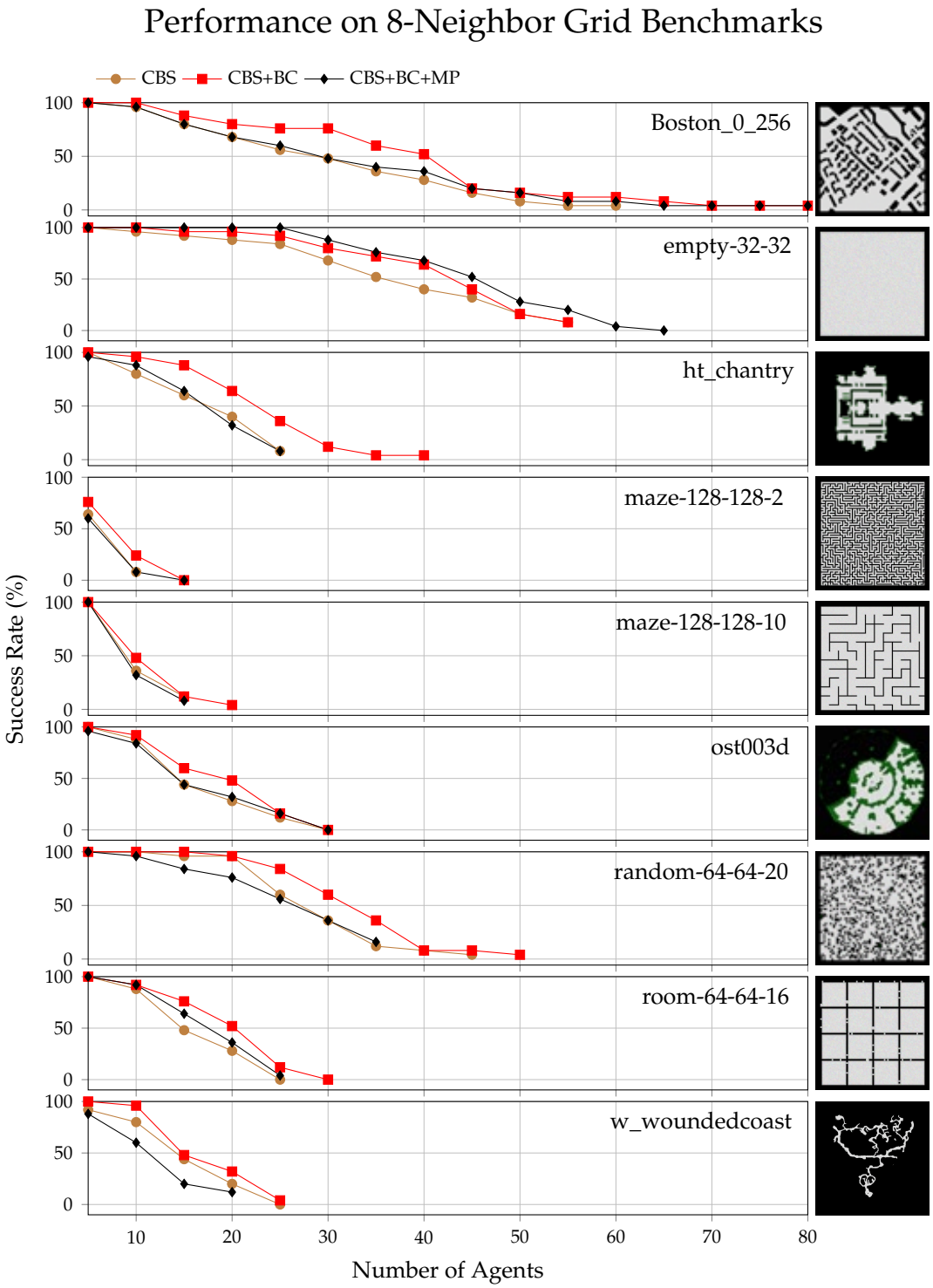


Figure 5.11.: Success rates of CBS, CBS+BC and CBS+MP+BC for benchmark problems on 8-neighbor grids.

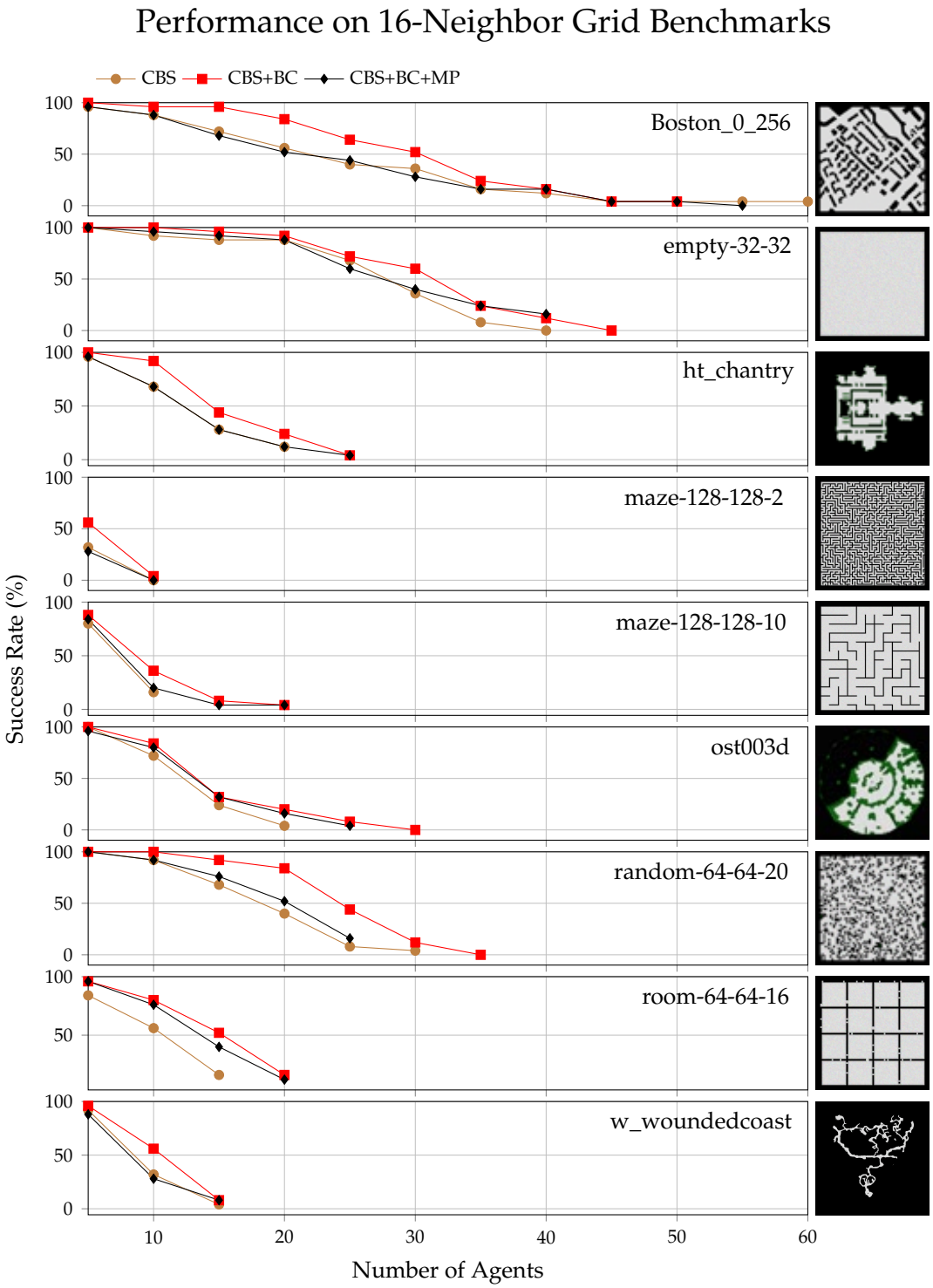


Figure 5.12.: Success rates of CBS, CBS+BC and CBS+MP+BC for benchmark problems on 16-neighbor grids.

5.8. Sub-Optimal Variants of CBS

Bounded and unbounded sub-optimal variants of CBS have also been formulated [15, 36, 196, 199]. The simplest of these variants is Greedy CBS (GCBS) [15] which is unbounded sub-optimal. GCBS changes the prioritization of the open list to use notions other than cost. The simplest, and most effective is an ordering by NC, the number of conflicts in $N.\Pi$. This helps CBS find a solution significantly faster, but no guarantee can be made on the level of optimality of the final solution. Some experiments were done with inflation of heuristics at the low-level, but generally speaking, longer, sub-optimal paths at the low level resulted in more conflicts at the high-level [15].

The Enhanced CBS (ECBS) [15] and Improved Enhanced CBS (IECBS) algorithms utilize an OPEN list and a FOCAL list [140] prioritized by an alternate objective that leads to faster solutions (e.g., NC). A sub-optimality bound parameter w is supplied. Candidates from the top of the OPEN list with f-cost inside the w-bound are inserted into the FOCAL list and re-prioritized. In this way, solutions are found much faster, with guaranteed optimality bounds.

This thesis contributes two additional techniques for sub-optimal CBS. The first technique is *constraint layering*, the second is *sub-optimal* constraints. Both approaches are shown to significantly increase performance.

5.9. Constraint Layering

Conflict-Based Search with Constraint Layering (CBS+CL) [196] uses a hierarchy of edge subgraphs (subgraphs with edge deletions) to simplify the planning process. CBS+CL plans individual agents in the environment, successively removing artificially added constraints from each agent as conflicts are discovered. Initially, for instance, an agent may be constrained to plan only in a 4-neighbor gird. But when collisions must be resolved, constraints on the agent's motion are relaxed to allow movement on an 8-neighbor grid, giving it more freedom to move and avoid conflicts. Our

results show that this approach allows us to solve up to 2.4 times more agents in the same amount of time when compared to regular CBS on the original graph and over 50% more agents in the same amount of time when compared to regular CBS on edge subgraph abstractions.

Environment abstractions can be created by forming subgraphs via node contractions, edge deletions or adding new embeddings to the original search graph [72]. Various abstraction techniques have been published viz. Clique Abstraction, Sector Abstraction, Line Abstraction, STAR [73, 175] and JPS [67]. All of these abstractions rely on the downward refinement property which means that a path between nodes in an abstracted graph must be refinable to a path in the original graph.

CBS+CL combines the strengths of constrained and unconstrained environments. To mitigate the cost of search in environments with high branching factors it is helpful to perform the low-level search on a constrained/abstracted version of the original environment. By introducing movement constraints into the original environment we reduce the branching factor, allowing better performance in the CBS low-level search.

We formulate constrained environments as graph abstractions. The practice of constraining on an agent is analogous to removing edges from the planning graph. Conversely, adding edges to the planning graph is analogous to relaxing the constraints. Figure 5.13 shows a continuum with representations of more- and less- con-

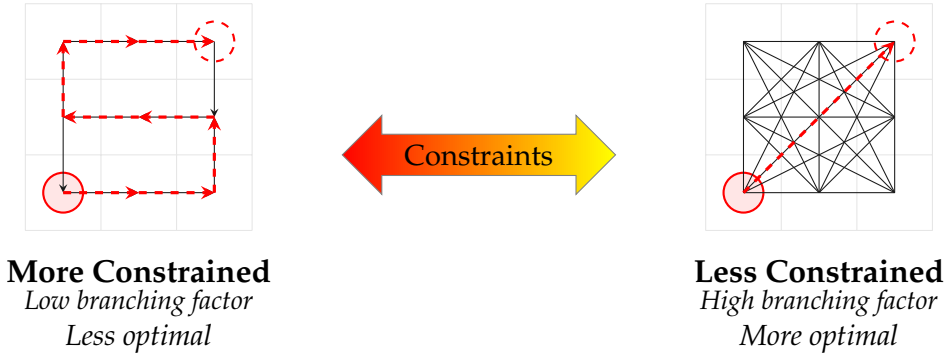


Figure 5.13.: Continuum of constraint abstraction.

strained environments. As the level of constraint increases (i.e., edges are removed), the branching factor is increased and vice-versa.

Formally, graphs formed by edge deletion are known as *edge subgraphs* [59]. An edge subgraph is a graph $G' \subseteq G$ s.t. $G' = (V, E'), E' = E \backslash X$ where X is the set of edges deleted from the initial graph G.

Search on edge subgraphs does not require special downward refinement as edge subgraphs contain a subset of the edges in the original domain. Additionally, this method of abstraction maintains the original edge lengths which allows conflict detection logic to be the same for all abstractions.

5.9.1. Abstraction For Conflict Avoidance

A sub-optimal method of conflict avoidance at the low-level is to direct the search in ways that produce fewer conflicts. Direction maps [80] have been proposed to augment heuristics. Direction maps provide an underlying flow field and agents are penalized if they do not follow the flow. Direction maps can be formulated as highways or circular movement patterns. In ECBS+HWY, [35] direction maps are used to influence agent movement to produce significant performance improvements.

Edge subgraphs are similar to flow fields because possible movements are restricted. Flow-Annotation Replanning (FAR) [201] performs static analysis of the search graph and creates edge subgraphs, favoring directed edges so that collisions are less likely.

Instead of augmenting heuristics or abstraction via node contraction, we have taken an approach similar to FAR, which is abstraction via the creation of edge subgraphs. However, unlike FAR, we do not perform static analysis and augmentation of the environment. Instead, we apply uniform edge deletions across the entire search graph.

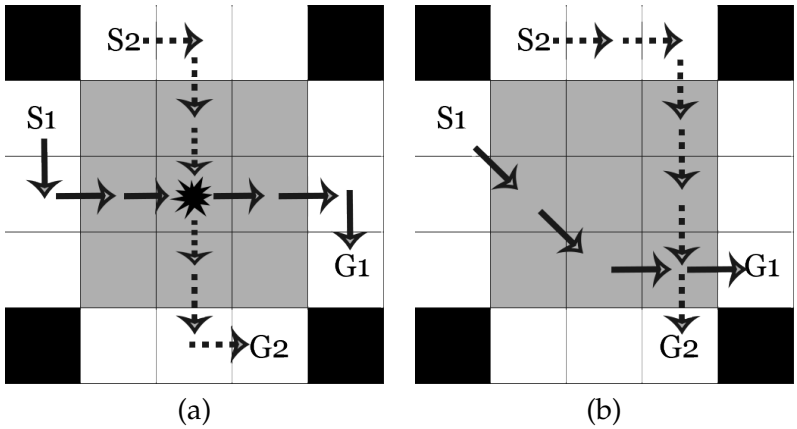


Figure 5.14.: Comparison of grid-based search with differing movement constraints. In (a), movement for both agents is constrained to be on 4-neighbor grids, thus all optimal path combinations will conflict in any of the grey squares. In (b), agent S_2 moves on a 4-neighbor grid, but agent S_1 moves on an 8-neighbor grid.

5.9.2. Constraint Layering for Conflict Avoidance

Consider an environment where agents are allowed to move on an 8-neighbor grid. The search will have a maximum branching factor of 9 if waiting is allowed. If we restrict agents to cardinal directions or wait, the maximum branching factor is reduced to 5 which may speed up pathfinding, however collisions become more likely. This scenario is illustrated in Figure 5.14(a), where both agents may only choose to move in a cardinal direction or wait. In this example, a rectangle conflict is immediately resolved in Figure 5.14b because agent 1 is allowed to use octile movement rules. Recall that conflict symmetries like rectangle conflicts can result in an exponential number of CT nodes at the high level. Instead, this approach allows immediate resolution of the conflict in many cases.

CBS+CL modifies the CBS algorithm as follows: We first define several environment abstractions (constraint "layers"), ranging from coarser (more constrained) to finer (less constrained). Then we set a conflict threshold for each environment. When the number of conflicts between two agents in any sub-tree of the CT meets the threshold for a particular environment, we still create two CT nodes as in traditional CBS,

except in CBS+CL we re-plan one agent path in each CT node using the new environment (line 21). If we set our thresholds to be incremental, the search environments become “layered" in the sense that as conflicts increase, we search on increasingly fine environments. Intuitively, highly-conflicting agents are increasingly given less constraints on their movement allowing them to circumvent conflicts easier.

5.9.3. Theoretical Analysis

CBS+CL has no optimality bound in general, though a specific bound may be provable depending on the characteristics of environment abstractions used. Though it is difficult to bound the optimality of CBS+CL, it is guaranteed to terminate with a solution if one exists [196]. Further, our empirical results show that CBS+CL does not, in practice, generate extremely costly paths for agents compared to optimal algorithms.

5.9.4. Airplane Domain Test Environment

In our testing, we apply both traditional CBS and our proposed improvements to a model of an aviation environment. We modeled the airspace with a three-dimensional grid, the width of which will allow an aircraft moving at maximum speed to negotiate a 90° turn. Each grid cell forms a cube with all three dimensions being the same.

In our model, we have a maximum branching factor of 63 - all combinations of heading changes: {0°, +45°, -45°, +90°, -90°, left shift, right shift}, speed changes: {no change, speed up, slow down} and height changes: {no change, climb, descend}. A *shift* is a maneuver where the aircraft moves in the diagonal direction but does not change heading. This environment which allows simultaneous change in heading, height and speed will be referred to in this thesis as the “base" environment. See Figure 5.15 for a representation of the movement model.

Our cost function is the same for all environment types and is based on fuel consumption. We base our fuel consumption on distance (liters per grid). The cost function is implemented with rules derived from [79]:

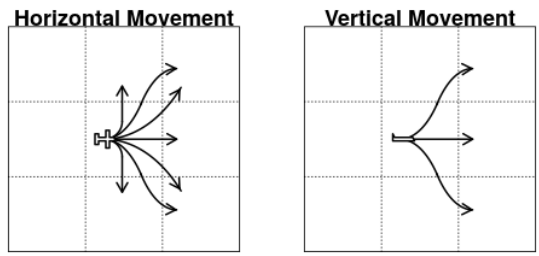


Figure 5.15.: Grid-based vertical and horizontal aircraft movement model for aircraft.

- Climbing adds fuel cost. (We used $c_{climb} = 0.001L$.)
- Descending saves fuel to half the cost of climbing. (We used $c_{desc} = -0.0005L$.)
- Traveling faster or slower than cruise speed decreases the fuel efficiency. We used 5 speeds with consumption rates of: $c_{speed} = [0.008, 0.0077, 0.006, 0.007, 0.008]$. The middle speed is cruise speed.
- Diagonal moves (45° turn and shift maneuver) multiply the fuel cost by $\sqrt{2}$ except for the cost of vertical movement, which is not affected by diagonal movement.

Given vector-valued aircraft states containing the variables: $\langle x, y, z, heading, speed \rangle$, the cost function $C(to, from)$ (the cost from the state "from" to the state "to") is described by:

$$C = c_{speed}(\text{to.speed}) \cdot \alpha + \beta$$

Where:

$$\alpha = \begin{cases} 1 & |\text{from.x} - \text{to.x}| \neq |\text{from.y} - \text{to.y}| \\ \sqrt{2} & \text{otherwise} \end{cases}$$

$$\beta = \begin{cases} c_{climb} & \text{to.z} - \text{from.z} > 0 \\ c_{desc} & \text{to.z} - \text{from.z} < 0 \\ 0 & \text{to.z} - \text{from.z} = 0 \end{cases}$$

Time is incremented based on speed and distance traveled. Thus, agents moving at higher speeds arrive at the next lattice point sooner and agents moving on a diagonal edge require additional traversal time.

Abstractions

All environments used in our experiments are abstractions formed by deleting edges from the “base environment". Because edge deletions never decrease path lengths from start to goal, admissible heuristics in the base environment remain admissible in an abstraction. It should be clear that the base heuristic may be weak in the abstracted environment, necessitating unique heuristics on a per-abstraction basis.

Highway Abstractions

We form highway abstractions by converting all undirected edges in the x, y and z dimensions to directed ones. We formulate vertically separated highways where agents flying at the same height fly in the same direction. The highway above or below a given height has edges pointing in an adjacent horizontal direction. Thus if an agent needs to turn, it must simultaneously change altitude to enter the highway for the desired direction. Thus we retain edges that simultaneously change altitude and heading into the highway just above and below a given highway.

Because agent heading is restricted, some goal or start states may be invalid. We cannot simply restrict all start/goal states to be within the highway abstraction, thus we relax the abstraction via adaptive dimensionality [62] for such states near the start and goal of the search:

- If an agent’s start state does not have a heading that conforms to the highway based on its height, it is only allowed to make moves that put it in alignment with the highway which lies in the direction of its goal.
- If an agent’s goal state is invalid with respect to the highway system, the agent is allowed to move freely when it is within 2 grid spaces from it’s goal.

5.9.5. Experimental Results and Analysis

All of our experiments are for a set of k agents with random start and goal locations inside an 80x80x20 three-dimensional grid world. Each configuration was run on a set of 100 MAPF instances with random start and goal positions. Our current implementation also assumes that agents disappear, i.e. do not remain in the collision space once reaching their goal. We tested on a progressively increasing number of agents. If a problem takes longer than five minutes to terminate, we mark it as a failure and set its completion time to five minutes. All experimental results shown were run on servers with Intel Xeon processors running at 2.4 GHz with 12 GB of memory.

Experimental Environments

All environments were benchmarked using the regular CBS algorithm. We also benchmarked against Greedy-CBS (GCBS) [15].

Base (8-Way) Environment This is the environment described at the beginning of this section. Agents are allowed to turn, change height and/or change speed simultaneously. The maximum branching factor is 63.

4-Way Environment This abstraction is the same as the "Base" environment except turns are restricted to 90°. The maximum branching factor is 27.

Simple Environment This abstraction restricts actions to one change per movement: heading, speed or height. The maximum branching factor is 11.

Highway-8 Environment We implemented an 8-directional highway system which enforces height-separated directional highways where agent heading is forced to be congruent to height modulo 8. The maximum branching factor is 9.

Highway-4 Environment This is similar to "Highway-8" except directions are restricted to height modulo 4. The maximum branching factor is 9.

5.9.6. Experimental Implementations

CBS We implemented CBS with the bypass enhancement [24] with no high-level heuristic and tie breaking toward CT nodes with lower conflict counts.

Greedy-CBS GCBS is the previous state-of-the-art for unbounded sub-optimal MAPF solvers which can be formulated for non-holonomic vehicles. It was shown that the performance of GCBS is on par with or better than the unbounded sub-optimal multi-agent A* variant MGS1 [172]. Our implementation GCBS uses NC for prioritization in the high level OPEN list. We include the performance of GCBS on the H4 environment as a benchmark because it is empirically the most performant abstraction.

5.9.7. Qualitative and Performance Results

Benchmark Environments

Figure 5.16 shows the results for mean time-to-solution. These results show that when using the traditional CBS algorithm, using any of the abstractions decreases the time-to-solution relative to the “Base" environment. The “Highway-4" abstraction is the best performer, allowing us to solve roughly twice as many agents in the same amount of time when compared to the “Base" environment.

We attribute the better runtimes to reduced branching factor and reduced conflicts due to highway traffic flows. We found that the number of conflicts that occurred in each environment indicated that the runtime strongly correlates to the number of conflicts. Using GCBS on the Highway-4 environment improved performance as well, allowing us to solve for about 30% more agents in the same amount of time.

Analysis of path lengths showed that solutions produced using the “Highway-4" environment are maximally 18% suboptimal when compared to the “Base" environment. GCBS seemed to have a minimal impact on solution quality. Table 5.8 shows the path quality and time to solution for various configurations. These results are from a subset of the 100 instances with 40 agents in which all environments were able to terminate with a result under the 5-minute time limit.

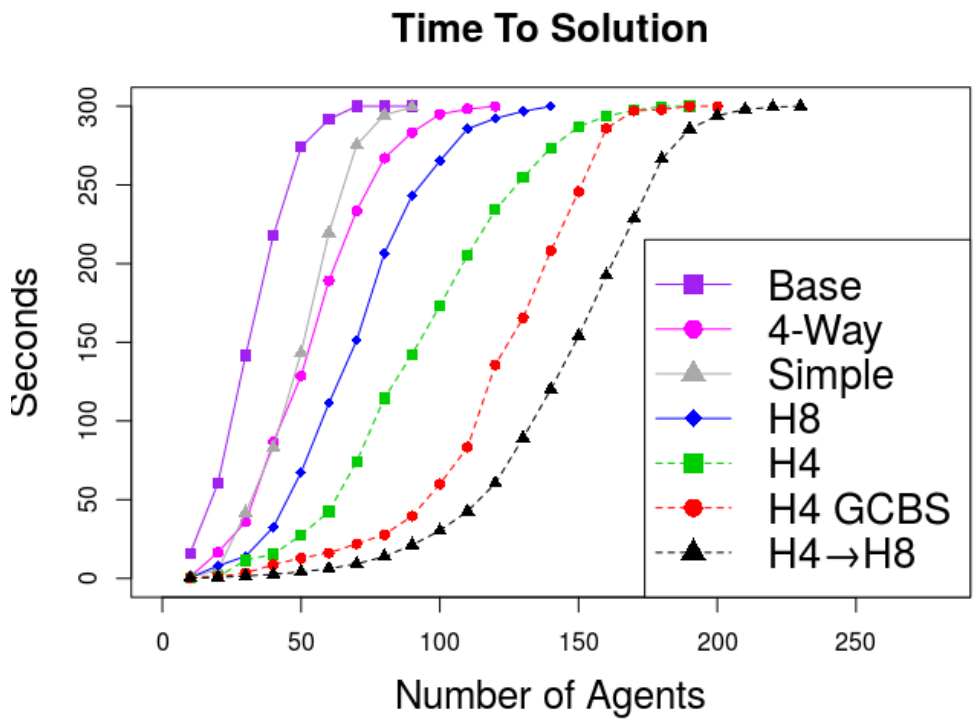


Figure 5.16.: Comparison of performance of different environment configurations

Layered Environments

Finally, we show our results for CBS+CL. We experimented with various abstraction layerings and switching thresholds. We found that it is generally most beneficial to set switching thresholds at increments of 1 so that upon encountering a conflict, the environment is switched immediately. We found that "Highway-4→Highway-8" had the best overall performance as shown in Figure 5.16. These combinations allowed us to find solutions for roughly 50% more agents when compared to just using "Highway-4", and 20% more agents when Compared to GCBS on "Highway-4".

Table 5.7 shows CT node and total low-level expansion counts for a set of problems which were solvable under the time limit by all configurations. Analysis of work done at the high and low level searches revealed that CBS+CL consistently lowered the number of conflicts found in the high-level search. Low-level expansion counts are

Table 5.7.: High and Low-Level Work by Configuration

Configuration	CT Nodes	Low-Level Expansions
H4→H8→Base	5.53	312099
H4→H8→Simple	5.61	294901
H4→H8→4-Way	5.61	301112
H4→H8	5.69	294712
H4→H8 GCBS	6.56	324818
H8→Simple→Base	8.62	303680
H4 GCBS	9.64	426309
H4	20.76	427492
H8 GCBS	28.62	305683
Base	29.32	2834728
4-Way	69.77	1219804
H8	84.62	471414
Simple	105.69	1372531

Table 5.8.: Quality and Performance by Configuration

Configuration	Sol. Cost	Optimality	Time
H4→H8 GCBS	4.44	0.83	0.84
H4→H8	4.43	0.83	0.85
H4→H8→4-Way	4.43	0.83	0.88
H4→H8→Base	4.43	0.83	0.91
H4→H8→Simple	4.43	0.83	0.92
H4	4.46	0.82	0.99
H4 GCBS	4.46	0.82	1.04
H8→Simple→Base	3.78	0.97	1.42
H8 GCBS	3.79	0.97	4.36
H8	3.79	0.97	6.92
Simple	3.72	0.99	10.77
4-Way	4.28	0.86	22.43
Base	3.67	1.00	67.86

affected both by the branching factors of the mixture of environment abstractions used in the test and the number of CT nodes. For example, although $H4 \rightarrow H8 \rightarrow Base$ had fewer conflicts overall than $H4 \rightarrow H8 \rightarrow Simple$, the number of expansions is higher due to the high branching factor induced by switching some of the agents into the Base environment. Notice however that both choices are better than using $H4$ alone.

We also experimented with GCBS+CL on "Highway-4→Highway-8" and found that it provided a slight improvement over CBS+CL on the same configuration. We be-

lieve the set of conflicts resolved by GCBS and CBS+CL have overlap, and thus using them in conjunction does not give a large incremental improvement.

Table 5.8 shows solution qualities for the same set of problems as Table 5.7. We found that in the "Highway-4→Highway-8" configuration, optimality varied between 18% and 3% sub-optimal depending on the percentage of agents in the search instance that switched to the "Highway-8" environment. Not only does switching into less-constrained abstractions reduce the number of conflicts and time-to-solution, it also improves the optimality of the solutions.

5.10. Sub-Optimal, Complete Constraints

We now introduce another novel contribution – sub-optimal constraints. In Section 5.4 we showed that biclique constraints are formulated to block large sets of mutually-conflicting actions to increase pruning of the CT. It is possible to further increase the number of blocked actions by relaxing the mutually disjunctive requirement, for example, by blocking *all* actions in the BCG. However, doing so may cause incompleteness in two ways: (1) termination at the low level without finding a path or (2) agents being constrained such that each low-level search returns a path which still conflicts with other agents. For example, if two agents continuoe to conflict over an over, but at increasingly later times. In situation (2) collisions tend to recur over and over at increasingly later times, causing the algorithm to run forever. For completeness, we must detect and avoid these two conditions. For this purpose, we introduce *conditional constraints*.

5.10.1. Conditional Constraints

Constraints for a CT node N apply *permanently* to N and are inherited by all CT nodes in the sub-tree of N. *Conditional* constraints are turned on by default, but may be turned off, meaning they no longer block any actions in N or its sub-tree. A constraint is turned off by omitting it from the low-level re-plan step after a split op-

eration. Per Lemma 5.3.1, if an action that is not in the biclique is blocked, CBS is no longer complete. To avoid this, mutually-conflicting actions from the MVB are always blocked *permanently* and other actions not in the MVB are blocked *conditionally*, so that those actions may be unblocked to avoid incompleteness.

Figure 5.17(a) and the corresponding BCG in (b) are shown for the same scenario as Figure 5.5: Actions corresponding to the MVB are permanent and shown with bold lines. All other actions in the BCG are shown with dashed lines – these are the set of conditional constraints.

Algorithm 5.3 contains pseudocode for the CT node expansion portion of CBS (see Algorithm 5.1 lines 5-14) with enhancements for biclique constraints. Further enhancements for implementing conditional constraints are highlighted in red. After detecting a conflict between two core actions (line 2) child nodes N_i, N_j are created as copies of N (line 4). Then the steps for creating permanent constraints are executed in the same manner as described in Section 5.3 (lines 8-12). Then conditional constraints are created from $U \setminus U'_t$ and $V \setminus V'_t$ where U and V are from the BCG, and U'_t and V'_t are from the TAB (lines 13,14). Then conditional constraints can later be turned off according to the two causes of incompleteness as follows:

Situation (1) will occur when a low-level re-plan for an agent returns no path because a conditional constraint blocked a feasible path (lines 23, 27). When this occurs,

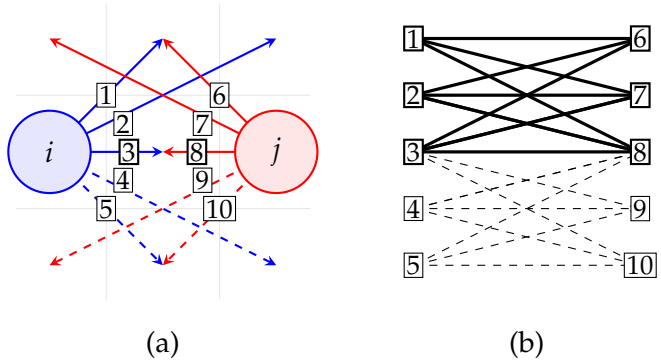


Figure 5.17.: Illustration of (a) sets of available actions for two agents and (b) the corresponding BCG.

conditional constraints are removed from $N_i.C$ and $N_j.C$ and the re-plan is performed again (lines 25, 29).

Situation (2) will occur when an agent is over-constrained such that it cannot arrive at its goal. Because this situation is caused by one of the conditional constraints, we use a strategy to turn them off *probabilistically*. Specifically, they are turned off with an increasing probability $\rho_{\text{off}} = \text{MIN}(1, {}^{(\Delta_i - 1)}/_{d_i})$ (line 16) where d_i (line 6) is the length of the path for agent i in the root CT node and Δ_i (line 5) is the number of conflicts with

Algorithm 5.3. Expand-CT-Node

1: Input: N – a CT node
2: $\langle a_i, a_j \rangle \leftarrow$find-conflict$(N.\Pi)$
3: **if** No conflict **return** $N.\Pi$ as goal
4: $N_i \leftarrow N; N_j \leftarrow N$ // Copy N to child nodes for split
5: Get conflict counts Δ_i, Δ_j: the number of conflicts from N to root
6: Get length of path d_i, d_j in CT root node for i and j
7: // Compute BCG and biclique for core action pair
8: $(U, V, E) \leftarrow CreateBCG(a_i, a_j)$
9: $(U'_t, V'_t) \leftarrow ComputeMaxVertexTAB(U, V, E)$
10: // Create constraints
11: $N_i.C \leftarrow N_i.C \cup CreatePermanentConstraints(U'_t)$
12: $N_j.C \leftarrow N_j.C \cup CreatePermanentConstraints(V'_t)$
13: $N_i.C \leftarrow N_i.C \cup CreateConditionalConstraints(U \setminus U'_t)$
14: $N_j.C \leftarrow N_j.C \cup CreateConditionalConstraints(V \setminus V'_t)$
15: // Create probabilistically filtered sets
16: $\rho_i \leftarrow \text{MIN}((\Delta_i - 1)/d_i, 1.0); \rho_j \leftarrow \text{MIN}[e]((\Delta_j - 1)/d_j, 1.0)$
17: Remove *conditional* constraints from $N_i.C$ with probability ρ_i
18: Remove *conditional* constraints from $N_j.C$ with probability ρ_j
19: // Re-plan with (filtered) constraint sets
20: $N_i.\Pi \leftarrow Replan(start_i, goal_i, N_i.C)$
21: $N_j.\Pi \leftarrow Replan(start_j, goal_j, N_j.C)$
22: // Check for no path and re-plan without conditional constraints
23: **if** $N_i.\Pi.\pi_i = \emptyset$ **then**
24: Remove *all* conditional constraints from $N_i.C$
25: $N_i.\Pi \leftarrow Replan(N.\Pi.\pi_i, N_i.C)$
26: **if** $N_j.\Pi.\pi_j = \emptyset$ **then**
27: Remove *all* conditional constraints from $N_j.C$
28: $N_j.\Pi.\pi_j \leftarrow Replan(N.\Pi.\pi_j, N_j.C)$
29: Add N_i, N_j to OPEN

agent i in CT nodes from N to the root. As the search progresses, if agent i has recurring conflicts, Δ_i will grow relative to d_i increasing ρ_{off}, resulting in a higher proportion of conditional constraints being turned off. Eventually, any conditional constraints causing situation (2) to occur will be turned off, allowing a goal to be found. We call this algorithm **CBS+TCC** (TAB with conditional constraints).

5.10.2. Theoretical Analysis

Theorem 5.10.1. *CBS+TCC is complete.*

Proof. First, no feasible solution is ever blocked by permanent constraints because they will never block a feasible solution per Lemma 5.3.1. Second, there are two cases to consider for any conditional constraint $c \in C_c$, where $C_c \subset C$ is the set of conditional constraints from N:

Case 1: *c blocks an action in a feasible solution.* If all feasible solutions are blocked, a conflict resulting from situation (1) or (2) will occur. In the case of (1), all conditional constraints are turned off immediately (including c), (lines 23,27) allowing a solution to be found. In the case of (2), if the probabilistic filtering (lines 17,18) does not turn off c at this stage, a new CT node will be created, increasing Δ_i. This situation may be repeated in subsequent CT nodes with increasing ρ_{off} until c is turned off. Because Δ_i is monotonically increasing, ρ_{off} will reach 1 after a finite number of steps, hence c is guaranteed to be turned off after a finite number of steps, (if a goal is not found in a different sub-tree of the CT first) allowing CBS to complete.

Case 2: *c blocks an action that causes a conflict.* If c is turned off before a goal is found, an agent may now be allowed to take an action which re-introduces a conflict into $N.\Pi$. In this case, either a goal node will be found in a different sub-tree, or the resulting conflict will eventually be detected in the sub-tree of N and a permanent constraint to avoid it will be created, allowing CBS to find a goal.

Eventually, in the worst case, all conditional constraints are turned off and the algorithm reduces to CBS+TAB which is guaranteed complete per Theorem 5.4.1. □

CBS+TCC can yield significant speed-ups over CBS+TAB because it preemptively blocks actions that are likely to lead to dead-ends in the CT, resulting in finding a feasible solution sooner. Optimality is not guaranteed because active conditional constraints may block an action in an optimal solution. An example is shown in Figure 5.18. The optimal solution as shown in (a) is unachievable when conditional constraints from Figure 5.17 are used for the problem. This is because action 5 is conditionally blocked for agent i and action 6 is permanently blocked for agent j after the first split, precluding at least one of the agents from taking an initial diagonal action. This is indicated by the 'x's on Figure 5.18(b). However, a sub-optimal, feasible solution such as in diagram (b) would be found by CBS+TCC immediately after the first split.

5.10.3. The Conflicting Paths Strategy

A more powerful blocking strategy called **CBS+TCP** (TAB with conflicting paths) blocks actions that conflict with the paths of all other agents (in addition to agents i and j). This technique has strong resemblances to prioritized planning algorithms [164, 189, 34, 33]. This is done during the CBS feasibility check routine. The first conflict encountered during the check is the *core conflict*. Mutually conflicting actions between agents i and j in the core conflict are blocked using permanent constraints (by computing the TAB for the core action pair). For every conflict between agent i or j and

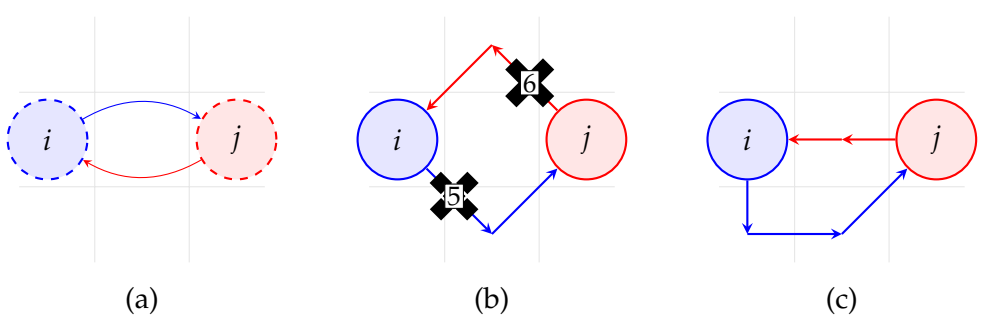


Figure 5.18.: Example showing (a) an example problem instance, (b) an optimal solution blocked by a conditional constraint and (c) a sub-optimal solution caused by the conditional constraint.

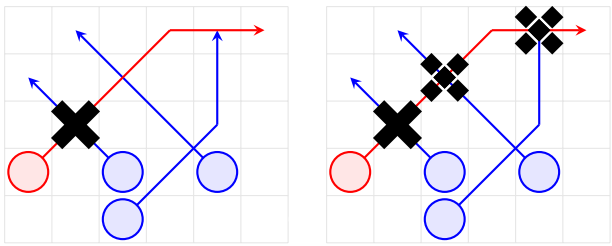
any other agent that is encountered thereafter, conditional constraints for all actions in the corresponding BCG are added to C_i or C_j.

Figure 5.19 (a) shows the regular constraint allocation strategy which adds permanent constraints for resolving only one conflict. This is indicated by the black 'x' over the collision area. Diagram (b) shows the CBS+TCP strategy which allocates extra conditional constraints for all conflicts beyond the core conflict as indicated by the dashed 'x's. With CBS+TCP, when agent i (resp. j) is re-planned as part of a split operation, it will attempt to avoid conflicts with all other agents (not just agent j). This technique can result in a significant performance improvement because of aggressive pruning high in the CT.

The same conditions for turning off conditional constraints in CBS+TCC are employed by CBS+TCP, hence it is complete but sub-optimal.

5.10.4. Empirical Results

We compare state-of-the-art, Greedy CBS (GCBS) [15], an unbounded, suboptimal variant of CBS using NC for prioritizing the OPEN list and time-range constraints with GCBS+TCC and GCBS+TCP which are GCBS with the new strategies discussed in Section 5.10. GCBS low-level prioritization on fewest conflicts with other agents is not performed because the CAT enhancement is not effective for 2^k neighborhoods with k of 3 and higher (see Section 3.5).



(a) (b)

Figure 5.19.: Illustration of (a) regular CBS constraint allocation and (b) allocation with the conflicting paths strategy.

Table 5.9.: Total problems solved in under 30 seconds

Type	Map	GCBS+Time		GCBS+TCC		GCBS+TCP	
		8	32	8	32	8	32
City	Berlin_1_256	2,473	1,121	4,413	2,970	**4,920**	3,068
	Boston_0_256	3,027	1,073	**6,021**	2,937	5,879	2,983
	Paris_1_256	2,833	1,153	6,115	3,011	**6,745**	3,027
DAO	brc202d	1,701	733	2,703	1,709	**3,385**	2,035
	den312d	849	451	1,885	1,919	**2,549**	2,457
	den520d	1,653	737	2,911	2,783	3,161	**3,235**
	lak303d	1,035	435	2,289	1,883	**2,635**	2,495
	orz900d	1,507	509	2,163	963	**2,393**	1,059
	ost003d	1,139	493	2,341	2,167	2,645	**2,687**
Dragon Age 2	ht_chantry	1,221	577	2,635	2,381	**3,327**	3,217
	ht_mansion_n	1,251	565	2,391	2,515	**2,847**	2,809
	lt_gallowstemplar	1,325	653	2,223	2,213	2,493	**2,497**
	w_woundedcoast	2,031	735	3,277	1,726	**3,853**	2,873
Open	empty-8-8	392	221	800	800	800	800
	empty-16-16	423	223	1,695	1,751	**2,147**	2,121
	empty-32-32	839	431	2,991	3,061	**3,765**	3,737
	empty-48-48	1,079	535	4,043	4,683	5,271	**5,833**
Open+ obstacles	random-32-32-10	689	381	2,787	2,723	3,283	**3,389**
	random-32-32-20	765	401	2,175	1,991	**2,657**	2,435
	random-64-64-10	1,261	595	4,869	5,105	5,891	**6,051**
	random-64-64-20	1,135	681	3,743	3,581	4,105	**4,185**
Maze	maze-32-32-2	447	261	999	917	1,103	**1,123**
	maze-32-32-4	339	222	651	631	**745**	665
	maze-128-128-10	981	435	1,793	1,711	**2,295**	2,155
	maze-128-128-2	601	301	1,117	1,025	**1,223**	1,171
Room	room-32-32-4	489	256	1,205	1,135	**1,395**	1,349
	room-64-64-16	709	317	1,503	1,465	**1,909**	1,677
	room-64-64-8	419	230	1,055	1,005	**1,163**	1,092

Table 5.9 shows results for the same set of benchmark problems. GCBS+TCP consistently outperforms the other variants. The improvement over GCBS is significant, up to 5×. Figure 5.20 shows success rate for a subset of the benchmark problems.

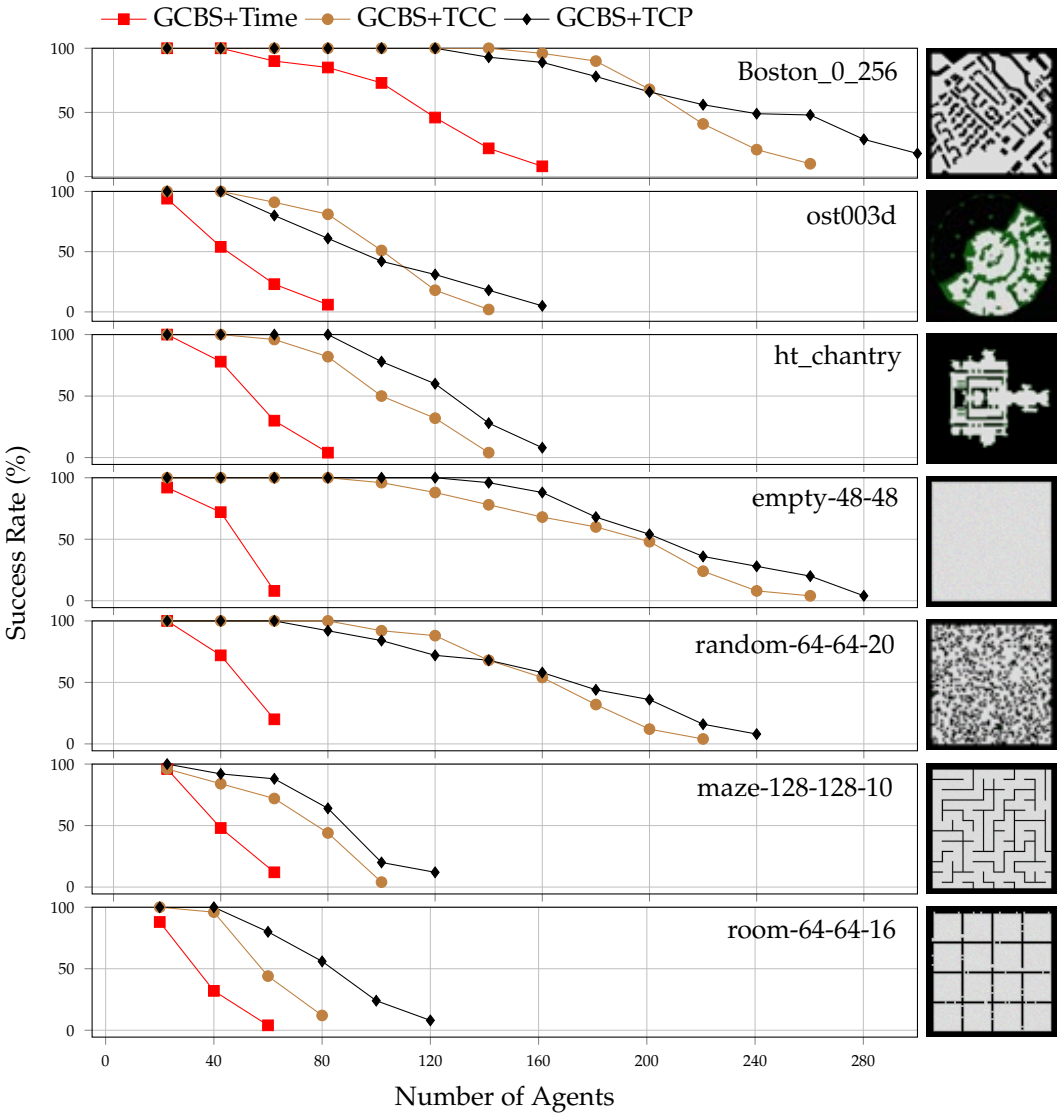


Figure 5.20.: Success rate of sub-optimal variants

Table 5.10.: Comparison of solution quality on 4- and 16-neighbor grids

	Optimal		Complete			
Configuration	CBS[4]	CBS[16]	GCBS[4]	GCBS[16]	GCBS+TCC[16]	GCBS+TCP[16]
Empty 8x8 (25 agents)	116	77 (67%)	132 (114%)	105 (91%)	107 (92%)	107 (92%)
Empty 64x64 (100 agents)	4,277	3,353 (78%)	4,283 (>100%)	3,355 (78%)	3,358 (79%)	3,358 (79%)
den520d (50 agents)	9,025	7,266 (81%)	9,028 (>100%)	7269 (81%)	7292 (81%)	7321 (81%)
brc202d (50 agents)	21,072	18,894 (90%)	21,090 (>100%)	18,899 (90%)	18,980 (90%)	18,922 (90%)
ost003d (50 agents)	7,889	6,148 (78%)	7,899 (>100%)	6,154 (78%)	6,293 (80%)	6,182 (78%)

GCBS+TCP is the strongest overall, with its most significant gains in maps with wide open spaces.

Table 5.10 shows mean solution costs where CBS[4], and GCBS[4] are for 4-neighbor grids, and CBS[16], GCBS[16], GCBS+TCC[16] and GCBS+TCP[16] are for 16-neighbor grids. The solution quality compared to optimal costs in 4-neighbor grids (the underlined values) is shown next to each statistic as a percentage in parenthesis. Solutions in 8x8 grids show the highest percentages of sub-optimality. This is due to the high agent density. Both strategies do not significantly degrade the overall solution quality when compared to GCBS[16], usually 1% of optimality or less. GCBS+TCP, which shows a significant speedup over GCBS+TCC, does not show any significant degradation in solution quality.

Path quality in 16-neighbor grids is better than for 4-neighbor grids [151], and this phenomenon is reproduced here – CBS[16] consistently yields higher quality solutions than CBS[4], and all sub-optimal variants consistently report better solution quality than CBS[4]. This is a key highlight because it means that if sub-optimal results are acceptable, when given a choice between a low-fidelity, unit-cost movement model and a higher-fidelity non-unit cost movement model, a higher fidelity model can yield both higher quality solutions *and* better runtime performance by using sub-optimal variants.

5.11. Theoretical Analysis of CBS for General MAPF

5.11.1. Computational Complexity

The computational complexity of the CBS high-level search for Classic MAPF is equivalent to the size of the CT at the time of termination which is $O(2^{k^2 {\mu_*}^3})$ [65] where k is the number of agents and μ_* is the makespan of the lowest-cost solution. The term ${\mu_*}^3$ represents the upper bound on the size of an MDD for a path of cost μ_* in 4-neighbor grids. The rationale behind using the MDD size in this equation is

that the total number of grid cell, time-step combinations in the path of an agent can be no larger than its MDD of makespan cost. Furthermore, two agents can not have more conflicts than the number of vertices and edges in one of the two agent's MDDs. There are $\binom{k}{2}$ pairs of agents that can have conflicts, hence the k^2 term. The low-level complexity is $O(\mu_* n)$ where n is the number of vertices in G. Normally a single-agent solver like A* would have a complexity of $O(n)$ because with duplicate detection, each vertex would only be expanded once in the worst case. However, because of the time component in the state space, A* may expand the same vertex (at different time steps) $O(\mu_*)$ times.

We now introduce our novel analysis for the complexity of CBS with General MAPF. The size of an MDD varies depending on the specific MAPF instance. For 2^k neighborhoods, the size of an MDD is $O({\mu_*}^2(\mu_* - \mu_0)/r)$ where μ_* is the makespan (e.g., largest cost of a single agent in the final solution), μ_0 is the smallest cost of an agent in the final solution and r is the inverse resolution of time. E.g., r=100 for a resolution of $1/100$. Inside of a cost of μ_*, an agent cannot move more than μ_* from the start vertex in any direction, hence the ${\mu_*}^2$ term. Assuming a finite set of edge costs, an agent may arrive at its goal with a cost in the range $[\mu_0, \mu_*]$. Similarly, it may occupy any vertex within a radius of μ_* inside a maximum time range of $(\mu_* - \mu_0)$ and the number of unique costs possible in that range is $(\mu_* - \mu_0)r$.

Increasing the connectivity of the graph (e.g., from 2^3 to 2^4) increases the number of unique costs in which a given vertex can be reached - but that number cannot exceed $(\mu_* - \mu_0)r$. For example with μ_*=10 and μ_0=5 and r=100 a goal may be reached at a maximum of 500 unique times. After simplification, the bound is $O({\mu_*}^3 r)$, hence the overall complexity of CBS for General MAPF is $O(2^{{\mu_*}^3 r})$. This bound holds for general graphs embedded in a metric space, provided that time and cost are synonymous.

5.11.2. Sufficient Conditions for Completeness and Optimality

Rigorous proofs for optimality and completeness have been shown for the basic CBS algorithm [160] and Multi-Constraint CBS [110]. The proofs for General MAPF are identical. In this thesis, we only provide a proof sketch covering these proofs. This section utilizes the symbols and definitions set forth in Section 2.3.1.

We start with some additional definitions:

- N is a CT node.
- $N.C$ is the set of constraints associated with a CT node which includes all constraints from itself and all constraints from its ancestor nodes from N to the root.
- $\mathbf{\Pi}$ (bold Π) is a *set* of solutions.
- $\mathbf{\Pi}^+$ is a set of feasible solutions.
- $\mathbf{\Pi}^*$ is the set of all lowest-cost feasible solutions.

Assumption 5.11.1. Assume that the problem graph G is finite and that the minimum edge cost is positive constant bounded.

Assumption 5.11.2. Assume that the algorithm used at the low level is optimal and complete and will either terminate with a lowest-cost path that is consistent with all constraints in $N.C$, or will terminate with no solution in the case that none exists.

This assumption is required to ensure that the low-level will terminate.

Theorem 5.11.3. *(Completeness)*

If constraint sets in each split are mutually disjunctive (see Section 5.3.2), CBS is guaranteed to find a feasible solution if one exists, otherwise it may not terminate.

Proof sketch. This is a proof by induction.

Base Case: Any solution in $\mathbf{\Pi}^+$ is permitted at the root because it contains no constraints.

Induction Step: Adding constraints to child nodes N_1 and N_2 will result in one of two cases for N_1 and N_2:

1. The child node permits any $\Pi \subseteq \mathbf{\Pi}^+$ because, by definition, mutually disjunctive constraint sets cannot block any feasible path combination.

2. The constraint(s) added completely block the low-level from finding a path that is consistent with the constraints and therefore the node is pruned from the CT.

General Case: Any further splits will continue to ensure that no $\Pi \in \mathbf{\Pi}^+$ is precluded by constraints. Hence, because new constraints are systematically added precluding infeasible solutions, eventually some $\Pi \in \mathbf{\Pi}^+$ will be discovered which is consistent with $N.C$ (via a series of nodes following case 1 of the induction step). Otherwise, the algorithm will terminate with no solution where each leaf node in the CT is pruned following case 2 in the induction step in the case no solution exists. Or else (assuming no temporally-relative duplicate pruning is applied) the algorithm will run forever, but only in the case no solution exists. See Section 3.3.1 on temporally-relative duplicate pruning for details on why it could run forever.

Finally, CBS is guaranteed to make progress toward a goal (if one exists) because (1) the low-level is guaranteed to terminate per Assumption 5.11.2 and (2) each split will eliminate at least one single-agent action in each branch. Therefore, after a finite number of steps, all conflicts will eventually be eliminated by some $N.C$ and a feasible solution will be found. Furthermore, since the CBS OPEN list is ordered by flowtime and adding of constraints eventually results in a cost increase after a finite number of steps, CBS cannot get stuck continuously adding constraints to some infeasible branch – eventually the cost will increase and a node from a different branch will come to the top of the OPEN list, causing a branch that allows a feasible solution to be explored and the goal found (if one exists) after a finite number of steps.

Theorem 5.11.4. *(Optimality)*

CBS will find an optimal solution assuming one exists if the high-level OPEN list is sorted by f-cost and an admissible heuristic is used.

Proof sketch. By Theorem 5.11.3, CBS will find a feasible solution in $\mathbf{\Pi}^+$ assuming one exists. Because of the ordering of the OPEN list, lower-cost solutions will be considered strictly before higher-cost ones, ensuring that a lowest-cost feasible solution in $\mathbf{\Pi}^* \subseteq \mathbf{\Pi}^+$ is found before a sub-optimal one. A sub-optimal solution cannot be considered before an optimal one because the heuristic is admissible [69]. Thus after a finite number of steps CBS will reach an optimal solution.

5.12. Summary

We have provided a description and pseudocode for the CBS algorithm. We have explained various conflict types, the notion of Cardinal conflicts and conflict symmetries. We highlighted how conflicts differ in General MAPF domains and specific approaches to deal with them, using appropriate constraints in CBS. We have defined the notion of correctness for constraint sets in CBS and the motivation for generating large sets of constraints for CBS with examples.

We have introduced a novel approach to generating large sets of constraints that are guaranteed to be correct using analysis with *bicliques* and bipartite conflict graphs. Additionally, we provided a new technique for more powerful, time-aware constraint generation using *time-annotated bicliques*. The empirical tests show that biclique-based constraint generation methods dominate other classic approaches to constraint generation in terms of reducing the amount of work performed and runtime necessary to find solutions.

We provide a new algorithm for performing *mutex propagation* in General MAPF domains. This allows more effective symmetry breaking in the case of conflict symmetries. We discussed options for low-level search routines in General MAPF and provided novel analysis for adaptation of *high-level heuristics* for General MAPF domains. We empirically compare CBS with constraints built using mutex propagation and constraints built with bicliques and find that CBS with bicliques dominates generally, with exceptions for some specific cases.

Next, we explored two new approaches to sub-optimal CBS: *constraint layering* and *conditional constraints*. The former allows refinement of solutions which can make the algorithm faster. The latter works by adding extraneous constraints to avoid conflicts and relaxing them as necessary to avoid incompleteness. Our results show that conditional constraints can boost performance significantly without significantly degrading solution quality.

Finally, we provided new *theoretical analysis* for the computational complexity, completeness and optimality of CBS in General MAPF domains.

6. Conflict-Based Increasing Cost Search

In this chapter, we take a deeper look at conflict symmetries in the context of continuous-time and formulate a new algorithm: Conflict-Based Increasing Cost Search (CBICS) [200] which combines elements of Conflict-Based Search (CBS) [160] and Increasing Cost Tree Search (ICTS) [163]. CBICS has been shown to be an improvement over CBS and ICTS in many settings.

6.1. Introduction

The main idea of CBICS is that useful information is learned by analyzing the path costs of individual agents and the pairwise solution costs of pairs of agents. This information can be exploited to avoid unnecessary work. In Figure 6.1(a), the path costs of the agents are $x{=}2, y{=}2, z{=}2$ when conflicts between agents are ignored. To resolve the conflict between agents y and z, at location B2 at the first time step, agent y must wait for agent z to enter B2, or vice versa. This results in the path cost combinations $\langle y{=}2, z{=}3\rangle$ and $\langle y{=}3, z{=}2\rangle$. Hence, $y{+}z{=}5$ in either case. We refer to this sum $lb_{i,j}$ (for arbitrary agents i and j) as the *pairwise lower bound*, since there is no feasible solution whose sum is less than this bound.

Without the information that $lb_{y,z}{=}5$, $lb_{x,z}$ also appears to be 5 (agent z must wait for agent x or vice versa). However, given $lb_{y,z}{=}5$, it must be that $lb_{x,z}{=}6$: If we fix $y{=}2$ and $z{=}3$, the path cost for agent x must be $x{\geq}4$. For example, if agent y follows the path *left, left* and agent z follows the path *wait, up, up*, then the only lowest-cost path for agent x is *down, right, up, right*. In general, by fixing the path costs of some agents, we can make inferences about the lower bound path costs of other agents.

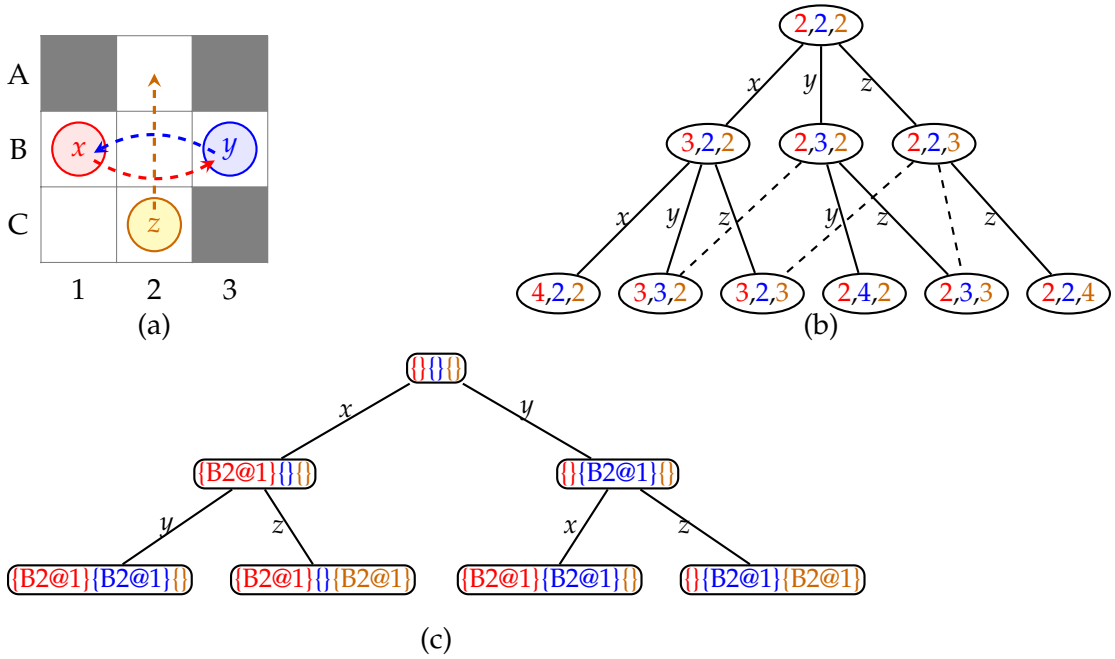


Figure 6.1.: (a) An example unit time MAPF instance, (b) a *partial* ICT with an implied δ of 1 for the MAPF instance and (c) a *partial* CT for the MAPF instance.

In CBICS we use *cost constraints* to fix the costs of certain agents. Cost constraints will be explained in more detail later. CBICS also uses *motion constraints* which are essentially the same as CBS "constraints" as discussed in Section 5.3. In order to clearly distinguish between the two types of constraints we use the terms "motion constraint" for constraints which constrict motion like CBS does, and "cost constraint" for constraints which restrict the upper-bound cost of an agent's path, similar to extended ICTS [198].

Consider the *partial* search trees for CBS (Figure 6.1(c)) and ICTS (Figure 6.1(b)). CBS adds only motion constraints at each depth of the CT, eventually resulting in enough cumulative motion constraints to eliminate infeasible path combinations. But little insight is gained at each depth in the CT. ICTS systematically increases the path cost for each agent, but does not learn that some subsets of path costs can *never* lead to a feasible solution.

CBICS gains insight about feasible path cost combinations and uses motion constraints *and* cost constraints in order to reduce the size of the search tree and find solutions more quickly.

6.2. CBICS High Level

CBICS searches the cost-range constraint tree (CRCT) as shown in Figure 6.2. A CRCT node is a tuple: $\langle R, M, LB, \Pi, SoC \rangle$ where $R=\{r_1, .., r_k\}$ is a set of path cost ranges for each agent where $r_i=[lb, ub)$, for example $r_i=[2, \infty)$. We use the shorthand 2+ for $[2, \infty)$ and 2 for $[2, 2]$. $M=\{M_1, .., M_k\}$ is a set of motion constraint sets for each agent. $LB=\{lb_{1,2}, lb_{1,3}, .., lb_{(k-1),k}\}$ is the set of lower bounds for the sum of path costs for all pairs of agents (pairwise path costs). Π is a solution. SoC is the sum-of-costs of all agents. For a CRCT node N we use the shorthand $N.r_i$ to refer to the cost range in $N.R$ for the ith agent, $N.M_i$ to mean the set of motion constraints in $N.M$ for the ith agent, $N.\pi_i$ to mean the path in $N.\Pi$ for the ith agent and $N.lb_{i,j}$ to mean the pairwise sum-of-costs lower bound for the ith and jth agents.

Each node in Figure 6.2 shows the path cost ranges R (in colors) and SoC or flowtime (in black parentheses) in the top row, pairwise cost lower bound information

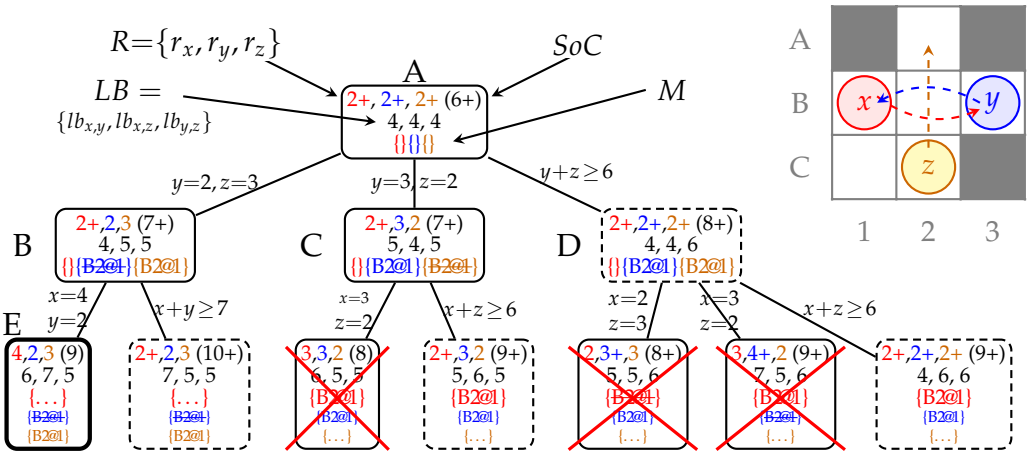


Figure 6.2.: The *entire* CRCT for the MAPF instance in Figure 6.1(a).

LB in the second row (in black) and motion constraint sets M in the remaining rows (in colors). When motion constraint sets contain more than one member, "..." is used to indicate this.

Recall that in Section 5.3 edge motion constraints $m \in N.M_i$ restrict agent i from performing one action at a specific time. A motion constraint is a tuple $m = \langle n, a \rangle$ where n is the agent number and $a = \langle s^t, s^{t+1} \rangle$ is the action for the agent to avoid. In this paper, the notation for motion constraints are in the format B1@0→B2@1 meaning the agent is prohibited from moving from location B1 at time step 0 and arriving at location B2 at time step 1. However, in Figure 6.2 motion constraints are shown with abbreviated notation like B2@1, which is shorthand for an action that ends at location B2 at time step 1. The start location can be inferred from the MAPF instance. The color indicates the agent number. When a motion constraint is struck through (e.g., ~~B2@1~~), it means that the constraint was conditionally removed. Conditional constraints are discussed later. The solution Π of each node is not shown.

Cost constraints restrict agents to paths with costs inside certain cost ranges. For $N.r_i = [lb, ub)$, $N.\pi_i$ is restricted to a path such that $c(N.\pi_i) \in [lb, ub)$. The pairwise path cost $N.lb_{i,j}$ similarly restricts paths based on the sum of two path costs. That is, for $N.lb_{i,j} = \ell$, $N.\pi_i$ and $N.\pi_j$ are restricted such that $c(\pi_i) + c(\pi_j) \geq \ell$.

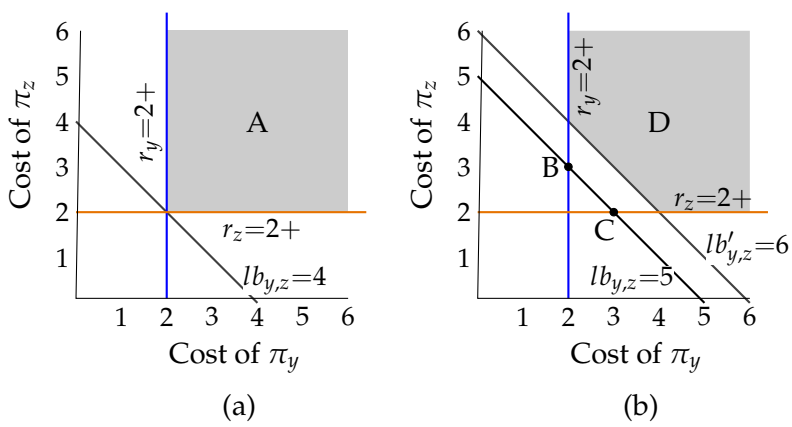


Figure 6.3.: (a) The combined cost range of agents y and z (a) for node A and (b) for nodes B, C and D of Figure 6.2.

The range of possible path costs for agents y and z for node A from Figure 6.2 is shown in Figure 6.3(a). The cost region (shown in gray) represents all values bounded by r_y=2+ represented by the blue line, r_z=2+ represented by the orange line, and all pairwise path costs $lb_{y,z}$=4 represented by the black line.

Pseudocode for CBICS is shown in Algorithm 6.1. On line 3, the root CRCT node N (e.g., node A in Figure 6.2) is constructed. $N.\Pi$ in the root node contains paths for all agents, planned for shortest paths without taking the other agents into account. Each $r_i \in N.R$ is set to $[c(\pi_i), \infty)$ respectively, SoC is $c(N.\Pi)$, $N.LB$ gets the sum of costs for each pair in $N.\Pi$ and all elements of $N.M$ are set to empty. CBICS checks for conflicts between the paths in Π (line 9). If no conflict is found, $N.\Pi$ is set as the new incumbent (line 11). This incumbent is needed due to the lazy evaluation of some nodes. Some nodes may have a cost increase after evaluation. The OPEN list is ordered by SoC. If no better solution exists in the OPEN list, the incumbent is returned as the solution (line 13). If a conflict is found, the *low-level* subroutine PAIRWISECONSTRAINTSEARCH is called for the two conflicting agents (generically, agent i and j) (line 16).

The PAIRWISECONSTRAINTSEARCH (PCS) which was initially discussed in Section 5.5.2 is an implementation of continuous-time mutex propagation which plans two conflicting agents (agents i and j) jointly to find a feasible solution and discover motion constraints and cost constraints at the same time. PCS takes as input two path cost constraints r_i and r_j, a pairwise cost constraint $lb_{i,j}$ (also known as the *current pairwise cost frontier*) and motion constraints M_i and M_j. Line 16 shows N as the input because N contains all of the information needed. The output is: (1) a pair of new motion constraint sets M'_i and M'_j, (2) a *set* of lowest-cost path pairs $P=\{\langle \pi_i, \pi_j \rangle_1, .., \langle \pi_i, \pi_j \rangle_n\}$ such that $\forall \langle \pi_i, \pi_j \rangle \in P$:

- π_i and π_j have no conflicts.
- π_i and π_j respectively conform to motion constraint sets M_i and M_j. (Paths do not violate any motion constraint inputs from the high-level.)

Algorithm 6.1. CBICS Algorithm

1: Input: a MAPF instance
2: $OPEN \leftarrow \emptyset$
3: Initialize the root node and add it to $OPEN$
4: $incumbent \leftarrow$ dummy with $SoC{=}\infty$
5: **while** $OPEN \neq \emptyset$ **do**
6: $\quad N \leftarrow OPEN.pop()$
7: $\quad$ **if** N has any empty $\pi_i \in N.\Pi$ **then**
8: $\quad\quad$ Re-plan each empty π_i with cost and motion constraints
9: $\quad A \leftarrow$ FINDCONFLICT$(N.\Pi)$ ▷ Find conflicting actions $\langle a_i, a_j \rangle$
10: $\quad$ **if** $A = \emptyset$ **then**
11: $\quad\quad incumbent \leftarrow N$ if $N.SoC < incumbent.SoC$
12: $\quad\quad$ **if** $incumbent.SoC \leq OPEN.top.SoC$ **then**
13: $\quad\quad\quad$ **return** $incumbent$
14: $\quad$ **else**
15: $\quad\quad P, M'_i, M'_j, lb'_{i,j} \leftarrow$ PAIRWISECONSTRAINTSEARCH$(start_i, start_j, goal_i, goal_j, N.M_i,$
16: $\quad\quad\quad N.M_j, N.r_i, N.r_j, N.lb_{i,j})$
17: $\quad\quad$ **if** $M'_i \neq \emptyset \vee M'_j \neq \emptyset$ **then** ▷ Conjunctive split
18: $\quad\quad\quad$ **for** $\langle \pi'_i, \pi'_j \rangle \in P$ **do** ▷ Nodes with pairwise costs $= lb_{i,j}$
19: $\quad\quad\quad\quad N' \leftarrow N$
20: $\quad\quad\quad\quad N'.r_i \leftarrow [c(\pi'_i), c(\pi'_i)]$
21: $\quad\quad\quad\quad N'.r_j \leftarrow [c(\pi'_j), c(\pi'_j)]$
22: $\quad\quad\quad\quad N'.lb_{i,j} \leftarrow c(\pi'_i) + c(\pi'_j)$
23: $\quad\quad\quad\quad N'.SoC \leftarrow N.SoC - c(N.lb_{i,j}) + N'.lb_{i,j}$
24: $\quad\quad\quad\quad N'.M_i \leftarrow N'.M_i \cup M'_i$ ▷ Cost-cond. constraints
25: $\quad\quad\quad\quad N'.M_j \leftarrow N'.M_j \cup M'_j$
26: $\quad\quad\quad\quad N'.\pi_i \leftarrow \pi'_i$
27: $\quad\quad\quad\quad N'.\pi_j \leftarrow \pi'_j$
28: $\quad\quad\quad\quad OPEN \leftarrow OPEN \cup N'$
29: $\quad\quad\quad N' \leftarrow N$ ▷ Node for path costs at and above the next frontier
30: $\quad\quad\quad N'.lb_{i,j} \leftarrow lb'_{i,j}$
31: $\quad\quad\quad N'.SoC \leftarrow N.SoC - c(N.lb_{i,j}) + N'.lb_{i,j}$
32: $\quad\quad\quad N'.M_i \leftarrow N'.M_i \cup M'_i$ ▷ Cost-cond. constraints
33: $\quad\quad\quad N'.M_j \leftarrow N'.M_j \cup M'_j$
34: $\quad\quad\quad N'.\pi_i \leftarrow \emptyset$ ▷ Will be replanned lazily on line 8
35: $\quad\quad\quad N'.\pi_j \leftarrow \emptyset$
36: $\quad\quad\quad OPEN \leftarrow OPEN \cup N'$
37: $\quad\quad$ **else** ▷ Disjunctive, CBS-style split for conflicting agents
38: $\quad\quad\quad N' \leftarrow N$ ▷ Create node for agent i
39: $\quad\quad\quad N'.r_i \leftarrow [\text{MIN}_{\pi'_i \in P} c(\pi'_i), \infty)$
40: $\quad\quad\quad N'.lb_{i,j} \leftarrow c(P_0.\pi'_i) + c(P_0.\pi'_j)$ ▷ P_0 is the first pair in P
41: $\quad\quad\quad N'.SoC \leftarrow N.SoC - c(N.\pi_i) + c(\pi'_i)$
42: $\quad\quad\quad N'.M_i \leftarrow N'.M_i \cup \{\langle i, a_i \rangle\}$ ▷ Regular motion constraint
43: $\quad\quad\quad N'.\pi_i \leftarrow \emptyset$ ▷ Will be replanned lazily on line 8
44: $\quad\quad\quad N'.SoC \leftarrow N.SoC - c(N.\pi_i) + c(\pi'_i)$
45: $\quad\quad\quad OPEN \leftarrow OPEN \cup N'$
46: $\quad\quad\quad N' \leftarrow N$ ▷ Create node for agent j
47: $\quad\quad\quad N'.r_j \leftarrow [\text{MIN}_{\pi'_j \in P} c(\pi'_j), \infty)$
48: $\quad\quad\quad N'.lb_{i,j} \leftarrow c(P_0.\pi'_j) + c(P_0.\pi'_j)$ ▷ P_0 is the first pair in P
49: $\quad\quad\quad N'.M_j \leftarrow N'.M_j \cup \{\langle j, a_j \rangle\}$ ▷ Regular motion constraint
50: $\quad\quad\quad N'.\pi_j \leftarrow \emptyset$ ▷ Will be replanned lazily on line 8
51: $\quad\quad\quad N'.SoC \leftarrow N.SoC - c(N.\pi_i) + c(\pi'_i)$
52: $\quad\quad\quad OPEN \leftarrow OPEN \cup N'$)
53: **return** "No solution"

- $c(\pi_i)+c(\pi_j)\geq lb_{i,j}$ (The sum of each pair of path costs conforms to the pairwise liwer bound cost constraint.)
- $c(\pi_i)\in r_i$ and $c(\pi_j)\in r_j$ (Each path cost conforms to the path cost constraints.)
- $\langle c(\pi_i), c(\pi_j)\rangle$ is unique in P. (P cannot contain more than one path pair with the same cost combination.)

and (3) $lb'_{i,j}$, the *next lowest pairwise cost frontier*. Further details of PCS are covered in Section 6.3.

For example, in Node A of Figure 6.2, based on the conflict between agents y and z, r_y=2+, r_z=2+, $lb_{y,z}$, $M_y=\emptyset$ and $M_z=\emptyset$ are passed to PCS in order to plan agents y and z jointly. PCS would then return two path pairs in P with path costs $\langle 2,3\rangle$ and $\langle 3,2\rangle$. These cost pairs correspond to points marked "B" and "C" in Figure 6.3(b). The resulting pairwise cost frontiers, $lb_{y,z}$=5 and $lb'_{y,z}$=6 are also shown as diagonal black lines. These two path pairs in P are the basis for generating the two child nodes B and C of node A in Figure 6.2. Additionally, $lb'_{y,z}$=6 is the basis for generating a third child, node D in Figure 6.2.

6.2.1. Motion Constraint Sets for Splitting

CBICS generates child nodes based on outputs from PCS. Recall from the discussion of CBS that during a split, agents i and j receive a set of new motion constraints M_i and M_j, respectively. This helps the agents to avoid a conflict. Completeness is ensured only when M_i and M_j are mutually disjunctive (See Section 5.3.2). In this chapter we also use the term *valid* to describe constraint sets that are mutually disjunctive.

There are two possibilities for PCS outputs M'_i, M'_j: (1) motion constraints are found for at least one of the two agents or (2) no motion constraints are found. In case (1) CBICS performs a *conjunctive split*. In case (2), CBICS performs a (CBS-style) *disjunctive split*.

6.2.2. Conjunctive Splitting

In a conjunctive split, the same motion constraint sets are applied to all child nodes. See lines 18-36 of Algorithm 6.1. A child node N' is created for each path pair π'_i, π'_j in P, where $N'.r_i$ and $N'.r_j$ are assigned the costs $c(\pi'_i)$ and $c(\pi'_j)$, respectively, the pairwise lower bound, $N'.lb_{i,j}$ gets $c(\pi'_i)+c(\pi'_j)$, $N'.\Pi$ is updated with π'_i and π'_j and $N'.M$ is updated with M'_i and M'_j (lines 18-28).

For example, node A of Figure 6.2 is split based on the conflict between agents y and z, producing two child nodes because PCS returned two path pairs in P with costs $\langle 2,3 \rangle$ and $\langle 3,2 \rangle$. Child nodes B and C take on *fixed* values for r_y and r_z (r_y=2, r_z=3 and r_y=3, r_z=2, respectively). Child B and C also get $lb_{y,z}$=5, and the same sets of constraints are added to both. CBICS also replaces $B.\pi_y$, $B.\pi_z$, $C.\pi_y$ and $C.\pi_z$ with the respective paths from P.

CBICS generates an additional node whose cost constraints represent the cost range based on the next pairwise cost frontier, $lb'_{i,j}$ (lines 29-36). Note that paths for this node are not filled in, but are lazily planned on line 8. The creation of the additional node is illustrated by node D, which is generated based on $lb'_{y,z}$=6, and also receives the same motion constraint sets as nodes B and C. The cost range for node D in Figure 6.2 is shown as the shaded region in Figure 6.3(b).

6.2.3. Disjunctive Splitting

Lines 37-52 show the steps for a disjunctive split. In a disjunctive split, only two nodes are created, one for each agent in conflict. Each node then gets motion constraints for one agent respectively. The cost constraints are not updated in a disjunctive split.

6.2.4. Cost-Conditional Motion Constraints

In order to ensure completeness, motion constraints used in a conjunctive split must be *cost-conditional* motion constraints. Recall that *conditional* motion constraints (See

Section 5.10.1) are similar to regular constraints, except that they can be turned off, meaning that they are omitted from the low-level search based on some criteria. The need for cost-conditional motion constraints in conjunctive splitting scenarios can be understood using the example in Figures 5.7(a) and (b). As discussed in Section 5.5, CBS with mutex propagation [218] constructs motion constraints for the mutexed MDD nodes shown with dashed outlines shown in Figure 5.7(a). Figure 5.7(b) shows the same analysis if the cost limit is increased to 3 for agent x. The MDD node B2 at time step 1 for agent z is no longer mutex with all MDD nodes for agent x, hence the motion constraints for agent x remain valid, but the motion constraints for agent z are no longer valid. This can be seen by comparing the nodes with dashed outlines between Figure 5.7 (a) and (b).

We therefore create motion constraints that are conditional on the cost of the other agents. The analysis in Figure 5.7(b) leads us to define a *cost-conditional* motion constraint $\langle n, a, ref, c \rangle$, where n is the agent being constrained, a is the action to be avoided, *ref* is the *reference agent* for the constraint, and c is the upper cost bound of the reference agent. Hence, the motion constraint is only valid when the cost of the reference agent is no greater than c, otherwise it is turned off. Referring back to Figure 5.7(b), a cost-conditional motion constraint $\langle n{=}x, a{=}B1@0{\rightarrow}B2@1, ref{=}z, c{=}2 \rangle$ would be returned by PCS.

6.2.5. High-Level Search

The CRCT is searched in a best-first fashion using the sum-of-costs as the primary sorting criterion. A heuristic such as WDG (See Section 5.7.1) should be used. Some nodes in the CRCT which are generated before node E (the goal node in Figure 6.2) are pruned quickly because the combination of costs is infeasible. These pruned nodes are shown with red 'X's in Figure 6.2. For example, the left child of node C is pruned because setting r_x=3 and r_y=3 makes conflict-free paths for agents x and y impossible.

The search continues in this fashion, finding the current pairwise cost frontier, creating child nodes for fixed costs on the frontier and one child node representing pairwise costs at and above the next lowest pairwise cost frontier.

Prior to calling PCS, CBICS checks each $m_i \in M_i$ and $m_j \in M_j$ against the upper bound of each $r \in R$. All m_i with $m_i.ref = j$ and $m_i.c < r_j.ub$ are removed from the constraint set (i.e., turned off). Then PCS is executed. For example, the analysis in Figure 5.7(a) creates four conditional motion constraints based on the actions terminating at dashed nodes.

1. $\langle n{=}x,\ a{=}B1{\rightarrow}B2@0,\ ref{=}z,\ c{=}2\rangle$
2. $\langle n{=}x,\ a{=}B2{\rightarrow}B3@1,\ ref{=}z,\ c{=}2\rangle$
3. $\langle n{=}z,\ a{=}C2{\rightarrow}B2@0,\ ref{=}x,\ c{=}2\rangle$
4. $\langle n{=}z,\ a{=}B2{\rightarrow}B3@1,\ ref{=}x,\ c{=}2\rangle$.

If PCS were called with parameters r_x=2 and r_z=3, constraints (1) and (2) would be omitted because r_z=3 is greater than c=2 for both of them.

The upper cost bound of a conditional motion constraint $m_i.c$ is valid iff it is mutually disjunctive with all of the conflicting agent (agent j)'s actions at the same time when agent j's path cost is less than or equal to $m_i.c$. Hence, we say cost-conditional motion constraint sets are *valid* iff no solution exists when both agents i and j violate any (non-turned off) motion constraint from M_i and M_j simultaneously. By *non-turned off*, we mean motion constraints m_i for which $m_i.c \leq r_j.ub$ where $r_j.ub$ is the upper bound of the cost range for agent j (and analogously for m_j).

In node E in Figure 6.2, because of the motion constraints and the cost constraints for agent y from node B, only one feasible solution for agents x and y exists with costs $\langle 4, 2\rangle$. This results in pushing the individual cost limit for agent x from 2+ (in node B) to 4 (in node E) and $lb_{x,y}$ from 4+ (in node B) to 6 (in node E). Because of the cost increase for agent x, we can infer that $lb_{x,z}$=7 (because r_x=4 and r_z=3).

In this instance, the goal node E is found sooner than would have been the case with either CBS or ICTS for two reasons: (1) CBICS can apply motion constraints to multiple agents at the same time. Hence, both agents y and z get motion constraints in nodes B, C and D where CBS would have applied them only to one agent. This often results in resolving more conflicts per node. (2) CBICS can increase the path costs of multiple agents at the same time, by specific amounts. ICTS would have increased the cost of only one agent and only by the fixed amount δ.

6.3. Pairwise Constraint Search

PCS plans two conflicting agents jointly to find feasible solutions and discover motion constraints and cost constraints at the same time. Discovering valid sets of motion constraints involves analysis of bipartite conflict graphs [199]. This is illustrated in Figure 5.8. As shown in Figures 5.1 and 5.2 it is not necessary to use vertex motion constraints – edge motion constraints capture enough information to make CBS and CBICS complete. For this reason, analysis for vertex motion constraints are omitted from this analysis for General MAPF instances.

Section 5.5.2 introduced analysis of biclique constraints and bi-connected bicliques as shown in Figure 5.8. This analysis is projected into the future using mutex propagation and novel *inherited mutexes* as shown in Figure 5.9.

Note that Figure 5.9 limits the MDDs to costs that make the problem infeasible as originally proposed for mutex propagation [218]. However, PCS continues to explore costs until all feasible cost combinations (hence all feasible MDD sets) are found. Figure 6.4 shows the next step, which would be carried out directly after the analysis in Figure 5.9 is complete. The cost limit of agent y is increased to 3.0, which allows for a feasible solution. [1] At the end of the process shown in Figure 6.4, motion constraint sets include $\{2, 4\}$ and $\{5, 7, 8, 11, 12, 14\}$. Note that the constraint set for agent y is reduced from $\{2, 3, 4\}$ from Figure 5.9(c.iii) to $\{2, 4\}$.

[1] Though it is not shown in Figure 6.4, PCS would also test increasing the cost limit of agent x separately.

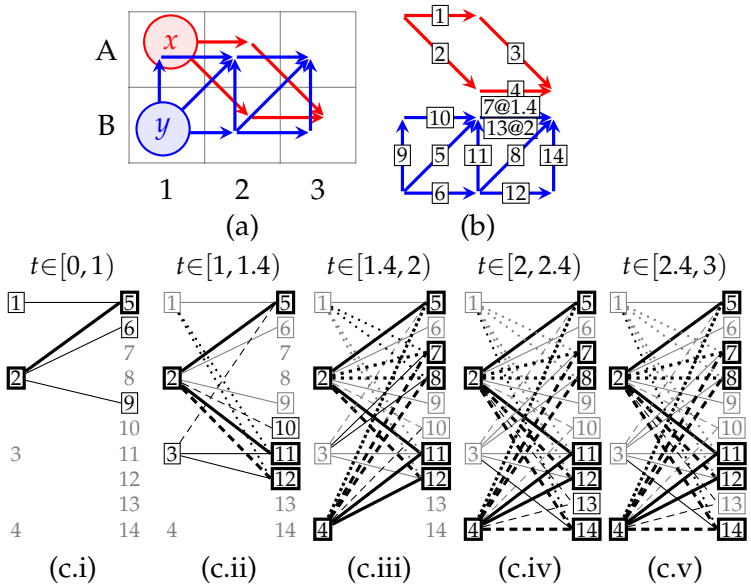


Figure 6.4.: An example of how PCS finds constraints. (a) Paths with a cost limit of 2.4 for agent x and 3.0 for agent y, (b) an enumeration of actions available to the agents including diagonal actions with wait actions omitted, (c) continuous-time mutex propagation.

If we take a close look at action number 2, we notice that, assuming agent y promises to never increase it's path cost above 3.0, we can always block agent x from taking action number 2. We find a similar, respective situation for action number 5. An important point to notice is that if neither agent increases its path cost above 3.0, action number 2 and action number 5 can be be blocked *conjunctively*. This is because action number 2 and 5 can never be contained in the paths of either agent if the cost constraints are honored. This situation is called *mutually cost-conjunctive*. The same can be said for all nodes in the BBC shown in Figure 6.4(c.v).

Formally, we say that two sets of motion constraints are **mutually cost-conjunctive** if, given individual cost constraints on agent's paths, no path for either agent can ever violate a motion constraint and be part of a feasible solution. This is in contrast to the mutually disjunctive property, in which one of the two agents can violate a motion constraint and still be part of a feasible solution. Mutually disjunctive is a subset of mutually cost-conjunctive, meaning all mutually cost-conjunctive sets are also mutu-

ally disjunctive, but not necessarily the other way around. Mutually cost-conjunctive constraint sets additionally guarantee resolution of conflict symmetries and may be used conjunctively with cost-conditional constraints.

Observation 6.3.1. *While the nodes of* any *biclique in a BCG are guaranteed to yield mutually disjunctive constraints, constraints generated from the nodes of a BBC are mutually cost-conjunctive. Hence, BBC-based constraint sets are strictly more efficient than biclique-based constraint sets.*

The result of applying the motion constraints (with the assumption of the cost restrictions of 2.4 and 3.0 for agents x and y respectively) shown in Figure 6.4 simultaneously is that the problem is resolved immediately – agent x is forced to take the path $[1, 3]$ and agent y is forced to take the path $[9, 10, 13]$. In contrast, using disjunctive constraints from Figure 5.9(c.iii) would result in an additional two levels of the high-level tree to be explored in the worst case.

The steps of mutex propagation as shown in Figures 5.9 and 6.4 are illustrative, however, the process is meticulous and computationally expensive. Instead, we present a faster approach to the PCS algorithm which does not explicitly build the MDDs, but accomplishes the same result with a single, integrated two-agent search with some bookkeeping.

The pseudocode for PCS shown in Algorithm 6.2 is based on A*, where the state space is the joint state space for the two agents. A state in the joint state space is $S=\{s_i, s_j\}$, where s_i and s_j are single-agent states. It respects motion constraints M_i and M_j for the agents during successor generation (line 5). It respects cost constraints by pruning successors (line 41). PCS terminates after all lowest cost combinations of solutions have been found and the first next-lowest cost solution is found (line 9) or when OPEN is empty.

During the successor generation phase (lines 4-9), successors are generated to maintain time overlap between the actions of the agents. *Time overlap* means that for $a_i=\langle s_i, s_i' \rangle$ and $a_j=\langle s_j, s_j' \rangle$, $s_i.t \in [s_j.t, s_j'.t]$ or $s_i'.t \in [s_j.t, s_j'.t]$. This is only needed for contin-

uous time [198]. Successor nodes are marked as infeasible if their parent is infeasible. This is similar to how MP checks the mutexes of predecessor MDD nodes for computing propagated mutexes. PCS performs mutex propagation, but it does this without the use of MDDs. Instead, it keeps a list of motion constraints, updating the upper cost bound $m'.c$ as necessary.

Consider the example instance in Figure 6.1(a). If PCS were planning for agents x and y, the root state would be S=$\langle s_x$=B1@0, s_y=B3@0, f=$(2,2)$, $feasible$=$true\rangle$, where s_x and s_y are the states of agents x and y and f is $f(s_x)$ and $f(s_y)$ respectively. $f(s_i)$ is the *f-cost* of s which is a lower-bounded estimate of the cost of a path from $start_i$ to $goal_i$ that passes through s_i. The successors of this joint state as produced by Algorithm 16, lines 4-9 would be:

1. $\langle s_x$=C1@1, s_y=B3@1, f=$4,3$, $feasible$=$true\rangle$
2. $\langle s_x$=C1@1, s_y=B2@1, f=$4,2$, $feasible$=$true\rangle$
3. $\langle s_x$=B1@1, s_y=B3@1, f=$3,3$, $feasible$=$true\rangle$
4. $\langle s_x$=B1@1, s_y=B2@1, f=$3,2$, $feasible$=$true\rangle$
5. $\langle s_x$=B2@1, s_y=B3@1, f=$2,3$, $feasible$=$true\rangle$
6. $\langle s_x$=B2@1, s_y=B2@1, f=$2,2$, $feasible$=$false\rangle$

Lines 15-18 will get an existing motion constraint (for updating) or add a new motion constraint (as an initial mutex) with an initial upper cost bound set to the f-cost of the opposing agent (line 18).

Continuing the example, five new cost-conditional motion constraints will be created from the successor nodes, whose upper cost bound $m'.c$ will be updated.

1. $\langle n$=x, a=B1@0$\rightarrow$B1@1, ref=y, c=$3\rangle$
2. $\langle n$=x, a=B1@0$\rightarrow$B2@1, ref=y, c=$3\rangle$
3. $\langle n$=x, a=B1@0$\rightarrow$C1@1, ref=y, c=$3\rangle$

Algorithm 6.2. Pairwise Constraint Search Algorithm

```
Input: start_i, start_j, goal_i, goal_j, M_i, M_j, r_i, r_j, lb_{i,j}
M_i ← M_i \ {∀m_i ∈ M_i; m_i.ref≠j ∨ m_i.c<r_j.ub}                ▷ "Turn off" conditional constraints
M_j ← M_j \ {∀m_j ∈ M_j; m_j.ref≠i ∨ m_j.c<r_i.ub}
OPEN ← ⟨start_i, start_j⟩
M'_i ← ∅, M'_j ← ∅, P ← ∅, B_i ← ∅, B_j ← ∅
while OPEN ≠ ∅ do
    S = ⟨s_i, s_j⟩ ← OPEN.pop()
    if S'.feasible ∧ S' is goal ∧
      P≠∅ ∧ f(s'_i)+f(s'_j) > all path costs in P then
        M'_i ← M'_i \ {∀m'_i∈M'_i : m'_i.c<r_j.lb ∨ m'_i.a ∉ B_i}
        M'_j ← M'_j \ {∀m'_j∈M'_j : m'_j.c<r_i.lb ∨ m'_j.a ∉ B_j}
        return P, M'_i, M'_j, lb'_{i,j}=f(s'_i)+f(s'_j)
    S' ← TIMEAWAREJOINTEXPANSION(S, M_i, M_j)                    ▷ See Algorithm 6.3
    for S' ∈ S' do
        if ∃m'_i ∈ M'_i such that m'_i.a = ⟨s_i, s'_i⟩ then
            M'_i ← M'_i \ m'_i
        else
            m'_i ← ⟨n=i, a=⟨s_i, s'_i⟩, ref=j, c=f(s'_j)⟩
        if S'.feasible = false then
            if not m'_i.costIsCapped then
                m'_i.c ← MAX(f(s'_j), m'_i.c)                    ▷ Increase range
        else
            if f(s'_j) ≤ m'_i.c then
                m'_i.c ← f(s'_j) − ε                             ▷ Decrease range permanently
                m'_i.costIsCapped ← true
        M'_i ← M'_i ∪ m'_i
        Analogously for s'_j
        if S' is goal ∧f(s'_i)≤r_i.ub ∧ f(s'_j)≤r_j.ub then
            if S'.feasible ∧ ⟨f(s'_i), f(s'_j)⟩ is unique in P∧
             f(s'_i)≥r_i.lb ∧ f(s'_j)≥r_j.lb ∧ f(s'_i)+f(s'_j)≥lb_{i,j} then
                P ← P ∪ path to S'                               ▷ Save unique-cost solutions
            else if ¬S'.feasible then
                while ¬S.feasible do                             ▷ Add to infeasible action sets
                    B_i ← ⟨S.s_i, s'_i⟩
                    B_j ← ⟨S.s_j, s'_j⟩
                    s'_i ← S.s_i
                    s'_j ← S.s_j
                    S ← S.parent
            if f(s'_i)<r_i.ub ∧ f(s'_j)<r_j.ub then
                OPEN ← OPEN ∪ S'                                 ▷ Add to open
        else if f(s'_i)≤r_i.ub ∧ f(s'_j)≤r_j.ub then
            if S'∈OPEN then                                      ▷ G-cost is included in duplicate check
                if ¬S'.feasible then
                    OPEN ← OPEN \ S'                             ▷ Replace with feasible joint state
            else
                OPEN ← OPEN ∪ S'                                 ▷ Add to open
return ∅, ∅, ∅, ∞                                                ▷ No solution
```

Algorithm 6.3. Joint Expansion with Temporally-Relative Duplicate Pruning

1: Input: $S = \langle s_i, s_j \rangle$, M_i, M_j
2: $t_{last_i} \leftarrow \underset{m \in M_i}{\text{MAX}}(m.a.s'.t)$ ▷ End time of latest constraint
3: $t_{last_j} \leftarrow \underset{m \in M_j}{\text{MAX}}(m.a.s'.t)$
4: **if** $s_i.time < s_j.time$ **then** ▷ Get successors for continuous time
5: $\quad \mathbf{s}'_i \leftarrow successors(s_i, M_i)$ ▷ Get successors which do not violate motion constraints
6: $\quad \mathbf{s}'_j \leftarrow \{s_j\}$
7: **else**
8: $\quad \mathbf{s}'_j \leftarrow successors(s_j, M_j)$ ▷ Get successors which do not violate motion constraints
9: $\quad \mathbf{s}'_i \leftarrow \{s_i\}$
10: $\mathbf{S}' \leftarrow \mathbf{s}'_i \times \mathbf{s}'_j$ ▷ Cartesian product
11: **for** $S' \in \mathbf{S'}$ **do** ▷ Mark feasibility
12: $\quad S'.feasible \leftarrow S.feasible$
13: $\quad$ **if** $s'_i \in S'$ conflicts with $s'_j \in S'$ **then**
14: $\quad\quad S'.feasible \leftarrow false$
15: **if** $\text{MAX}(s_i.t) > \text{MAX}(r_i.lb, t_{last_i}) \wedge \text{MAX}(s_j.t) > \text{MAX}(r_j.lb, t_{last_j}) \wedge$
16: $\quad f(s_i) > r_i.lb \wedge f(s_j) > r_j.lb \wedge f(s_i) + f(s_j) > lb_{i,j}$ **then**
17: $\quad t_{min} \leftarrow \underset{s \in S}{\text{MIN}}(s.t)$
18: $\quad \mathbf{S_\Delta} \leftarrow \emptyset$
19: $\quad$ **while** $S.parent \neq \emptyset$ **do**
20: $\quad\quad \mathbf{S_\Delta} \leftarrow \mathbf{S_\Delta} \cup \{\forall s \in S; (s.v, s.t - t_{min}\}$ ▷ Make times relative
21: $\quad\quad S \leftarrow S.parent$
22: $\quad$ **for** $S' \in \mathbf{S'}$ **do**
23: $\quad\quad S'_\Delta \leftarrow \{\forall s \in S; (s.v, s.t - t_{min}\}$ ▷ Make times relative
24: $\quad\quad$ **if** $S'_\Delta \in \mathbf{S_\Delta}$ **then**
25: $\quad\quad\quad \mathbf{S}' \leftarrow \mathbf{S}' \setminus S'$ ▷ Remove temporally-relative duplicates
26: **return** $\mathbf{S}'$

4. $\langle n{=}y, a{=}\text{B3@0}{\rightarrow}\text{B3@1}, ref{=}x, c{=}4\rangle$

5. $\langle n{=}y, a{=}\text{B3@0}{\rightarrow}\text{B2@1}, ref{=}x, c{=}4\rangle$

The logic for updating $m'.c$ occurs at lines 19-25. Consider what happens for motion constraint 1, listed above. Because agent x's action B1@0→B1@1 does not conflict with either of agent y's actions B3@0→B3@1 and B3@0→B2@1, the cost is capped so that it cannot grow (see line 25) and its upper cost limit, $m'.c$ is reduced, (after comparing to both states) to $2{-}\epsilon$ where 2 is the f-cost of agent y's B3@0→B2@1 action, and ϵ is a small constant. Now consider what happens for motion constraint 2 which is for agent x's action B1@0→B2@1. It does not conflict with agent y's action B3@0→B3@1, but it does conflict with B3@0→B2@1. Thus, the cost gets capped at $3{-}\epsilon$, (for the f-cost of B3@0→B3@1), but does not get decreased when checked versus B3@0→B2@1. If the actions had been checked in the reverse order, the result would be the same. By inspection, it is apparent that as long as agent y's cost does not go above $3{-}\epsilon$, agent x can never perform the action B1@0→B2@1 without conflict.

Subsequent expansions will either increase or decrease $m'.c$ values appropriately so that each $m'.c$ is the top of the continuous conflicting path cost range for the opposing agent. Note that the resulting $m'.c$ for some cost-conditional motion constraints will be less than the lowest possible path cost for the opposing agent, hence are unusable and must be omitted entirely (line 10). For example, $m'.c$ for motion constraint 1 is $2{-}\epsilon$ which is below the cost of a shortest path for agent y, and can be omitted from M_x.

In cases where heuristics that inform f-costs are not exact (without the additional logic described here), PCS could include motion constraints in M'_i and M'_j which block actions for paths with costs that are outside the cost bounds of r_i and r_j respectively. This would lead to incompleteness. In order to remove motion constraints which fall outside the cost bounds, PCS computes the sets of infeasible actions (literally $feasible{=}false$) B_i and B_j for agents i and j at lines 5 and 32-38. These sets include only infeasible actions included in infeasible paths to the goal that fall inside the cost

bounds. Finally, in addition to removing all m' where $m'.c$ is less than the lower cost bounds, we also remove any m' where $m'.a$ is not in the set of infeasible actions B_i and B_j respectively (lines 10 and 9).

In summary, PCS performs the mutex propagation process until it finds all feasible cost combinations, in order to determine mutually cost-conjunctive motion constraint sets. Additionally, it determines cost limits for all cost-conditional motion constraints in these sets. This is done using bookkeeping in a continuous A* search, instead of incrementally building MDDs as with traditional mutex propagation.

6.4. Sufficient Conditions for Completeness and Optimality

For proving optimality and completeness, we use the symbol definitions from Section 2.3.1. We also use the following definitions:

- $F(\mathbf{a}) = \left[\underset{a \in \mathbf{a}}{\text{MIN}} f(a),\ \underset{a \in \mathbf{a}}{\text{MAX}} f(a)\right]$ is the inclusive f-cost range of a set of actions.
- $\text{MAPF}_\text{Q} \subset \text{MAPF}_\text{R}$ is the subset of all instances of MAPF_R such that all edge weights are restricted to the set of non-negative rational numbers $\mathbb{Q}_+$. That is, $\forall e \in E$; $w(e) \in \mathbb{Q}_+$. This is a reasonable assumption since the most commonly used numerical representations in computing systems are limited to rational numbers.

Heuristic properties:

- $h(s)$ is considered to be an *admissible* heuristic if $\forall s\ h(s) \leq c^*(s, goal)$, otherwise it is *inadmissible*.
- An admissible heuristic $h(s)$ is *consistent* if $\{\forall (s, s'), (s.v, s'.v) \in E\}$, $h(s) \leq h(s') + c(s, s')$, otherwise it is *inconsistent*.
- For consistent heuristics, we define an *exactness* measure $e(h) = c^* - h$ which is the difference between h and a perfect heuristic. For a *perfect heuristic* $e(h) = 0$ and an uninformed heuristic (where $h = 0$) $e(h) = c^*$.

Based on these heuristic properties, we define MAPF instances which inherit all of the properties of MAPF_Q:

- **MAPF instance $I_{\text{L-ADM}}$**: This is an instance of MAPF_Q with admissible low-level/single-agent heuristics.
- **MAPF instance $I_{\text{L-IADM}}$**: This is an instance of MAPF_Q with one or more inadmissible low-level/single-agent heuristics.

In order to prove optimality and completeness for CBICS, we rely on the following properties of PCS. Proofs of correctness for each of these properties can be found in Appendix C.

1. PCS is guaranteed to return a set of *context-optimal* paths in P. Context optimal means that PCS returns a set of lowest-cost path pairs, which conform to motion and cost constraints.
2. PCS is guaranteed to terminate in the case that no solution exists.
3. PCS is guaranteed to compute valid sets of cost-conditional motion constraints.

We begin the proof of correctness of CBICS by showing with the following three lemmas that a path to a goal node in the CRCT can never be blocked due to cost-conditional motion constraints in conjunctive splits, permanent motion constraints in disjunctive splits, nor by cost bound changes.

Lemma 6.4.1. *Sets of valid cost-conditional motion constraints applied conjunctively never block a path to any goal node in the CRCT.*

Proof. Let π_i and π_j be any pair of non-conflicting paths for agents i and j. Also let M_i and M_j be sets of cost-conditional motion constraints for the agents. Each $m_i \in M_i$ has a cost upper-bound $m_i.c$ contingent on $c(\pi_j)$ and analogously for M_j. If π_i and π_j are non-conflicting, it follows that one of eight cases holds:

1. $c(\pi_j) \leq m_i.c$ and $c(\pi_i) \leq m_j.c$ and no motion constraints are violated
2. $c(\pi_j) > m_i.c$ and π_i violates a motion constraint in M_i
3. $c(\pi_i) > m_j.c$ and π_j violates a motion constraint in M_j
4. $c(\pi_j) > m_i.c$ and $c(\pi_i) > m_j.c$ and only one of π_i or π_j violates a motion constraint.
5. $c(\pi_j) \leq m_i.c$ and $c(\pi_i) \leq m_j.c$ and one or more motion constraints are violated
6. $c(\pi_j) > m_i.c$ and π_j violates a motion constraint in M_j
7. $c(\pi_i) > m_j.c$ and π_i violates a motion constraint in M_i
8. $c(\pi_j) > m_i.c$ and $c(\pi_i) > m_j.c$ and neither π_i or π_j violates a motion constraint.

In case 1 no cost-conditional motion constraints are turned off and no motion constraints are violated. In cases 2, 3 and 4 all violated motion constraints are turned off (see Algorithm 6.2 lines 2 and 3) due to the reference agent's increased cost, and thus are allowed. Cases 5, 6 and 7 can only occur if M_i and M_j contain *invalid* cost-conditional motion constraints and will never occur if PCS is implemented correctly. In case 8, no motion constraints are turned on and none are violated. In cases 1, 2, 3, 4 and 8 no feasible combination of paths is ever blocked, and cases 5, 6 and 7 violate the assumption of valid constraints and can never happen, therefore, no feasible solution is ever blocked. □

Lemma 6.4.2. *Sets of regular motion constraints applied disjunctively never block a path to any goal node in the CRCT.*

Proof. When constraints are applied disjunctively to resolve a conflict $\langle a_i, a_j \rangle$ in $N.\Pi$ between agent i and agent j, motion constraints M_i and M_j blocking only mutually-conflicting actions are applied to agent i in the first child node N_i and agent j in the second child node N_j (see Algorithm 6.1 lines 37-52). This means agent j is allowed to perform a_j in N_i because agent i is blocked from executing any action that might conflict with it and vice-versa.

Proof by induction. In this proof, we assume a feasible solution exists for the problem instance.

Base Case: A conflict in the root node is resolved by creating two child nodes. A feasible solution must lie in the sub-tree of N_i or N_j. By contradiction, if a feasible solution does not lie in either N_i or N_j, then the constraints added to N_i and N_j do not resolve a conflict. But this is impossible based on the assumptions about M_i and M_j.

Induction Step: A feasible solution lies in one of the two child nodes of N_i or one of the two child nodes of N_j.

General Case: Because a feasible solution exists, there must be a finite number of conflicts to be resolved. A feasible solution must lie down at least one path from the root to a leaf. If the addition of some set of motion constraints during a split results in blocking an agent from reaching its goal, there must be a node in a different sub-tree that allows the agent to reach its goal.

□

Lemma 6.4.3. *CBICS will never preclude a feasible cost combination from being explored.*

Proof. Proof by induction.

Base Case: The root node includes all possible feasible cost ranges because it includes the lowest feasible individual and pairwise costs for all agents and has unlimited upper bounds. See Figure 6.3(a).

Induction Step: Assuming PCS finds a solution and returns P and $lb'_{i,j}$ for some conflicting pair of agents i and j, the sum of the path costs for any pair of paths in P represents the current pairwise cost frontier $lb_{i,j}$. The original cost range from the base case is perfectly sub-divided, without missing any part of the cost range because CBICS will generate nodes for all cost combinations equal to $lb_{i,j}$ (Algorithm 6.1 lines 18-28). It will also generate a node for the inclusive unbounded cost area greater than or equal to the next pairwise cost frontier $lb'_{i,j}$ (lines 37-52). See Figure 6.3(b). In the case PCS does not find a solution, no child nodes are created, eliminating the infeasible cost combination from consideration.

By contradiction, assume some feasible pairwise cost $lb''_{i,j}$ between $lb_{i,j}$ and $lb'_{i,j}$ exists. This cannot happen because PCS is guaranteed to find lowest-cost solutions which respect the cost constraints per Lemma C.0.4.

General Case: When a node is expanded by CBICS, a new set of nodes for the next pairwise cost frontier and unbounded area above it will be generated, perfectly sub-dividing again. Therefore, no feasible individual or pairwise cost combinations are ever precluded from evaluation. □

We have shown that goal nodes cannot be blocked. Now, we continue by showing that making progress toward the goal is guaranteed.

Lemma 6.4.4. *Cost constraint lower bounds will increase after a finite number of steps.*

Proof. First, we show that cost constraint lower bounds cannot decrease, then we show that the cost constraint lower bounds are guaranteed to increase after a finite number of steps.

Let N be a CRCT node with individual cost constraints $r_1, .., r_k$ and pairwise cost constraints $lb_{1,2}, lb_{1,3}, .., lb_{(k-1),k}$ for each pair of agents. After conflict detection is performed (Algorithm 6.1 line 9) PCS is invoked (line 16). Per Algorithm 6.2 line 30 PCS cannot return any solution with costs less than $r_i.lb, r_j.lb$ and $lb_{i,j}$. Thus there are three possible outcomes:

1. no solution is found by PCS such that $c(\pi_i) \in r_i$, $c(\pi_j) \in r_j$ and $c(\pi_i) + c(\pi_j) \geq lb_{i,j}$
2. there is no increase in any $c(\pi_i)$, $c(\pi_j)$ or $c(\pi_i) + c(\pi_j)$ for $\langle \pi_i, \pi_j \rangle \in P$
3. at least one of $c(\pi_i)$, $c(\pi_j)$ or $c(\pi_i) + c(\pi_j)$ is increased.

No cost decrease with respect to N is possible in any case, hence each child node of N can never have decreased cost constraint lower bounds (see lines 20-22, 30, 39 and 41).

Now we show that cost will increase after a finite number of steps. In case 3, the cost constraint lower bounds clearly increase, however, we must show that case 2 cannot recur indefinitely. With each split, either cost-conditional motion constraints are

added to all child nodes (see Algorithm 6.1 lines 18-28) or regular motion constraints are added disjunctively to the child nodes (see Algorithm 6.1 lines 37-52).

Though many shortest paths may exist for a single agent, there is only one cost of a shortest path. Because there is a finite single-agent branching factor, the number of equal-cost shortest paths is finite. After a split, motion constraints are always added which block one or more actions on a shortest path. In the case of a disjunctive split, the motion constraints are based on conflicts in shortest paths in $N.\Pi$. In the case of a conjunctive split, PCS is guaranteed to have selected motion constraints in shortest paths because OPEN is ordered by g-cost. If no motion constraints are found by PCS, a disjunctive split is performed instead, hence constraints blocking actions in shortest paths are always added to child nodes.

Because there are a finite number of shortest paths of particular cost, and because motion constraints are added cumulatively, eventually all shortest paths will be blocked and the cost will increase.

In the case of a conjunctive split, conditional constraints remain "turned on" unless the reference agent's cost increases. So either constraints accumulate to increase the cost of agent i or the cost of agent j increases. In either case, the cost increases for at least one agent. □

Theorem 6.4.5. *CBICS is optimal and complete.*

Proof. Lemma 6.4.1 and Lemma 6.4.2 show that neither conjunctive nor disjunctive splits ever block a feasible solution. Lemmas 6.4.3 and 6.4.4 show that no feasible cost combination is ever precluded from consideration and cost constraints will increase after a finite number of steps. Hence, in cases where PCS returns increased costs or new motion constraints, CBICS will progress toward the goal - either the cost goes up, or invalid motion combinations at the current cost level are resolved by motion constraints.

Finally, optimality is ensured by prioritizing the OPEN list by the sum-of-costs. Because under some circumstances paths are re-planned lazily (Algorithm 6.1 line 8),

an incumbent is stored as soon as a feasible solution is found and a solution is accepted as soon as no better solution (in terms of sum-of-costs) exists in OPEN. Because CBICS guarantees no feasible solution is ever precluded from being found, the use of the prioritized OPEN list ensures that no sub-optimal solution is ever returned. □

6.4.1. Additional Enhancements

Some additional enhancements make CBICS more efficient. Instead of letting it dynamically find the pairwise costs and cost-conditional motion constraints during the search, it is possible to run a preprocessing step to discover the costs and/or motion constraints at the root node (see line 3 of Algorithm 6.1).

The first variant is called the preprocessing-cost (PCST) enhancement. This preprocessing step first performs a conflict check between all initial paths. Any pairs of paths that have a conflict are then planned together with PCS, but with mutex propagation turned off so that invalid joint states are discarded instead of propagated. This allows PCS to find the pairwise costs faster. The pairwise costs are then updated in the root node. This preprocessing can help achieve a more accurate heuristic for nodes early in the search, and may help prune nodes based on cost. However, this preprocessing step can be computationally costly – especially when a large number of conflicting agent pairs exist.

Another variant is called the preprocessing-constraints (PCON) enhancement. This preprocessing enhancement is similar to PCST, except PCS (with mutex propagation turned on) is run for all conflicting agent pairs. The pairwise costs are updated in the root node just like PCST. Additionally, *all* cost-conditional motion constraints found for any agents are added to the root node. This can result in more constrained paths for agents high in the tree, which pushes up costs faster, leading to more accurate heuristics and pruning. PCON is even more computationally expensive than PCST, and the extra time spent performing mutex propagation may, or may not result in finding motion constraints.

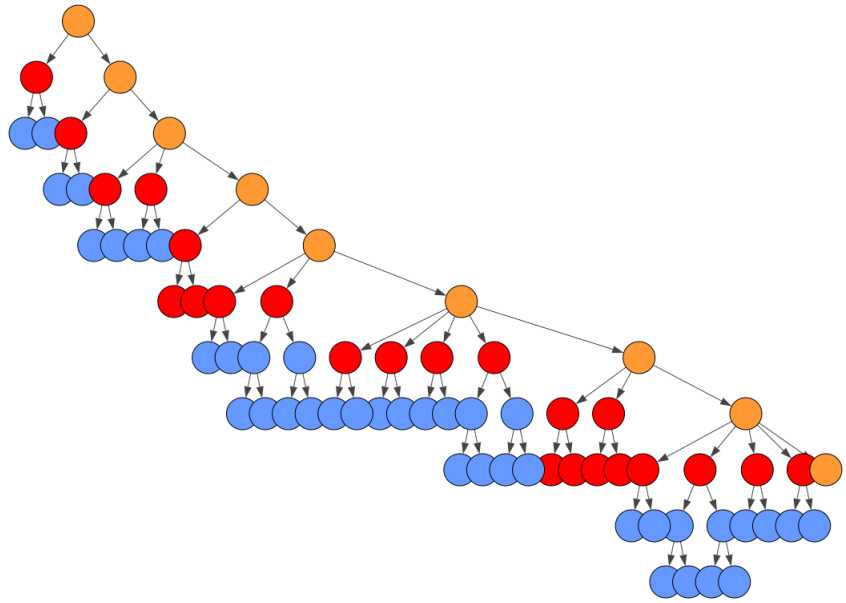


Figure 6.5.: Structure of a CBICS tree.

In some scenarios, $|P|$, the number of unique-cost path combinations found by PCS can be large, which for conjunctive splits, results in a large branching factor at the high level. In order to keep the branching factor small, some of the cost combinations can be combined into a single node. Of all approaches we tried, the best approach is to create four nodes from P: (1) a node with the minimum cost for r_i and maximum cost for r_j. (2) a node with the maximum cost for r_i and minimum cost for r_j. (3) a node that combines all costs in between (but not equal to) the minimum and maximum costs. (4) the node for where the sum of costs is at least $lb'_{i,j}$. For example, with the following cost combinations: $\langle 2,5\rangle$, $\langle 3,4\rangle$, $\langle 4,3\rangle$, $\langle 5,2\rangle$, four nodes with the following path cost ranges would be generated: $\langle r_i{=}2, r_j{=}5\rangle$, $\langle r_i{=}[3,4], r_j{=}[3,4]\rangle$, $\langle r_i{=}5, r_j{=}2\rangle$, $\langle r_i{=}2{+}, r_j{=}2{+}\rangle$. This is called the *combination* enhancement.

6.4.2. Analysis

Figure 6.5 shows an example of a typical CBICS tree. Orange nodes represent conjunctive nodes with an unlimited upper cost bound (unlimited nodes), red nodes

represent conjunctive nodes with finite cost bounds (limited nodes), and blue nodes represent disjunctive nodes. The sub-trees of limited nodes tend to be much smaller than the right sub-trees and are often searched to completion – eliminating the sub-trees from consideration before moving on to expanding unlimited nodes. In practice, conjunctive splits occur most often at unlimited nodes. This is because lower bounds on individual costs are rarely increased for unlimited nodes and, when the costs are low, PCS (and mutex propagation in general) is more likely to find motion constraints. Conversely, as costs increase, PCS is less likely to find motion constraints and disjunctive splitting is used. Often, the pattern results in the sub-trees of limited nodes resembling a regular CBS tree. However, occasionally a sufficient number of disjunctive motion constraints are added in a sub-tree to allow PCS to find motion constraints, triggering a conjunctive split.

When compared to CBS, which has a branching factor of two, CBICS has a larger branching factor on average. With the combination enhancement, the branching factor can be no larger than four. However, unlike CBS, nodes in CBICS have cost constraints which tend to eliminate a significant proportion of the sub-trees quickly. Thus, while CBS and ICTS tend to build fuller binary and k-ary-like trees respectively, CBICS tends to build an unbalanced search tree.

The branching factor of the joint state space for PCS is $b{=}(b_{base})^2$, where b_{base} is the single-agent branching factor. The depth of the solution Δ, can be no larger than $\text{MAX}(d_i, d_j)$ where d_i and d_j are the lengths (number of states) in π_i and π_j in the unit time case. Δ can be no larger than $d_i{+}d_j$ in the continuous time case because of a phenomenon related to operator decomposition [171]. Hence, the computational complexity of PCS is no worse than for regular mutex propagation, namely $O(b^{\Delta})$ in unit time domains, and $O(b^{2\Delta})$ for continuous time domains. This means that the complexity of PCS is exponential in the worst case (in Δ), meaning, running PCS for long paths (e.g., in large maps) can take significantly longer to complete. However, a strong low-level heuristic can reduce the complexity significantly.

6.5. Empirical Results

First we analyze enhancements to CBICS, then compare it to CBS. All experiments were run using an Intel i9 processor at 2.4GHz. The implementation is publicly available[2].

The experiments come from the MAPF benchmarks set [173] which includes various grid maps and MAPF instances with randomly selected start and goal vertices for agents. There are 25 problem instances for each map. The benchmarks were run by starting with five agents, incrementally increasing the number of agents and recording the runtime and number of instances that were solvable within 30 seconds. Any MAPF instance that was not solved within this time limit was marked as a failure.

All MAPF instances were solved for circular agents with a radius of $1/(2\sqrt{2})$ cells, a fixed wait time of 1, the sum-of-costs objective, and agents do not disappear at their goals, but will block others when waiting at their goal. We have run experiments on 4-, 8- and 16-neighbor grid maps which are 2^k neighborhoods [151]. The 4-neighborhood is the Classic MAPF domain. 8- and 16-neighborhoods are General MAPF domains because the edge lengths and weights are not uniform. All experiments, including for Classic MAPF, were run on an implementation which has collision detection, the conflict avoidance table and other aspects optimized for continuous time. This causes differences in our results versus MAPF solvers which are optimized for Classic MAPF.

6.5.1. CBICS Enhancements

Figures 6.6, 6.7 and 6.8, show results for 4-, 8-, and 16-neighbor grids on various benchmark instances. The y-axis shows the success rate where 100% means all 25 instances were solved within the time limit and 0% means none were solved inside the time limit. The x-axis shows the number of agents in the problem instance. As the number of agents is increased, the problem instance is exponentially harder to solve in terms of worst-case computational complexity.

[2]https://github.com/thaynewalker/hog2/tree/id/apps/CBICS

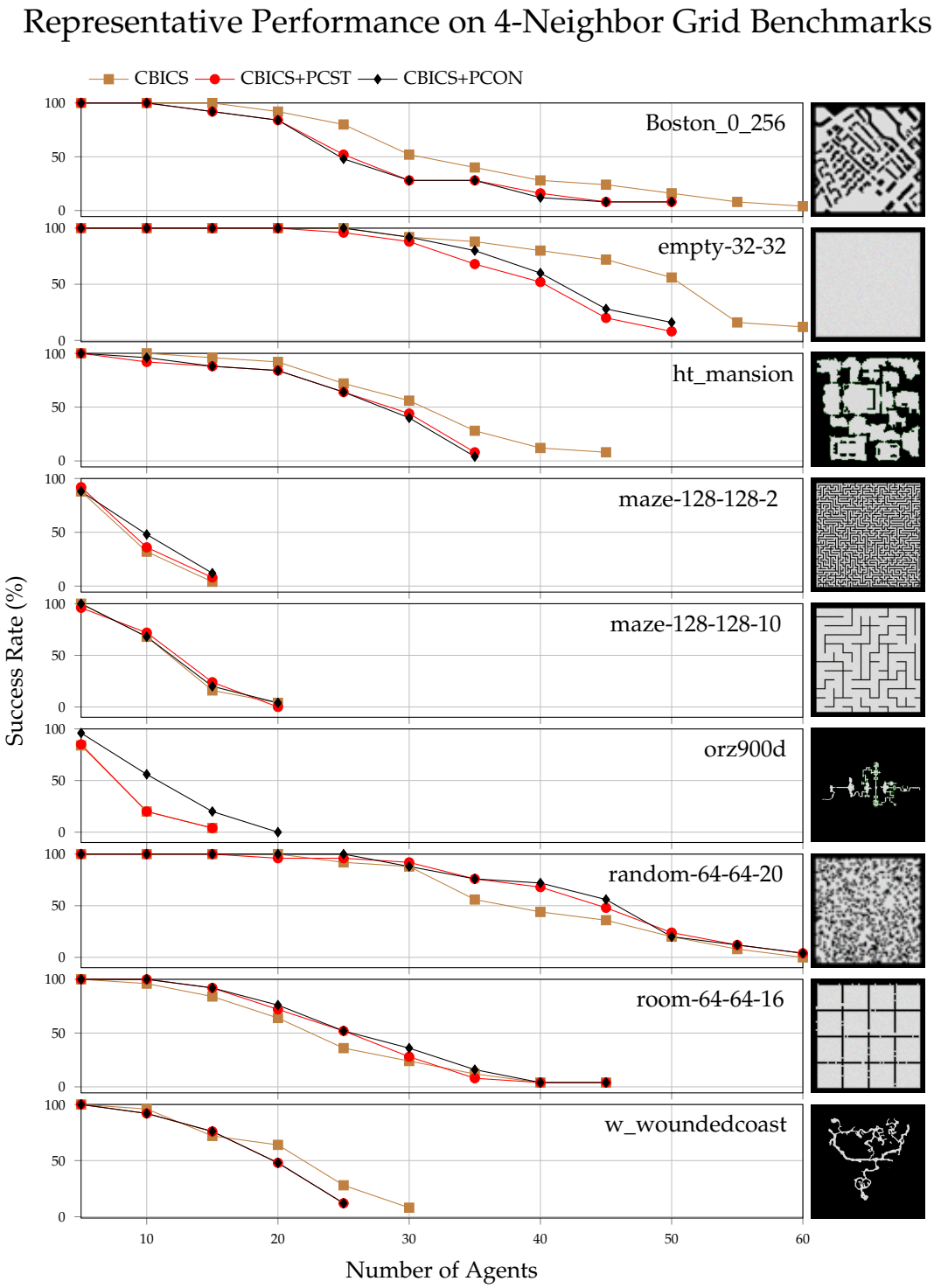


Figure 6.6.: Success rates of CBICS, CBICS+PCST and CBICS+PCON of CBICS for benchmark problems on 4-neighbor grids.

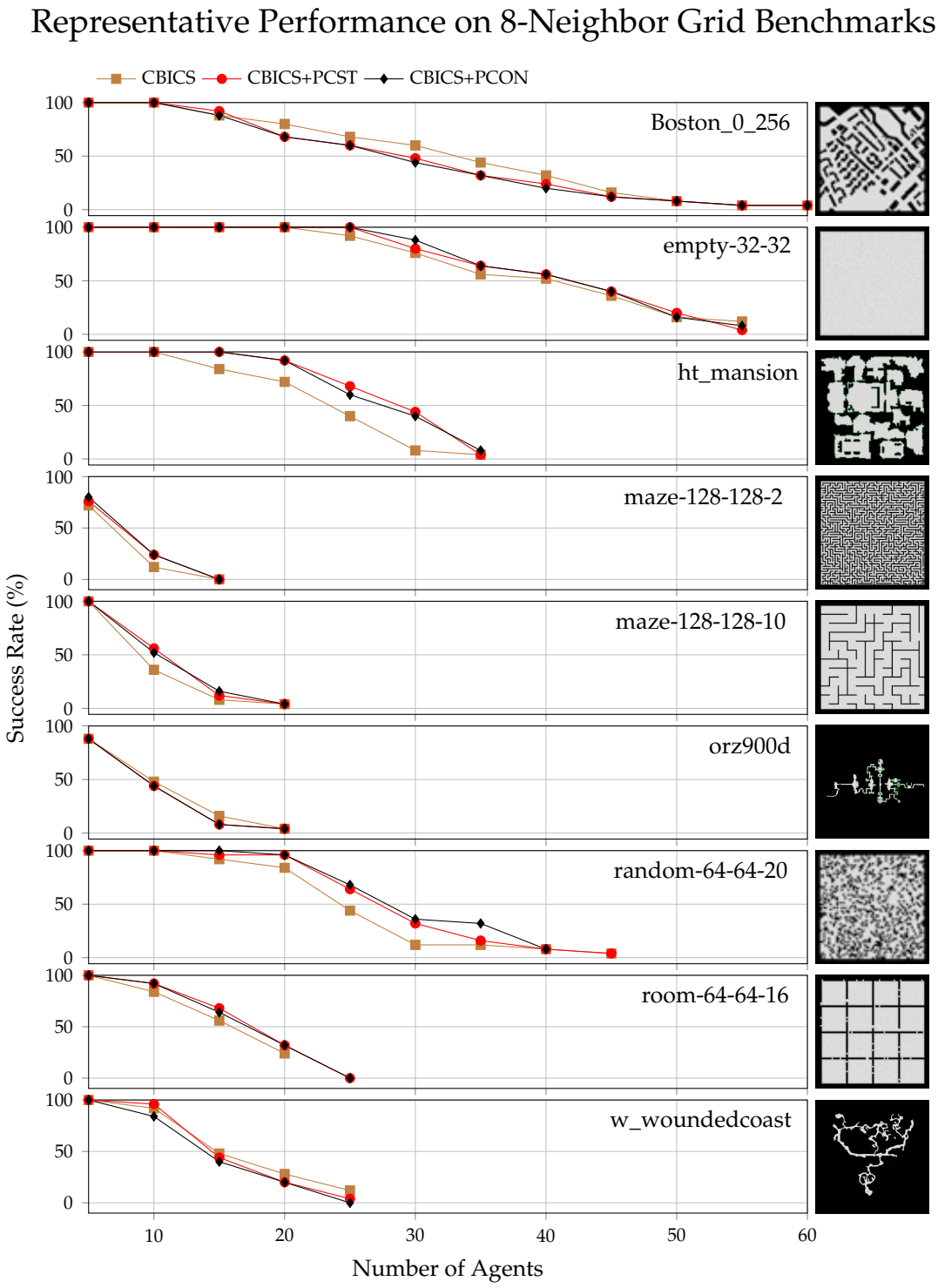


Figure 6.7.: Success rates of CBICS, CBICS+PCST and CBICS+PCON for benchmark problems on 8-neighbor grids.

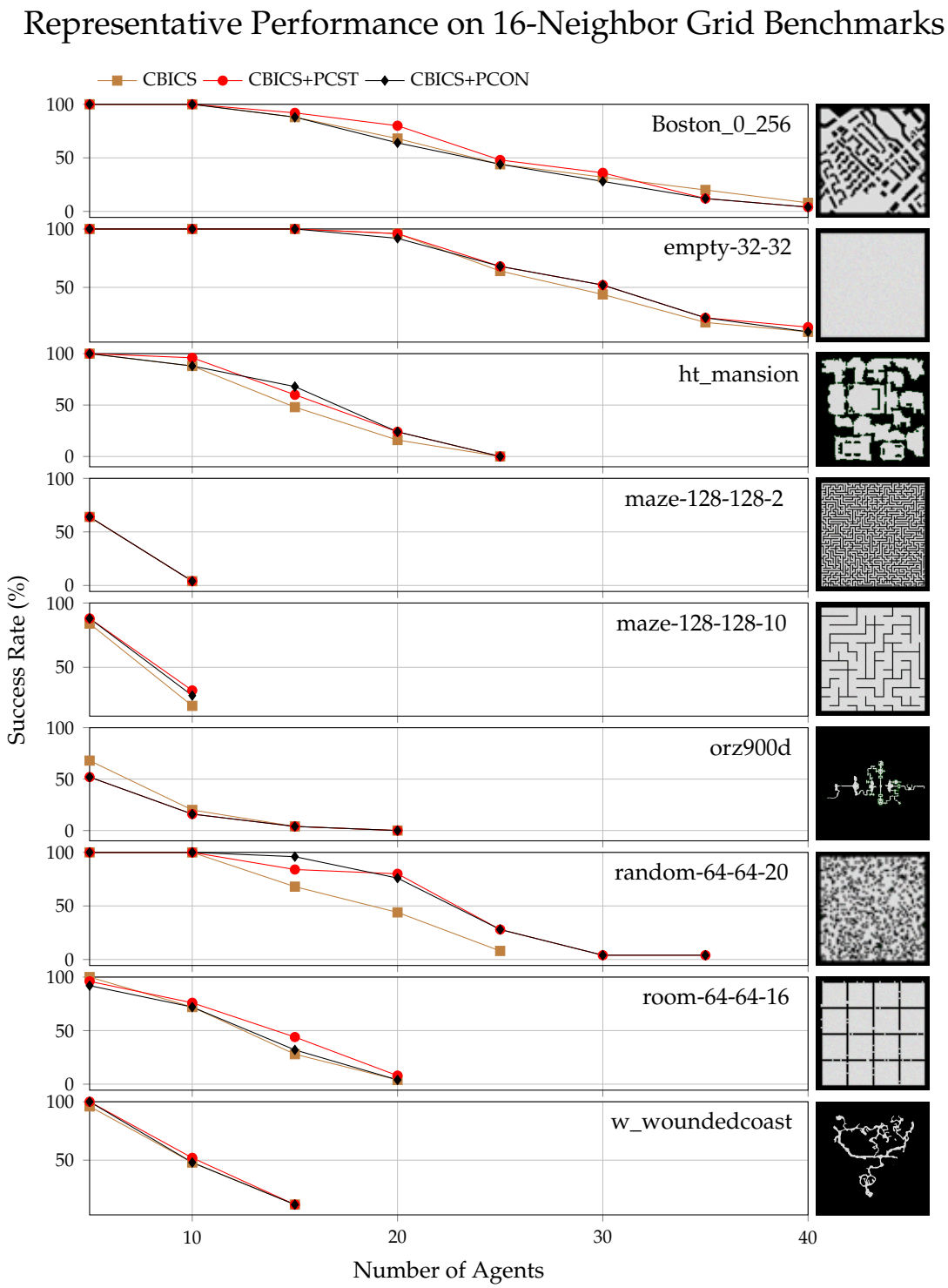


Figure 6.8.: Success rates of CBICS, CBICS+PCST and CBICS+PCON for benchmark problems on 16-neighbor grids.

In general, CBICS+PCON is stronger than CBICS+PCST, especially in the lower branching factors, while CBICS+PCST is slightly stronger in the higher branching factors. The modified PCS variant used by PCST which prunes infeasible joint states from the search, and this pruning is more significant in higher branching factors.

CBICS without enhancements is strongest in cases with a significant amount of *spatial* conflict symmetries (see Section 5.2.2). For example, maps with wide open spaces (e.g., where spatial conflict symmetries are common) and few narrow corridors (e.g., where temporal conflict symmetries are common) – especially in low branching factors, where the conflict symmetries involve a higher number of states (see Figure 2.4). Although PCON will eliminate all of the lowest-cost conflict symmetries in the root node, it turns out that only eliminating some of them is necessary to solve the problem for spatial conflict symmetries. For instance, if agent x, y and z each have pairwise conflicts, it is often the case that increasing the cost for only one of the agents (e.g., agent x) solves the symmetry for two of the three pairwise conflicts. PCST and PCON in this case perform more work than is necessary by solving for all conflicts.

Conversely, PCST and PCON are effective in situations with fewer spatial conflict symmetries and more temporal conflict symmetries such as with the higher branching factors, dense random obstacles (e.g., random-64-64-20) and mazes. With temporal conflict symmetries, costs usually need to be increased multiple times, and it is less often the case that increasing the cost for one particular agent resolves conflicts with multiple agents. Thus both preprocessing enhancements are beneficial in these settings because they push up the cost and add a more significant set of constraints at the root.

6.5.2. Comparison with CBS

In Section 5.4.5 we showed that CBS with biclique constraints dominates ICTS and CBS with edge and vertex constraints. In Section 5.7.3 we showed that CBS with biclique constraints is stronger than CBS with mutex propagation in most cases. We now compare CBICS with CBS. The setup and implementation are the same, where we

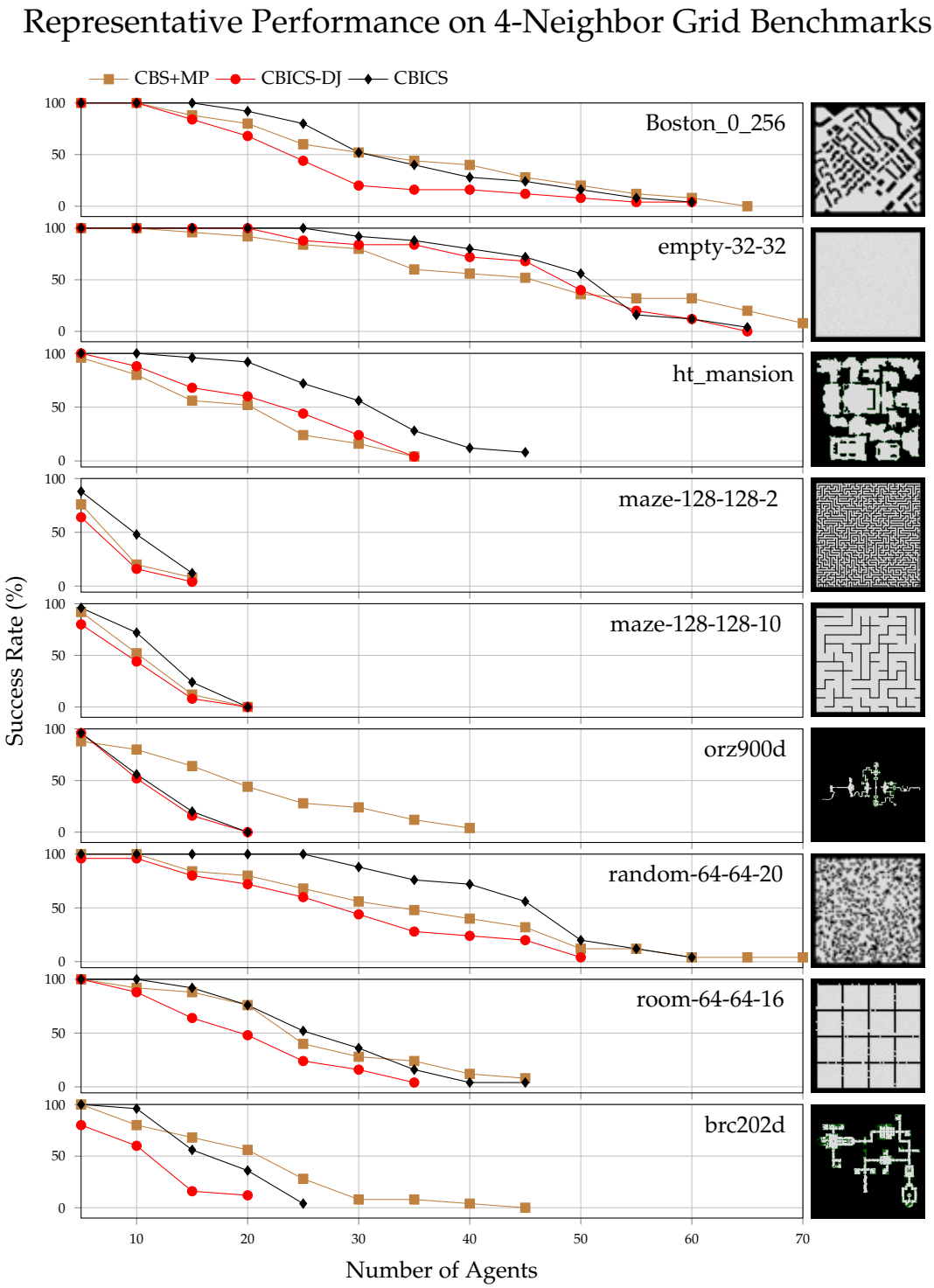


Figure 6.9.: Success rates of CBS+BC and two variants of CBICS for benchmark problems on 4-neighbor grids.

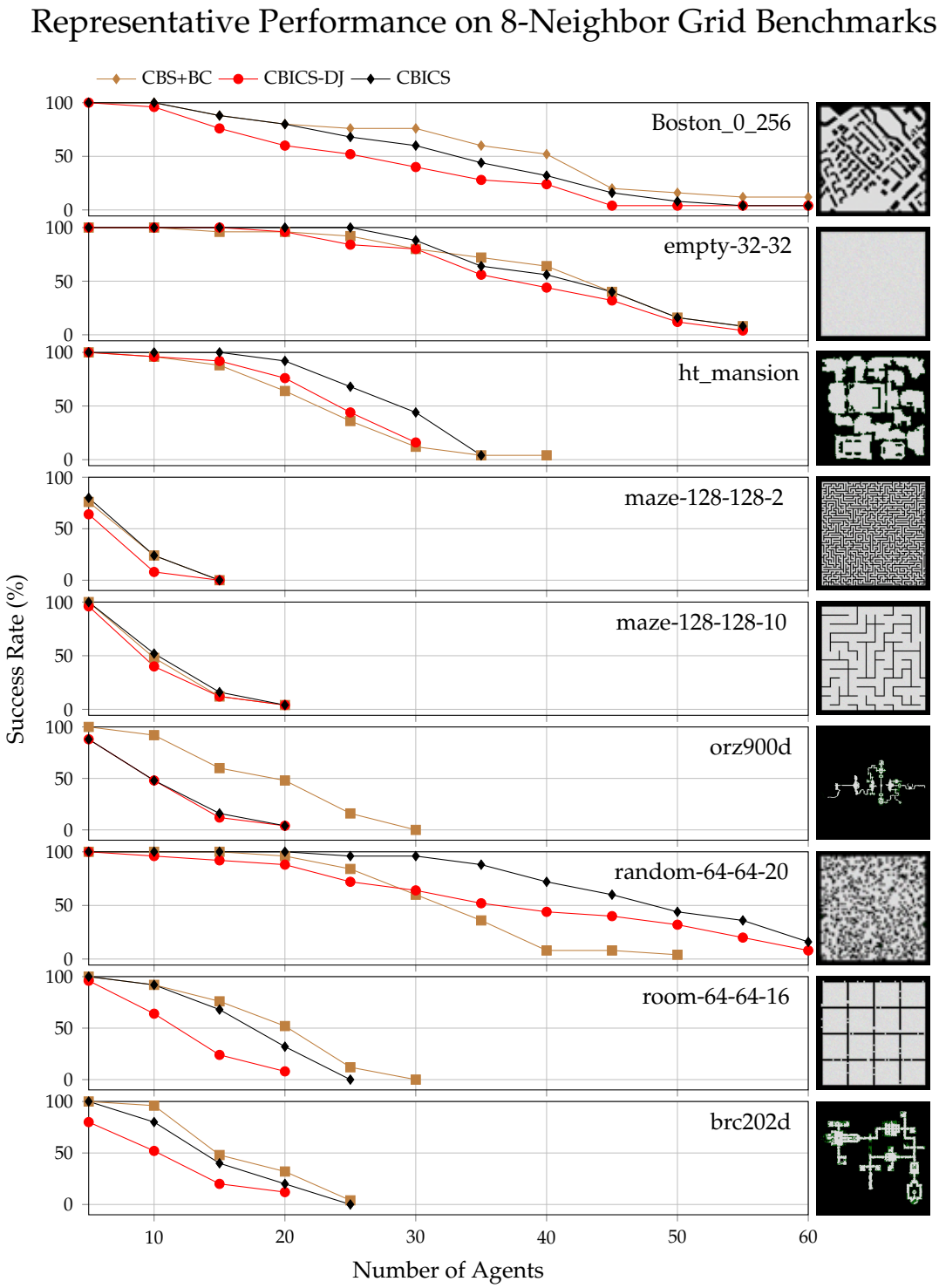


Figure 6.10.: Success rates of CBS+BC, CBS+BC+MP and two variants of CBICS for benchmark problems on 8-neighbor grids.

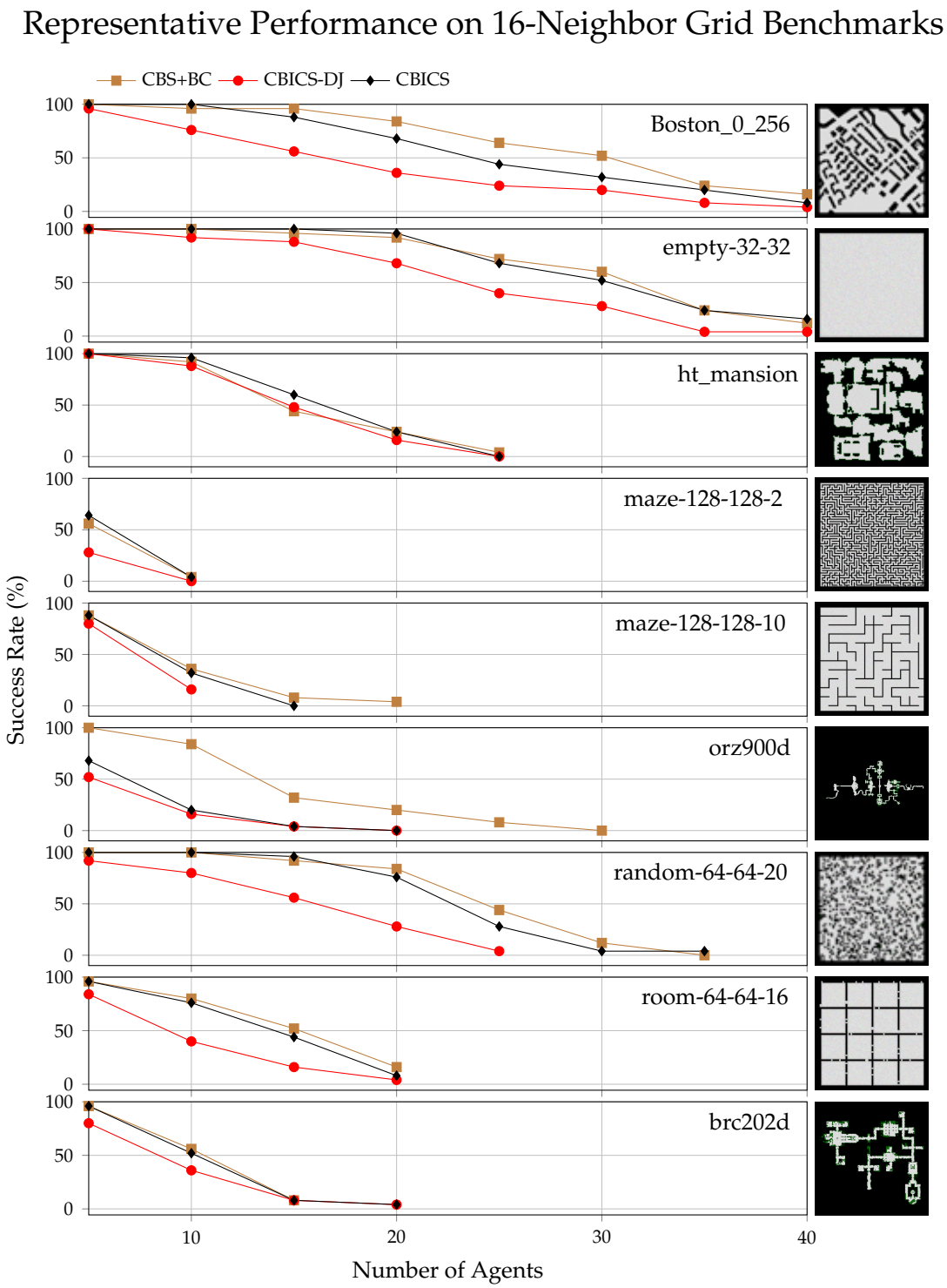


Figure 6.11.: Success rates of CBS+BC, CBS+BC+MP and two variants of CBICS for benchmark problems on 16-neighbor grids.

measure the success of each variant's ability to solve problem instances within the time limit, increasing the numbers of agents.

In Figures 6.9, 6.10 and 6.11, we show the performance of CBS, CBICS-DJ and CBICS. CBICS-DJ is a variant of CBICS in which only disjunctive splits are allowed. This variant is similar to CBS+MP except it uses cost constraints. By comparing CBICS with CBICS-DJ, we can evaluate the effectiveness of conjunctive constraints. For Figure 6.9, the variant of CBS used is CBS+MP because CBS+MP is only applicable to continuous time domains. In Figures 6.10 and 6.11, CBS+BC is used because it dominates CBS+BC+MP in most General MAPF domains (see Figures 5.11 and 5.12). For CBICS-DJ, we ran additional tests for CBICS-DJ+PCST and CBICS-DJ+PCON. For CBICS we reference the same tests as in Figures 6.6, 6.7 and 6.8. In each plot, we display results for the most performant variant (i.e., PCST or PCON). This is to compare the best performer in each category without making the plots too cluttered.

CBICS clearly dominates CBICS-DJ in all benchmarks. CBICS dominates CBS in many benchmarks, especially in 4-neighbor grids. However, CBS tends to be stronger as the base branching factor increases, especially in larger maps. This is because there are fewer symmetries in general in the higher branching factor settings, hence fewer conflict symmetries. Because the computational complexity of PCS is exponential in the search depth (or path length), CBICS variants' performance is not as good in large maps. Although BC constraints are not as powerful as MP constraints, BC has a lower computational complexity, and its complexity is not affected by the length of the path. Hence we see better performance in large maps with CBS+BC (i.e., Boston_0_256, orz900d and brc202d).

Tables 6.1, 6.2 and 6.3 show a summary of the total number of agents solved for in all of the MAPF benchmarks by the specific algorithm variant. The general trends shown in the plots are also apparent in these tables: CBS+BC dominates in the larger maps, especially as the single-agent branching factor increases. CBICS works best in smaller maps with narrower spaces. CBS+MP does best in small maps with large empty spaces and with small single-agent branching factors. The preprocessing

Table 6.1.: Total problems solved in under 30 seconds on 4-neighbor grid MAPF benchmarks

Map	CBS			CBICS		
		BC	MP		PCST	PCON
Berlin_1_256	1315	–	**1339**	918	744	737
Boston_0_256	**897**	–	825	844	688	701
Paris_1_256	1257	–	**1326**	758	581	576
Total	3469	–	**3490**	2520	2013	2014
brc202d	320	–	291	**388**	309	322
den312d	556	–	452	545	**606**	594
den520d	**780**	–	745	645	557	556
lak303d	322	–	396	434	**473**	450
orz900d	**325**	–	324	278	204	212
ost003d	**563**	–	504	**563**	468	463
Total	**2866**	–	2712	2853	2617	2597
empty-8-8	430	–	475	499	**506**	503
empty-16-16	758	–	778	814	785	**851**
empty-32-32	1154	–	**1463**	1232	988	1042
empty-48-48	1366	–	**1740**	1342	1000	1070
Total	3708	–	**4456**	3887	3279	3466
ht_chantry	528	–	484	493	**530**	500
ht_mansion	776	–	**817**	783	680	671
lt_gallowstemplar	**583**	–	485	482	530	516
w_woundedcoast	**556**	–	514	514	485	479
Total	**2443**	–	2300	2272	2225	2166
maze-128-128-10	283	–	281	316	**330**	313
maze-128-128-2	237	–	194	231	246	**261**
maze-32-32-2	244	–	254	303	**323**	316
maze-32-32-4	312	–	283	308	**335**	325
Total	1076	–	1012	1158	**1234**	1215
random-32-32-10	889	–	1035	**1079**	990	1053
random-32-32-20	667	–	695	687	752	**775**
random-64-64-10	**1485**	–	1406	1329	1055	1080
random-64-64-20	1029	–	886	1021	1089	**1111**
Total	4070	–	4022	**4116**	3886	4019
room-32-32-4	405	–	419	430	**465**	458
room-64-64-16	592	–	660	610	655	**679**
room-64-64-8	**417**	–	350	359	395	387
Total	1414	–	1429	1399	1515	**1524**

Table 6.2.: Total problems solved in under 30 seconds on 8-neighbor grid MAPF benchmarks

Map	CBS			CBICS		
		BC	BC+MP		PCST	PCON
Berlin_1_256	1115	**1211**	1040	1025	906	870
Boston_0_256	748	**924**	804	800	739	715
Paris_1_256	986	**1150**	933	888	843	824
Total	2849	**3285**	2777	2713	2488	2409
brc202d	303	322	271	**375**	308	292
den312d	467	**556**	465	453	505	520
den520d	612	**649**	575	605	525	524
lak303d	336	**424**	335	374	416	412
orz900d	258	259	259	**276**	262	262
ost003d	417	**475**	422	429	464	460
Total	2393	**2685**	2327	2512	2480	2470
empty-8-8	416	**436**	430	421	421	428
empty-16-16	489	580	**603**	563	575	597
empty-32-32	907	1024	**1123**	996	1022	1035
empty-48-48	1186	**1357**	1353	1337	1212	1205
Total	2998	3397	**3509**	3317	3230	3265
ht_chantry	446	**580**	439	446	531	510
ht_mansion	606	**723**	637	579	706	696
lt_gallowstemplar	516	**569**	514	470	508	496
w_woundedcoast	359	408	303	408	**411**	375
Total	1927	**2280**	1893	1903	2156	2077
maze-128-128-10	258	285	250	254	**289**	286
maze-128-128-2	169	197	166	181	**200**	**200**
maze-32-32-2	238	290	252	252	**264**	249
maze-32-32-4	246	**273**	249	260	266	271
Total	911	**1045**	917	947	1019	1006
random-32-32-10	714	**827**	751	710	718	729
random-32-32-20	481	**593**	464	456	504	493
random-64-64-10	1051	1211	1083	1056	1166	**1214**
random-64-64-20	712	**827**	655	649	726	750
Total	2958	**3458**	2953	2871	3114	3186
room-32-32-4	316	**388**	330	321	334	359
room-64-64-16	414	**506**	453	407	444	437
room-64-64-8	267	**360**	283	289	319	314
Total	997	**1254**	1066	1017	1097	1110

Table 6.3.: Total problems solved in under 30 seconds on 16-neighbor grid MAPF benchmarks

Map	CBS			CBICS		
		BC	BC+MP		PCST	PCON
Berlin_1_256	756	**899**	687	697	642	628
Boston_0_256	611	**743**	598	609	648	619
Paris_1_256	735	**846**	712	780	699	695
Total	2102	**2488**	1997	2086	1989	1942
brc202d	224	251	233	**265**	237	240
den312d	371	**447**	366	367	422	445
den520d	361	**484**	418	449	435	396
lak303d	232	279	246	263	**307**	297
orz900d	175	181	163	**200**	180	180
ost003d	324	**389**	366	360	381	373
Total	1687	**2031**	1792	1904	1962	1931
empty-16-16	352	**451**	408	385	410	404
empty-32-32	677	770	726	749	**774**	767
empty-48-48	954	1103	**1041**	984	977	978
empty-8-8	322	355	351	345	**356**	350
Total	2305	**2679**	2526	2463	2517	2499
ht_chantry	325	**409**	336	350	409	401
ht_mansion	344	**427**	400	382	423	**427**
w_woundedcoast	238	278	229	**286**	275	281
lt_gallowstemplar	397	**453**	402	411	429	418
Total	1304	**1567**	1367	1329	1536	1527
maze-128-128-10	197	**247**	220	210	227	217
maze-128-128-2	124	148	124	155	159	**163**
maze-32-32-2	189	**231**	195	209	220	214
maze-32-32-4	183	216	209	208	222	**228**
Total	693	**842**	748	782	828	822
random-32-32-10	477	**606**	535	499	523	533
random-32-32-20	384	**479**	377	364	415	415
random-64-64-10	638	**766**	714	643	714	724
random-64-64-20	464	**623**	505	483	579	586
Total	1963	**2492**	2131	1989	2231	2258
room-32-32-4	249	**319**	269	273	288	292
room-64-64-16	278	**389**	353	324	355	333
room-64-64-8	230	**305**	239	269	279	275
Total	757	**1013**	851	866	922	900

Map	E-ICTS		CBS+BC		CBS+BC+MP		CBICS	
	4	16	4	16	4	16	4	16
empty-8-8	**219**	28369	-	**2048**	4221	41994	592	6647
random-32-32-20	16	12722	-	938	5	114	**4**	**48**
maze-32-32-4	103	**167**	-	774	291	589	**80**	428
room-32-32-8	169	10132	-	798	84	465	**15**	**212**

Table 6.4.: Nodes expanded for small-map benchmarks with 10 agents.

enhancements for CBICS are most effective on maps with narrow spaces like mazes and with low single-agent branching factors.

Table 6.4 shows the average number of high-level node expansions for some benchmarks with 10 agents. When an algorithm was not able to solve an instance, the number of node expansions at the timeout was used instead. All algorithms expanded more nodes for the empty-8x8 map than for the other maps because of high agent density, requiring the resolution of many conflicts and cost increases to find a solution. Also, a smaller map means shorter path lengths and the low level of each algorithm thus does less work, enabling it to expand more nodes in less time.

Mutex propagation is an effective yet computationally expensive algorithm. Although CBS+BC+MP uses the same implementation as CBICS, its average number of expansions is consistently larger than for CBICS because of conjunctive motion constraints and pruning of infeasible cost combinations. Additionally, when running PCS with cost limits, a lot of work is reduced, allowing CBICS to evaluate nodes faster.

Map	E-ICTS		CBS+BC		CBS+BC+MP		CBICS	
	4	16	4	16	4	16	4	16
empty-8-8	**0.6**	5.4	-	**3.0**	3.4	4.5	1.8	3.8
random-32-32-20	**0.7**	29.7	-	28.3	9.9	48.2	4.0	**17.7**
maze-32-32-4	89.4	151.5	-	**20.1**	**35.8**	48.8	38.7	42.7
room-32-32-8	**0.6**	**1.5**	-	27.1	27.5	41.8	25.8	36.5

Table 6.5.: Low-level runtime for benchmarks with 10 agents.

Table 6.5 shows the average runtime per high-level node expanded for the low-level of each algorithm in milliseconds. For instances that were unsolvable, the runtime of all expanded high-level nodes were used. E-ICTS generally has a very small runtime per node evaluated because it uses small, cost-exact pairwise checks and eliminates infeasible combinations before doing a final k-agent search. The majority of its low-level searches never perform the full k-agent search because the pairwise tests fail quickly. PCS evaluates the entire pairwise space, including the infeasible combinations. This causes longer low-level runtimes, but more information is gleaned in the process. The runtime per node of CBS+BC+MP is consistent across all 32x32 maps, but the low-level runtime per node of CBICS has more variance. This is because *limited* nodes and the nodes in their sub-trees have tighter cost constraints, and so can be evaluated faster. The *unlimited* nodes in CBICS, having unlimited upper cost bounds, are similar to CBS+BC+MP nodes and tend to take longer to evaluate.

6.6. Summary

In this chapter, we introduced a new algorithm: *Conflict-Based Increasing Cost Search* (CBICS). In the process, we introduced several new concepts. We introduced *cost constraints* for restricting the path costs of agents. We introduced *conjunctive splitting* using novel *cost-conditional motion constraints*. The theory for conjunctive splitting relies on novel analysis of *bi-connected bicliques (BBC)*. We introduced the novel algorithm *Pairwise Constraint Search (PCS)*, an extension of mutex propagation which performs analysis of BBCs to produce cost-conditional motion constraints.

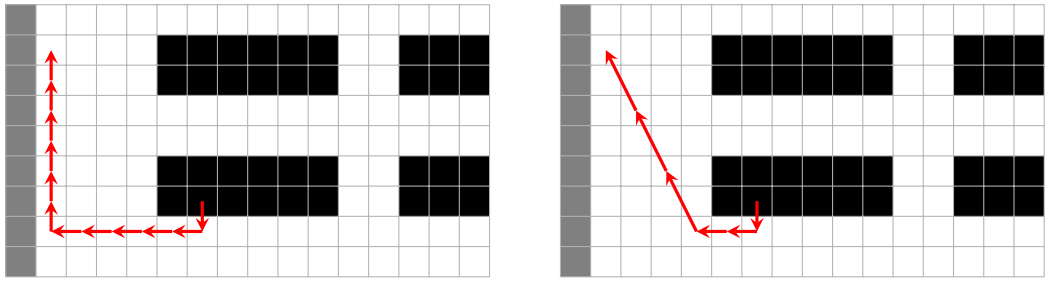
We developed theory for the computational complexity, completeness and optimality of CBICS and PCS. We developed enhancements for CBICS, namely, *preprocessing costs (PCST)* and *preprocessing constraints (PCON)* and tested their effectiveness. Finally, we provided an empirical comparison between CBS and CBICS and found that CBICS has distinct advantages in certain types of maps.

7. Revisiting Our Motivating Examples

In Chapter 1 we introduced the challenging warehouse automation and airport surface operations (ASO) domains. We now revisit these examples, showcase experiments on problem instances inspired by these domains and discuss how solving them in General MAPF settings improves state-of-the-art.

7.1. The Warehouse Domain

In Chapter 1 we stated: *Assuming that robots are capable of turning to arbitrary angles, paths that include non-Cardinal directions could result in significant cost savings over solutions that allow Cardinal angles only*. Figure 7.1 shows two example paths, Figure 7.1(a) shows a path which is restricted to Cardinal directions only (a 4-neighbor grid) and Figure 7.1(b) shows a path which uses a 16-neighbor grid. Our hypothesis is that higher-connected grids will allow shorter paths and cost savings in general, but due to



(a) (b)

Figure 7.1.: An example of (a) a path which uses actions on a 4-neighbor grid and (b) a path which uses actions on a 16-neighbor grid.

the higher branching factors they may require an increase in computation time. To test this hypothesis, we ran a set of experiments, comparing both the runtimes and quality (in terms of path length) of solutions.

For our first test, we set up experiments for four warehouse maps and recorded the success rate where the solver was able to find a solution in under thirty seconds for specific numbers of agents. Figures 7.2, 7.3 and 7.4 show the results.

We see that CBICS is generally more performant than CBS+MP in warehouse-10-20-10-2-1 in all settings, but is dominated by CBS+MP otherwise. This suggests that CBICS is sensitive to both map size and corridor widths. This same phenomenon was discussed in Chapter 6. However, it is clear that CBS with biclique constraints is superior to the other approaches in all cases, though bicliques are not useful in 4-neighbor grids as discussed in Chapter 5.

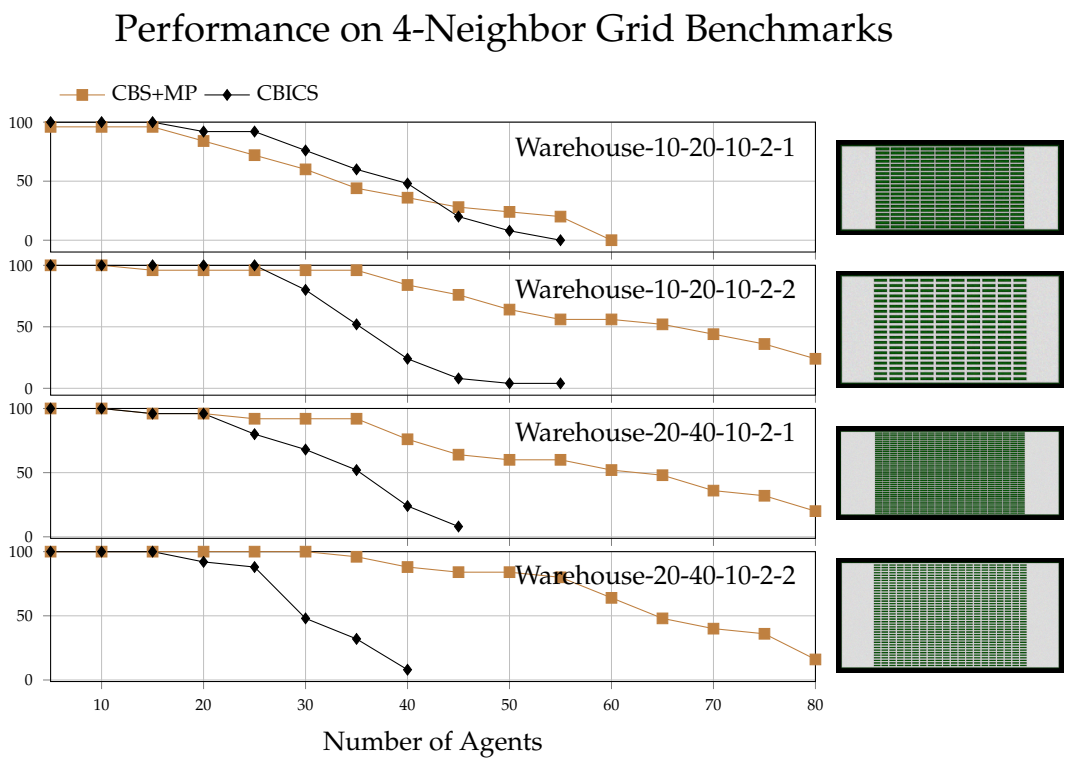


Figure 7.2.: Success rates of CBS and CBICS for warehouse benchmark problems on 4-neighbor grids.

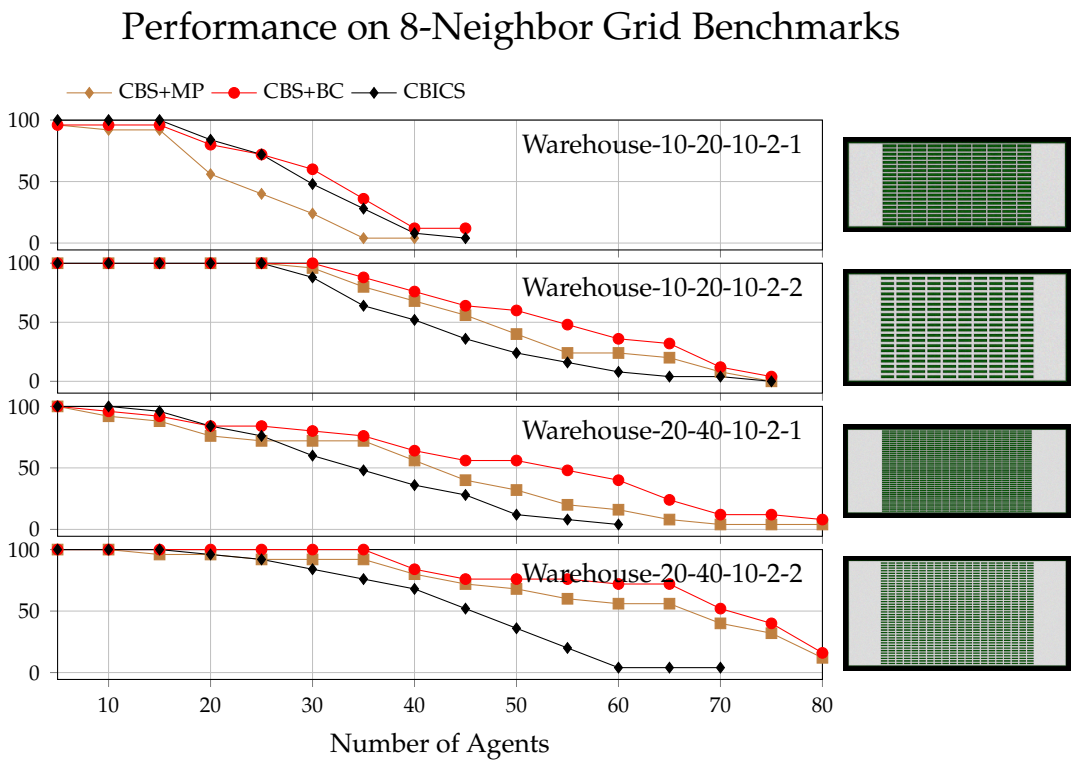


Figure 7.3.: Success rates of CBS and CBICS for warehouse benchmark problems on 8-neighbor grids.

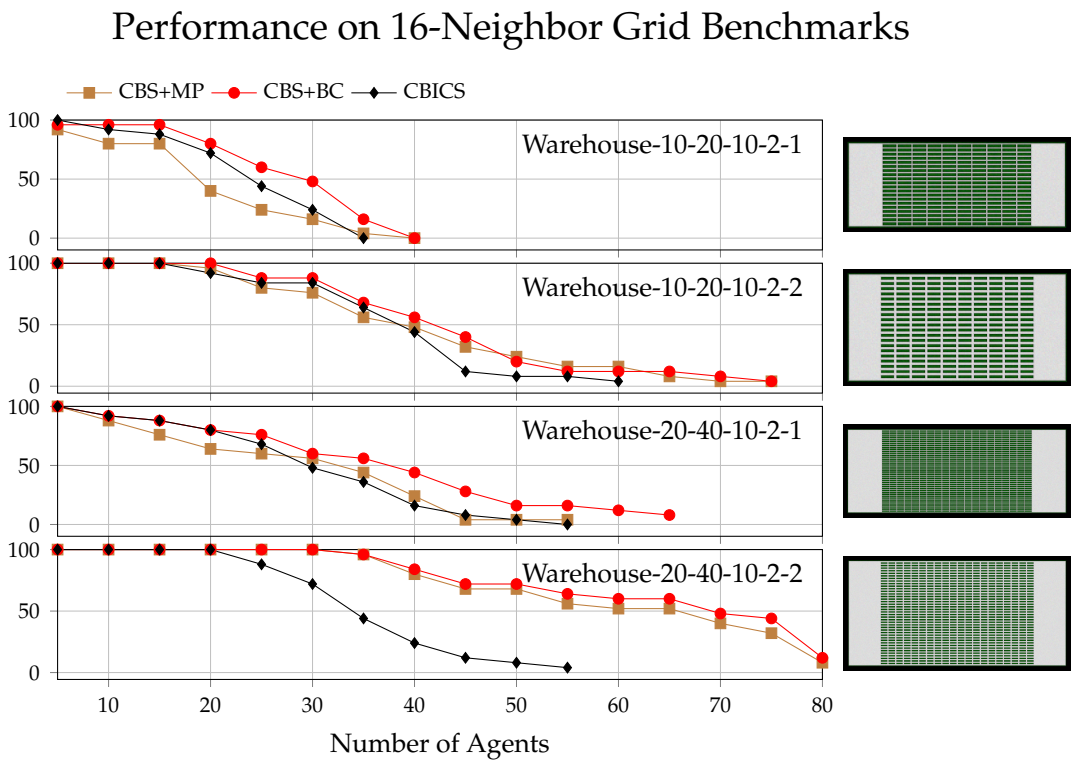


Figure 7.4.: Success rates of CBS and CBICS for warehouse benchmark problems on 16-neighbor grids.

Map	Agents	Solution Cost			Runtime (sec)		
		4-neighbor	8-neighbor	16-neighbor	4-neighbor	8-neighbor	16-neighbor
warehouse-10-20-10-2-1	30	2,391 (100%)	2,291 (95%)	**2,259** (94%)	**10.7** (100%)	15.7 (147%)	20.0 (187%)
warehouse-10-20-10-2-2	40	4,057 (100%)	3,751 (92%)	**3,653** (90%)	11.2 (100%)	**11.0** (98%)	16.4 (146%)
warehouse-20-40-10-2-1	30	5,225 (100%)	**5,043** (96%)	5,094 (96%)	**4.2** (100%)	9.4 (224%)	14.7 (350%)
warehouse-20-40-10-2-2	60	10,590 (100%)	9,889 (92%)	**9,596** (91%)	**16.2** (100%)	17.2 (106%)	18.5 (114%)

Table 7.1.: Average solution cost and runtime for warehouse benchmarks with varying grid connectivity.

Table 7.1 shows mean solution costs and mean runtimes for the warehouse benchmarks. Each column shows a percentage compared to the results for 4-neighbor grids. Our hypothesis from Chapter 1 is correct: paths are shorter when agents are planned on grids that allow more direct movement toward the goal. In Table 7.1 we see that with the higher-neighbor grids, paths in warehouses with corridor widths of 1 (*warehouse-10-20-10-2-1* and *warehouse-20-40-10-2-1*) show at least a 4% reduction in solution cost and paths in warehouses with corridor widths of 2 (*warehouse-10-20-10-2-2* and *warehouse-20-40-10-2-2*) show at least an 8% reduction in solution cost on average. These reduced costs could mean significant savings in terms of saved time and energy expenditures for full-time operations.

Although planning in the grids with higher branching factors yields lower cost solutions, computing these solutions requires more computation time. The mean run-

Map	Agents	Optimal Solution Cost			Sub-optimal Solution Cost		
		4-neighbor	8-neighbor	16-neighbor	4-neighbor	8-neighbor	16-neighbor
warehouse-10-20-10-2-1	30	2,391 (100%)	2,291 (95%)	**2,259** (94%)	2,476 (104%)	2,371 (99%)	**2,339** (98%)
warehouse-10-20-10-2-2	40	4,057 (100%)	3,751 (92%)	**3,653** (90%)	4,115 (101%)	3,801 (94%)	**3,719** (92%)
warehouse-20-40-10-2-1	30	5,225 (100%)	**5,043** (96%)	5,094 (96%)	5,311 (102%)	**5,115** (98%)	5,189 (99%)
warehouse-20-40-10-2-2	60	10,590 (100%)	9,889 (92%)	**9,596** (91%)	10,721 (101%)	10,001 (94%)	**9,766** (92%)

Table 7.2.: Average optimal and sub-optimal solution costs for warehouse benchmarks with varying grid connectivity.

Map	Agents	Runtime for Optimal (sec)			Runtime for Sub-optimal (sec)		
		4-neighbor	8-neighbor	16-neighbor	4-neighbor	8-neighbor	16-neighbor
warehouse-10-20-10-2-1	30	10.7 (100%)	15.7 (147%)	20.0 (187%)	4.0 (37%)	9.7 (91%)	13.7 (128%)
warehouse-10-20-10-2-2	40	11.2 (100%)	11.0 (98%)	16.4 (146%)	3.6 (32%)	5.0 (45%)	6.1 (54%)
warehouse-20-40-10-2-1	30	4.2 (100%)	9.4 (224%)	14.7 (350%)	3.0 (71%)	8.6 (204%)	12.5 (298%)
warehouse-20-40-10-2-2	60	16.2 (100%)	17.2 (106%)	18.5 (114%)	8.3 (64%)	16.7 (103%)	16.5 (102%)

Table 7.3.: Average runtime for warehouse benchmarks with varying grid connectivity.

times in Table 7.1 for 8- and 16-neighbor grids show that up to 3.5× more computation is required for these solutions. This leads us to our next conjecture which is (as we showed in Section 4.6.4), running sub-optimal search in grids with high branching factors can yield lower-cost solutions in less time than running optimal search on 4-neighbor grids. The results for solution costs are shown in Table 7.2, and runtimes are shown in Table 7.3. All percentages shown in parenthesis are relative to optimal costs and runtimes in optimal 4-neighbor settings.

The average sub-optimal results appear to inflate the costs by about 1-4% when compared to their optimal counterparts. However, in all cases, the sub-optimal solutions are better than for optimal 4-neighbor grids. The runtimes are still affected by the corridor width. In corridors with a width of 1, (with the exception of *warehouse-10-20-10-2-1* in 8-neighbor grids) we see that even when we plan sub-optimally, the average runtimes are not better than planning optimally in 4-neighbor grids. This is due to the fact that in corridors with a width of 1, agents are forced to use cardinal-direction actions only for a significant portion of their path, regardless of the grid connectivity.

In benchmarks with a corridor width of two, however, the cost savings is up to 8% and the runtime is reduced by as much as 50%. For example, in *warehouse-10-20-10-2-2*, 16-connected grids, we see a reduction of cost by 8% and a runtime which is only 54% of the counterpart in optimal 4-neighbor grids. In *warehouse-20-40-10-2-2*, the solution costs were improved by up to 8% and the runtimes were only increased by a fraction of a second. These results occur because the wider corridor widths are less congested and allow the agents to utilize non-Cardinal movements in the corridors.

Overall, our hypothesis from Chapter 1 is correct: paths are shorter when agents are planned on grids that allow more direct movement toward the goal. Furthermore, we have shown empirically that running sub-optimally in grids with higher connectivity can reduce both the runtime *and* the solution costs in specific settings.

7.2. The Airport Surface Operations Domain

We now analyze the performance of MAPF algorithms in the ASOAgent domain. Figure 7.5 shows an aerial view of Edmonton International Airport (YEG). Our objective is to plan routes from gates to one of the two runways and vice versa. The inset on Figure 7.5 measures the distance between the two ends of the runways to be over 5km. Aircraft typically taxi on the tarmac using lane lines as shown in Figure 7.6. We manually constructed a directed graph abstraction from the satellite image by labeling each lane line intersection with a vertex number as shown in Figure 7.6. Then we connected the vertices with edges as shown in Figures 7.7 and 7.8.

Figure 7.5.: Edmonton International Airport: View from satellite

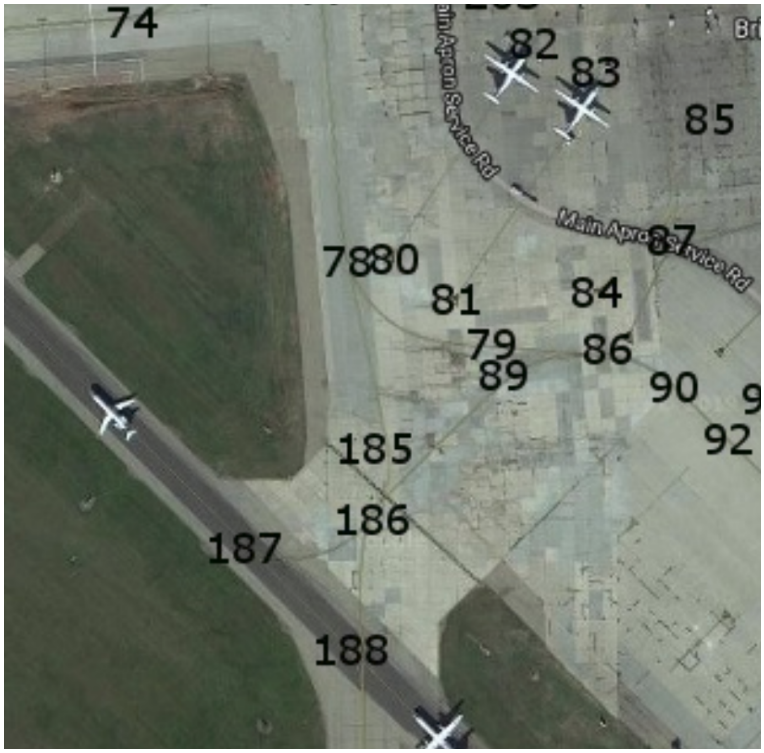


Figure 7.6.: Edmonton International Airport: Closeup of tarmac lane lines with numbered intersections

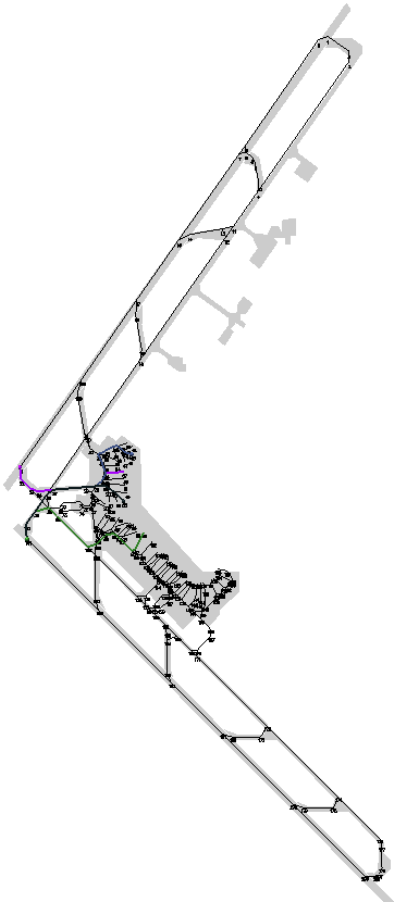


Figure 7.7.: Edmonton International Airport: View in simulation.

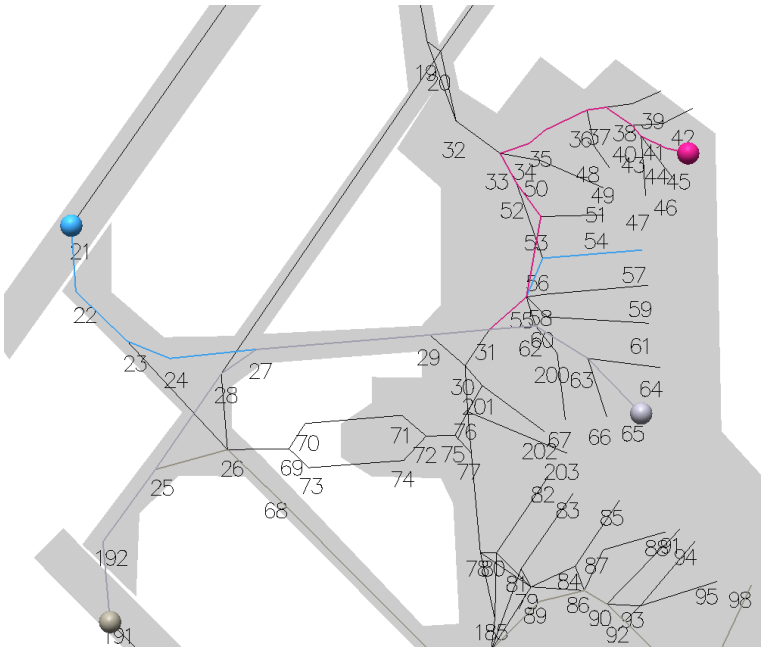


Figure 7.8.: Edmonton International Airport: Closeup of tarmac graph in simulation.

For the simulation, in the directed graph, we modeled two co-located vertices at each gate *gate vertices*, for example *42-in* and *42-out*, such that one vertex is connected to a directed edge leading into the gate and one vertex is connected to the directed edge leading out of the gate. There is no edge connecting the two co-located gate vertices. Thus, once an aircraft enters a gate it cannot leave (until another planning cycle). This prevents the search from trying to temporarily "tuck" an aircraft into an arbitrary gate, to let other aircraft pass by. If an aircraft pulls into a gate, it must remain there.

There are also directed edges leading into the entrance of a runway, and directed edges leading out of the exits of a runway and entrances are not connected to the exits. This discourages the search from arbitrarily moving agents onto the runway in order to let another agent pass by. Additionally, agents disappear once arriving at their respective goals to represent that they have taken off or parked, respectively.

Figure 7.7 shows the full scale of the simulation environment. Figure 7.8 shows a close up of a section near the center gate area. Edge weights are modeled as the length of the edge to the nearest meter. Aircraft are modeled as circles. Our simulation

allows an arbitrary radius for each aircraft, but our experiments were run with a radius of 10 meters.

Figure 7.9 shows the performance of CBS and CBICS variants on the SOC domain. This domain is challenging for MAPF algorithms. This is due to the extensive use of one-lane corridors and tightly congested intersection areas. CBS+BC shows the best overall performance with the lowest runtimes and highest success rate. All of CBS+BC+MP, and the CBICS variants have a very low number of high-level expansions when compared to CBS+BC. This is because the former three rely on continuous-time mutex propagation, which is a computationally expensive routine. In spite of the relatively high node count produced by CBS+BC, the nodes are much less expensive to evaluate, and solutions can be found quicker on average.

All of the algorithms are unable to solve all 25 problem instances for more than three agents. We conjecture that this difficulty arises due to fixed wait times. Because of the abundant one-lane corridors in this scenario, often the only recourse of action available to the agents is to wait at the start location for a significant amount of time for other agents to clear the corridors. We believe that allowing arbitrary wait times by converting the low-level to a SIPP-based algorithm would help solve these instances much faster. However, we leave this for future work.

7.3. Summary

In this chapter, we revisited the motivating examples from Chapter 1, namely: warehouse automation and airport surface operations (ASO). We showed that converting the warehouse domain from a unit-cost to a non-unit cost one, we can improve overall solution costs. Optimal solutions take longer, but sub-optimal ones can take less time depending on the configuration but always yield cost improvements. The ASO domain remains a challenge due to the very congested nature of the graph, it remains future work to investigate computing arbitrary wait times for aircraft at their gates.

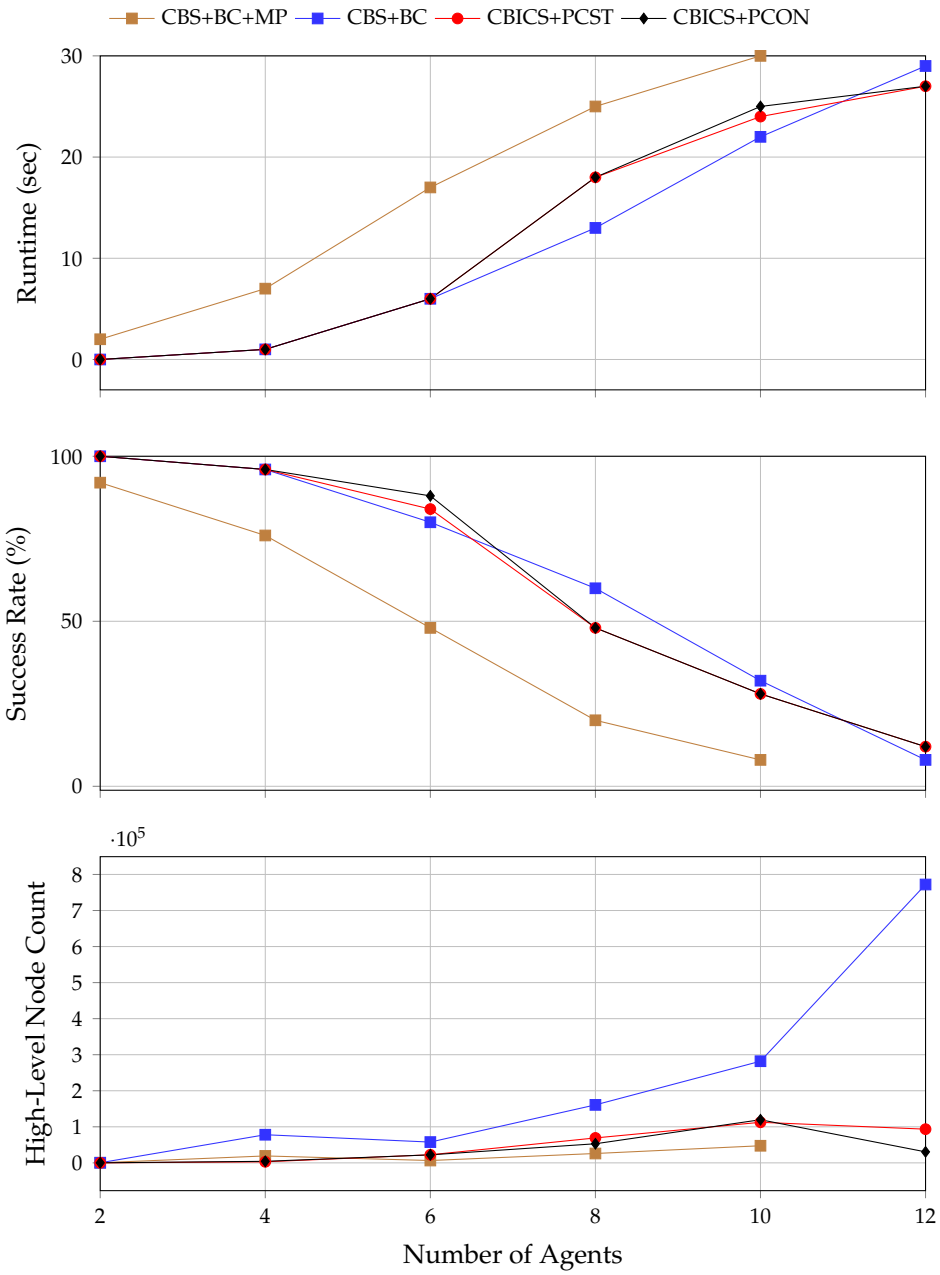


Figure 7.9.: Success rates of CBS variants and variants of CBICS for the airport surface operations domain.

8. Summary

This thesis focused on Multi-Agent Pathfinding (MAPF) in continuous time domains (General MAPF). In Chapter 1 we introduced motivating examples and an overview of this thesis. In Chapter 2 search problems were explained, MAPF was defined and prior work was reviewed and categorized. A novel definition of General MAPF and novel concepts for it were presented in Chapter 3. Chapters 4 and 5 reviewed existing work and introduced novel extensions for the Increasing Cost Tree Search (ICTS) and Conflict-Based Search (CBS) algorithms for General MAPF. Chapter 6 introduced a new algorithm, Conflict-Based Increasing Cost Search (CBICS) for General MAPF that combines the strengths of ICTS and CBS.

8.1. Summary of Contributions

At the beginning of the work represented by this thesis, search-based solutions geared for General MAPF were relatively sparse: no formulation of optimal search-based algorithms were known and no formal definition of General MAPF existed.

In Sections 1.1.1 and 1.1.2, we introduced two motivating examples: The warehouse domain, in which we conjectured that although treatment as a Classic MAPF problem was sufficient, treatment as a General MAPF problem could yield more efficient plans in terms of time and fuel. Also, the airport surface operations domain, which exhibits non-uniformly weighted edges and therefore cannot be fit into a Classic MAPF paradigm without considerable changes to the planning graph.

We have provided extensive background information on MAPF, including a list of MAPF problem variants in Section 2.3.4 and variations on cost and movement

for traditional MAPF in Section 2.3.5. We reviewed current and prior work on the MAPF problem and provided a *novel taxonomy* of various algorithms in Section 2.4. We provided pseudocode for important sub-routines and reviewed other techniques related to the MAPF problem throughout Chapter 2.

We provided a new formal definition of *General MAPF* in Section 3.1 which allows for a wider range of applications by allowing actions of arbitrary duration and velocity; and agents of arbitrary size and shape. Because of this, conflict detection takes a considerable amount of extra attention. We provided a short survey of conflict detection techniques in Section 3.2, including extended derivations in Appendix A.

We explained the concept of durative conflicts in Section 3.4. We provided a new *analysis of completeness* for General MAPF in Section 3.3 and a new approach for guaranteeing completeness for MAPF in general called *temporally-relative duplicate pruning (TRDP)* in Section 3.3.1 with detailed proofs in Appendix B. We proved that completeness can be guaranteed in General MAPF, assuming action durations are rational and temporally-relative duplicate pruning is performed. Finally, in Section 3.4.2 we showed that allowing arbitrary wait times yields multiple benefits, including better cost quality for solutions, exponentially less work, and may allow solvability when fixed wait actions will not.

In Section 3.5 we provided a *new statistic* for describing the ratio of equivalent-cost alternate paths in a General MAPF domain which is useful in determining whether to use a conflict avoidance table with dynamically centralized algorithms.

In Section 4.1 we explained the Increasing-Cost Tree Search Algorithm (ICTS). We presented the first proofs for completeness and optimality of ICTS in Section 4.2. We introduced *novel extensions for General MAPF* domains in Section 4.3. These extensions are general, and allow ICTS to be used with virtually any General MAPF instance. We showed that it has better performance than multi-agent A* and comparable performance to CBS in many settings in Section 4.6.

We introduced two new *bounded sub-optimal variants* of ICTS: ϵ-ICTS and w-ICTS in Section 4.4. We showed that these enhancements allow significant performance

improvements over optimal ICTS without a significant impact on solution quality in Section 4.6.

We explained the Conflict-Based Search (CBS) algorithm in Section 5.1. We explained various conflict types in Section 5.2.1 and conflict symmetries in Section 5.2.2. In Section 5.3 we highlighted how conflicts differ in General MAPF domains and specific approaches to deal with them, using appropriate constraints in CBS. We have defined the notion of correctness for constraint sets in CBS and the motivation for generating large sets of constraints for CBS in Section 5.3.2.

We introduced a novel approach to generating large sets of constraints that are guaranteed to be correct using analysis with *bicliques* and bipartite conflict graphs in Section 5.4. Additionally, we provided a new technique for more powerful constraint generation using *time-annotated bicliques*. We introduce three approaches to biclique constraint generation and in Section 5.4.5 show that biclique constraints dominate other classic approaches to constraint generation in terms of reducing the amount of work performed and runtime necessary to find solutions.

In Section 5.5 we provide a new algorithm for performing *mutex propagation* in General MAPF domains. This allows more effective symmetry breaking in the case of conflict symmetries. We discussed options for low-level search routines in General MAPF in Section 5.6 and provided novel analysis for the adaptation of *high-level heuristics* for General MAPF domains in Section 5.7.1. In Section 5.7.3 we empirically compare CBS with constraints built using mutex propagation and constraints built with bicliques and find that CBS with bicliques dominates generally, with exceptions for some specific cases.

In Section 5.8 we explored two new approaches to sub-optimal CBS: *constraint layering* and *conditional constraints*. The former allows refinement of solutions which can make the algorithm faster. The latter works by adding extraneous constraints to avoid conflicts and relaxing them as necessary to avoid incompleteness. Our results in Section 5.10.4 show that conditional constraints can boost performance significantly without significantly degrading solution quality. Finally, we provided new theoreti-

cal analysis for the computational complexity, completeness and optimality of CBS in General MAPF domains in Section 5.11.

Finally, the bulk of this work culminated in the formulation of a novel algorithm: Conflict-Based Increasing Cost Search in Chapter 6. Our new formulation of the high-level search of CBICS in Section 6.2 introduced *cost constraints* for restricting the path costs of agents and *conjunctive splitting* using novel *cost-conditional motion constraints*. The theory for conjunctive splitting relies on novel analysis of *bi-connected bicliques (BBC)* and the novel algorithm *Pairwise Constraint Search (PCS)*, an extension of mutex propagation which performs analysis of BBCs to produce cost-conditional motion constraints explained in Section 6.3.

We explained theory for the computational complexity, completeness and optimality of CBICS and PCS in Section 6.4. In Section 6.4.1 we explained enhancements for CBICS, namely, *preprocessing costs (PCST)* and *preprocessing constraints (PCON)* and tested their effectiveness in Section 6.5.1. Finally, we provided an empirical comparison between CBS and CBICS in Section 6.5.2 and found that CBICS has distinct advantages in certain types of maps.

All of these contributions are novel to the best of our knowledge. This research contributes scientific knowledge of MAPF and MAPF algorithms to allow solutions for more realistic real-world scenarios where previous algorithms could not.

8.2. Open Questions and Future Work

While the contributions of this thesis are a significant start, we believe we have only scratched the surface. We still have many open questions and unfinished work.

8.2.1. Open Questions

It has been shown that feasible solutions to Classic MAPF can be found in polynomial time [97]. An answer to the question of whether solving General MAPF instances is tractable is an open question.

Currently, it is difficult to detect conflict symmetries in General MAPF without performing a computationally expensive search like PCS. While detection of conflict symmetries in Classic MAPF is computationally efficient due to the uniform nature of grid vertices [113, 111, 112], such detection mechanisms remain elusive in General MAPF, perhaps because uniform spacing of vertices is not certain with all problem instances.

8.2.2. Future Work

For future work, there are several avenues. Significant successes have been achieved for Classic MAPF by hybridizing reduction-based solvers such as CSP [57], MIP [100] and SAT [179] with concepts from CBS and ICTS, however, further work is required to generalize these hybrid approaches for General MAPF.

At the end of Chapter 6 we saw that the airport surface operations domain continues to be a significant challenge. We conjecture that adding arbitrary wait times to the existing techniques will help immensely. Especially the technique of using SIPP with CBS [6] could be combined with biclique constraints and mutex propagation in CBS and possibly CBICS.

We discovered that the mechanism used to drive up costs in CBICS which informs pruning and heuristics could also be applied to ICTS to inform both pruning and possible novel ICTS heuristics. The theory behind Temporally Relative Duplicate Pruning (TRDP) suggests that it can be applied to algorithms like CBICS, CBS and ICTS with beneficial results. The details of how to implement TRDP in these algorithms remains future work.

Finally, application of General MAPF algorithms for execution on robots in General MAPF domains will require further work on online MAPF [182, 120] and extensions to the MAPF-POST algorithm [75] for General MAPF.

The work covered by this thesis lays a solid foundation for all of this future work.

Bibliography

[1] Airports Council International. *2017 Aircraft Movements – Annual Traffic Data – ACI World*. https://aci.aero/data-centre/annual-traffic-data/aircraft-movements/2017-aircraft-movements-annual-traffic-data/. [Online; accessed 2-January-2020]. 2017.

[2] S Akishita, S Kawamura, and K Hayashi. "Laplace potential for moving obstacle avoidance and approach of a mobile robot". In: *Japan-USA Symposium on flexible automation, A Pacific rim conference*. 1990, pp. 139–142.

[3] Faten Aljalaud and Nathan R Sturtevant. "Finding Bounded Suboptimal Multi-Agent Path Planning Solutions Using Increasing Cost Tree Search". In: *Sixth Annual Symposium on Combinatorial Search*. 2013.

[4] Fadi A Aloul, Bashar Al Rawi, and Mokhtar Aboelaze. "Identifying the shortest path in large networks using Boolean satisfiability". In: *2006 3rd International Conference on Electrical and Electronics Engineering*. IEEE. 2006, pp. 1–4.

[5] Ofra Amir, Guni Sharon, and Roni Stern. "Multi-Agent Pathfinding as a Combinatorial Auction." In: *AAAI Conference on Artificial Intelligence*. 2015, pp. 2003–2009.

[6] Anton Andreychuk et al. "Multi-Agent Pathfinding with Continuous Time". In: *International Joint Conference on Artificial Intelligence*. 2019, pp. 39–45.

[7] Franklin Antonio. "Faster line segment intersection". In: *Graphics Gems III (IBM Version)*. Elsevier, 1992, pp. 199–202.

[8] Francisco Javier Pulido Arrebola. "New Techniques and Algorithms for Multi-objective and Lexicographic Goal-Based Shortest Path Problems". PhD thesis. Universidad de Málaga, 2015.

[9] Dor Atzmon, Amit Diei, and Daniel Rave. "Multi-Train Path Finding". In: *International Symposium on Combinatorial Search*. 2019, pp. 125–129.

[10] Dor Atzmon et al. "Conflict-Free Multi-Agent Meeting". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 31. 2021, pp. 16–24.

[11] Dor Atzmon et al. "Probabilistic robust multi-agent path finding". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30. 2020, pp. 29–37.

[12] Dor Atzmon et al. "Robust Multi-Agent Path Finding". In: *International Conference on Autonomous Agents and Multiagent Systems*. 2018, pp. 1862–1864.

[13] Dor Atzmon et al. "Robust Multi-Agent Path Finding and Executing". In: *Journal of Artificial Intelligence Research* 67 (2020), pp. 549–579.

[14] Manlio Bacco et al. "Smart farming: Opportunities, challenges and technology enablers". In: *2018 IoT Vertical and Topical Summit on Agriculture-Tuscany (IOT Tuscany)*. IEEE. 2018, pp. 1–6.

[15] Max Barer et al. "Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem". In: *International Symposium on Combinatorial Search*. 2014, pp. 961–962.

[16] Laura Barnes, MaryAnne Fields, and Kimon Valavanis. "Unmanned ground vehicle swarm formation control using potential fields". In: *Control & Automation, 2007. MED'07. Mediterranean Conference on*. IEEE. 2007, pp. 1–8.

[17] Roman Barták, Ji?í Švancara, and Marek Vlk. "A Scheduling-Based Approach to Multi-Agent Path Finding with Weighted and Capacitated Arcs". In: *International Conference on Autonomous Agents and Multiagent Systems*. 2018.

[18] Roman Barták et al. "Modeling and solving the multi-agent pathfinding problem in picat". In: *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2017, pp. 959–966.

[19] Jur Van den Berg, Ming Lin, and Dinesh Manocha. "Reciprocal velocity obstacles for real-time multi-agent navigation". In: *International Conference on Robotics and Automation*. IEEE. 2008, pp. 1928–1935.

[20] Subhrajit Bhattacharya, Maxim Likhachev, and Vijay Kumar. "Multi-agent path planning with multiple tasks and distance constraints". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE. 2010, pp. 953–959.

[21] Zahy Bnaya and Ariel Felner. "Conflict-oriented windowed hierarchical cooperative A*". In: *International Conference on Robotics and Automation*. IEEE. 2014, pp. 3743–3748.

[22] Adi Botea, Davide Bonusi, and Pavel Surynek. "Solving Multi-Agent Path Finding on Strongly Biconnected Digraphs". In: *Journal of Artificial Intelligence Research* (2018), pp. 273–314.

[23] Adi Botea et al. "Pathfinding in games". In: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2013.

[24] Eli Boyarski et al. "Don't Split, Try To Work It Out: Bypassing Conflicts in Multi-Agent Pathfinding". In: *International Conference on Automated Planning and Scheduling*. 2015, pp. 47–51.

[25] Eli Boyarski et al. "ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding." In: *International Joint Conference on Artificial Intelligence*. 2015, pp. 223–225.

[26] Eli Boyarski et al. "Iterative-Deepening Conflict-Based Search." In: *IJCAI*. 2020, pp. 4084–4090.

[27] Jack E Bresenham. "Ambiguities in incremental line rastering". In: *IEEE Computer Graphics and Applications* 7.5 (1987), pp. 31–43.

[28] Michael Brunner, Bernd Brüggemann, and Dirk Schulz. "Hierarchical rough terrain motion planning using an optimal sampling-based method". In: *International Conference on Robotics and Automation*. IEEE. 2013, pp. 5539–5544.

[29] Bernard Chazelle. "Approximation and decomposition of shapes". In: *Algorithmic and Geometric Aspects of Robotics* 1 (1985), pp. 145–185.

[30] Hsing-Chia Chen and Wen-Hsiang Tsai. "Optimal security patrolling by multiple vision-based autonomous vehicles with omni-monitoring from the ceiling". In: *Proceedings of 2008 International Computer Symposium, Taipei, Taiwan*. 2008.

[31] Jingwei Chen and Nathan R Sturtevant. "Necessary and Sufficient Conditions for Avoiding Reopenings in Best First Suboptimal Search with General Bounding Functions". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 3688–3696.

[32] Shushman Choudhury et al. "Efficient large-scale multi-drone delivery using transit networks". In: *Journal of Artificial Intelligence Research* 70 (2021), pp. 757–788.

[33] Satyendra Singh Chouhan and Rajdeep Niyogi. "DiMPP: a complete distributed algorithm for multi-agent path planning". In: *Journal of Experimental & Theoretical Artificial Intelligence* (2017), pp. 1–20.

[34] Satyendra Singh Chouhan and Rajdeep Niyogi. "DMAPP: A Distributed Multi-agent Path Planning Algorithm". In: *Australasian Joint Conference on Artificial Intelligence*. Springer. 2015, pp. 123–135.

[35] Liron Cohen and Sven Koenig. "Bounded Suboptimal Multi-Agent Path Finding Using Highways". In: *International Joint Conference on Artificial Intelligence*. 2016, pp. 3978–3979.

[36] Liron Cohen et al. "Improved Solvers for Bounded-Suboptimal Multi-Agent Path Finding." In: *IJCAI*. 2016, pp. 3067–3074.

[37] Liron Cohen et al. "Optimal and Bounded Sub-Optimal Multi-Agent Motion Planning". In: *International Symposium on Combinatorial Search*. 2019, pp. 44–51.

[38] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[39] Jorge Cortes et al. "Coverage control for mobile sensing networks". In: *IEEE Transactions on robotics and Automation* 20.2 (2004), pp. 243–255.

[40] Michael Cui et al. "Compromise-free Pathfinding on a Navigation Mesh." In: *IJCAI*. 2017, pp. 496–502.

[41] Rina Dechter and Judea Pearl. "Generalized best-first search strategies and the optimality of A". In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 505–536.

[42] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[43] Kurt Dresner and Peter Stone. "A multiagent approach to autonomous intersection management". In: *Journal of artificial intelligence research* 31 (2008), pp. 591–656.

[44] Eva Dyllong and Cornelius Grimm. "Verified Adaptive Octree Representations of Constructive Solid Geometry Objects." In: Citeseer.

[45] Stefan Edelkamp and Stefan Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.

[46] Esra Erdem et al. "A General Formal Framework for Pathfinding Problems with Multiple Agents." In: *AAAI Conference on Artificial Intelligence*. 2013.

[47] Michael Erdmann and Tomas Lozano-Perez. "On multiple moving objects". In: *Algorithmica* 2.1-4 (1987), p. 477.

[48] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[49] James P Evans and Ralph E Steuer. "A revised simplex method for linear multiple objective programs". In: *Mathematical Programming* 5.1 (1973), pp. 54–72.

[50] Ariel Felner, Richard E Korf, and Sarit Hanan. "Additive pattern database heuristics". In: *Journal of Artificial Intelligence Research* 22 (2004), pp. 279–318.

[51] Ariel Felner et al. "Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding". In: *International Conference on Automated Planning and Scheduling*. 2018, pp. 83–87.

[52] Ariel Felner et al. "Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges". In: *International Symposium on Combinatorial Search*. 2017, pp. 29–37.

[53] Paolo Fiorini and Zvi Shiller. "Motion planning in dynamic environments using the relative velocity paradigm". In: *International Conference on Robotics and Automation*. IEEE. 1993, pp. 560–565.

[54] Paolo Fiorini and Zvi Shiller. "Motion planning in dynamic environments using velocity obstacles". In: *The International Journal of Robotics Research* 17.7 (1998), pp. 760–772.

[55] Paolo Fiorini and Zvi Shiller. "Motion planning in dynamic environments using velocity obstacles". In: *The International Journal of Robotics Research* 17.7 (1998), pp. 760–772.

[56] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic". In: *International Conference on Intelligent Robots and Systems*. IEEE. 2014, pp. 2997–3004.

[57] Graeme Gange, Daniel Harabor, and Peter J Stuckey. "Lazy CBS: Implict Conflict-Based Search Using Lazy Clause Generation". In: *International Conference on Automated Planning and Scheduling*. 2019, pp. 155–162.

[58] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. W.H. Freeman, New York, 2002.

[59] J. Gaschnig. "A Problem Similarity Approach to Devising Heuristics: First Results". In: *Readings in Artificial Intelligence*. 1981.

[60] Robert Ghrist, Jason M O'Kane, and Steven M LaValle. "Computing Pareto optimal coordinations on roadmaps". In: *The International Journal of Robotics Research* 24.11 (2005), pp. 997–1010.

[61] Elmer G Gilbert and SM Hong. "A new algorithm for detecting the collision of moving objects". In: *International Conference on Intelligent Robots and Systems*. IEEE. 1989, pp. 8–14.

[62] Kalin Gochev et al. "Path Planning with Adaptive Dimensionality." In: *International Symposium on Combinatorial Search*. AAAI Press, 2011.

[63] M. Goldenberg et al. "Enhanced Partial Expansion A*". In: *Journal of Artificial Intelligence Research (JAIR)* 50 (2014), pp. 141–187.

[64] Oded Goldreich. "Finding the shortest move-sequence in the graph-generalized 15-puzzle is NP-hard". In: *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*. Springer, 2011, pp. 1–5.

[65] Ofir Gordon, Yuval Filmus, and Oren Salzman. "Revisiting the Complexity Analysis of Conflict-Based Search: New Computational Techniques and Improved Bounds". In: *arXiv preprint arXiv:2104.08759* (2021).

[66] Pierre Hansen. "Bicriterion path problems". In: *Multiple criteria decision making theory and application*. Springer, 1980, pp. 109–127.

[67] Daniel Damir Harabor and Alban Grastien. "Online Graph Pruning for Pathfinding On Grid Maps." In: *AAAI Conference on Artificial Intelligence*. 2011.

[68] TE Harris and FS Ross. *Fundamentals of a method for evaluating rail net capacities*. Tech. rep. RAND CORP SANTA MONICA CA, 1955.

[69] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[70] Mark C Hendricks. *Rotated Ellipses And Their Intersections With Lines*. 2012.

[71] Patrick Chisan Hew. "The length of shortest vertex paths in binary occupancy grids compared to shortest r-constrained ones". In: *Journal of Artificial Intelligence Research* 59 (2017), pp. 543–563.

[72] Jörg Hoffmann. "Where 'ignoring delete lists' works: local search topology in planning benchmarks". In: *Journal of Artificial Intelligence Research* 24 (2005), pp. 685–758.

[73] Robert Holte et al. "Speeding Up Problem Solving by Abstraction: A Graph Oriented Approach". In: *Artificial Intelligence Journal* (1996).

[74] Wolfgang Hönig et al. "Multi-agent path finding with kinematic constraints". In: *Twenty-Sixth International Conference on Automated Planning and Scheduling*. 2016.

[75] Wolfgang Hönig et al. "Persistent and robust execution of mapf schedules in warehouses". In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 1125–1131.

[76] Shuli Hu et al. "Jump Point Search with Temporal Obstacles". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 31. 2021, pp. 184–191.

[77] Taoan Huang, Bistra Dilkina, and Sven Koenig. "Learning to Resolve Conflicts for Multi-Agent Path Finding with Conflict-Based Search". In: *AAAI Conference on Artificial Intelligence*. 2021.

[78] Yong K Hwang and Narendra Ahuja. "Gross motion planning – a survey". In: *ACM Computing Surveys (CSUR)* 24.3 (1992), pp. 219–291.

[79] Terry Jameson. *A Fuel Consumption Algorithm for Unmanned Aircraft Systems*. Tech. rep. DTIC Document, 2009.

[80] M.R. Jansen and N.R. Sturtevant. "Direction maps for cooperative pathfinding". In: *Artificial Intelligence and Interactive Digital Entertainment*. 2008.

[81] Pablo Jiménez, Federico Thomas, and Carme Torras. "3D collision detection: a survey". In: *Computers & Graphics* 25.2 (2001), pp. 269–285.

[82] Pablo Jiménez, Federico Thomas, and Carme Torras. "3D collision detection: a survey". In: *Computers & Graphics* 25.2 (2001), pp. 269–285.

[83] Berit Johannes. "Scheduling parallel jobs to minimize the makespan". In: *Journal of Scheduling* 9.5 (2006), pp. 433–452.

[84] Omri Kaduri, Eli Boyarski, and Roni Stern. "Algorithm selection for optimal multi-agent pathfinding". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30. 2020, pp. 161–165.

[85] Omri Kaduri, Eli Boyarski, and Roni Stern. "Experimental Evaluation of Classical Multi Agent Path Finding Algorithms". In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 12. 1. 2021, pp. 126–130.

[86] Sertac Karaman and Emilio Frazzoli. "Incremental sampling-based optimal motion planning". In: *Robotics: Science and Systems*. 2010.

[87] Narendra Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 1984, pp. 302–311.

[88] Lydia E Kavraki et al. "Probabilistic roadmaps for path planning in high- dimensional configuration spaces". In: *IEEE transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.

[89] Oussama Khatib. "Real-time obstacle avoidance for manipulators and mobile robots". In: *Autonomous robot vehicles*. Springer, 1986, pp. 396–404.

[90] Mokhtar M. Khorshid, Robert C. Holte, and Nathan R. Sturtevant. "A Polynomial-Time Algorithm for Non-Optimal Multi-Agent Pathfinding". In: *International Symposium on Combinatorial Search*. 2011.

[91] Stefan Kiel, Wolfram Luther, and Eva Dyllong. "Verified distance computation between non-convex superquadrics using hierarchical space decomposition structures". In: *Soft Computing* 17.8 (2013), pp. 1367–1378.

[92] Stephen Kloder and Seth Hutchinson. "Path planning for permutation-invariant multirobot formations". In: *IEEE Transactions on Robotics* 22.4 (2006), pp. 650–665.

[93] Sinan Kockara et al. "Collision detection: A survey". In: *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*. IEEE. 2007, pp. 4046–4051.

[94] Sinan Kockara et al. "Collision detection: A survey". In: *2007 IEEE International Conference on Systems, Man and Cybernetics*. IEEE. 2007, pp. 4046–4051.

[95] Andrew Kopeikin et al. "Unmanned aircraft system swarm for radiological and imagery data collection". In: *AIAA Scitech 2019 Forum*. 2019, p. 2286.

[96] Richard E Korf. "Depth-first iterative-deepening: An optimal admissible tree search". In: *Artificial intelligence* 27.1 (1985), pp. 97–109.

[97] Daniel Martin Kornhauser, Gary L Miller, and Paul G Spirakis. "Coordinating pebble motion on graphs, the diameter of permutation groups, and applications". MA thesis. M. I. T., Dept. of Electrical Engineering and Computer Science, 1984.

[98] Aaron R Kraft. "Abstraction Hierarchies for Multi-Agent Pathfinding". PhD thesis. University of Denver, 2017.

[99] James J Kuffner and Steven M LaValle. "RRT-connect: An efficient approach to single-query path planning". In: *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*. Vol. 2. IEEE. 2000, pp. 995–1001.

[100] Edward Lam et al. "Branch-and-cut-and-price for multi-agent pathfinding". In: *International Joint Conference on Artificial Intelligence*. 2019, pp. 1289–1296.

[101] Jean-Claude Latombe. *Robot motion planning*. Vol. 124. Springer Science & Business Media, 2012.

[102] Steven M LaValle. "Rapidly-exploring random trees: A new tool for path planning". In: (1998).

[103] Steven M LaValle and Seth A Hutchinson. "Optimal motion planning for multiple robots having independent goals". In: *IEEE Transactions on Robotics and Automation* 14.6 (1998), pp. 912–925.

[104] Hak-Tae Lee and Thomas F Romer. "Automating the process of airport surface node-link model generation". In: *Journal of Guidance, Control and Dynamics* 34.4 (2011).

[105] Kian Seng Lee et al. "Autonomous patrol and surveillance system using unmanned aerial vehicles". In: *2015 IEEE 15th International Conference on Environment and Electrical Engineering (EEEIC)*. IEEE. 2015, pp. 1291–1297.

[106] Louise Leenen, Johannes Vorster, and Hermanus le Roux. "A constraint-based solver for the military unit path finding problem". In: *Proceedings of the 2010 spring simulation multiconference*. 2010, pp. 1–8.

[107] Jiaoyang Li et al. "Disjoint Splitting for Multi-Agent Path Finding with Conflict-Based Search". In: *International Conference on Automated Planning and Scheduling*. 2019, pp. 279–283.

[108] Jiaoyang Li et al. "Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search." In: *IJCAI*. Vol. 2019. 2019, pp. 442–449.

[109] Jiaoyang Li et al. "Lifelong multi-agent path finding in large-scale warehouses". In: *arXiv preprint arXiv:2005.07371* (2020).

[110] Jiaoyang Li et al. "Multi-Agent Pathfinding for Large Agents". In: *AAAI Conference on Artificial Intelligence*. 2019, pp. 7627–7634.

[111] Jiaoyang Li et al. "New Techniques for Pairwise Symmetry Breaking in Multi-Agent Path Finding". In: *International Conference on Automated Planning and Scheduling*. 2020, pp. 6087–6095.

[112] Jiaoyang Li et al. "Pairwise symmetry reasoning for multi-agent path finding search". In: *Artificial Intelligence* (2021), p. 103574.

[113] Jiaoyang Li et al. "Symmetry-Breaking Constraints for Grid-Based Multi-Agent Pathfinding". In: *AAAI Conference on Artificial Intelligence*. 2019, pp. 6087–6095.

[114] Yandong Liu et al. "A novel swarm robot simulation platform for warehousing logistics". In: *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE. 2017, pp. 2669–2674.

[115] Leonardo Lozano and Andrés L Medaglia. "On an exact method for the constrained shortest path problem". In: *Computers & Operations Research* 40.1 (2013), pp. 378–384.

[116] Tomas Lozano-Perez. "Spatial planning: A configuration space approach". In: *Autonomous robot vehicles*. Springer, 1990, pp. 259–271.

[117] Hang Ma. "A Competitive Analysis of Online Multi-Agent Path Finding". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 31. 2021, pp. 234–242.

[118] Hang Ma and Sven Koenig. "Optimal target assignment and path finding for teams of agents". In: *International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2016, pp. 1144–1152.

[119] Hang Ma, TK Satish Kumar, and Sven Koenig. "Multi-agent path finding with delay probabilities". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1. 2017.

[120] Hang Ma et al. "Lifelong multi-agent path finding for online pickup and delivery tasks". In: *arXiv preprint arXiv:1705.10868* (2017).

[121] Hang Ma et al. "Multi-Agent Path Finding with Deadlines". In: *International Joint Conference on Artificial Intelligence*. 2018, pp. 417–423.

[122] Hang Ma et al. "Overview: Generalizations of multi-agent path finding to real-world scenarios". In: *arXiv preprint arXiv:1702.05515* (2017).

[123] David R Mazur. "Combinatorics : A Guided Tour". In: (2010).

[124] Justin V Montoya et al. "Analysis of airport surface schedulers using fast-time simulation". In: *2013 Aviation Technology, Integration, and Operations Conference*. 2013, p. 4275.

[125] Edward F Moore. "The shortest path through a maze". In: *Proc. Int. Symp. Switching Theory, 1959*. 1959, pp. 285–292.

[126] Matthew Moore and Jane Wilhelms. "Collision detection and response for computer animation". In: *ACM Siggraph Computer Graphics*. ACM. 1988.

[127] Jonathan Morag et al. "Studying Online Multi-Agent Path Finding". In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 12. 1. 2021, pp. 228–230.

[128] Robert Morris et al. "Planning, scheduling and monitoring for airport surface operations". In: *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

[129] Robert Morris et al. "Self-driving aircraft towing vehicles: A preliminary report". In: *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[130] John M Mulvey. "A classroom/time assignment model". In: *European Journal of Operational Research* 9.1 (1982), pp. 64–70.

[131] Bernhard Nebel. "On the computational complexity of multi-agent pathfinding on directed graphs". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30. 2020, pp. 212–216.

[132] Jerzy Neyman and Egon S Pearson. "The testing of statistical hypotheses in relation to probabilities a priori". In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 29. Cambridge University Press. 1933, pp. 492–510.

[133] Van Nguyen et al. “Generalized target assignment and path finding using answer set programming”. In: *Twelfth Annual Symposium on Combinatorial Search*. 2019.

[134] T Alastair J Nicholson. “Finding the shortest route between two points in a network”. In: *the computer journal* 9.3 (1966), pp. 275–280.

[135] Nils J Nilsson. *A mobile automaton: An application of artificial intelligence techniques*. Tech. rep. SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER, 1969.

[136] Carole Nissoux, Thierry Siméon, and J-P Laumond. “Visibility based probabilistic roadmaps”. In: *International Conference on Intelligent Robots and Systems*. Vol. 3. IEEE. 1999, pp. 1316–1321.

[137] Colm Ó'Dúnlaing and Chee K Yap. “A "retraction" method for planning the motion of a disc”. In: *Journal of Algorithms* 6.1 (1985), pp. 104–111.

[138] Luigi Palmieri, Sven Koenig, and Kai O Arras. “RRT-based nonholonomic motion planning using any-angle path biasing”. In: *International Conference on Robotics and Automation*. IEEE. 2016, pp. 2775–2781.

[139] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[140] Judea Pearl and Jin H Kim. “Studies in semi-admissible heuristics”. In: *IEEE transactions on pattern analysis and machine intelligence* 4 (1982), pp. 392–399.

[141] Jufeng Peng and Srinivas Akella. “Coordinating multiple robots with kinodynamic constraints along specified paths”. In: *Journal of Artificial Intelligence Research* 24.4 (2005), pp. 295–310.

[142] Paul A Peterson and Michael C Loui. “The general maximum matching algorithm of Micali and Vazirani”. In: *Algorithmica* 3.1 (1988), pp. 511–533.

[143] Mike Phillips and Maxim Likhachev. "Sipp: Safe interval path planning for dynamic environments". In: *International Conference on Robotics and Automation*. IEEE. 2011, pp. 5628–5635.

[144] Andrew B Philpott. "Continuous-time shortest path problems and linear programming". In: *SIAM journal on control and optimization* 32.2 (1994), pp. 538–552.

[145] Poom Pianpak et al. "A distributed solver for multi-agent path finding problems". In: *Proceedings of the First International Conference on Distributed Artificial Intelligence*. 2019, pp. 1–7.

[146] Francisco-Javier Pulido, Lawrence Mandow, and José-Luis Pérez-de-la Cruz. "Dimensionality reduction in multiobjective shortest path search". In: *Computers & Operations Research* 64 (2015), pp. 60–70.

[147] Daniel Ratner and Manfred Warmuth. "The (n2- 1)-puzzle and related relocation problems". In: *Journal of Symbolic Computation* 10.2 (1990), pp. 111–137.

[148] Jingyao Ren et al. "MAPFAST: A Deep Algorithm Selector for Multi Agent Path Finding using Shortest Path Embeddings". In: *arXiv preprint arXiv:2102.12461* (2021).

[149] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. "Multi-objective Conflict-based Search Using Safe-interval Path Planning". In: *arXiv preprint arXiv:2108.00745* (2021).

[150] Aristides AG Requicha and Herbert B Voelcker. "Constructive solid geometry". In: (1977).

[151] Nicolas Rivera, Carlos Hernández, and Jorge A Baier. "Grid Pathfinding on the 2k Neighborhoods." In: *AAAI Conference on Artificial Intelligence*. 2017, pp. 891–897.

[152] Juan Jesús Roldán-Gómez, Eduardo González-Gironda, and Antonio Barrientos. "A survey on robotic technologies for forest firefighting: Applying drone

swarms to improve firefighters' efficiency and safety". In: *Applied Sciences* 11.1 (2021), p. 363.

[153] Malcolm Ryan. "Constraint-Based Multi-Robot Path Planning". In: *International Conference on Robotics and Automation*. IEEE. 2010, pp. 922–928.

[154] Lorenzo Sabattini, Cristian Secchi, and Cesare Fantuzzi. "Arbitrarily shaped formations of mobile robots: artificial potential fields and coordinate transformation". In: *Autonomous Robots* 30.4 (2011), p. 385.

[155] Qandeel Sajid, Ryan Luna, and Kostas E Bekris. "Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives." In: *International Symposium on Combinatorial Search*. 2012.

[156] Guillaume Sartoretti et al. "Primal: Pathfinding via reinforcement and imitation multi-agent learning". In: *IEEE Robotics and Automation Letters* 4.3 (2019), pp. 2378–2385.

[157] Jürgen Scherer et al. "An autonomous multi-UAV system for search and rescue". In: *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*. 2015, pp. 33–38.

[158] Tomer Shahar et al. "Safe Multi-Agent Pathfinding with Time Uncertainty". In: *Journal of Artificial Intelligence Research* 70 (2021), pp. 923–954.

[159] G. Sharon et al. "Conflict-Based Search For Optimal Multi-Agent Path Finding". In: *AAAI Conference on Artificial Intelligence*. 2012, pp. 563–569.

[160] Guni Sharon et al. "Conflict-Based Search for Optimal Multi-Agent Pathfinding". In: *Artificial Intelligence Journal* 219 (2015), pp. 40–66.

[161] Guni Sharon et al. "Pruning techniques for the increasing cost tree search for optimal multi-agent pathfinding". In: *Fourth Annual Symposium on Combinatorial Search*. 2011.

[162] Guni Sharon et al. "The Increasing Cost Tree Search for Optimal Multi-agent Pathfinding". In: *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI, 2011.

[163] Guni Sharon et al. "The Increasing Cost Tree Search for Optimal Multi-agent Pathfinding". In: *Artificial Intelligence Journal* (2013), pp. 470–495.

[164] David Silver. "Cooperative Pathfinding." In: *Artificial Intelligence and Interactive Digital Entertainment*. 2005, pp. 117–122.

[165] Jerry Slocum and Dic Sonneveld. "The 15 puzzle: how it drove the world crazy". In: *The puzzle that started the craze of* (1880).

[166] JW Smeltink et al. "Optimisation of airport taxi planning". In: (2003).

[167] Jamie Snape et al. "The hybrid reciprocal velocity obstacle". In: *IEEE Transactions on Robotics* 27.4 (2011), pp. 696–706.

[168] Kiril Solovey and Dan Halperin. "K-Color Multi-Robot Motion Planning". In: *The International Journal of Robotics Research* 33.1 (2014), pp. 82–97.

[169] Peng Song and Vijay Kumar. "A Potential Field Based Approach to Multi-Robot Manipulation". In: *International Conference on Robotics and Automation*. Vol. 2. IEEE. 2002, pp. 1217–1222.

[170] Arvind Srinivasan et al. "Algorithms for Discrete Function Manipulation". In: *IEEE International Conference on Computer-Aided Design*. IEEE. 1990, pp. 92–95.

[171] Trevor Scott Standley. "Finding Optimal Solutions to Cooperative Pathfinding Problems." In: *AAAI Conference on Artificial Intelligence*. 2010, pp. 28–29.

[172] Trevor Scott Standley and Richard E. Korf. "Complete Algorithms for Cooperative Pathfinding Problems." In: *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI, 2011.

[173] Roni Stern et al. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks". In: *International Symposium on Combinatorial Search*. 2019, pp. 151–159.

[174] N. Sturtevant and M. Buro. "Improving collaborative pathfinding using map abstraction". In: *Artificial Intelligence and Interactive Digital Entertainment*. 2006, pp. 80–85.

[175] N. Sturtevant and R. Jansen. "An analysis of map-based abstraction and refinement". In: *Symposium on Abstraction, Reformulation and Approximation (SARA)* (2007), pp. 344–358.

[176] Avneesh Sud et al. "Real-time path planning in dynamic virtual environments using multiagent navigation graphs". In: *IEEE transactions on visualization and computer graphics* 14.3 (2008), pp. 526–538.

[177] Pavel Surynek. "A SAT-Based Approach to Cooperative Path-Finding Using All-Different Constraints." In: *International Symposium on Combinatorial Search*. 2012.

[178] Pavel Surynek. "An Optimization Variant of Multi-Robot Path Planning Is Intractable." In: *AAAI Conference on Artificial Intelligence*. 2010, pp. 1–3.

[179] Pavel Surynek. "Multi-agent path finding with continuous time and geometric agents viewed through satisfiability modulo theories (SMT)". In: *Twelfth Annual Symposium on Combinatorial Search*. 2019.

[180] Pavel Surynek et al. "Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective". In: *European Conference on Artificial Intelligence*. 2016, pp. 810–818.

[181] Richard S Sutton and Andrew G Barto. "Reinforcement learning: An introduction". In: *Robotica* 17.2 (1999), pp. 229–235.

[182] Jiří Švancara et al. "Online multi-agent pathfinding". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 7732–7739.

[183] Herbert G Tanner, Ali Jadbabaie, and George J Pappas. "Stable flocking of mobile agents, Part I: Fixed topology". In: *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*. Vol. 2. IEEE. 2003, pp. 2010–2015.

[184] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *Switching and Automata Theory, 1971., 12th Annual Symposium on*. IEEE. 1971, pp. 114–121.

[185] Larry Arthur Taylor. *Pruning duplicate nodes in depth-first search*. University of California, Los Angeles, 1997.

[186] Shyni Thomas, Dipti Deodhare, and M Narasimha Murty. "Extended Conflict-Based Search for the Convoy Movement Problem". In: *IEEE Intelligent Systems* 30.6 (2015), pp. 60–70.

[187] Sebastian Trüg, Jörg Hoffmann, and Bernhard Nebel. "Applying automatic planning systems to airport ground-traffic control–a feasibility study". In: *Annual Conference on Artificial Intelligence*. Springer. 2004, pp. 183–197.

[188] Richard Valenzano et al. "Using alternative suboptimality bounds in heuristic search". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 23. 1. 2013.

[189] Jur P Van Den Berg and Mark H Overmars. "Prioritized motion planning for multiple robots". In: *International Conference on Intelligent Robots and Systems*. IEEE. 2005, pp. 430–435.

[190] Wouter G Van Toll, Atlas F Cook IV, and Roland Geraerts. "A navigation mesh for dynamic environments". In: *Computer Animation and Virtual Worlds* 23.6 (2012), pp. 535–546.

[191] Kyle Vedder and Joydeep Biswas. "X*: Anytime multi-agent path finding for sparse domains using window-based iterative repairs". In: *Artificial Intelligence* 291 (2021), p. 103417.

[192] Juan Irving Solis Vidana et al. "Representation-Optimal Multi-Robot Motion Planning using Conflict-Based Search". In: *IEEE Robotics and Automation Letters* (2021).

[193] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. "Geometric containers for efficient shortest-path computation". In: *Journal of Experimental Algorithmics (JEA)* 10 (2005), pp. 1–3.

[194] Glenn Wagner and Howie Choset. "M*: A complete multirobot path planning algorithm with performance bounds". In: *International Conference on Intelligent Robots and Systems*. 2011.

[195] Glenn Wagner, Minsu Kang, and Howie Choset. "Probabilistic path planning for multiple robots with subdimensional expansion". In: *International Conference on Robotics and Automation*. IEEE. 2012, pp. 2886–2892.

[196] Thayne T. Walker, David Chan, and Nathan R. Sturtevant. "Using Hierarchical Constraints to Avoid Conflicts in Multi-Agent Pathfinding". In: *International Conference on Automated Planning and Scheduling*. 2017, pp. 316–324.

[197] Thayne T. Walker and Nathan R. Sturtevant. "Collision Detection for Agents in Multi-Agent Pathfinding". In: *arXiv preprint arXiv:1908.09707* (2019).

[198] Thayne T Walker, Nathan R Sturtevant, and Ariel Felner. "Extended Increasing Cost Tree Search for Non-Unit Cost Domains." In: *International Joint Conference on Artificial Intelligence*. 2018, pp. 534–540.

[199] Thayne T Walker, Nathan R Sturtevant, and Ariel Felner. "Generalized and Sub-Optimal Bipartite Constraints for Conflict-Based Search". In: *AAAI Conference on Artificial Intelligence*. 2020.

[200] Thayne T. Walker et al. "Conflict-Based Increasing Cost Search". In: *International Conference on Automated Planning and Scheduling*. 2021.

[201] Cindy Wang and Adi Botea. "Fast and Memory-Efficient Multi-Agent Pathfinding". In: *International Conference on Automated Planning and Scheduling*. 2008.

[202] Jiangxing Wang et al. "A new constraint satisfaction perspective on multi-agent path finding: preliminary results". In: *18th International Conference on Autonomous Agents and Multi-Agent Systems*. 2019.

[203] Ko-Hsin Cindy Wang, Adi Botea, et al. "Tractable Multi-Agent Path Planning on Grid Maps." In: *International Joint Conference on Artificial Intelligence*. Vol. 9. Pasadena, California. 2009, pp. 1870–1875.

[204] Xuan Wang and Qinghai Zuo. "Aircraft taxiing route planning based on airport hotspots". In: *AIP Conference Proceedings* 1839.1 (2017), p. 020088. DOI: 10.1063/1.4982453.

[205] Daniel S Weld. "Recent Advances in AI Planning". In: *AI magazine* 20.2 (1999), pp. 93–93.

[206] Emo Welzl. "Constructing the visibility graph for n-line segments in O (n2) time". In: *Information Processing Letters* 20.4 (1985), pp. 167–171.

[207] B de Wilde, Adriaan W Ter Mors, and Cees Witteveen. "Push and rotate: a complete multi-agent pathfinding algorithm". In: *Journal of Artificial Intelligence Research* 51 (2014), pp. 443–492.

[208] Steven A Wilmarth, Nancy M Amato, and Peter F Stiller. "MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space". In: *International Conference on Robotics and Automation*. Vol. 2. IEEE. 1999, pp. 1024–1031.

[209] Xiaolin Wu. "An Efficient Antialiasing Technique". In: *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '91. New York, NY, USA: ACM, 1991, pp. 143–152. ISBN: 0-89791-436-8. DOI: 10.1145/122718.122734.

[210] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating hundreds of cooperative, autonomous vehicles in warehouses". In: *AI magazine* 29.1 (2008), pp. 9–9.

[211] Hong Xu et al. "A warning propagation-based linear-time-and-space algorithm for the minimum vertex cover problem on giant graphs". In: *International Con-*

ference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Springer. 2018, pp. 567–584.

[212] Konstantin Yakovlev and Anton Andreychuk. "Any-Angle Pathfinding for Multiple Agents Based on SIPP Algorithm". In: *arXiv preprint arXiv:1703.04159* (2017).

[213] Zhang Yang. "A Model to Convert Airport Geographic and Geometric Information into a Node-Link Network". MA thesis. Blacksburg, Virginia, USA: Virginia Polytechnic Institute and State University, 2014.

[214] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. "A* with Partial Expansion for Large Branching Factor Problems." In: *AAAI/IAAI*. 2000.

[215] Jingjin Yu and Steven M LaValle. "Multi-agent path planning and network flow". In: *Algorithmic Foundations of Robotics X*. Springer, 2013, pp. 157–173.

[216] Jingjin Yu and Steven M LaValle. "Planning optimal paths for multiple robots on graphs". In: *International Conference on Robotics and Automation*. IEEE. 2013, pp. 3612–3617.

[217] Jingjin Yu and Steven M. LaValle. "Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs". In: *AAAI Conference on Artificial Intelligence*. 2013, pp. 1443–1449.

[218] Han Zhang et al. "Multi-Agent Pathfinding with Mutex Propagation". In: *International Conference on Automated Planning and Scheduling*. 2020.

A. Closed-Form Collision Detection for Circular Agents

Figure 3.4 shows an example of two-agent motion for fixed velocity (a) and initial velocity with fixed acceleration (b). Computing the time and duration of conflict for two circular agents can be done by solving equations for the squared distance between agents [48].

A.0.1. Constant Velocity

Given $P_1 = \langle x_1, y_1 \rangle$, the start position of agent 1, and $P_2 = \langle x_2, y_2 \rangle$, the start position of agent 2, velocity vectors $V_1 = \langle vx_1, vy_1 \rangle$, $V_2 = \langle vx_2, vy_2 \rangle$, and radii r_1, r_2 respectively, the location in time of an agent is defined as:

$$P' = P + Vt \tag{A.1}$$

The following equation specifies the squared distance between the centers of the agents over time:

$$sqdist(t) = {V_\Delta}^2 t^2 + 2V_\Delta \cdot P_\Delta t + {P_\Delta}^2 \tag{A.2}$$

where

$P_\Delta = P_1 - P_2$
$V_\Delta = V_1 - V_2$

Via substitution, this equation is simplified to a quadratic equation:

$$sqdist(t) = at^2 + bt + c_0 \tag{A.3}$$

where

$a = {V_\Delta}^2$
$b = 2V_\Delta \cdot P_\Delta$
$c_0 = {P_\Delta}^2$

A collision will occur when the squared distance between the agents is less than or equal to the squared sum of the radii, giving the following inequality.

$$at^2 + bt + c_0 \leq (r_1 + r_2)^2$$

Solving the inequality gives the equation for collision between the agent's edges:

$$0 \geq at^2 + bt + c_0 - (r_1 + r_2)^2$$

$$sqEdgeDist(t) = at^2 + bt + c \tag{A.4}$$

where

$c = P_\Delta^2 - (r_1 + r_2)^2$

Solving equation A.4 for t will determine the exact times where the squared distance between agent's edges is zero – the time when collision occurs. Section A.1 discusses the process for determining the conflict interval for using this equation.

A.0.2. Initial Velocity with Constant Acceleration

Equation (A.4) can be extended for constant acceleration. Given $P_1 = \langle x_1, y_1 \rangle$, the start position of agent 1, and $P_2 = \langle x_2, y_2 \rangle$, the start position of agent 2, velocity vectors $V_1 = \langle vx_1, vy_1 \rangle$, $V_2 = \langle vx_2, vy_2 \rangle$, acceleration vectors $A_1 = \langle ax_1, ay_1 \rangle$, $A_2 = \langle ax_2, ay_2 \rangle$ and radii r_1, r_2 respectively, the location in time of an agent is defined as:

$$P' = P + Vt + \frac{At^2}{2} \tag{A.5}$$

The following inequality specifies the collision condition as a quartic equation:

$$at^4 + bt^3 + ct^2 + dt + e_0 \leq (r_1 + r_2)^2 \tag{A.6}$$

where

$a = \frac{{A_\Delta}^2}{4}$
$b = A_\Delta \cdot V_\Delta$
$c = A_\Delta \cdot P_\Delta + {V_\Delta}^2$
$d = 2V_\Delta \cdot P_\Delta$
$e_0 = {P_\Delta}^2$

for

$P_\Delta = P_1 - P_2$
$V_\Delta = V_1 - V_2$
$A_\Delta = A_1 - A_2$

which gives the equation for the squared distance between circular edges:

$$sqEdgeDist(t) = at^4 + bt^3 + ct^2 + dt + e \tag{A.7}$$

where

$e = {P_\Delta}^2 - (r_1 + r_2)^2$

Again, solving for t will yield the time of collision, which is discussed further in the next section.

A.1. Computing the Exact Conflict Interval For Circular Agents

The exact conflict interval is determined by solving for the roots of (A.4) or (A.7) using the quadratic and quartic formulas respectively. These solutions assume that both agents are at P_1 and P_2 at the same time. However, if there is an offset in time, e.g. agent 1 starts moving at time t_1 and agent 2 starts moving at time t_2, then P_Δ must be adjusted to reflect this offset by projecting the position of the earlier agent to be at the position when the later agent starts its motion. If the earlier agent were agent 1, the adjustment would be as follows:

$$P_\Delta = P_1 + V_1(t_2 - t_1) - P_2 \tag{A.8}$$

Otherwise, the adjustment will be analogously done for agent 2. In the case of acceleration, the position and velocity must be adjusted (again, assuming agent 1 starts early) as:

$$P_\Delta = P_1 + V_1(t_2 - t_1) + \frac{A_1(t_2 - t_1)^2}{2} - P_2 \tag{A.9}$$

$$V_\Delta = V_1 + A_1(t_2 - t_1) - V_2 \tag{A.10}$$

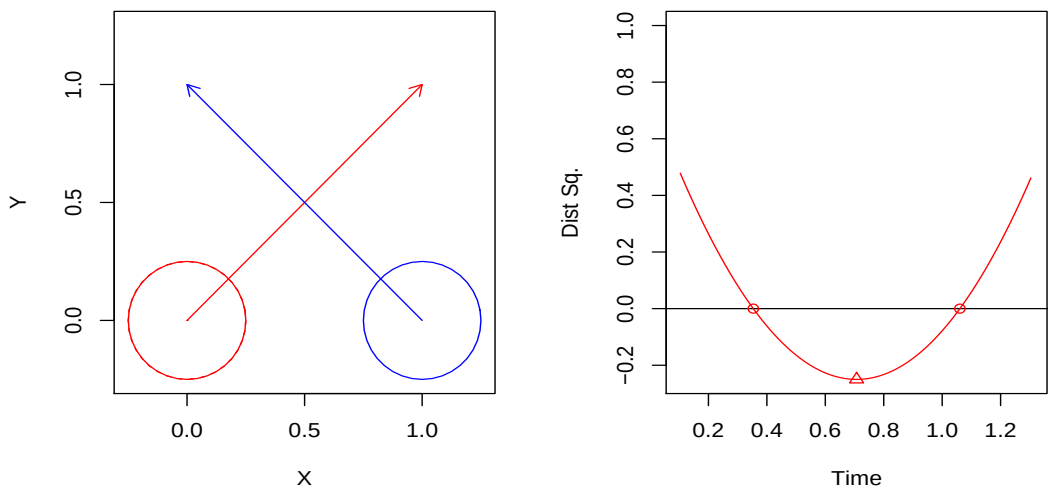


Figure A.1.: Agents Trajectories and Corresponding Squared Distance Plot

A.1.1. Constant Velocity

For the quadratic form, if the discriminant ($b^2 - 4ac$) is less than zero, V_1 and V_2 are parallel and no collision will ever occur. Assuming the discriminant is positive, the collision interval is defined as the roots of the quadratic formula:

$$t_{interval} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{A.11}$$

In the case of a double root, the edges of the agents just touch, but no overlap actually occurs (assuming open intervals). See Figure A.1 for an example of two-agent motion and the resulting squared-distance plot. When the distance is less than zero, there is overlap of the agents. Given this interval, it is possible to determine whether a collision will occur in the future and at what time, or if the agents are currently colliding.

A.1.2. Initial Velocity with Constant Acceleration

This case uses the quartic formula to find roots to (A.7). The quartic formula will yield 4 roots, some of which may be imaginary resulting in 0, 1 or 2 conflict intervals. Imaginary roots will tell us the time(s) at which agents are locally closest together, but do not actually overlap (local minima). Imaginary roots are always double roots, and can be discarded. If all 4 roots are imaginary, the agents never overlap. If there is a double real root, then the two agents touch edges at exactly one point in time, creating an instantaneous interval.

Because our equation is based on distance, the quartic function will always be concave up. Hence, the overlapping intervals can only be between roots 1,2 and 3,4. If roots 1,2 and/or 3,4 are real, then the agents continuously overlap between 1,2 and/or 3,4 respectively. Four real roots means that the objects overlap twice, continuously between root pairs 1,2 and 3,4. This is possible because agents may have curved trajectories. See Figure 3.4 (b) for an example.

A.2. Determining Exact Minimum Delay or Velocity Adjustment for Conflict Avoidance for Circular Agents

It is often useful, not just to determine if and when agents are going to collide, but to determine a delay time to avoid collision.

A.2.1. Exact Delay for Constant Velocity

In order to determine the minimum delay required for an agent to avoid conflict, we adjust (3) to incorporate $\delta = t_2 - t_1$, a delay variable, by plugging equation (A.8) into equation (A.2) to get:

$$sqEdgeDist(t, \delta) = At^2 + Bt\delta + C\delta^2 + Dt + E\delta + F \tag{A.12}$$

where

$$
\begin{aligned}
A &= {V_\Delta}^2 \\
B &= 2(V_1^2 - V_1 \cdot V_2) \\
C &= {V_1}^2 \\
D &= 2(P_2 \cdot V_2 - P_2 \cdot V_1 + P_1 \cdot V_2 - P_1 \cdot V_1) \\
E &= -2(P_2 \cdot V_1 + P_1 \cdot V_1) \\
F &= {V_\Delta}^2 - (r_1 + r_2)^2
\end{aligned}
$$

Equation (A.12) is the standard form of a conic section. Note that the sign of both A and C are positive, therefore, this conic section will always be an ellipse, except for two degenerate cases: (1) agents' motion is parallel and (2) at least one agent is waiting in place. Fortunately, both cases are easy to detect and solve. The conversion of (8) to canonical form for an ellipse will not be covered here, nor is it necessary.

Figure A.2(a) shows an example of agent trajectories, the squared distance plot (equation (A.4)) when $delay = 0$, and the resulting elliptical conic section (equation (A.12)). Note that the horizontal line at $\delta = 0$ passes through the ellipse at the exact same time points that the squared distance plot does. If agent 1 were to delay by ϵ, the horizontal line would move up, resulting in a different collision interval (see Figure A.2(b)). If agent 2 were to delay by ϵ, the horizontal line would move down, again resulting in a different collision interval. The question we want to solve is: what value of *delay* will result in no collision? In other words, we want to find the positive value of δ, such that the radii of the agents just touch, i.e. (A.12) yields a double root. The targeted delay interval is derived by determining the top and bottom extrema of the ellipse [70] which is described by:

$$delayRange = center_\delta \pm \frac{\sqrt{(2BD-4AE)^2+4(4AC-B^2)(D^2-4AF)}}{2(4AC-B^2)} \quad \text{(A.13)}$$

where $center_\delta$ is the y-coordinate of the ellipse center:

$center_\delta = \frac{BD-2AE}{4AC-B^2}$

The collision times of the endpoints of the delayRange (where horizontal lines touch the top and bottom of the ellipse) are computed via:

$$collisionTimes = \frac{-B(delayRange) - D}{2A} \tag{A.14}$$

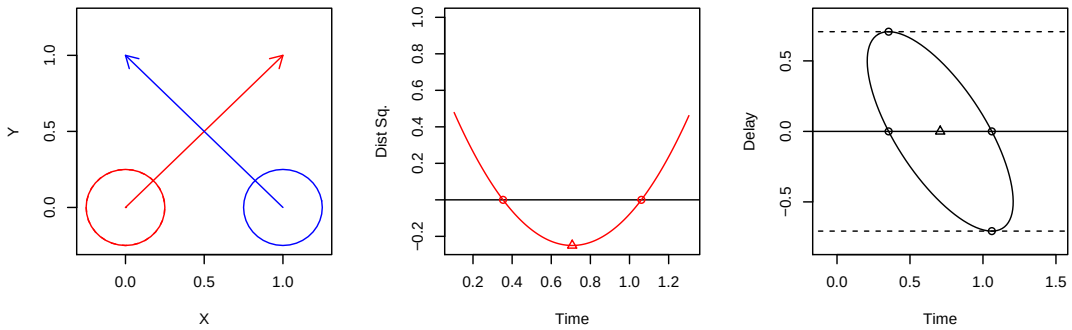
Note that (A.13) is undefined when the discriminant is negative, which can only happen for $a = 0$ or $c = 0$. This can only happen when agents' motion vectors are parallel (moving the same or opposite directions) or either agent is waiting in place. These cases are easy to detect.

When the motion is not of infinite length, i. e. segmented motion, we must also take into account the beginning and end of the duration of motion. Effectively, we treat agents as if they appear at their start time and disappear at their end time. When the movement of agents 1 and 2 start at t_1 and t_2 and end at t'_1 and t'_2 respectively, we measure time relative to $t_0 = MIN(t_1, t_2)$ and $t_{max} = MIN(t'_1, t'_2)$. In the case that $\delta = t_1 - t_2$ is outside of the range *delayRange* as calculated via (A.13), no collision will occur. If either of the collision times (as calculated in (A.14) for each point in *delayRange* occur before t_0, or after t_{max}, the delay times need to be re-computed for t_0 or t_{max} as necessary using (A.15). An example where t_{max} occurs too early is shown by the vertical dashed line in Figure A.3.

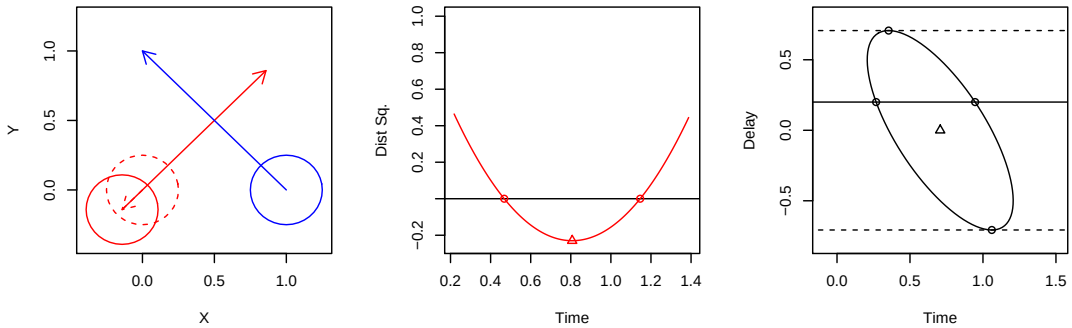
This yields the algorithm detailed in Algorithm A.1 for computing the unsafe interval for segmented motion. The algorithm is straightforward and utilizes the following additional formulas:

The value of δ, given a time which is derived from (A.12), solved for δ:

$$\delta = \frac{-\sqrt{(Bt+E)^2 - 4C(t(At+D)+F)} + Bt + E}{2C} \tag{A.15}$$



(a)



(b)

Figure A.2.: (a) Agent trajectories, squared distance plot and ellipse showing collision intervals for Varying *delay* and (b) the same trajectories where the red agent is delayed delayed by 0.2 seconds

, and for the leftmost t coordinate on the ellipse:

$$minCollisionTime = center_t - \frac{\sqrt{(2BE-4CD)^2+4(4AC-B^2)(E^2-4CF)}}{2(4AC-B^2)} \quad (A.16)$$

where $center_t$ is $\frac{BE-2CD}{4AC-B^2}$

At Algorithm A.1, lines 5-9 check the actual delay ($\delta = t2 - t1$) between the two agents against the unsafe delay range per equation (A.13). If there is no collision (e.g. in the case of parallel movement) or δ does not fall inside the unsafe range, no collision will occur. Lines 10-23 compute the unsafe time interval per equation (A.14) and then adjust the endpoints accordingly per t0 and tmax using equation (A.15).

The final result is the adjusted unsafe interval for agent 1. This interval can now be used to instruct agent 1 not to start execution of its action inside the interval (e.g. by starting its action sooner or later). Note that the unsafe interval for agent 2 is the negated interval for agent 1 – [-range[2],-range[1]].

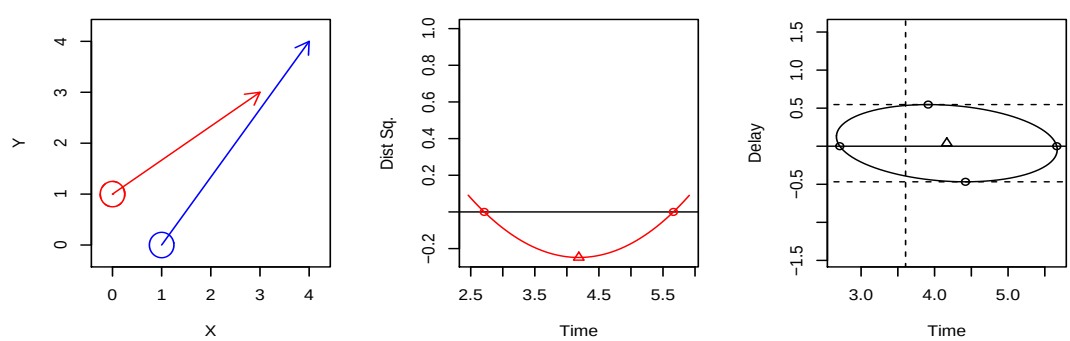


Figure A.3.: An example where the maximum delay time happens after the first agent arrives at its destination.

Algorithm A.1. Unsafe Interval Computation for Segmented Motion

```
INPUT: P1,P2,V1,V2,t1,t2,t1′,t2′,r1,r2
t0=MAX(t1,t2)
tmax←MIN(t1′,t2′)
δ=t2-t1
// Execute equation (A.13) to get unsafe delay range
range←delayRange(P1,P2,V1,V2,r1,r2)
if range=∅ or range[1]> δ or range[2]< δ then
    return NO COLLISION
// Execute equation (A.14) to get unsafe time range
collisionTimes←delayTimes(P1,P2,V1,V2,r1,r2)
minCollisionTime←MIN(collisionTimes)
maxCollisionTime←MAX(collisionTimes)
// Truncate delay for motion time segments
if minCollisionTime<t0 then
    // Get delay for t0 via (A.15)
    range[1]←delayAtTime(P1,P2,V1,V2,r1,r2,t0)
if maxCollisionTime<t0 then
    // Get delay for tmax via (A.15)
    range[2]←delayAtTime(P1,P2,V1,V2,r1,r2,t0)
// Return the unsafe interval by adding the delay to the start time
return [t0+range[1],MIN(tmax,t0+range[2])]
```

A.2.2. Exact Delay for Initial Velocity with Constant Acceleration

The equivalent conic equation for 4th order bivariates is called a quartic plane curve. A closed-form solution for unsafe intervals is still an open question. However, an interative solution has been formulated for the constant velocity case which is generalizable to this case [6].

The algorithm starts by evaluating (A.7) at $t0$, retrieving an initial upper bound from the interval which is closest to and greater than $t0$. Then performs a binary search, from both ends of the interval until the interval is determined within a predetermined accuracy threshold. Binary search is a well known algorithm and will not be repeated here.

A.2.3. Minimum Velocity Change for Constant Velocity

In order to determine the minimum velocity change necessary to avoid collision for segmented motion, a VO is created as shown in Figure A.4 which is similar to Figure 3.6, but with motion segments added. Motion segments are shown as dotted arrows with large points at the beginning and end of the segment. Velocities that lie on the segment are the only valid choices, hence a velocity that lies just outside of the VO as shown in diagram (b) is desirable for determining the minimum necessary change to avoid collision. There may be kinematic constraints on agents, such as a maximum velocity.

The following steps can be undertaken to determine the appropriate action for the agent, which may result in the agent waiting in place or using a new velocity:

1. Detect if a collision will occur inside the segments. This can be done via equation (A.4). We immediately return $\emptyset$ if no collision will occur.

2. Construct a VO, then compute a new velocity that lies on the segment and intersects with the edges of the VO as shown in Figure A.4 (b) for agent A. Ths can be done using a formula for the line intersection point [7] of the motion vector and both of the VO tangent lines.

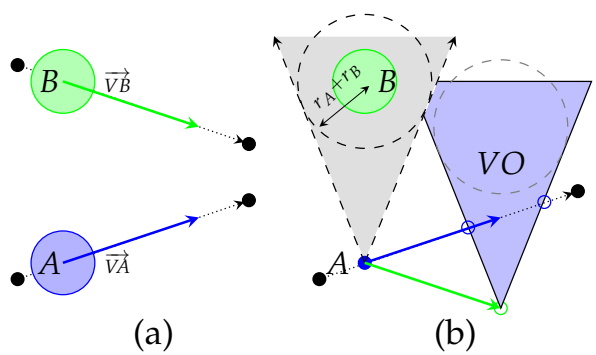


Figure A.4.: Velocity Obstacle (VO) construction based on (a) two agents moving on edges. (b) The minimum change for safe velocity is determined by the intersection points of the edge and the velocity obstacle.

- Return new velocity if either of the velocities at the intersection points are kinematically feasible.

3. Construct and check a VO for a new velocity for agent B.
   - Return new velocity if either of the velocities at the intersection points are kinematically feasible.
4. If the current state of the agent will allow it to wait in place, compute the delay using Algorithm A.1.
   - Return original velocity and new delay.
5. Otherwise, return NO SOLUTION

A.2.4. Summary

We provided derivations for computing the exact interval of collision between two agents with constant velocity or intial velocity with constant acceleration. We have additionally derived a formulation for computing unsafe intervals (the range of start times in which agents come into collision) for two circular agents with constant velocity and differing start times. An algorithm was then shown for computing the unsafe intervals in the case of segmented motion. Finally, an algorithm for computing safe velocities and delay times was outlined.

B. Proofs for Temporally-Relative Duplicate Pruning

Recall from Section 3.3.1 that the motivation for temporally-relative duplicate pruning is to ensure completeness in MAPF and MAPF_Q. We now introduce a novel proof of completeness for the case when no solution exists for MAPF and MAPF_Q based on temporally-relative duplicates. Later in this section, we extend this proof for CBICS.

Recall that temporally-relative duplicate pruning removes successors during expansion which have been visited before in a temporally relative sense. The procedure adjusts joint successor states $S' \in \mathbf{S}'$ of S to be temporally-relative $\Delta_t(\mathbf{S}')$, then compares them to temporally-relative states in $\Pi(S)$. Any member of $\Delta_t(\mathbf{S}')$ that is identical to any member of $\Delta_t(\Pi(S))$ is pruned from the search.

Temporally-relative duplicate pruning in MAPF and MAPF_Q has two effects:

1. It renders the search space finite.

2. It eliminates sub-optimal solutions from consideration in the search.

These claims do not generally hold for MAPF_R. However, we show that elimination of temporally-relative duplicates renders MAPF and MAPF_Q algorithms complete. This is done by proving the two claims above; namely that temporally-relative duplicate pruning renders the search space finite, guaranteeing termination even when no solution exists, and that it never eliminates optimal solutions from being found. We also show why termination cannot be guaranteed in MAPF_R in the case that a solution does not exist.

Assumption B.0.1. G, the graph for motion planning MAPF agents is finite.

Observation B.0.2. *Because G is finite, the single-agent branching factor is also finite.*

Observation B.0.3. *Because G is finite, arbitrary wait times are not allowed – only fixed duration wait actions.*

Assumption B.0.4. k, the number of agents is finite.

Observation B.0.5. *Because the single-agent branching factor is finite, and k is finite, the multi-agent branching factor is finite.*

Observation B.0.6. *Even if the graph G is finite, MAPF and MAPF$_Q$ have infinite state spaces due to the time dimension. The time domain for MAPF is $\mathbb{Z}_+$ which is countably infinite and the time domain for MAPF$_Q$ is $\mathbb{Q}_+$ which is also countably infinite.*

Observation B.0.7. *The search spaces for MAPF and MAPF$_Q$ (without temporally-relative duplicate pruning) are infinite. For example it is possible for agents to wait in place at their start location forever without coming into conflict.*

Lemma B.0.8. *Eliminating temporally-relative duplicates renders the search space for MAPF and MAPF$_Q$ finite.*

Proof. Let W be the set of edge weights for E: $W{=}\{\forall e \in E; w(e)\}$.

The domain of $s.t$ for the search space is finite iff each $\pi{\in}\Pi$ is finite in length. Finite-length paths however, cannot be guaranteed without augmentation per Observation B.0.7. But the domain of temporally-relative times $\Delta_t(S).t$ is finite. Per the definition of joint actions, each $a{\in}A$ has temporal overlap with all other actions in A, therefore each $s.t$ in $\Delta_t(S).t$ can be no larger than the largest value in W. Thus, the upper bound on the size of the domain of each $s.t$ in $\Delta_t(S).t$ is defined as

$$D_t{=}\frac{\mathrm{MAX}(W)}{\mathrm{GCD}(W)}$$

, the largest value in W divided by the greatest common denominator (GCD) of W. For example, given $W{=}\{.2, .5, 4\}$ the GCD is .1 and the largest value is 4, hence the size of

the domain for each $s.t$ is 40. Note that the GCD is not defined for irrational numbers, hence this result is not generally valid for MAPF_R. The upper bound on D_T, the size of the domain of $\Delta_t(S).t$ is the total number of k-length strings over D_t,

$$D_T = {D_t}^k$$

. This is because for a multi-agent state, each agent can take on any temporally-relative value in D_t. D_T is a finite value. In our example, D_t is 40 and for k=3 D_T=40^3. Note, D_T=1 for MAPF.

Concerning the domain of $S.v$, the possible vertex locations of the agents, the total number of valid configurations is the total number of k-permutations of vertices in the graph

$$D_V = \frac{|V|!}{(|V|-k)!}$$

which is $O(|V|^k)$, a finite value.

There are only $D_V D_T$ ways to arrange agents and temporally-relative times. This is a finite value. Thus removing temporally-relative duplicates ensures that the search space is finite. □

We now show that no optimal solution is removed by temporally-relative duplicate pruning.

Lemma B.0.9. *No optimal solution exists to a MAPF or MAPF_Q instance such that a temporally-relative state is visited twice.*

Proof. Let $\Pi = [S^0, .., S^d]$ be a solution of length d from start to goal for k agents.

Recall from Section 3.3.2 that in MAPF_Q it is possible for a set of vertices to be repeated more than once in Π in optimal solutions. However, the relative times of the single agent states in a joint state must be unique, which is proved as follows:

Assume that some temporally-relative joint state $\Delta_t(S)$ is visited twice in $\Delta_t(\Pi)$, once at index i and again at index j. Π cannot be an optimal solution because the concatenation of $\Pi_{[0,i)} + S^i + \Pi_{(j,d]}$ (after updating times in $\Pi_{(j,d]}$ by subtracting $S^j.t - S^i.t$)

is a solution with contiguous paths from start to goal which is shorter than Π. Hence, in MAPF_Q any Π that visits the same temporally-relative joint state twice cannot be a shortest path. □

The question of completeness and optimality ultimately depends on the specific design of MAPF algorithms. However, we can now show that temporally-relative duplicate pruning, cannot preclude a MAPF algorithm from being optimal and complete.

Theorem B.0.10. *Eliminating temporally-relative duplicates cannot preclude completeness and optimality in MAPF and* MAPF_Q.

Proof. For completeness, we must show two things: That an algorithm will terminate if no solution exists, and that if a solution exists, it will not be eliminated by temporally-relative duplicate pruning.

Per Lemma B.0.8, with elimination of temporally-relative duplicates there is a finite search space in MAPF and MAPF_Q. Therefore, in the case that no solution to the problem exists, eventually, the search space will be explored completely. Therefore, the algorithm is guaranteed to terminate.

Per Lemma B.0.9, no optimal solution can ever be eliminated by temporally-relative duplicate pruning. Although some sub-optimal solutions may be eliminated, optimal solutions will remain. Furthermore, if at least one solution exists for a problem instance, at least one of them must be an optimal solution. Therefore, temporally-relative duplicate pruning cannot preclude completeness nor optimality. □

C. Proofs for PCS

In this appendix, we prove the following properties of PCS:

1. PCS is guaranteed to return a set of *context-optimal* paths in P. Context optimal means that PCS returns a set of feasible lowest-cost joint state paths, which conform to motion and cost constraints.

2. PCS is guaranteed to terminate in the case that no such paths exist.

3. PCS is guaranteed to compute valid sets of cost-conditional motion constraints.

PCS Proof Part 1: Guarantees when a solution exists

For proving (1), we show that costs must increase toward optimal costs. Then we show that PCS will never prune a joint state that belongs to an optimal solution. Then we show that PCS will never prune a context-optimal path. Finally, that PCS will not violate motion constraints nor cost constraints.

Lemma C.0.1. *After a finite number of expansions in PCS, the quantity $g(S)$ is guaranteed to increase.*

Proof. Because all edge weights in G have positive values and waiting at the goal has no cost, for $s_i, s_j \in S$ at least one of their respective successors s'_i or s'_j is guaranteed to have increased cost: $g(s'_i) \geq g(s_i)$ and $g(s'_j) \geq g(s_j)$ s.t. $g(s'_i)+g(s'_j) > g(s_i)+g(s_j)$. Therefore, $g(S') > g(S)$ for all $S' \in \mathbf{S}'$ where $\mathbf{S}'$ is the set of all successors of S (See Algorithm 2.2). Because OPEN is ordered by g-cost, $g(S)$ is guaranteed to never decrease. Because the branching factor is finite, $g(S)$ is guaranteed to increase after a finite number of iterations (if a feasible solution is not found first). □

We have established that costs are guaranteed to increase in PCS, an essential part of making progress toward optimal solutions. We now show that if any optimal solutions exist, PCS will find them. First, we show that pruning in PCS retains optimal solutions.

Lemma C.0.2. *For any problem instance in $I_{\text{L-ADM}}$, PCS will not prune any joint state belonging to the set of all context-optimal paths.*

Proof. There are two reasons that states are pruned in PCS: (1) Temporally-relative duplicate pruning (see Algorithm 2.2 lines 16-25). (2) Pruning when individual f-costs are above $r_i.ub$ and $r_j.ub$ respectively (See Algorithm 6.2 line 41).

(1) Per Lemma B.0.9 no joint state belonging to a lowest-cost solution can be eliminated by temporally-relative duplicate pruning. However, as noted in Section 3.3.3 this does not apply to PCS because it operates on a subset of agents. Essentially, temporally-relative duplicate pruning could remove a joint state belonging to a lowest-cost solution if applied in the context of a subset of agents due to the influence of agents outside the subset. Therefore no pruning is allowed until after the last constraint time (See Algorithm 2.2 line 16) in order to ensure that no outside agents influence is disregarded in the context of PCS.

We can show this is correct. By contradiction, assume that some temporally-relative duplicate is pruned in PCS and it eliminates a lowest-cost solution in the context of all agents. It must be the case that a context-sub-optimal path is required by agents i and j (because temporally-relative duplicate pruning will never prune joint states from lowest-cost solutions per Lemma B.0.9). Setting cost constraints aside, the only reason that a context-sub-optimal path would be needed, is to avoid another agent from the larger context. However, if no motion constraint exists, it must be that either (I) no other agent has a conflict with agent i nor agent j or (II) such a conflict has not been discovered yet in the high-level search. Case (I) contradicts our assumption – if there is no conflict with another agent, temporally-relative duplicate pruning in PCS cannot preclude a context-optimal solution from being found. Case (II) will be resolved

only by the high-level detecting a conflict and re-engaging PCS for a new node with a motion constraint for the conflict. Hence, temporally-relative duplicate pruning after the last constraint time cannot preclude an optimal solution from being found.

Additionally, context-sub-optimal solutions need to be allowed when any of the cost lower bounds are are greater than the lowest possible cost combination. Therefore temporally-relative pruning is only allowed when f-costs are above the lower bounds in addition to the maximum motion constraint time (See Algorithm 2.2 line 16). Because f-costs are computed using an admissible heuristic, no joint state whose true f-cost is below the lower bound cost constraint can ever be pruned. Thus no context-sub-optimal solution is precluded unless its actual cost is above the lower bound cost constraints.

(2) is correct because in the case of an infinite $r_i.ub$ and $r_j.ub$ no pruning occurs. In the case $r_i.ub$ or $r_j.ub$ is finite, pruning states which have f-costs over these upper bounds respectively will not remove any lowest-cost path whose actual cost is in the intervals r_i and r_j because the f-costs are computed using an admissible heuristic. □

Note that Lemma C.0.2 still holds in the case of an inconsistent heuristic. Because OPEN is ordered by g-cost, re-expansions can never occur. F-costs are only used to prune, hence, because inconsistent heuristics are admissible, a lowest-cost path is still guaranteed not to be pruned.

Lemma C.0.3. *For any problem instance in $I_{\text{L-ADM}}$, PCS will not return any paths in P which violate motion constraints or are infeasible.*

Proof. PCS cannot return paths in P which violate motion constraints because TIME-AWAREJOINTEXPANSION does not generate any single-agent states which violate motion constraints (See Algorithm 2.2 lines 5, 8).

PCS cannot return paths in P which are infeasible because infeasible solutions are not allowed in P based on goal criteria (see Algorithm 6.2 line 30). States are marked as infeasible if the constituent actions are infeasible when generated (see Algorithm 2.2 line 13) or if all of their parents are infeasible. This is ensured by first marking children

of infeasible joint states as infeasible (see Algorithm 12) and replacing infeasible joint states in OPEN with any duplicates that are feasible (see Algorithm 44).

Let Π_c^* be the set of context-optimal joint state paths. There are two attributes for parents of a joint state which must be considered, namely, the parent's f-cost, and the parent's *feasibility*. If the parent's f-cost is greater than $c(\Pi_c^*)$ it will never be expanded. Let $\mathbf{S}$ the set of all parents of S' with f-cost less than or equal to $c(\Pi_c^*)$. The f-cost of S' is also less than or equal to $c(\Pi_c^*)$.

If the constituent actions of S' are infeasible, it will always be marked infeasible. Otherwise there are three cases that determine if it is marked feasible:

1. All $S \in \mathbf{S}$ are feasible.

2. All $S \in \mathbf{S}$ are infeasible.

3. At least one $S \in \mathbf{S}$ is feasible.

Case 1: S' will be marked feasible as soon as it is generated and will never change to infeasible.

Case 2: S' will be marked infeasible as soon as it is generated and will never change when it is revisited via a different parent.

Case 3: S' must eventually be marked feasible, otherwise, any optimal solution containing S' could be omitted from P, or some motion constraint M could be erroneously generated for actions terminating or originating from S'. This is the main part that needs to be proven.

If S' is initially marked feasible when it is generated, it will never change to infeasible. If S' is initially marked infeasible when it is generated, it is guaranteed to eventually be marked feasible. By contradiction, assume S' is not marked feasible before PCS terminates. It must be that S' was never visited via a feasible parent in $\mathbf{S}$. S' could not have been pruned because per Lemma C.0.2, S' never would have been generated in the first place if it falls under the criteria of a temporally-relative duplicate, and pruning by f-cost violates our assumptions about $\mathbf{S}$. Another possibility is that

some $h(S) \in \mathbf{S}$ is never expanded, but this violates our assumptions about the f-cost of $\mathbf{S}$ as well, since per Lemma C.0.4 all S would be expanded. If $h(S)$ under-estimates the cost to the goal, there are two possible outcomes: (I) it visits or generates a child at $S'.V$ with a different time and g-cost than S' or (II) it visits S'. (I) violates our assumptions about S, since S is not actually a parent of S' in this case. In case (II) S' will be set to feasible, contradicting our assumption.

Based on Algorithm 6.2 line 30, PCS will include paths in P for all joint-states that satisfy the goal conditions which are marked feasible, and these are guaranteed to have a traceable path back to the root joint state with only feasible actions. Additionally, no action for any path in P can contain single-agent actions which are blocked by motion constraints. □

Lemma C.0.4. *For any problem instance in $I_{\text{L-ADM}}$, PCS will find all context-optimal paths with unique cost combinations if any exist.*

Proof. Let $\mathbf{\Pi^*_c} = \{\Pi^*_1 = \{\pi_i, \pi_j\}, .., \Pi^*_n\{\pi_i, \pi_j\}\}$ be the set of all context-optimal paths for a two-agent MAPF instance. Let S^* be the last state that was expanded in Π^*. By contradiction, if PCS did not find any Π^*, it must be that:

(1) the successor $S'^* \in \Pi^*$ of S^* was not generated or

(2) S'^* was never considered for expansion (i.e., never came to the top of OPEN).

We can rule out (1) because there is a finite branching factor, and PCS generates all successors, (Algorithm 2.2, lines 5-8) S'^* is guaranteed to be generated. Additionally, S'^* cannot have been pruned Per lemma C.0.2. We can also rule out (2) because f-costs are computed using an admissible heuristic, therefore no S' having $f^*(s_i) \leq r_i.ub$ or $f^*(s_j) \leq r_j.ub$ where $f^* = c^*$ will ever be omitted from OPEN. Additionally, because costs are non-decreasing per Lemma C.0.1, S'^* will eventually arrive at the top of OPEN and because PCS will continue performing expansions until all joint states in open with f-cost less than or equal to $c*$ are expanded (See Algorithm 6.2 lines 30 and 9) S'^* is guaranteed to be expanded because $g(S'^*) \leq c*$. Otherwise, a feasible path must not exist.

□

Lemma C.0.5. *PCS will not include any solution in P which violates cost constraints.*

Proof. PCS cannot include a solution in P which violates cost constraints because the criteria for adding a solution to P ensures that cost constraints are met (see Algorithm 6.2 lines 28, 30). □

Theorem C.0.6. *For any problem instance in $I_{\text{L-ADM}}$, PCS returns all cost-combination-unique, context-optimal paths in P.*

Proof. Per Lemma C.0.2 no joint state in a lowest-cost solution will ever be pruned. Per Lemma C.0.3 no infeasible solution can be included in P. Per Lemma C.0.4 P will include all context-optimal solutions. Per Lemma C.0.5 no solution in P can violate the cost constraints. □

It does not matter whether PCS uses an consistent heuristic or not, as long as the heuristic is admissible, the guarantees hold.

PCS Proof Part 2: Guarantees when no solution exists

Recall the two parts to completeness are guaranteed termination when a solution exists and when no solution exists. Part 1 showed that PCS will find solutions if they exist and terminate. We now show the second part of completeness.

Theorem C.0.7. *PCS is complete for MAPF and $MAPF_Q$ in $I_{\text{L-ADM}}$.*

Proof. Per Theorem C.0.6 PCS is guaranteed to find solutions if any exist, subject to constraints. This satisfies the first requirement for completeness.

One of two cases guarantee termination in the case that no solution exists: (1) When both $lb_i.ub$ and $lb_j.ub$ are finite, PCS will never add states violating either of these cost bounds to OPEN, thus the search space is rendered finite because costs are guaranteed to increase per Lemma C.0.1. The g-cost for agent j or the g-cost for agent j or both increase with every expansion, eventually reaching $lb_i.ub$ and/or $lb_j.ub$. (2) when either $lb_i.ub$ and $lb_j.ub$ are infinite, because PCS performs temporally-relative duplicate

pruning when f-costs are above the lower cost bounds and when $S.t$ is greater than the maximum motion constraint time. Per Lemma B.0.8, the search space is rendered finite by the use of temporally-relative duplicate pruning. Guaranteeing that the search will terminate.

Thus, PCS is guaranteed to terminate whether a solution to $I_{\text{L-ADM}}$ exists or not. □

PCS Proof Part 3: Guarantees on valid motion constraint sets

We start by showing that PCS generates valid motion constraint sets. Recall that "*valid*" means any action a_i blocked by a cost-conditional motion constraint m_i conflicts with all actions available to the second agent at the same time. This is the mutually-disjunctive property. Because a potentially large set of actions could be used by agent j, we restrict the set based on f-costs. For a complete proof, we must show two things: (1) that the pair of agents' conflicting actions is computed correctly and (2) that the cost limit $m_i.c$ used to restrict the set is correct.

For the following two lemmas let a_i (resp. a_j) be an action for agent i composed of (s_i, s_i'). Let A_j be the set of all actions available to agent j that have time overlap with a_i (i.e., in the range $[s_i.t, s_i'.t]$). Let $A_j^c \subseteq A_j$ be the set of actions such that $\forall a_j \in A_j$, a_j either directly conflicts with a_i or is marked infeasible due to having all infeasible parents. Let $\overline{A_j} = A_j \setminus A_j^c$ be the set of non-conflicting actions. Let B_i^c and B_j^c be the sets of valid motion constraints in the sense that all $m_i \in B_i^c$ are mutually disjunctive with all $m_j \in B_j^c$ (equivalently all $m_i.a$ conflict with all $m_j.a$ and vice-versa) in the same time frame.

Lemma C.0.8. *PCS is guaranteed to compute a set of valid motion constraints.*

Proof. Because PCS initializes a motion constraint m_i when its constituent action a_i is in direct conflict with some action for agent j or its parents had conflicts (marked with $feasible = false$), it will never omit a possible motion constraint from consideration (see Algorithm 2.2 lines 12 and 30 and Algorithm 6.2 line 18.

When $f(s'_i) \leq r_i.ub$ and respectively for s'_j, successors are added to open (see Algorithm 6.2 lines 40 and 46), therefore only m_i with $f(m_i.a{=}a_i) \leq r_i.ub$ will be considered as a possible motion constraint. When $f(a_i){=}f^*(a_i)$, a_i is guaranteed to be on a path within the cost range r_i. Thus m_i is guaranteed to be mutually disjunctive (and thus a member of B^c_i) because PCS is guaranteed to expand and perform conflict checks with all a_j with cost $\leq r_j.ub$.

However, when the f-cost is under-estimated, that is, $f(a_i) < f^*(a_i)$ and its true f-cost is greater than $r_i.ub$, a_i is not on a path in the cost range r_i. In this case, it is possible that some prefix path through a_i was not expanded in the search. Therefore, it is possible that some parent of a_i does not conflict with a parent of a_j, hence m_i with $m_i.a{=}a_i$ would not be mutually conflicting with A_j in the right time frame, hence not a member of B^c_i.

Therefore, PCS eliminates any m such that $f(m.a) < f^*(m.a)$ by computing the sets B_i and B_j (see Algorithm 6.2 lines 32-38) which include all infeasible actions in paths with costs in r_i and r_j respectively. Thus B_i (resp. B_j) contains only infeasible actions with $f(a_i){=}f^*(a_i)$. B^c_i (resp. B^c_j) is a subset of B_i because B_i may contain infeasible actions that are not mutually disjunctive. Because all actions in M_i are mutually disjunctive with those in M_j, then $B^c_i{=}M_i \cap B_i$. Because PCS computes this intersection (see Algorithm 6.2 lines 10 and 11) only valid motion constraints are returned. □

Now that we have shown that the correct set of motion constraints is computed, we must show that the cost limits $m_i.c$ are correct. That is, for each m_i in the set B^c_i, $m_i.c$ is set such that it will be turned off if a path found for agent j could contain an action that does not conflict with $m_i.a$. Otherwise, m_i could block an action in a feasible solution, resulting in incompleteness or sub-optimality.

Lemma C.0.9. *PCS is guaranteed to compute correct cost limits for cost conditional motion constraints.*

Proof. The cost range for a constraint which blocks a_i is:

$$F(A_j^c) = F(A_j) \backslash F(\overline{A_j})$$

Thus $F(A_j^c)$ is the entire conflicting cost range minus the non-conflicting cost range. In Algorithm 6.2, the upper bound on this range is computed on line 21 (for $F(A_j)$) and line 24 subtracts lower bound (for $F(\overline{A_j})$).

The *exact* valid cost range is $F^*(A_j^c)$ where F^* is a *perfect* cost range, computed using a perfect heuristic. In this case, $m_i.c{=}F^*(A_j^c).ub$, therefore, if $r_j.ub{\leq}F^*(A_j^c).ub$ (which causes m_i to be "turned on", blocking a_i) no $a_j{\in}\overline{A_j}$ can be expanded by PCS (since the f-costs for these actions are higher than $r_j.ub$), hence no feasible solution will be blocked. Conversely, when $r_j.ub{>}F^*(A_j^c).ub$ (causing m_i to be "turned off") $a_j{\in}\overline{A_j}$ will be expanded and m_i (being turned off) will not block any feasible solution.

In the case that single-agent heuristics are admissible but under-informed $F(A_j^c).ub{\leq}F^*(A_j^c).ub$. Because $\forall a_j{\in}\overline{A_j}$, $f(a_j){\notin}F^*(A_j^c)$ it follows that $\forall a_j{\in}\overline{A_j}$, $f(a_j){\notin}F(A_j^c)$. Still, no $a_j{\in}\overline{A_j}$ can be expanded when m_i is turned on since the f-costs for these actions are higher than $r_j.ub$. This may result in some $a_j{\in}\overline{A_j}$ not being expanded when m_i is turned off, but it still cannot lead to blocking a feasible solution.

□

Theorem C.0.10. *PCS is guaranteed to compute valid cost-conditional motion constraints.*

Proof. Per Lemma C.0.8 PCS computes the correct (mutually disjunctive) set of motion constraints and per Lemma C.0.9 the cost conditions of the motion constraints are guaranteed to only be turned on when they cannot block a feasible solution. Therefore PCS computes valid cost-conditional motion constraints. □