Efficient and Effective Techniques for Large-Scale Multi-Agent Path Finding

by

Jiaoyang Li

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

August 2022

Acknowledgements

I had never expected my Ph.D. life to be such a wonderful experience. I am grateful for receiving a lot of help from my colleagues, friends, and family.

First and foremost, I want to express my deepest gratitude to my advisor Sven Koenig for guiding me through every step of doing research. Sven has deeply impacted and inspired me. His enthusiasm, dedication, and perfectionism in research have shaped my personality as a researcher. He offers the greatest freedom for me to work on anything that excites me, but he is also always there when I need his advice. I could not have hoped for a better advisor.

I am grateful to have the rest of my dissertation committee, namely T. K. Satish Kumar, Nora Ayanian, Satyandra K. Gupta, and Brian C. Williams, for their valuable time, guidance, support, and feedback. I especially want to thank Satish for helping me with my writing during the early years of my Ph.D. I also thank Bistra Dilkina for helpful feedback on an earlier version of this dissertation.

At USC, I have worked with great colleagues in our research group. I want to extend my appreciation to Hang Ma for introducing me to MAPF during my visit to USC as an undergraduate student. Since then, Hang has continually provided me with invaluable advice, shared his first-hand experience with me, and answered thousands of my questions with lots of patience. I would also like to thank Shao-Hung Chan, Weizhe Chen, Liron Cohen, Wolfgang Hönig, Taoan Huang, Christopher Leet, Kexuan Sun, Tansel Uras, Hong Xu, Han Zhang, Hejia Zhang, Yi Zheng, and other colleagues at USC for the many casual, enjoyable discussions and fruitful collaborations that directly influenced this dissertation.

Next, I would like to thank my long-time collaborators from Monash University, Australia. I am extremely grateful to Peter J. Stuckey and Daniel Harabor for the countless hours of brainstorming with me. Despite the continents and time differences between us, we have met virtually every week for more than four years. I learned a great deal from their visionary insights and hands-on experiences. Our ongoing and fruitful collaboration generated numerous research ideas and led to a considerable part of this dissertation, including Chapters 4 and 6. I also thank other folks in the optimization group at Monash University for being a great educational resource in my studies on optimization and for supporting me during my one-year visit to Monash University. My special thanks go to Zhe Chen, who has offered me a lot of help and spent many hours coding and debugging with me. Part of the implementation of MAPF-LNS was done by him.

I would also like to thank my collaborators from Ben-Gurion University, Israel. I am extremely grateful to Ariel Felner for introducing me to the world of heuristic search and for teaching me how to write good papers and do proper research. I still remember that when I wrote my first paper, Ariel Skyped with me at 6 am his time every morning and walked me through my draft sentence by sentence. I am grateful to Ariel and Eli Boyarski for their contribution to the work presented in Chapter 3. I also thank Dor Atzmon, Roni Stern, and many others from Ben-Gurion University for inspiring discussions on heuristic search and for taking care of me during my visits.

I want to extend my sincere thanks to Wheeler Ruml from the University of New Hampshire for teaching me bounded-suboptimal searches and for sharing his knowledge and experience with me. Working with Wheeler is an inspirational experience, and our collaboration led to the work presented in Chapter 5.

While this dissertation focuses on the algorithmic side of MAPF, I have also worked with many excellent researchers with different expertise on the applicational side of MAPF that provided valuable insights and contributed indirectly to this dissertation. Many thanks to Joseph W. Durham, Andrew Tinka, and Scott Kiesel from Amazon Robotics for the warehouse robots project, Robert Morris from NASA Ames for the airport surface operation project, Jingkai Chen from MIT for the multi-robot motion planning and robotic arm assembly projects, Le Hai Vu and The Hoang from

Monash University for the autonomous intersection management project, and Naveed Haghani, Nathan R. Sturtevant, Pavel Surynek, Thayne T. Walker, Julian Yarkony, and many others for other MAPF-related projects.

I have been fortunate to work with many talented Masters and undergraduate students, including Eugene Lin, Minghua Liu, Sumanth Varambally, Jiangxing Wang, Qinghong Xu, Moli Yang, Shuyang Zhang, Xinyi Zhong, and many others. I learned a lot from them and the mentoring experience.

Finally, I would like to thank my family and friends. I am indebted to my parents for their unconditional love and support. Their love and support define who I am today.

The research presented in this dissertation was supported by National Science Foundation under grant numbers 1319966, 1409987, 1724392, 1817189, 1837779, and 1935712 as well as a gift from Amazon.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Abbreviations

MAPF	Multi-Agent Path Finding; See Definition 2.1.
CBS	Conflict-Based Search; See Algorithm 2.1.
PP	Prioritized Planning; See Section 2.2.3.
PP^R	Prioritized Planning with random restarts; See Section 2.2.3.
PPS	Parallel-Push-and-Swap; See Section 2.2.3.
CT	Constraint Tree; See Section 2.3.1.
ICBS	Improved CBS, i.e., CBS with the conflict prioritization technique; See Section 2.3.2.1.
MDD	Multi-Valued Decision Diagram; See Definition 2.3.
CG	Admissible CBS heuristic based on cardinal conflict graphs; See Section 3.1.
CBSH	ICBS with the CG heuristic; See Section 3.1.1.
DG	Admissible CBS heuristic based on pairwise dependency graphs; See Section 3.2.
WDG	Admissible CBS heuristic based on weighted pairwise dependency graphs; See Section 3.3.
CBSH2	ICBS with the WDG heuristic; See Section 3.3.1.

CBSH2-RTC	CBSH2 with all symmetry reasoning techniques (and bypassing conflicts); See Algorithm 4.6.
ECBS	Enhanced CBS; See Section 5.1.
EES	Explicit Estimation Search (EES); See Section 5.2.2.
EECBS	Explicit Estimation CBS; See Algorithm 5.1.
EECBS(*w*)	EECBS with suboptimality factor *w*.
LNS	Large Neighborhood Search; See Chapter 6.
ALNS	Adaptive LNS; See Section 6.1.3.4.
CP	Number of colliding pairs; See Section 6.2.1.
PMDO	Pathfinding with Dynamic Obstacles; See Section 6.2.2.
SIPPS	Safe Interval Path Planing with Soft constraints; See Algorithm 6.3.
EECBS*	EECBS with SIPPS.
EECBS*(*w*)	EECBS with SIPPS and suboptimality factor *w*.

Abstract

Recent advances in robotics have laid the foundation for building large-scale multi-agent systems. One fundamental task in many multi-agent systems is to navigate agents in a shared environment to their target locations without colliding with each other or obstacles. Applications include evacuation, formation control, object transportation, traffic management, search and rescue, autonomous driving, drone swarm coordination, video game character control, and warehouse automation, to list a few. One well-studied abstract model for this problem is known as *Multi-Agent Path Finding* (MAPF). It is defined on a general graph with given start and target vertices for agents on this graph. Each agent is allowed to wait at its current vertex or move to an adjacent vertex from one discrete timestep to the next one. We are asked to find a path for each agent such that no two agents are at the same vertex or cross the same edge at any timestep (because this would result in a collision) and minimize the sum of the path costs of all agents or similar optimization criteria.

MAPF is NP-hard to solve optimally (and, in some cases, even bounded-suboptimally) in general. Existing algorithms for solving MAPF either have limited scalability (such as optimal and bounded-suboptimal algorithms), generate costly solutions (such as rule-based algorithms), or may fail to find any solutions for hard MAPF instances (such as prioritized algorithms). In this dissertation, we develop fundamental techniques to overcome the shortcomings of these MAPF algorithms, namely techniques that reduce the runtimes of optimal and bounded-suboptimal MAPF algorithms and techniques that improve the solution quality and success rates of rule-based, prioritized, and other non-optimal MAPF algorithms.

The state-of-the-art optimal and bounded-suboptimal MAPF algorithms often use the framework of planning paths for each agent independently and systematically resolving collisions afterward. Intelligently searching the collision-resolution space often finds good solutions faster than traditional methods that search the joint-state space of the agents. But still, searching the collision-resolution space can be computationally expensive for two reasons.

The first reason is due to a phenomenon called pairwise symmetry, which occurs when two agents have many different paths to their target vertices, all of which appear promising, but every combination of them results in a collision. So, when we search the collision-resolution space, we need to enumerate many (or, in most cases, an exponential number of) combinations of such colliding paths before finding a pair of collision-free paths for the two agents. Therefore, in this dissertation, we identify a number of different symmetries and show that they arise commonly in practice and enumerating them often leads to timeout failures. We develop symmetry-breaking techniques to detect the symmetries during the search and resolve them in a single branching step while preserving the (bounded-sub)optimality and completeness of the optimal and bounded-suboptimal MAPF algorithms.

The second reason is due to the blind search in the extremely large size of the collision-resolution space. Although resolving pairwise symmetries can reduce the size of the collision-resolution space, when solving MAPF instances with many agents, the space is still too large for the (bounded-sub)optimal MAPF algorithms to find a solution and prove its (bounded sub)optimality with a reasonable runtime. On the one hand, in order to prove (bounded sub)optimality, (bounded-sub)optimal MAPF algorithms have to examine a large number of states (= sets of paths) with small costs in the collision-resolution space and resolve their collisions. Therefore, in this dissertation, we develop admissible heuristics for each state, which underestimate the minimum cost increase of the paths in the state when all their collisions are resolved, to reduce the number of states that (bounded-sub)optimal MAPF algorithms need to examine and thus speed up their search. On the other hand, in order to find a solution fast, bounded-suboptimal MAPF algorithms often view the states with costs within the desired suboptimality bound and small numbers of collisions as "promising" states and examine them first. But the identification of such "promising" states can be misleading because (1) the cost of a state can increase and exceed the desired suboptimality bound after resolving the collisions in the state, and (2) one might have to resolve more collisions than the current collisions in the state (because new collisions can appear when resolving the current ones). Therefore, in this dissertation, we apply an online learning method to learn how the costs and numbers of collisions change during the search and compute an informed learned heuristic (that may be inadmissible) based on them to guide the bounded-suboptimal MAPF algorithms to find a solution within the desired suboptimality bound fast. We also apply a search strategy called Explicit Estimation Search, which cleverly trades off the effort of proving bounded suboptimality and finding solutions.

Sometimes, we are interested in a good solution but not necessarily proof of how good the solution is. For this reason, people often use rule-based algorithms, prioritized algorithms, or bounded-suboptimal algorithms with large suboptimality factors to rapidly find solutions for challenging MAPF instances. In order to overcome the shortcomings of these algorithms in terms of generating costly solutions, we adapt a stochastic local search algorithm, called Large Neighborhood Search (LNS), to improving the quality of a given MAPF solution over time by repeatedly replanning paths for subsets of agents to reduce the overall path costs. In order to overcome their shortcomings in terms of failing to find any solutions, we adapt LNS to repairing an infeasible solution (i.e., a set of paths that contain collisions) by repeatedly replanning paths for subsets of agents to reduce the overall number of collisions among the paths.

In summary, we develop three types of techniques, namely symmetry reasoning, heuristics, and LNS, to improve MAPF algorithms. Specifically, in this dissertation, we first introduce three admissible heuristics and show that they can speed up the optimal MAPF algorithm CBS by up to 50 times. We then develop three symmetry-reasoning techniques and show that they can speed up CBS and its variant with the admissible heuristics by up to 4 orders of magnitude. Their combination results in the state-of-the-art optimal MAPF algorithm CBSH2-RTC, which handles up to thirty times more agents than the previously best variant of CBS. Next, we develop a learned

but inadmissible heuristic to speed up a bounded-suboptimal variant of CBS while preserving its bounded-suboptimality guarantee. We combine the resulting algorithm with many CBS improvements, including admissible heuristics and symmetry reasoning, to obtain the state-of-the-art bounded-suboptimal MAPF algorithm EECBS, which dominates other bounded-suboptimal MAPF algorithms in all tested scenarios. It can find solutions whose costs are provably at most 2% worse than optimal for MAPF instances with a thousand agents within just a minute, while state-of-the-art optimal MAPF algorithms can handle no more than two hundred agents under the same condition. Last but not least, we develop MAPF-LNS and MAPF-LNS2 based on LNS to reduce the cost of a (feasible) MAPF solution and repair an infeasible MAPF solution, respectively. MAPF-LNS significantly outperforms other anytime MAPF algorithms in terms of scalability, runtime to the initial solution, and speed of improving the solution. It reduces the solution cost of non-optimal MAPF algorithms by up to 36 times within just a minute and up to 110 times within five minutes. In comparison to other non-optimal MAPF algorithms (including EECBS with different suboptimality factors), MAPF-LNS2 solves 80% of the most challenging instances in the MAPF benchmark suite (while none of the other algorithms solve more than 65%) within five minutes and finds lower-cost solutions than the other algorithms in most cases.

Chapter 1

Introduction

Multi-Agent Path Finding (MAPF) [163] is the problem of finding collision-free paths for multiple agents on a given graph while minimizing the sum of their travel times, their makespan, or similar optimization criteria. It is NP-hard to solve optimally on general graphs [205], planar graphs [203] and grids [15]. It is also NP-hard to find feasible solutions on directed graphs [129] and NP-hard to find solutions of makespans no larger than 4/3 times the optimal makespan on general graphs [118]. In this dissertation, we mainly focus on minimizing the sum of travel times because this optimization criterion is widely used in many applications. Nevertheless, most techniques developed in this dissertation can be easily generalized to other optimization criteria.

MAPF is a core problem in a variety of real-world applications, including (but not limited to) evacuation [128, 122], automated warehousing and manufacturing [199, 120, 115, 32, 113, 38], automated valet parking [131], autonomous road intersection management [56], traffic management [79, 106, 110], drone swarm coordination [83], search and rescue [143], formation control [108], and video game character control [121, 156]. Although the number of agents involved in these applications varies from dozens of agents to thousands of agents, finding high-quality solutions with small runtimes is important for all of them.

For example, today, in automated warehouses, hundreds of robots already autonomously move inventory pods or flat packages from one location to another in a known, congested environment [199]. Fiducial markers are put on the floor to delineate a four-neighbor grid, and the robots navigate NORTH, SOUTH, WEST, or EAST according to these fiducial markers. Figure 1.1 shows

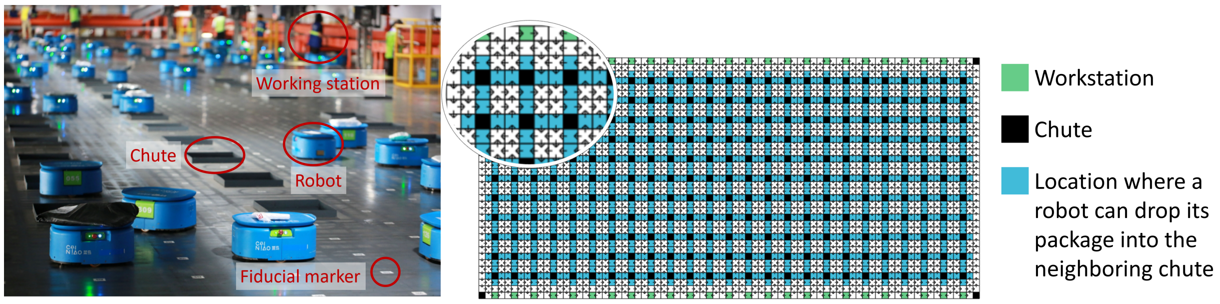

Working station
Chute
Robot
Fiducial marker
Workstation
Chute
Location where a robot can drop its package into the neighboring chute

Figure 1.1: Illustration of a sorting center and its layout, with the left figure reproduced from [94].

a visualization and an example layout of an automated sorting center, where robots deliver flat packages from workstations to designated chutes. Finding low-cost paths for the robots fast is essential for the effectiveness of such systems because lower-cost paths result in higher throughputs or lower operating costs (as fewer robots are required) and finding the paths fast prevents robots from waiting for replanning to finish and thus from being idle. Figure 1.2 compares the effectiveness of a MAPF algorithm against that of a single-agent algorithm, which plans paths for each agent independently and resolves collisions as they arise by issuing wait commands during execution [181]. The MAPF algorithm considers the potential collisions among agents during planning and thus results in the agents moving along paths of lower costs than the single-agent algorithm. As a result, the throughput of the single-agent algorithm starts to drop after 600 robots due to severe traffic congestion, while that of the MAPF algorithm continues to increase until at least 1,000 robots (= 38.9% empty cells on the map). However, on the other hand, this MAPF algorithm needs, for example, 21 seconds on average to generate one set of collision-free paths for 1,000 robots [113], which limits its applicability to real-time systems. Therefore, the focus of this dissertation is to push the limits of the MAPF algorithms by developing techniques to find lower-cost paths faster.

The current state-of-the-art MAPF algorithms can be classified into three categories. Optimal algorithms can find optimal solutions for small-sized problems (e.g., with dozens of agents). Bounded-suboptimal algorithms, i.e., algorithms that find solutions of costs no more than a given factor away from optimal, can find near-optimal solutions for small to medium-sized problems (e.g., with a few hundred agents). Unbounded-suboptimal algorithms can solve very large practical

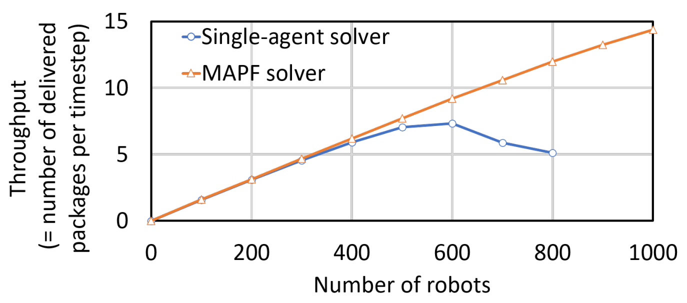


Figure 1.2: Throughput comparison between a single-agent algorithm from [192] and a MAPF algorithm from [113] on the map shown in Figure 1.1.

problems (e.g., with hundreds or even thousands of agents) but usually find low-quality solutions. As a result, existing algorithms for solving MAPF either have limited scalability (such as optimal and bounded-suboptimal algorithms), generate costly solutions (such as rule-based algorithms), or could fail to find any solutions for hard problems (such as prioritized algorithms).

Due to these computational challenges and the substantial interest in MAPF applications, we develop techniques to improve MAPF algorithms in all categories. Specifically,

- we develop techniques to reduce the runtimes of optimal and bounded-suboptimal MAPF algorithms with a focus on developing heuristics and symmetry reasoning techniques to speed up the state-of-the-art MAPF algorithm CBS [153] and its variants for small- and medium-sized problems so that they can scale up to more agents than possible before and still provide guarantees on the solution quality; and

- we also develop techniques to improve the solution quality and success rates of bounded- and unbounded-suboptimal MAPF algorithms with a focus on developing large-neighborhood-search techniques to improve the solutions generated by these algorithms (namely, reducing the costs of feasible solutions and the numbers of collisions in infeasible solutions) over time for large-sized problems so that they can find higher-quality solutions with higher success rates.

Figure 1.3 summarizes the focus of this dissertation. Here, efficiency refers to small runtimes, and effectiveness refers to small solution costs. We validate the following hypothesis:

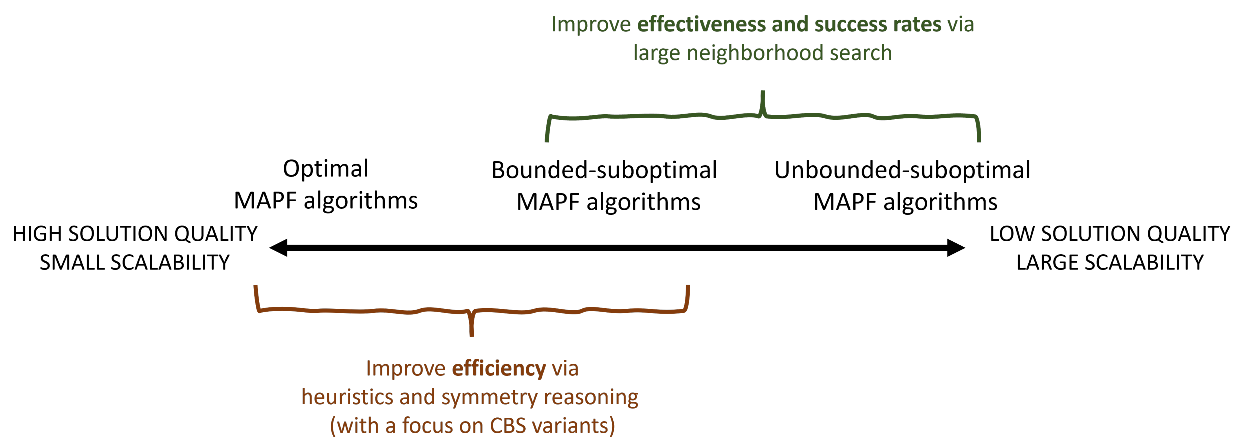


Figure 1.3: Summary of the focus of the dissertation.

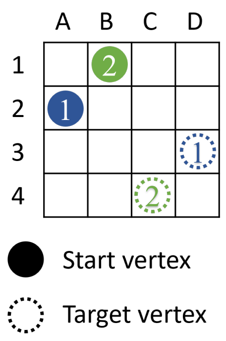
One can improve the efficiency of (bounded-sub)optimal MAPF algorithms via heuristics and symmetry reasoning and the effectiveness and success rates of non-optimal MAPF algorithms via large neighborhood search.

1.1 Motivations

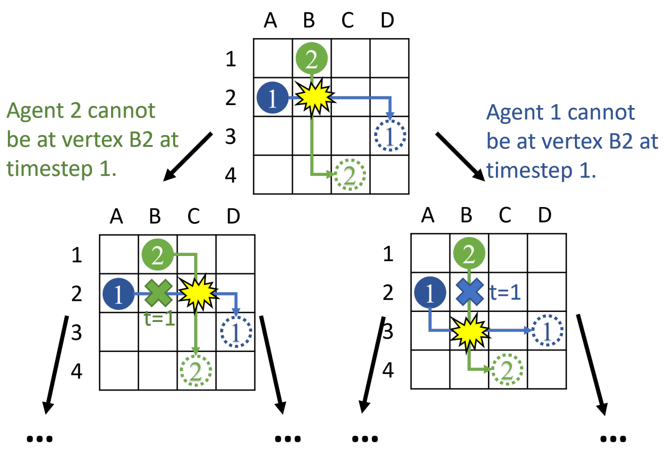
In this section, we first explain why we choose CBS to represent the optimal and bounded-suboptimal MAPF algorithms. We then explain why we choose heuristics and symmetry reasoning to reduce the runtimes of optimal and bounded-suboptimal variants of CBS. We last explain why we choose large neighborhood search to improve the success rates and solution quality of bounded- and unbounded-suboptimal MAPF algorithms.

1.1.1 Why CBS

In terms of reducing the runtimes of optimal and bounded-suboptimal algorithms, we focus on the popular optimal algorithm Conflict-Based Search (CBS) [153] and its optimal and bounded-suboptimal variants. CBS is a two-level search-based MAPF algorithm that resolves collisions by adding constraints on the high level and computing paths consistent with those constraints on the low level. Its central idea is to plan paths for each agent independently by ignoring the other agents



(a) Two-agent MAPF instance. At every timestep, an agent can either wait at its current cell or move to the up, down, left, or right cell.


A B C D
1 2 3 4
Agent 2 cannot be at vertex B2 at timestep 1.
Agent 1 cannot be at vertex B2 at timestep 1.
t=1
t=1
...

(b) CBS tree.

Figure 1.4: High-level tree of CBS for a two-agent MAPF instance on a 4×4 four-neighbor grid.

initially and then resolving collisions by branching. Each branch is a new candidate plan wherein one agent or the other is forced to find a new path that avoids the chosen collision. Below is an example.

Example 1.1. In order to solve the MAPF instance shown in Figure 1.4a, CBS first plans a shortest path for each agent by ignoring the other agent, e.g., [A2, B2, C2, D2, D3] for agent 1 and [B1, B2, B3, B4, C4] for agent 2. It then checks for collisions and finds a collision at vertex B2 at timestep 1. It tries two ways to resolve the collision, namely by prohibiting either agent 1 or agent 2 from being at vertex B2 at timestep 1. In each case, it replans the paths and repeats the procedure by checking for collisions again. Figure 1.4b shows the resulting high-level search tree of CBS. CBS traverses the tree in a best-first manner and terminates when it expands a node whose paths have no collisions. □

We focus on CBS, instead of other optimal or bounded-suboptimal MAPF algorithms, for the following two reasons:

- *CBS is efficient*. CBS represents the class of the state-of-the-art optimal and bounded-suboptimal MAPF algorithms. The state-of-the-art optimal MAPF algorithms are either variants of CBS or deploy strategies similar to CBS, such as lazy CBS [67], BCP [95], and

SMT-CBS [168]. The bounded-suboptimal variants of CBS, such as ECBS [16], also represent the state-of-the-art bounded-suboptimal MAPF algorithms. So, the techniques that we develop to speed up CBS have the potential to speed up a variety of the state-of-the-art optimal and bounded-suboptimal MAPF algorithms.

- *CBS is flexible*. CBS uses a simple and flexible framework that can be easily adapted to many generalized MAPF problems. For instance, we can replace the single-agent pathfinder on the low level of CBS with a domain-specific one (such as a single-robot motion planner) to handle agent with different navigation constraints [49, 37, 158] and redefine the collisions to take into account the shapes of the agents and their robustness requirement [105, 8]. More examples are listed in Section 2.3.4. So, the techniques that we develop to speed up CBS have the potential to speed up its variants for solving a variety of generalized MAPF problems.

In fact, as presented in the extension sections of Chapters 3 to 5, our techniques developed for CBS have already been shown effective in improving the efficiency for some state-of-the-art MAPF algorithms other than CBS and some generalized MAPF problems.

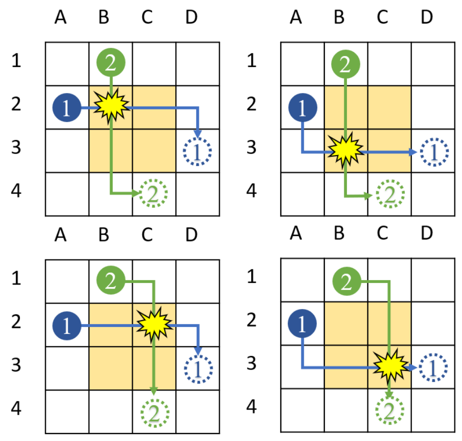
1.1.2 Why Heuristic Search and Symmetry Reasoning

The high level of CBS solves a state-space search problem at its core, where the state space is the collision-resolution space, as each branching operator resolves one collision between two agents. Verifying that a solution to a state-space search problem is optimal requires expanding every node whose f-value is less than the optimal solution cost C^*. Therefore, the larger admissible heuristics we use, the fewer nodes we are required to expand. Existing variants of CBS do not use any admissible heuristics that estimate future work. Therefore, we develop admissible heuristics for CBS nodes, motivated by cost partitioning and pattern databases from the AI planning community, to find provably optimal solutions with fewer node expansions and (thus) faster.

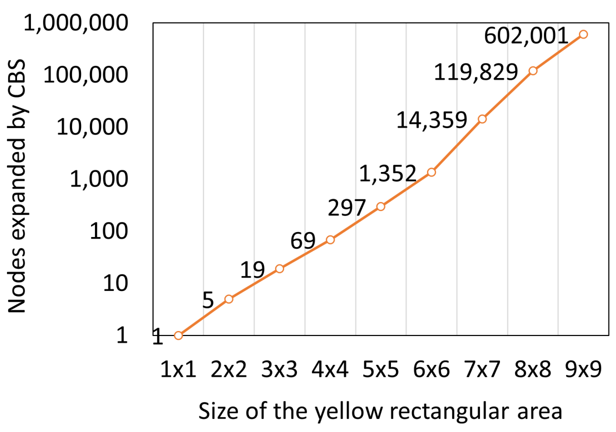
Cost partitioning [91, 201] is a general and principled approach for constructing informed admissible heuristics for optimal classic planning by combining information from multiple admissible heuristics. It distributes operator costs among the heuristics, allowing to add up the heuristic estimates admissibly. A popular way to obtain such additive heuristics is to use pattern databases [138], a technique that pre-computes and stores perfect cost estimates for sub-problems in a lookup table. We apply such ideas to CBS and treat the problem of finding collision-free paths for pairs of agents as the sub-problems. However, since the state space of the high-level search of CBS is infinitely large, it is impractical to construct pattern databases a priori. Therefore, we either analyze the collisions between each pair of agents on the fly to quickly obtain an admissible estimate (instead of a perfect estimate) or construct the pattern databases on the fly by finding the optimal collision-free paths for each pair of agents seen so far. The former method has lower runtime overhead but produces less-informed admissible heuristics than the latter method.

The issue of symmetry is widely recognized as of fundamental importance in constraint satisfaction problems and, in general, many combinatorial problems [44]. Symmetry breaking [20, 188] is a powerful and successful technique in the constraint programming community to reduce the search space exponentially. This is important for a state-space search problem because a smaller search space usually results in a smaller number of nodes with f-values less than C^*. In pathfinding problems, symmetries have so far been studied only for single agents, e.g., by exploiting grid symmetries [76]. In MAPF, the situation is more complicated: Agents have to cooperate to avoid collisions, and this gives rise to collision symmetries, which occur when two agents have many paths to their target vertices, but every combination of them results in a collision. Below is an example of rectangle symmetry.

Example 1.2. Figure 1.5a enumerates 4 possible combinations of the shortest paths of agents 1 and 2 in the MAPF instance shown in Figure 1.4a. Every combination has a collision inside the yellow rectangular area. CBS has to try many such combinations before realizing that the optimal resolution is to let one of the agents wait for one timestep. Figure 1.5b shows that the size of the CBS tree grows exponentially with the size of the yellow rectangular area. □



(a) Rectangle symmetry example.



(b) Numbers of nodes expanded by CBS.

Figure 1.5: Numbers of nodes expanded by CBS for resolving a rectangle symmetry between two agents.

There are many other classes of collision symmetries. Each of them arises commonly in practice and can produce an exponential explosion in the collision-resolution space, leading to unacceptable runtimes for CBS. These collision symmetries are conceptually different from the symmetries in the literature (for solving other problems), which all focus on problem/state/solution symmetries in the sense that permuting some variables, values, propositions, or operators results in identical problems/states/solutions. Therefore, we develop a variety of new reasoning techniques that detect the collision symmetries efficiently as they arise and resolve them using specialized symmetry-breaking constraints to eliminate all permutations of pairwise colliding paths in a single branching step.

For many problems of practical interest, even after we apply admissible heuristics and symmetry reasoning techniques, there are still too many nodes with f-values less than C^* to allow the search to complete with a reasonable runtime. A third way to speed up a state-space search algorithm is to trade off solution quality with runtime by finding a bounded-suboptimal solution, i.e., a solution whose cost is at most $w \cdot C^*$, where w is a user-specified suboptimality factor. In general, finding bounded-suboptimal solutions involves two sub-tasks, namely providing proof that solutions of cost less than C_1 (with $C_1 \leq C^*$) do not exist and finding a solution that is of cost C_2 (with $C^* \leq C_2 \leq w \cdot C_1$). Therefore, we study three methods to speed up bounded-suboptimal

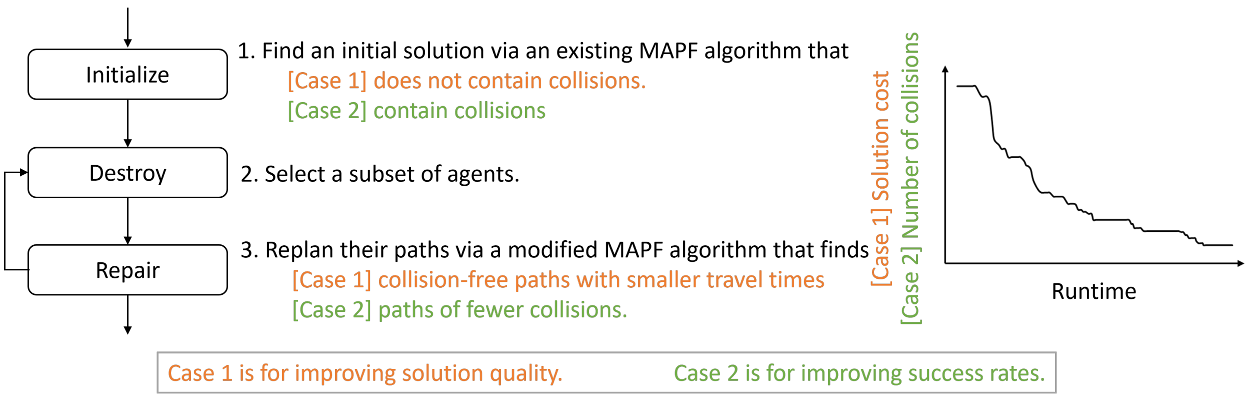

Initialize
Destroy
Repair
1. Find an initial solution via an existing MAPF algorithm that
[Case 1] does not contain collisions.
[Case 2] contain collisions
2. Select a subset of agents.
3. Replan their paths via a modified MAPF algorithm that finds
[Case 1] collision-free paths with smaller travel times
[Case 2] paths of fewer collisions.
[Case 1] Solution cost
[Case 2] Number of collisions
Runtime
Case 1 is for improving solution quality.
Case 2 is for improving success rates.

Figure 1.6: Illustration of LNS for improving MAPF solutions.

variants of CBS. First, we reuse the aforementioned admissible heuristics and symmetry reasoning to speed up the procedure of providing proof. Second, we develop informed (but not necessarily admissible) heuristics to guide the search to find a solution whose cost is sufficiently small (but not necessarily optimal). Third, we deploy a clever strategy to trade off the efforts of providing proof and finding a solution.

1.1.3 Why Large Neighborhood Search

Unlike optimal and bounded-suboptimal MAPF algorithms, whose state-of-the-art variants all employ similar structures, unbounded-suboptimal MAPF algorithms employ ideas and structures of various kinds. Therefore, we are interested in developing algorithm-independent techniques to improve the solution quality and success rates of bounded- and unbounded-suboptimal MAPF algorithms. Large Neighborhood Search (LNS) [154], a well-known meta-heuristic framework from the constraint programming and operations research communities, is a good fit for this purpose. Starting from a given solution, LNS *destroys* part of the solution, called a *neighborhood*, and treats the remaining part of the solution as fixed. What results is a simpler form of the original problem to solve. It then *repairs* the solution and replaces the old solution with the repaired solution if the repaired solution is better. LNS repeats this procedure until it meets some stopping criterion. Figure 1.6 shows an illustration of our LNS-based MAPF framework.

In Case 1, starting from an initial solution obtained from any bounded- or unbounded-suboptimal MAPF algorithm, we reduce the solution cost as time progresses by repeatedly re-planning the paths for subsets of agents using LNS. The benefits of this framework are two-fold. First, the initial solution can be obtained from any existing non-optimal MAPF algorithm, so any improvements to these non-optimal MAPF algorithms improve the performance of our framework as well. Second, the anytime behavior allows us to find an initial MAPF solution fast so that a feasible solution is usually available, even for extremely challenging MAPF problems, and make full use of the runtime budget by keeping to reduce the solution cost until it converges to near-optimal or we timeout.

In Case 2, the existing MAPF algorithm fails to find collision-free paths (within the given runtime limit) due to its incompleteness or large runtime. Therefore, starting from an infeasible solution (i.e., a set of paths that contain collisions) obtained from any existing MAPF algorithm, we reduce the number of collisions in the infeasible solution as time progresses by repeatedly replanning the paths for subsets of agents using LNS. The procedure terminates when the number of collisions is zero. Since the modified MAPF algorithm in this framework solves MAPF instances with only a small subset of agents at a time, this framework empirically scales to a larger number of agents than existing MAPF algorithms.

1.2 Contributions

In this dissertation, we make the following contributions to improving MAPF algorithms:

1. We design three admissible heuristics for CBS: The cardinal conflict graph (CG) heuristic is based on cardinal collisions (i.e., collisions where, if they are resolved, the length of the shortest path of at least one of the two agents involved in the collision increases by at least one); the dependency graph (DG) heuristic is based on whether two agents have a pairwise dependency (i.e., whether all combinations of the shortest paths of the two agents are in collision); and the weighted dependency graph (WDG) heuristic is based on the minimum

cost increase that is required to resolve all collisions between two agents. The WDG heuristic is guaranteed to be at least as informative as the DG heuristic, which in turn is guaranteed to be at least as informative as the CG heuristic. We empirically show that the addition of admissible heuristics can reduce the number of expanded nodes and runtime of CBS by up to a factor of fifty. More details are given in Chapter 3.

2. We develop efficient symmetry reasoning techniques to reason about three classes of collision symmetries: Rectangle symmetry arises when two agents attempt to cross each other in an open area and repeatedly collide with each other along many different shortest paths; target symmetry arises when one moving agent repeatedly collides with another stopped agent at its target vertex along many different paths of increasing lengths; and corridor symmetry arises when two agents moving in opposite directions repeatedly collide with each other inside a narrow passage along many different paths of increasing lengths. These symmetries are common in practice and produce an exponential explosion in the collision-resolution space. We develop reasoning techniques to detect each class of symmetries efficiently and resolve them by specialized constraints that can eliminate, in a single step, all combinations of colliding paths. We empirically show that the addition of symmetry reasoning can reduce the number of nodes expanded by CBS by up to four orders of magnitude and improve its scalability (in terms of the number of agents) by up to thirty times. We also show that the combination of the symmetry reasoning and the WDG heuristic can further speed up CBS, resulting in the best performing optimal MAPF algorithm CBSH2-RTC. More details are given in Chapter 4.

3. We use an online learning framework to learn one-step errors of the hand-crafted heuristics used in ECBS [16], a bounded-suboptimal variant of CBS, and compute a learned, well-informed, but not necessarily admissible heuristic based on the learned errors to help ECBS focus on promising search directions. ECBS uses focal search [134] to determine which node to expand next. Focal search uses an admissible heuristic for bounding the solution cost and another heuristic for determining which nodes are closer to goal nodes. It can be

hindered when these heuristics are negatively correlated. We overcome this issue by using online learning to inadmissibly estimate the cost of the solution under each CBS node and Explicit Estimation Search (EES) [174], an improved variant of focal search, to choose which node to expand next. While ECBS uses focal search that commits all the search efforts to finding solutions, we use EES that cleverly trades off the effort of finding solutions and proving bounded suboptimality. We also investigate recent improvements in CBS and adapt them to our proposed version of bounded-suboptimal CBS, including the admissible heuristics and the symmetry reasoning techniques that we developed for CBS. We find that all three techniques, namely admissible heuristics, symmetry reasoning, and learned inadmissible heuristics with EES, can speed up ECBS. Their combination works the best and can solve MAPF instances with up to a thousand agents within just one minute.[1] The resulting algorithm EECBS runs significantly faster than the state-of-the-art bounded-suboptimal MAPF algorithms ECBS, BCP-7 [95], and eMDD-SAT [172] on a variety of MAPF instances. More details are given in Chapter 5.

4. We build two MAPF frameworks based on Large Neighborhood Search (LNS) for solving challenging MAPF problems. We first build an anytime MAPF framework MAPF-LNS that quickly finds an initial solution via an existing non-optimal MAPF algorithm and then subsequently improves the solution to near-optimal as time progresses. Starting from an initial solution, MAPF-LNS repeatedly selects subsets of agents and replans their paths to reduce the overall solution cost. We then build a MAPF framework MAPF-LNS2 to find the initial solution fast. Starting from an initial infeasible solution, MAPF-LNS2 repeatedly selects subsets of agents and replans their paths to reduce the overall number of collisions. For both MAPF-LNS and MAPF-LNS2, we develop three different methods for selecting subsets of agents and use adaptive LNS [146] to determine which method to use at each LNS iteration. Empirically, we compare MAPF-LNS to the state-of-the-art anytime MAPF algorithm Anytime BCBS [47] and report significant gains in scalability (in terms of the number of

[1]The state-of-the-art optimal MAPF algorithms can handle no more than 200 agents [95].

agents), runtime to the first solution, and speed of improving solutions. We compare MAPF-LNS2 to state-of-the-art non-optimal MAPF algorithms, including prioritized planning with random restarts [157] and Parallel Push and Swap [149] as well as EECBS as proposed in Chapter 5. MAPF-LNS2 solves 80% of the random-scenario instances with the largest number of agents from the MAPF benchmark suite with a runtime limit of just five minutes, significantly outperforming these existing algorithms. More details are given in Chapter 6.

Chapter 2

Background

In this chapter, we first present a formal definition of MAPF in Section 2.1. We then review existing algorithms for solving MAPF in Section 2.2. We finally provide a survey on CBS and its variants in Section 2.3.

2.1 Definition of Multi-Agent Path Finding (MAPF)

MAPF is a broad family of problems with many variants [163]. In this dissertation, we use a classic formulation that considers: (i) vertex and swapping conflicts, (ii) the "stay at target" assumption, and (iii) the objective of minimizing the sum of (individual path) costs.

Definition 2.1 (Multi-Agent Path Finding)**.** *The Multi-Agent Path Finding (MAPF) problem takes as input a graph (or, synonymously, map) $G = (V, E)$ and a set of m agents $A = \{a_1, \dots, a_m\}$. Each agent a_i has a start vertex $s_i \in V$ and a target vertex $g_i \in V$. Time is discretized into timesteps. At each timestep, every agent either* moves *to an adjacent vertex or* waits *at its current vertex. Both types of actions have unit costs. A* path *p_i for agent a_i is a sequence of vertices which are adjacent, i.e., $(p_i[t], p_i[t+1]) \in E$, with $p_i[t] = p_i[t+1] \in V$ indicating a wait action. The* length *(or, synonymously,* cost *or* travel time*) of path p_i is the number of constituent edges or actions, which we measure as $length(p_i) = |p_i| - 1$. Meanwhile, $dist(x, y)$ indicates the distance from vertex x to vertex y, i.e., the length of the shortest path from vertex x to vertex y. Each agent begins at its start vertex s_i and must end at its target vertex g_i. Agents* arrive *at their target vertices if they can wait* *conflict-free until the arrival time of the last agent. Agents waiting at their target vertices without leaving them again have zero costs. There are two types of conflicts: A* vertex conflict $\langle a_i, a_j, v, t\rangle$ *occurs when agents a_i and a_j attempt to occupy vertex $v \in V$ at the same timestep t; and an* edge conflict $\langle a_i, a_j, u, v, t\rangle$ *occurs when agents a_i and a_j attempt to traverse the same edge $(u, v) \in E$ in opposite directions at the same timestep t (or, more precisely, from timestep $t-1$ to timestep t). A* plan *is a set of paths, one for each agent. A* solution *is a conflict-free plan. Our task is to find a solution $P = \{p_i \mid a_i \in A\}$ while minimizing its* sum of costs $\sum_{a_i \in A} length(p_i)$.

MAPF is an idealized abstraction of multi-agent navigation problems in many real-world applications. Different applications have different practical constraints and challenges, and much research has been done to bridge the gap. We list three examples below.

- *Motion constraints*. MAPF solutions are discretized both in time and space, while robots need to navigate in continuous time and space satisfying their motion constraints, e.g., speed and acceleration limits. This issue can be addressed by (1) adding a post-processing step that transfers discrete MAPF solutions to robust, persistent, and executable commands for robots [81, 84, 208], (2) modifying the single-agent pathfinding solver inside MAPF algorithms to directly generate dynamically feasible trajectories for robots [49, 4, 37, 158, 197], or (3) using MAPF solutions as heuristics to guide motion planners to generate dynamically feasible trajectories for robots [100].

- *Task assignment*. Before solving a MAPF problem, one needs to assign target vertices to robots. This issue can be addressed by (1) inserting task-assignment algorithms into MAPF algorithms to solve task assignment and pathfinding simultaneously [117, 82, 78, 170, 215] or (2) developing frameworks that interleave the execution of task-assignment algorithms with that of MAPF algorithms [120, 115, 108].

- *Uncertainty*. When executing commands, the travel times of robots are usually non-deterministic. This issue can be addressed by (1) modifying the cost and conflict-detection functions of MAPF algorithms to generate MAPF solutions that minimize the conflict probability and/or the expected solution cost [185, 106, 11, 164, 10, 151, 135] or (2) using execution policies and fast replan mechanisms to execute MAPF solutions with robustness and deadlock-freeness guarantees [34, 119, 50, 23].

Despite the various approaches for handling the various practical constraints, all these works either directly use MAPF algorithms to generate MAPF solutions and then post-process them or modify some parts of the MAPF algorithms to take the practical constraints into account. So, developing efficient and effective MAPF algorithms is crucial for all of them. Therefore, although we demonstrate the techniques only in the context of the simple, classic MAPF formulation, similar ideas can be, or have already been, applied to more complicated, realistic MAPF formulations.

2.1.1 MAPF Instances Used in the Experiments in This Dissertation

In the examples and experiments of this dissertation, we always use four-neighbor grids as graphs. In our experiments, the maps are from three sources:

- Map `lak503d` from the 2D pathfinding benchmarks[1] [165] (used in Chapter 3);
- all 33 maps from the MAPF benchmarks[2] [163] (used in Chapters 4 to 7); and
- empty and random maps that we generate, where empty maps are four-neighbor grids with no blocked cells, and random maps are four-neighbor grids with randomly blocked cells (both used in Chapter 3).

If the map is not from the MAPF benchmarks, we generate random start and target vertices for the agents. Otherwise, we use the "random" scenarios from the MAPF benchmark, in which the start and target vertices of the agents are also generated randomly. Given the number of agents m, we always use the start and target vertices listed in the first m rows in each "random" scenario file.

[1] `https://movingai.com/benchmarks/grids.html`
[2] `https://movingai.com/benchmarks/mapf.html`

In case the number of agents in the “random” scenarios is smaller than what we need, we generate random start and target vertices for the agents.

The experimental setup (such as the computer, the MAPF instances, and the codebase) used within each chapter is always the same. However, the experimental setup is slightly different in different chapters. To allow us to compare the algorithms developed in different chapters, we always bring the best variant of the algorithms from one chapter to the next and compare (and, when possible, even combine) it with the new algorithms developed in the next chapter in its empirical sections. Moreover, we also run all the best variants using with the same experimental setup in Chapter 7.

We implemented all algorithms in C++. More details on the experimental setup can be found in the empirical evaluation sections in each chapter.

2.2 Overview of MAPF Algorithms

In this section, we review representative optimal, bounded-suboptimal, and unbounded-suboptimal MAPF algorithms. We show that the state-of-the-art optimal MAPF algorithms are either CBS variants or deploy strategies similar to CBS. The bounded-suboptimal variants of CBS also represent the state-of-the-art with respect to bounded-suboptimal MAPF algorithms. So, the heuristics and symmetry reasoning techniques that we develop to speed up CBS have the potential to speed up a variety of state-of-the-art optimal and bounded-suboptimal MAPF algorithms. We also show that unbounded-suboptimal MAPF algorithms suffer from either incompleteness or poor solution quality, which our Large Neighborhood Search technique can overcome.

2.2.1 Optimal MAPF Algorithms

Optimal MAPF algorithms include search-based algorithms (that either search the joint-state space or are variants of CBS that search the conflict-resolution space) and compilation-based algorithms (that reduce MAPF to other well-studied combinatorial optimization problems, such as integer linear programming problems, Boolean satisfiability problems, and constraint programming problems, and use off-the-shelf solvers to find optimal solutions for them).

2.2.1.1 Search-Based Algorithms

A* A straightforward way of solving MAPF optimally is to use A* in the joint-state space, where the *joint states* are different ways of placing all m agents in m out of $|V|$ vertices, one agent per vertex, and the operators between joint states are non-conflicting combinations of actions that the agents can take. Since the size of the joint-state space grows exponentially with the number of agents, numerous techniques have been developed to speed up A* for solving MAPF, such as independence detection [161], operator decomposition [161], partial expansion [71], and sub-dimensional expansion [184]. Their different combinations result in representative A*-based MAPF algorithms, such as OD [161], OD+ID [161], EPEA* [71], M* [184], rM* [184], ODrM* [63], and EPERM* [16].

ICTS Increasing Cost Tree Search (ICTS) [152] is a two-level optimal MAPF algorithm that is conceptually different from A* but still searches the joint-state space. Its high level searches the increasing cost tree, where each node corresponds to a set of costs, one for each agent, and a child node differs from its parent node by increasing the cost of one of the agents by one. When expanding a node, its low level searches the joint-state space to determine whether there exists a MAPF solution such that the cost of the path for each agent is equal to the corresponding cost in the node. A number of pruning techniques have been developed to speed up the low-level search [152].

CBS Conflict-Based Search (CBS) [153] is a two-level optimal MAPF algorithm that resolves conflicts by adding constraints on the high level and computing paths consistent with those constraints on the low level. Its central idea is to plan paths for each agent independently by ignoring other agents and then resolve conflicts by branching. More details on CBS are described in Section 2.3.

Summary Empirically, in terms of runtimes, although many of the A* and ICTS variants are competitive with (vanilla) CBS [61], they are worse than some advanced variants of CBS [104, 90]. This is not surprising because, as the number of agents and the congestion level increase, the effectiveness of the speedup techniques mentioned in the A* and ICTS paragraphs is limited. Thus they all suffer from the exponential explosion of the joint-state space.

2.2.1.2 Compilation-Based Algorithms

ILP MAPF can be encoded as an integer multi-commodity flow problem [206] and then solved by Integer Linear Programming (ILP) solvers. Such methods are competitive with and sometimes even outperform search-based optimal MAPF algorithms on small maps. However, they do not scale well on large maps because the ILP encoding requires a Boolean variable for each agent being at each vertex at each timestep. Branch-and-Cut-and-Price (BCP) [96, 95, 97] is a more efficient ILP-based optimal MAPF algorithm based on branch-and-price and one of the current state-of-the-art optimal MAPF algorithms. Like CBS, BCP is a two-level algorithm whose low level uses search algorithms to solve single-agent pathfinding problems and whose high level uses ILP to assign paths to agents and resolve conflicts.

SAT MAPF can also be encoded as a Boolean satisfiability problem (SAT) [171]. Like the basic ILP encoding, the basic SAT encoding requires a Boolean variable for each agent being at each vertex at each timestep, and thus its efficiency drops as the size of the map grows. SMT-CBS [167] is a more efficient SAT-based algorithm based on satisfiability modulo theories. Like CBS, SMT-CBS ignores all conflicts initially and adds conflict-resolution constraints only when necessary.

CP Like the basic ILP- and SAT-based MAPF solvers,one can also directly encode MAPF as a constraint satisfaction problem and then solve it by an off-the-shelf Constraint Programming (CP) solver [148, 17]. But, again, there is a more efficient CP-based algorithm, called lazy CBS [67], that uses the CBS framework. It uses the same high-level search tree as CBS but traverses it using

	Joint-state space	Conflict-resolution space
Search-based algorithms	OD+ID [161], ICTS [152], EPEA* [71], M* [184]	CBS [153], ICBS [28], MA-CBS [153]
ILP-based algorithms	ILP [206]	BCP [96, 95]
SAT-based algorithms	SAT-MDD [171]	SMT-CBS [167]
CP-based algorithms	CSP [148], SM-OPT [17]	Lazy CBS [67]
ASP-based algorithms	ASP [58, 130, 68, 72]	-

Table 2.1: Summary of optimal MAPF algorithms.

iterative deepening depth-first search with lazy clause generation instead of best-first search. Lazy CBS is also one of the state-of-the-art optimal MAPF algorithms.

Summary Although the formulations of different compilation-based optimal MAPF algorithms are different, the ones based on the direct encoding of the joint-state space (i.e., one variable per agent per vertex per timestep) do not scale to large maps. The leading compilation-based optimal MAPF algorithms, although deploying different techniques from different disciplines, all reason about the conflict-resolution space, like CBS.[3] A summary of optimal MAPF algorithms is shown in Table 2.1.

2.2.2 Bounded-Suboptimal MAPF Algorithms

Bounded-suboptimal MAPF algorithms trade off solution quality and runtime by finding a solution whose cost is at most w times the optimal cost, where w is a user-specified *suboptimality factor*. In the current literature, bounded-suboptimal MAPF algorithms are always variants of optimal MAPF algorithms. Examples include

- the A* variants weighted OD [16], weighted EPEA* [16], inflated M* [184], and inflated ODrM* [63],
- the ICTS variant suboptimal ICTS [2],

[3] In addition to ILP, SAT, and CP, MAPF can also be encoded as an Answer Set Programming (ASP) problem [58, 130, 68, 72]. However, the focus of most of these works is to build a general ASP model that can solve not only classic MAPF but also many of its variants. There does not exist any ASP-based optimal MAPF algorithm that reasons about the conflict-resolution space.

- the CBS variants Weighted CBS (WCBS) [16], Bounded CBS (BCBS) [16], and Enhanced CBS (ECBS) [16], and
- the SAT variants eMDD-SAT [172] and eSMT-CBS [169].

Although bounded-suboptimal MAPF algorithms are not as well explored as optimal MAPF algorithms, similar performance differences are observed, i.e., the current state-of-the-art bounded-suboptimal MAPF algorithms are CBS variants or employ ideas similar to CBS.

2.2.3 Unbounded-Suboptimal MAPF Algorithms

Unbounded-suboptimal MAPF algorithms include variants of bounded-suboptimal algorithms with infinite suboptimality factors (such as GCBS [16] and uMDD-SAT [169]), prioritized algorithms, and rule-based algorithms. We provide additional details on prioritized and rule-based algorithms in the following.

Prioritized Algorithms Prioritized algorithms (or, sometimes, people call them prioritized planning) [59] plan conflict-free paths based on a priority ordering of the agents. They plan the path for each agent individually in the order from high priority to low priority while avoiding conflicts with the (already-planned) paths of higher-priority agents. Priorities can be pre-assigned in many ways [59, 195, 179, 191, 191, 198, 213] or determined on the fly via hill climbing [22] or systematic search [180, 123]. Agents can also exchange priorities for different path segments [157]. Although prioritized algorithms can scale to MAPF instances with large numbers of agents on large maps, they do not provide completeness or solution quality guarantees and thus can find costly solutions or fail to find any solutions for MAPF instances with high agent density. In this dissertation, we use PP to represent prioritized planning with a random priority ordering. PP^R describes prioritized planning with random restarts, that repeatedly runs PP until a MAPF solution is found.

Rule-Based Algorithms Rule-based MAPF algorithms use pre-determined movement rules to coordinate agents, and many of them guarantee to find solutions in polynomial time under some (weak) assumptions. For example, the algorithms in [13, 124] guarantee to find solutions in linear time for MAPF instances on trees. MAPP [193] guarantees to find solutions in polynomial time for "slidable" MAPF instances on grids. BIBOX [166] guarantees to find solutions in polynomial time for MAPF instances on bi-connected graphs with at least two more vertices than agents. Push-and-Swap [116] and Parallel-Push-and-Swap [149] guarantee to find solutions in polynomial time for MAPF instances on general graphs with at least two more vertices than agents. PIBT [132] and winPIBT [133] guarantee to find solutions in polynomial time for MAPF instances on bi-connected graphs if agents are allowed to leave their target vertices after completing their paths. Although no existing rule-based MAPF algorithms guarantee to find bounded-suboptimal solutions, some recent ones, such as Partition-and-Flow [204] and Walk-Stop-Count-and-Swap [190], guarantee to find solutions for MAPF instances on grids whose suboptimality (= the solution cost divided by the optimal cost) does not increase as the instance size increases, i.e., the solutions are provably $O(1)$-suboptimal, for some optimization criteria. In general, although rule-based MAPF algorithms run in polynomial time and can scale to very challenging instances, their solution quality is usually substantially worse than that of other types of MAPF algorithms. In this dissertation, PPS stands for Parallel-Push-and-Swap and represents the class of rule-based MAPF algorithms since it is one of the best performing rule-based MAPF algorithms empirically.

2.3 Overview of Conflict-Based Search (CBS) and Its Variants

We now introduce CBS in detail and present its improvement techniques and variants for both classic and generalized MAPF problems.

2.3.1 Vanilla CBS

Conflict-Based Search (CBS) [153] is a two-level search algorithm for solving MAPF optimally. On the low level, CBS invokes *space-time A** [157] (i.e., A* that searches the space-time space, whose states are vertex-timestep pairs) to find a shortest path for a single agent that satisfies the constraints added by the high level, breaking ties in favor of the path that has the fewest conflicts with the (already planned) paths of the other agents. A *constraint* is a space-time restriction introduced by the high level to resolve conflicts. Specifically, a *vertex constraint* $\langle a_i, v, t\rangle$ prohibits agent a_i from being at vertex $v \in V$ at timestep t. Similarly, an *edge constraint* $\langle a_i, u, v, t\rangle$ prohibits agent a_i from traversing edge $(u, v) \in E$ from vertex u to vertex v at timestep t (or, more precisely, from timestep $t-1$ to timestep t).

On the high level, CBS performs a best-first search on a binary *constraint tree* (CT). Each CT node N contains a set of constraints $N.constraints$ and a *plan* $N.plan$, which is a set of shortest paths, one $N.plan[a_i]$ for each agent $a_i \in A$, that satisfy the constraints in $N.constraints$ but are not necessarily conflict-free. The root CT node contains an empty set of constraints. The cost of a CT node N is defined as the sum of costs of the paths in $N.plan$, i.e., $cost(N) = \sum_{a_i \in A} length(N.plan[a_i])$. CBS always expands the CT node with the smallest cost, breaking ties in favor of the CT node that has the fewest conflicts in its plan, and terminates when the plan of the CT node chosen for expansion is conflict-free and thus corresponds to an optimal solution.

When expanding a CT node, CBS checks for conflicts in its plan. It chooses one of the conflicts (by default, arbitrarily) and resolves it by *branching*, i.e., by *splitting* the CT node into two child CT nodes. In each child CT node, CBS adds a constraint to prohibit one agent from the conflict from using the conflicting vertex or edge at the conflicting timestep. The path of this agent does not satisfy the new constraint and is replanned by the low-level search. All other paths remain unchanged. If the low-level search cannot find any path, then there does not exist any solution that satisfies the constraints of this child CT node, and CBS thus prune the node.

Algorithm 2.1 presents the pseudo-code of CBS. On Line 10, if the conflict is a vertex conflict $\langle a_i, a_j, v, t\rangle$, then the constraints are $\langle a_i, v, t\rangle$ and $\langle a_j, v, t\rangle$; otherwise, i.e., the conflict is an edge conflict $\langle a_i, a_j, v, u, t\rangle$, then the constraints are $\langle a_i, v, u, t\rangle$ and $\langle a_j, u, v, t\rangle$.

Theorem 2.1. *CBS is guaranteed to terminate with an optimal solution for a given MAPF instance if one exists.*

Algorithm 2.1: CBS for solving MAPF optimally.

```
Input: MAPF instance $(G,A)$
1  Generate root CT node $R$ with an empty set of constraints;
2  for $a_i \in A$ do $R.plan[a_i] \leftarrow$ LowLevelSearch$(a_i, G, R)$;
3  $R.conflicts \leftarrow$ all conflicts in $R.plan$;
4  OPEN $\leftarrow \{R\}$;
5  while OPEN $\neq \emptyset$ do
6      $N \leftarrow \arg\min_{N \in \text{OPEN}} cost(N)$;        // Break ties by CT nodes with fewer conflicts
7      OPEN $\leftarrow$ OPEN $\setminus \{N\}$;
8      if $N.conflicts = \emptyset$ then return $N.plan$;
9      $conflict \leftarrow$ a conflict in $N.conflicts$;
10     Generate the two constraints $constraint_1$ and $constraint_2$ for resolving $conflict$;
11     for $i = 1, 2$ do
12         $N' \leftarrow$ a copy of $N$;
13         $N'.constraints \leftarrow N.constraints \cup constraint_i$;
14         $a_j \leftarrow$ the agent on the agent on which $constraint_i$ is imposed;
15         $N'.plan[a_j] \leftarrow$ LowLevelSearch$(a_j, G, N')$;
16         if $N'.plan[a_j]$ does not exist then continue;
17         $N'.conflicts \leftarrow$ all conflicts in $N'.plan$;
18         OPEN $\leftarrow$ OPEN $\cup \{N'\}$;
19 return "No Solution";
```

Proof Sketch. CBS guarantees its completeness by exploring both ways of resolving every conflict. In other words, when CBS expands a CT node, any solution that satisfies the constraints of the CT node must satisfy the constraints of at least one of its child CT nodes. So, branching excludes only plans with conflicts. CBS guarantees optimality by performing best-first searches on both its high and low levels. Please refer to [153] for a detailed proof. □

In this dissertation, the *completeness* of CBS means that CBS is guaranteed to terminate with a solution for a given MAPF instance if one exists. It does not mean that CBS is guaranteed to terminate and return failure if no solution exists. In fact, if a given MAPF instance is unsolvable, CBS may not terminate. However, there exist linear-time algorithms, such as [207], to determine the solvability of MAPF instances. So, if solvability is an issue, one can run such an algorithm before CBS.

2.3.2 CBS Improvements

Researchers have proposed many techniques in the past few years to improve CBS, such as disjoint splitting [103], merge and restart [29], and iterative deepening [30], and temporal jump point search [85]. We discuss two such techniques in detail below, namely prioritizing conflicts and bypassing conflicts, as they will be used in Chapters 3 to 5.

2.3.2.1 Prioritizing Conflicts

Prioritizing conflicts [29] determines which conflict to resolve first. It classifies conflicts into three types, and, here, we provide a generalized definition that applies to not only vertex and edge conflicts but also the symmetric conflicts that will be introduced in Chapter 3.

Definition 2.2 (Cardinal, Semi-Cardinal, and Non-Cardinal Conflicts)**.** *A conflict is* cardinal *iff replanning for any of the two agents involved in the conflict (with the corresponding constraint) increases the sum of costs. A conflict is* semi-cardinal *iff replanning for one agent involved in the conflict increases the sum of costs while replanning for the other agent does not. Finally, a conflict is* non-cardinal *iff replanning for any of the two agents involved in the conflict does not increase the sum of costs.*

We can significantly improve the efficiency of CBS by letting it resolve cardinal conflicts first, then semi-cardinal conflicts, and finally non-cardinal conflicts because generating child CT nodes with larger costs first can improve the *lower bound* of the CT, i.e., the minimum cost of the leaf CT nodes, faster and thus produce smaller CTs (since an optimal solution can be found only after the lower bound of the CT is equal to the optimal cost). We refer to CBS with the conflict prioritization technique as *Improved CBS* (ICBS).

ICBS builds MDDs to classify conflicts.

Definition 2.3 (MDD)**.** *A* Multi-Valued Decision Diagram *(MDD) [152] MDD_i of agent a_i at a CT node is a directed acyclic graph that consists of all shortest paths of agent a_i that satisfy the*

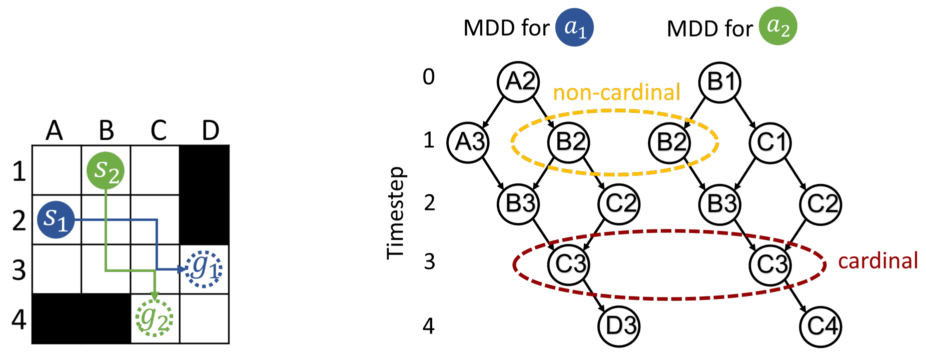


Figure 2.1: Example of using MDDs to identify cardinal and non-cardinal conflicts. We omit the timesteps of the MDD nodes in the right figure.

constraints of the CT node. The MDD nodes at depth t in MDD_i correspond to all vertices at timestep t in these paths.

Definition 2.4 (Singleton)**.** *If MDD_i has only one MDD node (v,t) at depth t, then we call this MDD node a* singleton*, and all shortest paths of agent a_i are at vertex v at timestep t.*

So, a vertex conflict $\langle a_i, a_j, v, t\rangle$ is cardinal iff the MDDs of both agents have singletons at depth t, and an edge conflict $\langle a_i, a_j, u, v, t\rangle$ is cardinal iff the MDDs of both agents have singletons at both depths $t-1$ and t. Semi-/non-cardinal vertex/edge conflicts can be identified analogously.

Example 2.1. Consider the MAPF instance shown in Figure 2.1. Agents a_1 and a_2 have two vertex conflicts, one at vertex B2 at timestep 1 and the other one at vertex C3 at timestep 3. According to the MDD of each agent, MDD node (B2, 1) is not a singleton for either MDD, while MDD node (C3, 3) is one for both MDDs. As a result, the conflict at vertex B2 at timestep 1 is non-cardinal, while the conflict at vertex C3 at timestep 3 is cardinal. □

2.3.2.2 Bypassing Conflicts

Bypassing conflicts [27] is a conflict-resolution technique that, instead of splitting a CT node, modifies the paths of the agents involved in the chosen conflict in the CT node. When expanding a CT node N and generating its child CT nodes, if the cost of a child CT node N' is equal to $cost(N)$ and the number of conflicts in $N'.plan$ is smaller than that in $N.plan$, then CBS replaces the paths in N with the paths in N' and discards all generated child CT nodes. Otherwise, it splits CT node N as before. Bypassing conflicts often produces smaller CTs and decreases the runtime of CBS.

2.3.3 Suboptimal Variants of CBS

Barer et al. [16] extend CBS to its bounded-suboptimal variants. The bounded suboptimality is achieved by using focal search [134], instead of best-first search, on both the high and low levels of CBS. A focal search maintains a FOCAL list that consists of a subset of the nodes in the OPEN list, namely those nodes whose costs are within a constant factor of the lowest cost of any node in OPEN. The focal search always expands a node with the best user-provided heuristic value in FOCAL. Barer et al. [16] use the number of conflicts as heuristics in both the high- and low-level focal searches. They propose two bounded-suboptimal variants of CBS, namely Bounded CBS (BCBS), that takes two suboptimality factors as input parameters, one used for its high-level focal search and one used for its low-level focal search, and Enhanced CBS (ECBS), that takes one suboptimality factor as input parameter, used for both its high- and low-level focal searches. Empirically, the efficiency of ECBS dominates that of BCBS. ECBS runs substantially faster than CBS and can be further sped up by techniques like node selection [86], merge and restarts [48, 35], highways [46], and flex distribution [36]. More details of ECBS will be introduced in Section 5.1.

There are also some unbounded-suboptimal variants of CBS. Greedy CBS (GCBS) [16] uses the number of conflicts, instead of the sum of costs/path cost, as the optimization objective in both its high- and low-level best-first searches. Priority-Based Search (PBS) [123] combines CBS with prioritized planning. Its high level is similar to the high level of CBS, except that the constraint added to each child CT node is that one agent from the conflict has a higher priority than the other. Its low level is similar to prioritized planning, i.e., it plans a shortest path for each agent that is consistent with the partial priority ordering generated by the high level. PBS empirically outperforms many variants of prioritized planning solvers in terms of success rates and solution quality, despite being still incomplete and suboptimal in theory.

2.3.4 Variants of CBS for Generalized MAPF Problems

CBS is such a simple and flexible framework that it can be adapted to solving a large variety of generalized MAPF problems. We roughly divide these problems into four categories.

- Generalized agent models: MAPF with agents of different shapes [177, 105, 9], MAPF with agents of different priorities [131], and MAPF with payload transfer [118].
- Generalized task models: MAPF with temporally-constrained target vertices [122, 127, 210], MAPF with target assignment [117, 82, 78, 170, 215], and MAPF with streams of target vertices [189].
- Generalized navigation constraints: MAPF with formation control [121, 108, 143], MAPF with transit networks [43], MAPF with non-unit/continuous/stochastic traversal time [119, 8, 5, 186, 10, 187, 151], and MAPF with kinematic constraints/motion planning [49, 4, 158, 93].
- Other generalized MAPF problems: MAPF with multiple objectives [144], MAPF with meeting requirements [12, 73], MAPF with layout design [18], and explainable MAPF via segmentation [92].

In addition to modifying CBS directly, some researchers design hierarchical frameworks for generalized MAPF problems and use CBS as a sub-solver. Examples include MAPF with motion planning [83, 37], spatially-partitioned MAPF systems [209, 101], and MAPF for warehouse applications [120, 115, 32, 113, 74, 88].

CBS and its variants can even be extended to solving problems that are not directly related to MAPF, such as task assignment with time window constraints [42], pipe routing [19], and virtual network embedding [214].

Chapter 3

Speeding up Optimal CBS via Admissible Heuristics

The best-first search of the high level of all existing CBS variants uses only the cost of a CT node as its priority. (Recall that the cost of a CT node is the sum of costs of the paths in its plan.) This value can be regarded as the g-value of the node. We want to add an *admissible* (i.e., non-overestimating) h-value to its priority to make it more informed. We introduce three different admissible heuristics for CBS by aggregating potential cost increases between pairs of agents:

- *Cardinal conflict Graph (CG) heuristic*, which makes use of information about whether resolving the current conflicts (i.e., the conflicts in the plan of the current CT node) increases the sum of costs of the paths of the agents.
- *Dependency Graph (DG) heuristic*, which makes use of information about whether resolving the future as well as current conflicts increases the sum of costs of the paths of the agents.
- *Weighted Dependency Graph (WDG) heuristic*, which makes use of information about how much resolving the current and future conflicts increases the sum of costs of the paths of the agents.

Among the three heuristics, CG has the smallest runtime overhead but also the smallest h-values, while WDG has the largest runtime overhead but also the largest h-values. Overall, WDG results in a better efficiency than DG in most cases, which in turn results in a better efficiency than CG in most cases. The addition of admissible heuristics can reduce the number of expanded nodes and the runtime of CBS (or, more precisely, ICBS) by up to a factor of fifty.

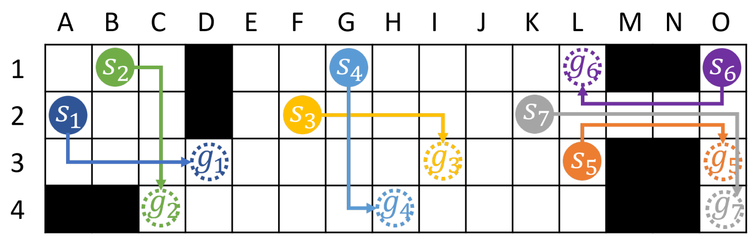


Figure 3.1: Example of a seven-agent MAPF instance.

This chapter closely follows [62, 102].

3.1 The CG Heuristic

Recall the definition of cardinal conflicts in Definition 2.2. If the plan of a CT node N contains one or more cardinal conflicts, then an h-value of one is admissible for CT node N because the cost of any of its descendants in the CT with a conflict-free plan is at least $N.cost + 1$. The reason is that the paths in their solutions cannot be shorter than the ones in the plan of CT node N since the same and perhaps more constraints are imposed on the agents, and the length of the path of at least one of the conflicting agents has to increase by at least one.

However, if the plan of a CT node contains x cardinal conflicts, then an h-value of x is not necessarily admissible for the CT node because, for example, if both agents a_1 and a_2 have a cardinal conflict with agent a_3, then we might be able to resolve both cardinal conflicts by finding a new path for agent a_3 that is only one unit longer than its shortest path. We therefore need to reason about the dependencies among the cardinal conflicts to calculate admissible h-values. We achieve this by aggregating cardinal conflicts via cardinal conflict graphs.

3.1.1 Cardinal Conflict Graphs

We use a *Cardinal Conflict Graph* $G_C = (V_C, E_C)$ of CT node N. Each vertex $v_i \in V_C$ corresponds to an agent a_i that is involved in at least one cardinal conflict. Each edge $e = (v_i, v_j) \in E_C$ expresses that there is at least one cardinal conflict between agents a_i and a_j. Figure 3.2 shows the cardinal

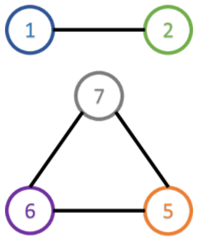


Figure 3.2: Cardinal conflict graph of the root CT node for the MAPF instance in Figure 3.1.

conflict graph of the root CT node for the MAPF instance in Figure 3.1. Similar conflict graphs have been used in the context of heuristic search for sliding tile puzzles [60] and cost-optimal planning [139].

The path length of at least one agent of each conflicting agent pair in a cardinal conflict has to increase by at least one. Thus, a *Minimum Vertex Cover* (MVC) (i.e., a set of vertices such that each edge is incident on at least one vertex in the set) of the cardinal conflict graph represents a minimum set of agents that takes non-shortest paths to resolve all cardinal conflicts. In other words, the size of a MVC of the cardinal conflict graph of CT node N is an admissible h-value. We refer to this heuristic as the *CG heuristic* h_{CG}. For example, the size of the MVC of the graph in Figure 3.2 is 3. That is, $h_{CG} = 3$ is an admissible h-value for the root CT node of CBS for the MAPF instance in Figure 3.1. We refer to ICBS with the CG heuristic as CBSH, which will be used as a sub-solver in Section 3.3.3 and a baseline optimal MAPF algorithm in Chapter 4.

Finding the MVC of a general graph is NP-hard [200]. However, we can update the MVC incrementally by utilizing some properties of the high-level search of CBS.

Property 3.1. *The size of the MVC of the cardinal conflict graph of a CT node N is either one unit larger than, the same as, or one unit smaller than that of its parent CT node.*

Proof. When CBS generates a CT node N, it replans the path of only one agent. Consequently, only edges incident on the vertex corresponding to this agent can appear in or disappear from the cardinal conflict graph. Let v be this vertex and $\mathscr{V}$ and $\mathscr{V}'$ be the MVC of the cardinal conflict graph of CT node N and its parent CT node, respectively. $\mathscr{V}' \cup \{v\}$ is guaranteed to be a vertex cover of the cardinal conflict graph of CT node N, which indicates that $|\mathscr{V}| \leq |\mathscr{V}'| + 1$. Similarly,

$\mathscr{V} \cup \{v\}$ is guaranteed to be a vertex cover of the cardinal conflict graph of the parent CT node of CT node N, which indicates that $|\mathscr{V}'| \leq |\mathscr{V}| + 1$. Therefore, $|\mathscr{V}'| - 1 \leq |\mathscr{V}| \leq |\mathscr{V}'| + 1$. □

Property 3.1 can be exploited to calculate the h-value of CT node N with an algorithm that determines in time $O(2^q n)$ whether a given graph with n vertices has a vertex cover of size q [55], by executing it at most twice (namely for $q = h - 1$ and, if that is unsuccessful and $h < m - 1$, also for $q = h$, where h is the h-value of the parent of CT node N). Recall that m is the number of agents, so $m - 1$ is an upper bound on the h-values since the size of the MVC of the cardinal conflict graph with at most m vertices is at most $m - 1$.

Since the runtime of the above algorithm grows exponentially in the size of the MVC, we propose a greedy method for calculating a weaker admissible heuristic based on cardinal conflict graphs in polynomial time. *Disjoint cardinal conflicts* are cardinal conflicts between disjoint pairs of agents, i.e., pairs of agents that do not contain common agents. If the plan of a CT node N contains x disjoint cardinal conflicts, then $h = x$ is admissible for CT node N since the path length of at least one agent of each agent pair has to increase by at least one. Thus, we can use the size of a *matching* (i.e., a set of edges without common vertices) in the cardinal conflict graph of node N as its admissible h-value. We find a maximal matching in linear time as follows: We repeatedly choose an arbitrary edge (representing a cardinal conflict) in the conflict graph, increase the h-value of CT node N by one and then delete all edges that are incident on both vertices of the chosen edge from the conflict graph, until no edge remains. We refer to the resulting heuristic as the *greedy CG heuristic* h'_{CG}. For example, the size of a greedy matching on the graph in Figure 3.2 is 2. That is, $h'_{CG} = 2$ is an admissible h-value for the root CT node of CBS for the MAPF instance in Figure 3.1. From the relationship between minimum vertex cover and maximal matching, we know that $h'_{CG} \leq h_{CG} \leq 2h'_{CG}$.

3.1.2 Constructing Cardinal Conflict Graphs

The construction of a cardinal conflict graph is straightforward. For every conflict in the plan of a CT node N, we check whether it is cardinal by building the MDDs (defined in Definition 2.3) for both conflicting agents. Since conflict prioritization (introduced in Section 2.3.2.1) also needs to classify conflicts, building cardinal conflict graphs incurs almost no runtime overhead if we already use the conflict-prioritization technique in CBS.

3.1.3 Properties of Cardinal Conflict Graphs

We have argued above that an h-value of 1 for one cardinal conflict is admissible. The following theorem answers the question whether it is possible to find a larger h-value for a cardinal conflict. The proof of this theorem is given in Appendix A.

Theorem 3.1. *Suppose that CBS chooses to resolve a conflict between agents a_i and a_j at timestep t at a CT node N and successfully generates two child CT nodes N_1 (with an additional constraint imposed on a_i) and N_2 (with an additional constraint imposed on a_j). If the conflict occurs after one of the agents, say a_i, completes its path, i.e., $t \geq length(N.plan[a_i])$, then $cost(N_1) = cost(N) + t + 1 - length(N.plan[a_i])$ and $cost(N_2) \in \{cost(N), cost(N)+1\}$. Otherwise (i.e., the conflict occurs before both agents have completed their paths), $cost(N_1), cost(N_2) \in \{cost(N), cost(N)+1\}$.* $\square$

Therefore, if both child CT nodes of CT node N are successfully generated (which always happens in practice), a conflict can be regarded as an admissible h-value of at most 1. Then, the CG heuristic is the best admissible heuristic for N that can be obtained from the cardinal conflict graph. So, new directions need to be explored if we want to obtain better heuristics.

3.2 The DG Heuristic

The CG heuristic considers cardinal conflicts only in $N.plan$. To improve on that, we also need to consider conflicts in future plans, i.e., plans of the descendants of CT node N. For example, in Figure 3.1, if CBS resolves the non-cardinal conflict $\langle a_3, a_4, \text{G2}, 1\rangle$ by adding a constraint for one of the agents, a new conflict will occur no matter which new shortest path CBS picks. In fact, any two shortest paths of agents a_3 and a_4 conflict in one of the four vertices {G2, G3, H2, H3}. Therefore, an h-value of 1 is admissible here. This is not captured by CG because the conflicts

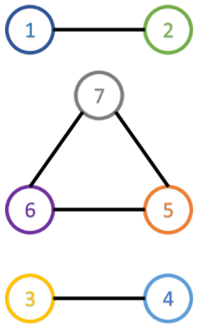


Figure 3.3: Pairwise dependency graph of the root CT node for the MAPF instance in Figure 3.1.

are initially non-cardinal. Inspired by this example, we generalize the cardinal conflict graph to a *pairwise dependency graph*, whose edges reflect that all pairs of shortest paths of the corresponding two agents have conflicts.

3.2.1 Pairwise Dependency Graphs

In a *pairwise dependency graph* $G_D = (V_D, E_D)$ of CT node N, each agent a_i induces a vertex $v_i \in V_D$. An edge (v_i, v_j) is in E_D iff agents a_i and a_j are *dependent*, i.e., all pairs of their shortest paths that satisfy the constraints in $N.constraints$ have conflicts. Similarly to the cardinal conflict graph, for each edge $(v_i, v_j) \in E_D$, the path length of at least one agent, a_i or a_j, has to increase by at least 1 in any solution that can be found under CT node N. Hence, the size of the MVC of G_D is an admissible h-value for CT node N. We refer to this heuristic as the *DG heuristic* h_{DG}. We use the same algorithm as before to determine an MVC. Its complexity is $O(2^q|V_D|)$, where q is the size of the MVC. Figure 3.3 shows the pairwise dependency graph of the root CT node for the MAPF instance in Figure 3.1, and thus $h_{DG} = 4$.

Property 3.2. *The DG heuristic strictly dominates the CG heuristic.*

Proof. This is true because the cardinal conflict graph is a sub-graph of the pairwise dependency graph. □

Like for the CG heuristic, we can also calculate a lower bound on the DG heuristic using the greedy matching method, resulting in the *greedy DG heuristic* h'_{DG}.

3.2.2 Constructing Pairwise Dependency Graphs

To construct the pairwise dependency graph G_D of CT node N, we analyze the dependencies between all pairs of agents. We first classify all pairs of agents into three categories based on their conflicts in $N.plan$:

(1) The two agents do not have any conflicts.

(2) They have at least one cardinal conflict.

(3) They have only semi-cardinal or non-cardinal conflicts.

If agents a_i and a_j are in Category (1), then they are *independent* as their paths in $N.plan$ are conflict-free. Hence, $(v_i, v_j) \notin E_D$. If they are in Category (2), by the definition of cardinal conflicts in Definition 2.2, then they are surely dependent. Hence, $(v_i, v_j) \in E_D$. If they are in Category (3), then we do not know whether they are dependent or independent. To provide an answer, we try to *merge* the MDDs of the two agents into a joint MDD using the method described in [152]. The two agents are dependent iff their joint MDD is empty. Details of the merging are given in Section 3.2.3.

Since each CT node has an additional constraint imposed on only one agent, we only need to look at the dependencies between this agent and all other agents and can copy the edges for the other pairs of agents from the pairwise dependency graph of the parent CT node. Of course, at the root CT node, we still need to look at the dependencies for all pairs of agents. CG already builds MDDs to classify conflicts. So, for DG, we get these MDDs for free. The only runtime overhead of DG over CG comes from merging MDDs.

3.2.3 Merging MDDs

The *joint MDD* of the MDDs of agents a_i and a_j at CT node N consists of all combinations of conflict-free shortest paths of agents a_i and a_j that satisfy the constraints in $N.constraints$. Nodes at depth t of the joint MDD correspond to all joint states (where a joint state is a tuple of two vertices and a timestep) where agents a_i and a_j can be at timestep t along such a

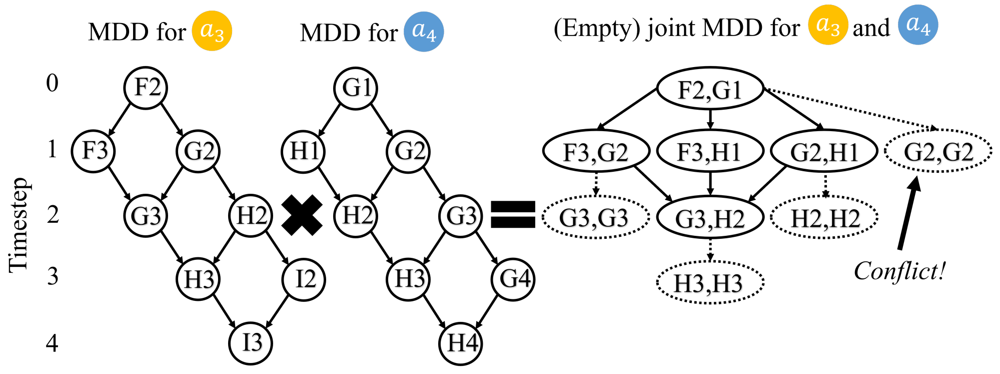


Figure 3.4: MDDs and joint MDD for agents a_3 and a_4 in the MAPF instance in Figure 3.1. We omit the timesteps of the MDD nodes in the figure.

pair of shortest paths without conflicts. If $length(N.plan[a_i]) \neq length(N.plan[a_j])$, a path of $|length(N.plan[a_i]) - length(N.plan[a_j])|$ dummy target vertices is added to the sink node of the shallower MDD (representing the agent waiting at its target vertex) so that both MDDs have the same depth. The joint MDD is built level by level. The merging procedure starts at the joint state (s_i, s_j) at level 0. Suppose that we already have a joint state (v_i, v_j) at level t and want to add its child nodes at level $t+1$. Each pair in the cross product of the child nodes of vertex v_i at level t in the MDD of agent a_i and the child nodes of vertex v_j at level t in the MDD of agent a_j should be examined (i.e., to check if it leads to vertex or edge conflicts). Only conflict-free pairs are added. Agents a_i and a_j are dependent iff their joint MDD is *empty*, i.e., does not contain state (g_i, g_j) at level $\max\{length(N.plan[a_i]), length(N.plan[a_j])\}$.

Example 3.1. Figure 3.4 shows an example of merging MDDs. The joint MDD of agents a_3 and a_4 starts at (F2, G1) at level 0. At level 1, we try all combinations of vertices at level 1 in both MDDs and add them to the joint MDD except for the pair (G2, G2), which represents a conflict state. We repeat this procedure at levels 2 and 3 until all branches of the joint MDD reach conflict states and cannot be further developed. Therefore, in this example, the joint MDD is empty, and thus agents a_3 and a_4 are dependent. □

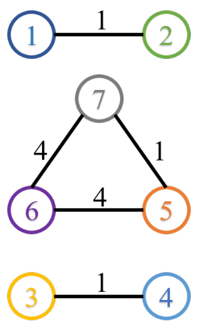


Figure 3.5: Weighted pairwise dependency graph of the root CT node for the MAPF instance in Figure 3.1.

3.3 The WDG Heuristic

For a CT node N and two agents a_i and a_j, we refer to the difference between the minimum sum of costs of their conflict-free paths that satisfy $N.constraints$ and the sum of costs of their paths in $N.plan$ as Δ_{ij} ($\Delta_{ij} \geq 0$). Agents a_i and a_j are dependent iff $\Delta_{ij} > 0$.

Although G_D captures the information whether $\Delta_{ij} > 0$ for any pair of agents a_i and a_j, it does not capture the information how large the value of Δ_{ij} is. When $\Delta_{ij} > 0$, the DG heuristic uses only 1 (a lower bound on Δ_{ij}) as an admissible h-value. However, Δ_{ij} can be larger than 1. For instance, $\Delta_{56} = 4$ in Figure 3.1 because one of the agents must wait for 4 timesteps at its start vertex. Therefore, we introduce the WDG heuristic, which captures not only the pairwise dependencies between agents but also the extra cost that each pair of dependent agents will contribute to the total cost.

3.3.1 Weighted Pairwise Dependency Graphs

We generalize the pairwise dependency graph to a *weighted pairwise dependency graph* $G_{WD} = (V_D, E_D, W_D)$ for CT node N, where W_D represents the weights of the edges. It uses the same vertices and edges as pairwise dependency graph G_D. The weight on each edge $(v_i, v_j) \in E_D$ equals Δ_{ij}. Here, $\Delta_{ij} \geq 1$ since agents a_i and a_j are dependent. We also generalize the MVC to an *edge-weighted minimum vertex cover*.

Definition 3.1 (Edge-Weighted Minimum Vertex Cover)**.** Edge-Weighted Minimum Vertex Cover *(EWMVC) is an assignment of non-negative integers $\{x_i \in \mathbb{N} \mid v_i \in V_D\}$, one to each vertex, that minimizes their sum $\sum_{v_i \in V_D} x_i$ subject to the constraints that $x_i + x_j \geq \Delta_{ij}$ for all $(v_i, v_j) \in E_D$.*

x_i can be interpreted as the increase in the length of the path of agent a_i. The sum $\sum_{v_i \in V_D} x_i$ of the EWMVC of G_{WD} is an admissible h-value for CT node N since, for each edge $(v_i, v_j) \in E_D$, the sum of costs of the paths of agents a_i and a_j has to increase by at least Δ_{ij}. We refer to this heuristic as the WDG heuristic h_{WDG}.

Property 3.3. *The WDG heuristic strictly dominates the DG heuristic.*

Proof. This is true because the value of the WDG heuristic is equal to the sum $\sum_{v_i \in V_D} x_i$ of the EWMVC of G_{WD} when the weights of all edges are one. □

Figure 3.5 shows the weighted pairwise dependency graph of the root CT node of CBS for the MAPF instance in Figure 3.1. An example EWMVC for it is $x_1 = x_3 = x_5 = 1$, $x_2 = x_4 = x_7 = 0$, and $x_6 = 4$, which results in $h_{WDG} = 7$. We refer to ICBS with the WDG heuristic as CBSH2, which will be used as a baseline optimal MAPF algorithm in Chapter 4.

Calculating a EWMVC is NP-hard since calculating a MVC is NP-hard and a special case of calculating EWMVC when the weights of all edges are one. To calculate a EWMVC, we partition G_{WD} into its connected components and calculate the EWMVC for each component with a branch-and-bound algorithm that branches on the possible values of each x_i in the component and prunes nodes using the cost of the best result so far. The EWMVC of G_{WD} is the union of the EWMVCs of all components. Similar dependency graphs and EWMVCs for heuristic search have been used in the context of sliding tile puzzles [60] and cost-optimal planning [138].

As for the CG and DG heuristics, we can also calculate a lower bound on the WDG heuristic using a greedy weighted matching method, resulting in the *greedy WDG heuristic* h'_{WDG}. We find a greedy weighted matching in near-linear time as follows: We repeatedly choose the edge with the largest edge weight in G_{WD}, increase the h-value of node N by the weight of the chosen edge, and then delete all edges from G_{WD} that are incident on both vertices of the chosen edge, until no edge remains.

Property 3.4. *The WDG heuristic strictly dominates the greedy WDG heuristic.*

See Appendix B for the proof. From Property 3.4 and the fact that the WDG heuristic is admissible, we know that the greedy WDG heuristic is also admissible. For example, the size of a greedy weighted matching for the graph in Figure 3.5 is $4+1+1=6$. That is, $h'_{WDG}=6$ can be used as an admissible h-value for the root CT node of CBS for the MAPF instance in Figure 3.1.

3.3.2 Constructing Weighted Pairwise Dependency Graphs

We first construct the vertices and edges of G_{WD} for CT node N using the same method as in Section 3.2.2. To calculate the weight Δ_{ij} of every edge $(v_i, v_j) \in E_D$, we then run an optimal MAPF algorithm to find the minimum sum of costs of the conflict-free paths for agents a_i and a_j that satisfy $N.constraints$ (ignoring the other agents). Here, the pathfinding problem is a two-agent MAPF problem with the constraints from $N.constraints$ imposed on the two agents. Most optimal MAPF algorithms can be adapted to satisfy these constraints.

Similar to Section 3.2.2, for each non-root CT node, we need to find the edges and calculate the weights for only the agent that the new constraint is imposed on and can copy the rest of the edges and their weights from the parent CT node.

3.3.3 The Two-Agent MAPF Problem

We tried three search-based optimal MAPF algorithms for solving the two-agent MAPF problem in our experiments, namely CBSH, EPEA* [71] (i.e., A* with partial expansion introduced in Section 2.2.1.1), and ICTS [152] (introduced in Section 2.2.1.1). CBSH turned out to be significantly faster than the other two MAPF algorithms.

One enhancement that we use in CBSH for the two-agent MAPF problem is that we set the h-value of the root CT node to 1. This h-value is admissible because Δ_{ij} is at least 1. It can help CBSH to resolve cardinal rectangle conflicts or other symmetric conflicts more efficiently. We further use pathmax to ensure that the f-values of the CT nodes along a branch are non-decreasing. Recall the cardinal rectangle conflict discussed in Example 1.2. The number of CT nodes expanded by CBS grows exponentially with the size of the yellow rectangular area. CBSH performs slightly better, but its number of expanded CT nodes still grows exponentially (see Figure 4.1b) because most of the conflicts in the CT nodes are not cardinal. In such two-agent instances, the cost C^* of the optimal solution is always one unit larger than the cost of the root CT node. Since CBSH searches in a best-first manner, it has to expand all CT nodes of f-value $C^* - 1$, even if it has already generated a CT node of f-value C^* with an optimal solution. However, if the h-value of the root CT node is 1, all CT nodes generated by CBSH have an f-value of C^*. So, CBSH with a good tie-breaking rule (such as preferring the latest generated CT node) can quickly generate a CT node with an optimal solution and return this solution immediately. In our experiments, this speeds up CBSH for the two-agent MAPF problem by up to three orders of magnitude.

3.4 Runtime Reduction Techniques

Computing the CG, DG, and WDG heuristics incurs runtime overhead per CT node. In this section, we introduce a number of simple techniques to reduce the runtime overhead for the calculation of the heuristics.

3.4.1 Lazy Computation of Heuristics

The high-level search of CBS with admissible heuristics resembles an A* search, so techniques for speeding up A* can be applied here. Lazy A* [178] improves A* by evaluating expensive heuristics lazily. Instead of computing the expensive h-value $h_2(N)$ immediately after generating a new CT node N, lazy A* first computes a cheaper but less informed h-value $h_1(N)$ (or even uses zero) and inserts CT node N into OPEN. Only when lazy A* receives CT node N from OPEN, it computes $h_2(N)$ for CT node N and re-inserts it into OPEN. Empirically, the runtime overhead of the operations on OPEN (e.g., inserting or popping a CT node) is negligible for CBS with admissible heuristics.

Here, for simplicity, we view both the cardinal conflict graph and the pairwise dependency graph as an edge-weighted pairwise dependency graph all of whose edges have weight one. We use $G_{WD}(N) = (V_D(N), E_D(N), W_D(N))$ to denote the edge-weighted pairwise dependency graph of CT node N and $\mathrm{EWMVC}(G_{WD}(N))$ to denote the sum $\sum_{v_i \in V_{D(N)}} x_i$ of the EWMVC of graph $G_{WD}(N)$. Each of the CG, DG, or WDG heuristics is treated as h_2, and we define h_1 for a child CT node N' of CT node N as

$$h_1(N') = \max\{h(N) - \max_{j:(i,j)\in E_D(N)} \Delta_{ij}, cost(N) + h(N) - cost(N'), 0\}, \tag{3.1}$$

where i is the index of the agent whose path gets replanned at CT node N'. The first term $h(N) - \max_{j:(i,j)\in E_D(N)} \Delta_{ij}$ is a lower bound on $\mathrm{EWMVC}(G'_{WD})$, where

$$G'_{WD} = (V_D(N) \setminus \{i\}, \{(u,v) \in E_D(N) \mid u \neq i \wedge v \neq i\}, \{w(u,v) \in W_D(N) \mid u \neq i \wedge v \neq i\}), \tag{3.2}$$

and thus a lower bound on $\mathrm{EWMVC}(G_{WD}(N'))$ because

$$G'_{WD} = (V_D(N') \setminus \{i\}, \{(u,v) \in E_D(N') \mid u \neq i \wedge v \neq i\}, \{w(u,v) \in W_D(N') \mid u \neq i \wedge v \neq i\}). \tag{3.3}$$

Since $\mathrm{EWMVC}(G_{WD}(N'))$ is our admissible WDG heuristic of CT node N', the first term is admissible. The second term $cost(N) + h(N) - cost(N')$ is admissible because the f-value (i.e., $cost(N) + h(N)$) is non-decreasing from CT node N to CT node N' (known as the pathmax strategy).

3.4.2 Memoization

Memoization is an optimization technique for speeding up algorithms by caching the results of expensive function calls and returning the cached results when the same inputs occur again. Here, we use memoization to store the results of merging the MDDs and solving the two-agent MAPF problems. The inputs are the indices of two agents and the set of constraints imposed on them. The output is the existence of the corresponding edge and, if it exists, its edge weight. Empirically, both the memory overhead of caching and the runtime overhead of storing and retrieving the results are negligible, and the cached results are used frequently. This is so because CBS often resolves the same conflict in different branches, and many CT nodes thus have the same set of constraints imposed on the same pair of agents.

We also use memoization for the MDDs of each agent, where the inputs are the index of the agent and the set of constraints imposed on it, and the output is the MDD.

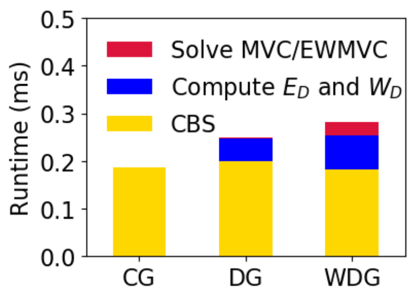
3.5 Empirical Evaluation

We experiment with ICBS (with the zero heuristic) and ICBS with the CG, DG, and WDG heuristics on four-neighbor grids. All ICBS algorithms use the two improvements discussed in Section 3.4, and the WDG heuristic uses the CBSH algorithm discussed in Section 3.3.3 to solve the two-agent MAPF problem. We generate 50 instances with random start and target vertices for each map and each number of agents. Our code is written in C++, and our experiments are conducted on a 2.80 GHz Intel Core i7-7700 laptop with 8 GB RAM.

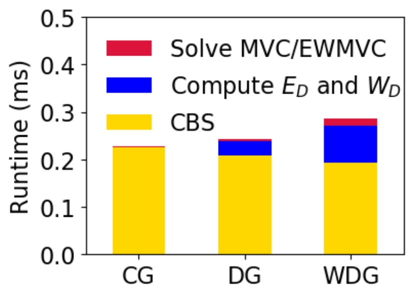
Empirically, since the cardinal conflict graph, the pairwise dependency graph, and the weighted pairwise dependency graph are always small and sparse, the runtime overhead of solving the NP-hard MVC and EWMVC problems are reasonable (as shown in Figure 3.6). A similar phenomenon was reported in [60]. We therefore always use the CG, DG, and WDG heuristics instead of their greedy versions.

Empty map				Dense map				20 agents			
Agents	CG	DG	WDG	Agents	CG	DG	WDG	Obs	CG	DG	WDG
30	0.2	1.0	1.2	16	3.9	3.9	11.6	0	0.1	0.5	0.5
40	0.5	1.7	2.0	20	4.8	4.8	15.2	10	1.0	1.3	2.1
50	0.6	2.3	2.8	24	6.9	7.0	22.2	20	3.0	3.1	6.2

Table 3.1: Average h-values of the root CT node. "Obs" represents the percentage of cells that are randomly blocked on a 20×20 grid.



(a) Empty map with 10, 20, 30, 40, 50, and 60 agents.



(b) Dense map with 10, 15, 18, 20, 22, and 24 agents.

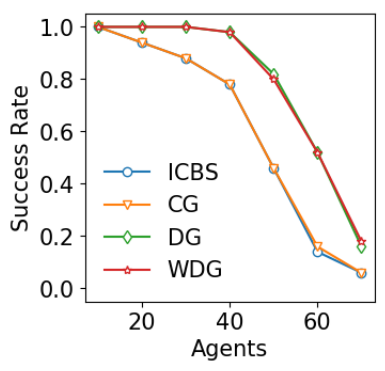
Figure 3.6: Average runtimes per expanded CT node on the empty and dense maps. We use 6 different numbers of agents for each map, resulting in 300 instances per map.

3.5.1 Small Maps

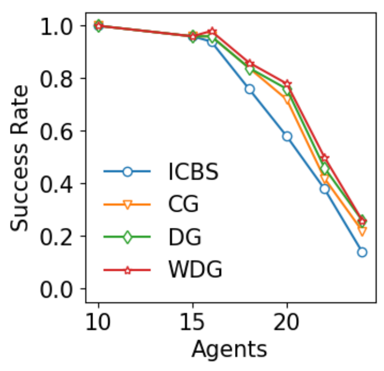
First, we test the algorithms on two 20×20 grids, namely an *empty map*, which is a 20×20 grid with no blocked cells, and a *dense map*, which is a 20×20 grid with 30% randomly blocked cells. We use a runtime limit of one minute for each algorithm on each instance.

h-values of the root CT node. Table 3.1 shows the h-values of the root CT node. On the empty map, DG is much larger than CG while WDG is only slightly larger than DG because agents on the empty map usually have many shortest paths, and thus Δ_{ij} is 0 or 1 in most cases. However, on the dense map, DG is only slightly larger than CG while WDG is much larger than both of them because most conflicts are cardinal, and the map contains many narrow passages, which induce a large Δ_{ij}. The last four columns show the results for 20 agents on grids with increasing obstacle densities to provide more details on the transition from empty grids to dense grids.

Runtime overhead of calculating the h-values. Figure 3.6 shows the breakdown of the runtimes per CT node. The CBS runtimes (yellow) of the three algorithms are slightly different because the



(a) Empty map.



(b) Dense map.

Figure 3.7: Success rates on the small maps.

m	Instances	ICBS	CG	DG	WDG
Number of expanded CT nodes ($\times$1000)					
30	44	3.6	2.6	**0.5**	**0.5**
40	39	8.9	7.0	**0.2**	**0.2**
50	23	12.4	10.1	**2.9**	**2.9**
Runtime (s)					
30	44	0.5	0.4	**0.1**	**0.1**
40	39	1.0	0.9	**0.1**	**0.1**
50	23	1.7	1.5	**0.6**	0.7

(a) Empty map.

m	Instances	ICBS	CG	DG	WDG
Number of expanded CT nodes ($\times$1000)					
16	47	20.2	9.6	7.8	**6.1**
20	29	20.2	13.6	10.7	**8.9**
24	7	79.6	47.4	33.2	**15.2**
Runtime (s)					
16	47	7.0	**2.4**	**2.4**	**2.4**
20	29	4.0	3.3	2.1	**1.9**
24	7	17.9	9.6	5.4	**3.0**

(b) Dense map.

Table 3.2: Average numbers of expanded CT nodes and average runtimes over instances solved by all algorithms.

different heuristics cause CBS to expand different sets of CT nodes. The runtimes of constructing G_D and G_{WD} (blue) are small due to the memoization technique, which saves more than 90% of the edge and weight computation time. Although we use simple algorithms for solving the NP-hard MVC and EWMVC problems, their runtimes (red) are also small due to the small sizes of G_D and G_{WD}. The lazy computation of heuristics also contributes to the reduction in the runtime overhead as the expensive heuristics are computed for only 65% of the generated CT nodes.

Overall performance. Figure 3.7 and Table 3.2 show the *success rates* (i.e., the percentages of solved instances within the runtime limit), the average numbers of expanded CT nodes, and the average runtimes of the algorithms. The numbers of expanded CT nodes are consistent with the computed h-values of the root CT node. That is, larger h-values usually lead to smaller numbers

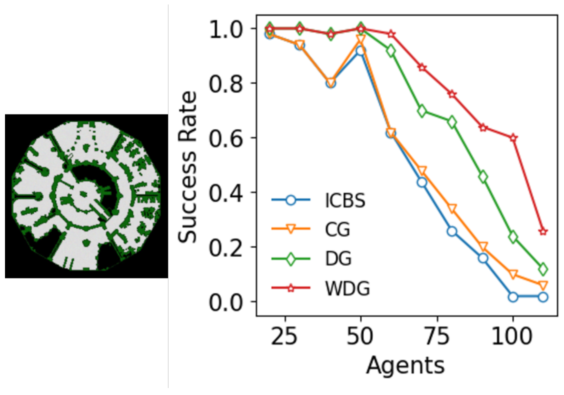


Figure 3.8: Success rates on the large map.

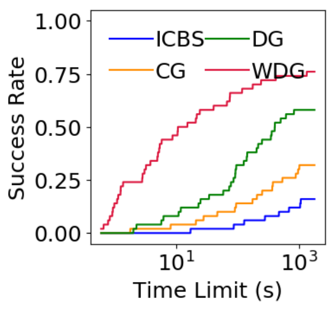
of expanded CT nodes. In terms of the success rates and runtimes, both DG and WDG outperform CG. In particular, DG runs slightly faster than WDG on the empty map as it has a smaller runtime overhead than WDG, while WDG runs much faster than DG on the dense map as it leads to a larger CT node reduction than DG. The usefulness of CG heavily depends on the particular instance. CG has almost the same efficiency as ICBS on the empty map where cardinal conflicts are rare. Overall, WDG improves the success rate of ICBS by up to 2 times (e.g., for 60 agents on the empty map). It also reduces the number of expanded CT nodes and the runtime of ICBS by up to 45 times and 10 times (e.g., for 40 agents on the empty map), respectively, for the instances solved by all algorithms (which are relatively easy).

3.5.2 Large Maps

Next, we test the algorithms on a *large map*, namely the benchmark game map `lak503d` from [165], which is a 192×192 grid with 51% blocked cells, see Figure 3.8(left).

Success rates. Figure 3.8 shows the success rates of ICBS with different heuristics with a runtime limit of one minute. As before, all heuristics improve the success rates of ICBS, and WDG has the largest improvements.

Results with longer runtime limits. Figure 3.9a shows the success rates on 50 instances with 100 agents on the large map for different runtime limits. As the runtime limits increase, the benefits



(a) Different runtime limits.

	CG	DG	WDG
All 50 instances			
h-value of the root CT node	9.4	10.1	**17.0**
Runtime per CT node expansion (ms)	**16.1**	21.7	21.9
Success rate	0.32	0.58	**0.76**
16 instances solved by all three algorithms			
Number of expanded CT nodes ($\times$1000)	19.9	6.9	**0.4**
Runtime (s)	319	141	**6**

(b) Runtime limit of thirty minutes.

Figure 3.9: Results for 100 agents on the large map.

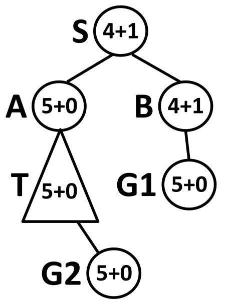
Empty map					Dense map					Large map				
m	CG	DG	WDG	h^*	m	CG	DG	WDG	h^*	m	CG	DG	WDG	h^*
30	0.2	1.0	1.2	1.7	16	3.9	3.9	11.5	18.6	60	3.6	4.0	6.7	7.6
40	0.5	1.6	2.0	3.3	20	4.7	4.7	14.0	23.2	80	5.7	6.5	10.9	12.2
50	0.5	2.2	2.6	4.7	24	6.5	6.5	18.9	28.5	100	8.6	9.2	15.6	18.0

Table 3.3: Average h- and h^*-values of the root CT node over instances of which the h^*-value is known, i.e., instances solved by at least one ICBS algorithm.

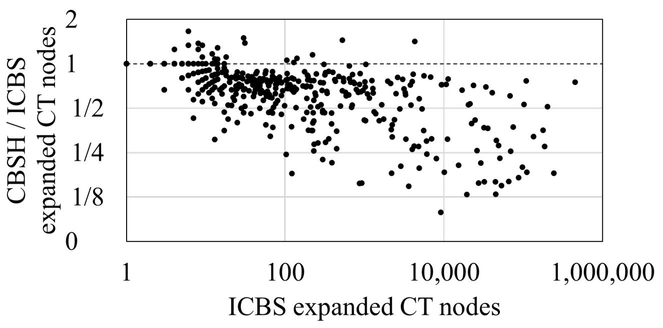
of our admissible heuristics increase. It is generally worth spending some extra runtime per CT node expansion to obtain a larger h-value since a larger h-value usually leads to an exponential reduction in the number of expanded CT nodes. Figure 3.9b shows the results for a runtime limit of thirty minutes. Although DG and WDG have a larger runtime overhead on this large map than on the small maps, WDG significantly outperforms DG, which - in turn - significantly outperforms CG in terms of both success rate and runtime. For example, compared with CG, WDG increases the success rate by a factor of 2 and runs faster by a factor of 50.

3.5.3 Comparison with the Perfect Heuristic

Table 3.3 compares the average h-values of the root CT node by ICBS algorithms using different heuristics with the average h^*-values of the root CT node (i.e., the optimal solution cost minus the cost of the root CT node). On the dense map, WDG is significantly smaller than h^* because agents are deeply coupled, and reasoning about only the pairwise dependencies between agents is insufficient. However, on the empty or large map, WDG is close to h^* because agents are less



(a) Illustration of a CT.



(b) Number of expanded CT nodes.

Figure 3.10: Comparison between ICBS and CBSH.

coupled, and reasoning about the pairwise dependencies between them is sufficient in many cases. In other words, h/h^* is closer to 1 on the empty map or the large map than the dense map. This explains why the h-values of WDG are much larger than those of CG on the dense map (as shown in Tables 3.1 and 3.3), but they reduce the numbers of expanded CT nodes only slightly over CG (as shown in Table 3.2).

3.5.4 Possible Slowdown

Depending on tie breaking, A* with admissible h-values can expand more nodes than with zero h-values if the admissible h-values of some non-goal nodes are zero. Zero h-values for non-goal nodes must exist if zero-cost edges are allowed and connected to goal CT nodes.

When a CT node N is split based on a semi-cardinal or non-cardinal conflict, at least one of its child CT nodes have the same cost as CT node N. That is, the corresponding CT edge can be regarded as a *zero-cost edge*. Admissible h-values of non-goal CT nodes have to be zero in case they are connected to goal CT nodes via one zero-cost edge or a sequence of zero-cost edges. Thus, ICBS with admissible heuristics can expand more CT nodes than ICBS. Figure 3.10a shows an example CT. The expressions inside the CT nodes are the sums of their g-values and h-values. CT nodes $G1$ and $G2$ are goal CT nodes. ICBS first expands CT node S. It then expands CT node B since it has a smaller g-value (and thus cost and priority) than CT node A. Finally, it expands CT node $G1$ (at which point it terminates) since it has the same g-value as CT node A but fewer conflicts (since the plans in goal CT nodes are conflict-free while the ones in non-goal CT nodes are not). ICBS with admissible heuristics first expands CT node S as well. It then expands node CT A if it has fewer conflicts than CT node B since it has the same sum of g- and h-values (and thus priority) as CT node B. (It would also expand CT node A in case it broke ties in favor of CT nodes with smaller h-values.) It can then expand the entire subtree T rooted at CT node A and finally CT node $G2$ (at which point it terminates). In this case, ICBS with admissible heuristics expands more CT nodes than ICBS. If the h-values of non-goal CT nodes were strictly larger than zero, then ICBS with admissible heuristics would avoid this issue since it would expand CT node B (instead of CT node A) and finally CT node $G1$ (at which point it would terminate).

However, in our experiments, CBSH expands more CT nodes than ICBS for fewer than 5% of the instances, and these cases do not significantly contribute to the average number of expanded CT nodes. Figure 3.10b shows the ratio of the number of expanded CT nodes by CBSH and ICBS (as a function of the number of expanded CT nodes by ICBS) on instances with 10 agents on 8×8 grids with 10% to 35% randomly blocked cells that ICBS solves with a runtime limit of five minutes. ICBS expands fewer CT nodes than CBSH for only 22 out of 447 instances.

3.6 Summary

In this chapter, we provided first evidence that admissible h-values are beneficial for CBS. We proposed three admissible heuristics, namely CG, DG, and WDG, by reasoning about the conflicts and pairwise dependencies between agents. Theoretically, the WDG heuristic is provably no smaller than the DG heuristic, which in turn is provably no smaller than the CG heuristic. Empirically, they all incur a small runtime overhead per expanded CT node, with the CG heuristic incurring the smallest and the WDG heuristic incurring the largest runtime overhead. They increase the success rates and efficiency of ICBS by up to a factor of fifty.

3.7 Extensions

The three heuristics are beneficial for CBS for the classic MAPF problem and have been further improved by us and other researchers. For example, Boyarski et al. [30] show that, when facing large MAPF instances, solving MVCs with integer linear programming solvers in an incremental way can calculate the CG heuristic faster. Boyarski et al. [31] improve the CG, DG, and WDG heuristics by using information about the cost changes when resolving certain conflicts. Mogali et al. [126] improve the DG heuristic by reasoning about groups of three (instead of two) agents using the Lagrangian Relax-and-Cut scheme.

The three heuristics are beneficial not only for using CBS to solve the classic MAPF problem but also for using CBS variants to solve generalized MAPF problems. For example, Andreychuk et al. [6] use the greedy CG heuristic to speed up a CBS variant for solving MAPF with continuous traversal time. Chen et al. [39] use the CG heuristic to speed up a CBS variant for solving MAPF with robustness guarantees in terms of navigation delays. Chen et al. [40] use the CG heuristic to speed up a CBS variant for solving MAPF with agents of different lengths. Li et al. [105] use a combination of the CG and WDG heuristics to speed up a CBS variant for solving MAPF with agents of different shapes. Zhong et al. [215] use the CG, DG, and WDG heuristics to speed up a CBS variant for solving MAPF with target assignment.

In Chapter 5, we will show that such admissible heuristics are also beneficial for bounded-suboptimal CBS variants.

Chapter 4

Speeding up Optimal CBS via Symmetry Reasoning

In this chapter, we show that one of the reasons why MAPF is so difficult to solve is a phenomenon called pairwise symmetry, which occurs when two agents have many different paths to their target vertices, all of which appear promising, but every combination of them results in a conflict. We identify several classes of pairwise symmetries, namely

- *rectangle symmetry*, which arises when two agents attempt to cross each other in an open area and repeatedly conflict with each other along many different shortest paths.
- *target symmetry*, which arises when one moving agent repeatedly conflicts with another agent waiting at its target vertex along many different paths of increasing lengths.
- *corridor symmetry*, which arises when two agents moving in opposite directions repeatedly conflict with each other inside a narrow passage along many different paths of increasing lengths.

We show that each symmetry arises commonly in practice and can produce an exponential explosion in the conflict-resolution space, leading to unacceptable runtimes for optimal MAPF algorithms.[1]

For each type of symmetry, we propose algorithmic reasoning techniques that can identify the situation at hand and resolve it in a single branching step by adding symmetry-breaking constraints.

[1]In Section 5.3.3, we will show that this issue also occurs for bounded-suboptimal MAPF algorithms.

We explore these ideas in the context of CBS (or, more precisely, CBSH). On the one hand, we provide a theoretical analysis that shows that our symmetry-reasoning techniques preserve the completeness and optimality of CBS. On the other hand, we evaluate the impact of these symmetry-reasoning techniques empirically and show that the symmetry-reasoning techniques can lead to an exponential reduction in the number of expanded CT nodes. We show that CBSH with our symmetry-reasoning techniques resolves most two-agent conflicts in just a single branching step. We also show that our symmetry-reasoning techniques substantially improve the runtimes and success rates of CBSH and CBSH2. For example, adding the symmetry-reasoning techniques to CBSH can reduce the number of CT nodes expanded CBSH by up to four orders of magnitude and increase its scalability up to thirty times. Combining the WDG heuristic and the symmetry-reasoning techniques leads to the best CBS-based optimal MAPF algorithm CBSH2-RTC.

In this chapter, we refer to a pair of a vertex $v \in V$ and a timestep $t \in \mathbb{N}$ as a *space-time node* (or *node* for short) (v,t). A MDD node (see Definition 2.3) is a space-time node. We say that a path (or agent) visits node (v,t) iff it visits vertex v at timestep t. We say that two paths (or agents) conflict at node (v,t) iff they conflict at vertex v at timestep t and refer to the node as the conflicting node. We say that a constraint *blocks* a path iff the path does not satisfy the constraint.

This chapter closely follows [104, 107, 111].

4.1 Background

We first explain how to design constraints to resolve conflicts in CBS without losing its completeness (and optimality) guarantees. We then review existing work that reasons about symmetries in MAPF and other areas of AI.

4.1.1 Principle of Designing Constraints for CBS

CBS is complete and optimal (recall Theorem 2.1). Since we will introduce new types of constraints to resolve symmetry conflicts in this chapter, we here provide the principle of designing constraints for CBS without losing its completeness or optimality guarantees.

Definition 4.1 (Mutually Disjunctive Constraints)**.** *Two constraints are* mutually disjunctive *iff any pair of conflict-free paths satisfies at least one of the two constraints, i.e., there does not exist a pair of conflict-free paths that violates both constraints. Moreover, two sets of constraints are* mutually disjunctive *iff each constraint in one set is mutually disjunctive with each constraint in the other set, i.e., any pair of conflict-free paths satisfies at least one set of constraints.*

In our previous work [105], we prove that using two sets of mutually disjunctive constraints to resolve a conflict at a CT node preserves the completeness and optimality of CBS. The key idea of the proof is to show that any solution that satisfies the constraints of a CT node also satisfies the constraints of at least one of its child CT nodes, as stated in Lemma 4.1.

Lemma 4.1. *For a given CT node N with constraint set C, if two constraint sets C_1 and C_2 are mutually disjunctive, any solution that satisfy C also satisfy at least one of the constraint sets $C \cup C_1$ and $C \cup C_2$.*

Proof. This is true because, otherwise, there would exist a pair of conflict-free paths that does not satisfy all constraints in C_1 and does not satisfy all constraints in C_2. That is, one of the paths violates a constraint $c_1 \in C_1$, and one of the paths violates a constraint $c_2 \in C_2$. Then, c_1 and c_2 are not mutually disjunctive, contradicting the assumption. □

By reusing the proof of Theorem 2.1, we have the following theorem.

Theorem 4.1. *Using two constraint sets to split a CT node preserves the completeness and optimality of CBS if the two constraint sets are mutually disjunctive and each of them blocks at least one path in the plan of the CT node.* □

Hence, the principle of designing constraints for CBS is to ensure that the two constraints (or constraint sets) that we use to split a CT node are mutually disjunctive and block some paths in the plan of the CT node.

4.1.2 Related Work on Symmetry Reasoning

We review existing methods that can eliminate (some) symmetries in MAPF and existing symmetry-reasoning work done in the context of other problems.

4.1.2.1 Existing Approaches for Eliminating Symmetries in MAPF

In pathfinding problems, symmetries have so far been studied only for single agents, e.g., by exploiting grid symmetries [76]. Some prior work can eliminate some symmetries in MAPF (but loses optimality or completeness guarantees) by preprocessing the input graphs. We describe two of them below.

Graph decomposition. Ryan [147] proposes several graph decomposition approaches for solving MAPF. Like our work, he detects special graph structures, including stacks, cliques, and halls. Unlike our work, he builds an abstract graph by merging each such sub-graph into a meta-vertex during preprocessing in order to reduce the search space. His work preserves completeness but not optimality. Our work, by comparison, focuses on exploiting the sub-graphs to break symmetries without preprocessing or sacrificing optimality.

Highways. Cohen et al. [46] propose highways to reduce the number of corridor conflicts (defined in Section 4.5). They assign directions to some corridor vertices (resulting in one or more highways) and make moving against highways more expensive than other movements. They show that highways can speed up ECBS, a bounded-suboptimal variant of CBS and introduced in Section 2.3.3. However, the utility of highways for optimal CBS is limited because they can only be used to break ties among multiple shortest paths and are not guaranteed to resolve all corridor conflicts. Similar ideas of introducing directions to the graph edges are also explored in flow annotation replanning [192], direction maps [89], and optimized directed roadmap graphs [77], none of which guarantee optimality.

4.1.2.2 Symmetry Reasoning in Other Areas of AI

Symmetry is a widely-used concept that has been studied in many AI communities.

For example, symmetry reasoning is a successful technique in planning [65, 66, 137, 53, 54, 155, 196, 69, 145]. Here, symmetries usually refer to *state symmetries* [137], which are defined as the automorphisms[2] of the state transition graph, i.e., a directed multigraph, where the set of vertices contains a vertex for every state and the set of edges contains a directed edge for every operator that leads from one state to another. However, the state transition graph is usually extremely large in practice, so existing work usually infers state symmetries from a compact description of it, such as a semantic description of the planning task [137] or a factored representation of the planning task [155]. State symmetries take the form of symmetry groups across states. If several states from a group are encountered, only one of them is explored. In addition, information obtained during the search at different symmetric states can also be used to improve heuristics [54].

Symmetry reasoning is also a successful technique in constraint programming [140, 20, 14, 64, 141, 142, 150, 188, 44, 99, 125]. Cohen et al. [44] propose a *microstructure* for constraint satisfaction problems, i.e., a hypergraph, where the set of vertices contains a vertex for every literal (i.e., variable-value pair) and the set of edges contains a (hyper-)edge among a set of literals that corresponds to either an assignment allowed by a specific constraint or an assignment allowed because there is no constraint between the associated variables. Then, they define a *constraint symmetry* as an automorphism of the microstructure, which is conceptually similar to the automorphism of the state transition graph used in planning. As for detecting symmetries in planning, for the sake of computational efficiency, existing work in constraint programming usually infers (subsets of) constraint symmetries from, for example, *variable symmetries* [188], i.e., the variables are

[2]An automorphism of a graph is a bijection on the vertices that preserves the edges (and hence also the non-edges).

interchangeable, or *value symmetries* [188, 99], i.e., the values are interchangeable. The detected symmetries can then be eliminated by adding symmetry-breaking constraints [64] or performing symmetry breaking during search [14].

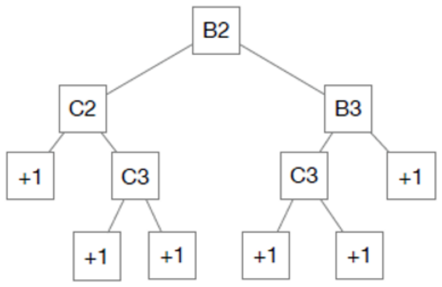
Similar symmetries have also been studied in propositional satisfiability problems [51, 3], model checking [57, 26], path and motion planning for single agents [41, 76], etc. Although symmetries have been widely studied in the literature, existing work always focuses on problem/state/solution symmetries in the sense that "renaming" (permuting somehow) some variables, values, propositions, or operators results in identical problems/states/solutions. However, we focus on conflict symmetries in this chapter instead. It is unclear how to translate knowledge in symmetry reasoning in these domains to improvements to CBS or other similar MAPF algorithms because they search the conflict-resolution space as opposed to the problem/state/solution space.

4.2 Rectangle Symmetry

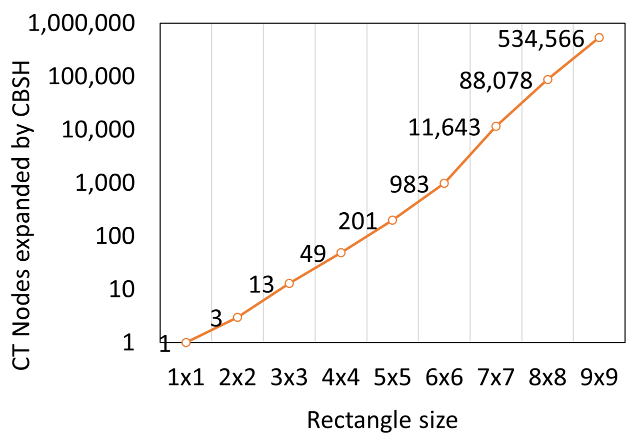
We start with some examples to show the motivation behind rectangle reasoning. Formal definitions of rectangle conflicts are introduced later. In this section, we focus on four-neighbor grids, as required by rectangle-reasoning techniques. In particular, for a space-time node S, we use $(S.x, S.y)$ to denote its cell and $S.t$ to denote its timestep.

Recall the MAPF instance discussed in Examples 1.1 and 1.2. Figure 4.1a shows the corresponding CT of CBS. Figure 4.1b shows the number of CT nodes expanded by CBSH when the yellow rectangular area in the MAPF instance in Figure 1.5a is larger, indicating that the size of the CT grows exponentially with the size of the rectangular area. So, even for a two-agent MAPF instance, CBSH can time out if the rectangle conflict is not detected. According to Definition 2.2, these rectangle conflicts are cardinal. However, reasoning about cardinal rectangle conflicts does not eliminate all rectangle symmetries in the conflict-resolution space.

Example 4.1. Consider the MAPF instances shown in Figure 4.2 and ignore R_s, R_g, R_1, and R_2 for now. Suppose that $S_1.t = S_2.t = 0$. The conflict in Figure 4.2b is not a cardinal rectangle conflict

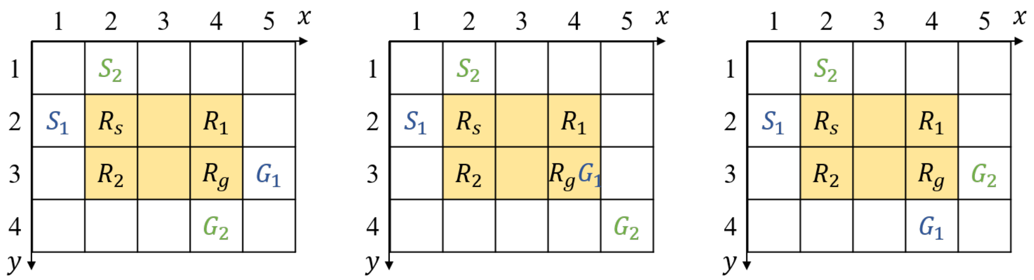


(a) CT generated by CBS for solving the two-agent MAPF instance in Figure 1.5a.



(b) Number of CT nodes expanded by CBSH empirically for the two-agent MAPF instance in Figure 1.5a with different (yellow) rectangle sizes.

Figure 4.1: Example of (cardinal) rectangle conflicts. In (a), each left branch constrains agent a_2, and each right branch constrains agent a_1. Each non-leaf CT node is marked with the cell of the chosen conflict. Each leaf CT node marked "+1" contains an optimal solution, whose sum of costs is one larger than the sum of costs of the plan of the root CT node.



(a) Cardinal conflict. (b) Semi-cardinal conflict. (c) Non-cardinal conflict.

Figure 4.2: Examples of different types of rectangle conflicts. The cells of the start and target nodes are shown in the figures. The timesteps of the start and target nodes are $S_1.t = S_2.t$ and $G_i.t = S_i.t + |G_i.x - S_i.x| + |G_i.y - S_i.y|$ for $i = 1,2$. The conflicting area is highlighted in yellow. R_s, R_g, R_1, and R_2 denote the four corner nodes of the rectangle.

because agent a_2 has *optimal bypasses*, i.e., shortest paths that do not traverse the rectangular area (e.g., path [(2, 1), (3, 1), (4, 1), (5, 1), (5, 2), (5, 3), (5, 4)]). However, if cell (5, 2) at timestep 4 and cell (5, 3) at timestep 5 are occupied by other agents, then the low-level search of CBS always finds a path for agent a_2 that conflicts with the path of agent a_1, because the low-level search uses the number of conflicts with other agents to break ties. Therefore, CBS generates again many CT nodes before finally finding conflict-free paths. □

We refer to the conflict in Figure 4.2b as a semi-cardinal rectangle conflict. Similarly, we refer to the conflict in Figure 4.2c, where both agents have optimal bypasses, as a non-cardinal rectangle conflict. Together with cardinal rectangle conflicts, we refer to these types of conflicts as rectangle conflicts. In Section 4.2.1, we introduce a rectangle-reasoning technique that can efficiently identify and resolve rectangle conflicts between entire paths. Then, in Section 4.2.2, we generalize the reasoning technique to rectangle conflicts between path segments. Both techniques are applicable only on four-neighbor grids. In Section 4.3, we generalize rectangle conflicts to cases where the *conflicting area* (i.e., the yellow area in Figure 1.5a) is not necessarily rectangular and propose a more general reasoning technique that can work on planar graphs. We evaluate the performance of all three rectangle-reasoning techniques in Section 4.3.4.

4.2.1 Rectangle Reasoning Technique I: For Entire Paths

Consider two agents a_1 and a_2. Let nodes S_1, S_2, G_1 and G_2 be their start and target nodes, respectively. For now, we assume that the start node is at the start vertex at timestep 0 and the target node is at the target vertex at the timestep when the agent completes its path. But, in the next subsection, we will relax this assumption to allow our method to detect rectangle symmetries between path segments. Below are the formal definitions of the conflicting area (i.e., yellow rectangular area) and the rectangle conflicts, with some examples shown in Figure 4.2.

Definition 4.2 (Conflicting Area)**.** *Given start and target nodes S_1, S_2, G_1, and G_2 for agents a_1 and a_2, respectively, we define the* conflicting area *as the intersection cells of the rectangular area with diagonal corners $(S_1.x, S_1.y)$ and $(G_1.x, G_1.y)$ and the rectangular area with diagonal corners $(S_2.x, S_2.y)$ and $(G_2.x, G_2.y)$.*

Definition 4.3 (Rectangle Conflict)**.** *Two agents are involved in a* rectangle conflict *iff*

1. *they have at least one vertex conflict along their paths,*

2. *both paths are* Manhattan-optimal*, i.e., for each agent, the length of its path is equal to the Manhattan distance between the cells of its start and target nodes, and*

3. *both paths move in the same x-direction and the same y-direction.*

Property 4.1. *Given a rectangle conflict between two agents, the distances from the cell of the start node of one agent to any cell x inside the conflicting area is equal to the distance from the cell of the start node of the other agent to cell x.*

Proof. Suppose that the vertex conflict in Condition 1 is $\langle a_1, a_2, v, t\rangle$. From Condition 2, we know that $t = dist(s_1, v) = dist(s_2, v)$. Then, from Conditions 2 and 3 and a simple geometric analysis, we know that $dist(s_1, x) = dist(s_2, x)$ holds for every cell x inside the conflicting area. □

From Property 4.1, we know that, if two agents have a rectangle conflict, all their paths from their start nodes to their target nodes reach the same cell inside the conflicting area at the same timestep. We therefore define the rectangle, which is a set of nodes located inside the conflicting area, for a rectangle conflict as follows. Please refer to Figure 4.2 for illustration.

Definition 4.4 (Rectangle). *Given start and target nodes S_1, S_2, G_1, and G_2 for agents a_1 and a_2 with a rectangle conflict, respectively, we define the* rectangle *as a set of nodes whose cells are the cells in the conflicting area and whose timesteps are the timesteps when a shortest path of agent a_1 or agent a_2 reaches the cell of the node. The four corner nodes of the rectangle are referred to as R_s, R_g, R_1, and R_2, where R_s and R_g are the corner nodes whose cells are closest to the cells of the start and target nodes, respectively, and R_1 and R_2 are the other corner nodes whose cells are on the borders opposite of the cells of S_1 and S_2, respectively. The border from R_1 to R_g and the border from R_2 to R_g (or, more precisely, the nodes in the rectangle whose cells are on the straight line segment from the cell of R_1 to the cell of R_g and from the cell of R_2 to the cell of R_g), are called the exit borders of agents a_1 and a_2 and denoted by R_1R_g and R_2R_g, respectively.*

The mathematical equations for computing the corner nodes are shown in Appendix C.1. Intuitively, every path for agent a_1 that moves from node S_1 to a node on border R_1R_g conflicts with every path for agent a_2 that moves from node S_2 to a node on border R_2R_g. We want to design branching methods that eliminate all such conflicting paths in a single branching step. In the following three subsections, we present in detail how to efficiently identify, classify, and resolve rectangle conflicts.

4.2.1.1 Identifying Rectangle Conflicts

Rectangle conflicts occur only when two agents have one or more vertex conflicts. Assume that agents a_1 and a_2 have a semi-/non-cardinal vertex conflict (which ensures that Condition 1 in Definition 4.3 holds). Here, we do not consider cardinal vertex conflicts because resolving a cardinal vertex conflict with vertex constraints can already increase the path cost of one of the agents in the child CT nodes and thus eliminate the rectangle symmetry. They have a rectangle conflict iff

$$|S_1.x - G_1.x| + |S_1.y - G_1.y| = G_1.t - S_1.t > 0 \tag{4.1}$$

$$|S_2.x - G_2.x| + |S_2.y - G_2.y| = G_2.t - S_2.t > 0 \tag{4.2}$$

$$(S_1.x - G_1.x)(S_2.x - G_2.x) \geq 0 \tag{4.3}$$

$$(S_1.y - G_1.y)(S_2.y - G_2.y) \geq 0. \tag{4.4}$$

Equations (4.1) and (4.2) ensure that Condition 2 in Definition 4.3 holds. Equations (4.3) and (4.4) ensure that Condition 3 in Definition 4.3 holds.

4.2.1.2 Resolving Rectangle Conflicts

Consider the examples in Figure 4.2. For cardinal rectangle conflicts, all pairs of the shortest paths have conflicts. For semi- and non-cardinal rectangle conflicts, although agents have shortest paths that are conflict-free, all pairs of the shortest paths that visit the corresponding exit borders of the agents have conflicts. We therefore propose to resolve a rectangle conflict by forcing one of the agents to leave its exit border later or take a detour. Formally, we introduce the *barrier constraint* $B(a_i, R_i, R_g) = \{\langle a_i, (x,y), t\rangle \mid ((x,y),t) \in R_iR_g\}$ for $i = 1,2$, which is a set of vertex constraints that prohibits agent a_i from occupying any node along its exit border R_iR_g. When resolving a rectangle conflict, we generate two child CT nodes and add $B(a_1, R_1, R_g)$ to one of them and $B(a_2, R_2, R_g)$ to the other one.

For instance, for the cardinal rectangle conflict in Figure 4.2a, the two barrier constraints are $B(a_1, R_1, R_g) = \{\langle a_1, (4, 2+n), 3+n\rangle \mid n = 0, 1\}$ and $B(a_2, R_2, R_g) = \{\langle a_2, (2+n, 3), 2+n\rangle \mid n = 0, 1, 2\}$. Barrier constraint $B(a_i, R_i, R_g)$ for $i = 1, 2$ blocks all shortest paths for agent a_i that reach its target node G_i via the rectangle. Thus, agent a_i for $i = 1, 2$ is replanned with a longer path that does not conflict with the current path for the other agent (recall that the low-level search of CBS breaks ties in favor of the path that has the minimum number of conflicts with the paths of the other agents). The rectangle conflict is thus resolved in a single branching step. For the semi-cardinal rectangle conflict in Figure 4.2b, the barrier constraints are the same. So, one of the barrier constraints $B(a_1, R_1, R_g)$ forces agent a_1 to take a longer path and very likely leads to conflict-free paths. The other barrier constraint $B(a_2, R_2, R_g)$ blocks only some of the shortest paths of agent a_2 and forces agent a_2 to not use its exit border, which increases the chances for agent a_2 to find a shortest path that does not conflict with agent a_1. The analysis for resolving the non-cardinal rectangle conflict in Figure 4.2c is analogous.

We use barrier constraints to resolve a rectangle conflict at a CT node N only if $B(a_i, R_i, R_g)$ for $i = 1, 2$ blocks the path for agent a_i in the plan of CT node N because, otherwise, the child CT node may have the same plan as CT node N. If this condition does not hold, we do not regard the conflict as a rectangle conflict. For example, given the instance shown in Figure 4.2c, if the paths of the two agents are [(1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)] and [(2, 1), (2, 2), (3, 2), (4, 2), (5, 2), (5, 3)], respectively, then we regard the conflict $\langle a_1, a_2, (2, 2), 1\rangle$ as a regular vertex conflict and resolve it by vertex constraints $\langle a_1, (2, 2), 1\rangle$ and $\langle a_2, (2, 2), 1\rangle$.

We add barrier constraints on the exit borders of the agents instead of their entry borders because there might be an optimal solution that violates both "entry-border" barrier constraints. For instance, given the rectangle conflict shown in Figure 4.2a, if we use "entry-border" barrier constraints $B(a_1, R_s, R_2) = \{\langle a_1, (2, 2+n), 1+n\rangle \mid n = 0, 1\}$ and $B(a_2, R_s, R_1) = \{\langle a_2, (2+n, 2), 1+n\rangle \mid n = 0, 1, 2\}$, then the pair of paths [(1, 2), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3)] for agent a_1 and [(2, 1), (2, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)] for agent a_2 is an optimal solution that violates both "entry-border" barrier constraints.

4.2.1.3 Classifying Rectangle Conflicts

To classify a rectangle conflict, we need to know whether the path length of agent a_i for $i = 1,2$ would increase after adding barrier constraint $B(a_i, R_i, R_g)$. Because of Condition 2 in Definition 4.3, all paths between the start and target nodes of agent a_i are within the S_i-G_i rectangle. We thus only need to compare the length and width of the rectangle with those of the S_1-G_1 and S_2-G_2 rectangles. Consider the two equations

$$R_i.x - R_g.x = S_i.x - G_i.x \tag{4.5}$$

$$R_i.y - R_g.y = S_i.y - G_i.y. \tag{4.6}$$

Equation (4.5) holds when the length of the rectangle is equal to the length of the S_i-G_i rectangle for $i = 1,2$, and Equation (4.6) holds when the width of the rectangle is equal to the width of the S_i-G_i rectangle for $i = 1,2$. Also, since the rectangle is the intersection of the S_1-G_1 and S_2-G_2 rectangles, its length and width cannot be larger than the lengths and widths, respectively, of the S_1-G_1 and S_2-G_2 rectangles. Therefore, if one of Equations (4.5) and (4.6) holds for $i = 1$ and the other one holds for $i = 2$, the rectangle conflict is cardinal; if only one of them holds for $i = 1$ or $i = 2$, it is semi-cardinal; otherwise, it is non-cardinal. For example, in Figure 4.2a, $R_2.x - R_g.x = S_2.x - G_2.x = -2$ and $R_1.y - R_g.y = S_1.y - G_1.y = -1$, so the conflict is cardinal.

4.2.1.4 Theoretical Analysis

Now, we present a sequence of properties of Rectangle Reasoning Technique I and prove its completeness and optimality.

Property 4.2. *If Rectangle Reasoning Technique I identifies a rectangle conflict between agents a_1 and a_2, then any path of agent a_1 that visits a node on its exit border R_1R_g also visits a node on* *its entry border R_sR_2, and any path of agent a_2 that visits a node on its exit border R_2R_g also visits a node on its entry border R_sR_1.* □

Property 4.2 is straightforward to prove, but the proof is lengthy. We thus include the formal proof only in Appendix C.2.

Property 4.3. *For all combinations of paths of agents a_1 and a_2 with a rectangle conflict found by Rectangle Reasoning Technique I, if one path violates barrier constraint $B(a_1,R_1,R_g)$ and the other path violates barrier constraint $B(a_2,R_2,R_g)$, then the two paths have one or more vertex conflicts where the conflicting node is within the rectangle.*

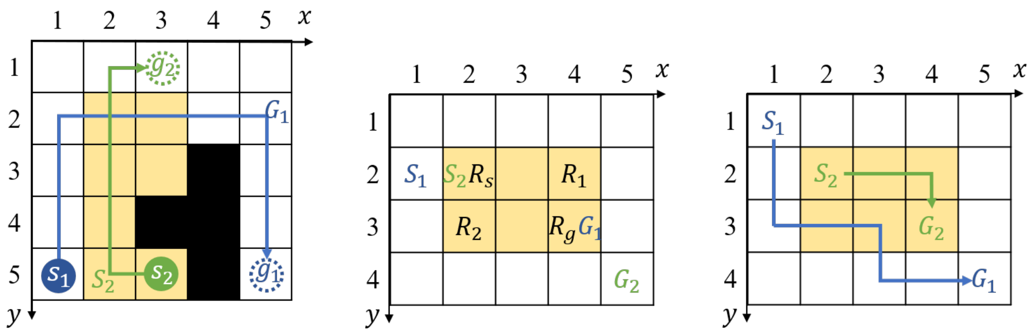
Proof. According to Property 4.2, any path that violates $B(a_1,R_1,R_g)$ must visit a node on border R_sR_2 and a node on border R_1R_g, and any path that violates $B(a_1,R_1,R_g)$ must visit a node on border R_sR_1 and a node on border R_2R_g. Since R_sR_2 and R_sR_1 are the opposite sides of R_1R_g and R_2R_g of the conflicting area, respectively, such two paths must cross each other, i.e., they visit a common cell within the conflicting area. According to Property 4.1, they must visit this cell at the same timestep. Therefore, the property holds. □

Property 4.3 implies that barrier constraints $B(a_1,R_1,R_g)$ and $B(a_2,R_2,R_g)$ are mutually disjunctive (recall Definition 4.1). According to Theorem 4.1 and the fact that $B(a_i,R_i,R_g)$ for $i=1,2$ blocks the current path for agent a_i, using them to split a CT node preserves the completeness and optimality of CBS.

Theorem 4.2. *Using Rectangle Reasoning Technique I preserves the completeness and optimality of CBS.* □

4.2.2 Rectangle Reasoning Technique II: For Path Segments

Rectangle Reasoning Technique I does not reason about obstacles and constraints, so it applies only to rectangle conflicts for entire paths. In some cases, however, rectangle conflicts exist for path segments but not for entire paths, such as the cardinal rectangle conflict in Figure 4.3a. Since the



(a) Cardinal conflict. (b) Semi-cardinal conflict. (c) No rectangle conflict.

Figure 4.3: Examples of rectangle conflicts for path segments. The cells of the start and target nodes are shown in the figures. In (a), the cells of S_1 and G_2 are indicated by s_1 and g_2. In (b) and (c), $G_i.t = S_i.t + |G_i.x - S_i.x| + |G_i.y - S_i.y|$ for $i = 1,2$. In (b), $S_1.t = S_2.t - 1$. In (c), $S_1.t = S_2.t - 2$.

paths are not Manhattan-optimal, Rectangle Reasoning Technique I fails to identify this rectangle conflict. Therefore, we extend the rectangle reasoning technique to reasoning about rectangle conflicts between two path segments, each of which starts at a singleton (recall Definition 2.4) and ends at another singleton. Since all shortest paths of an agent must visit all of its singletons, we can regard the two singletons as the start and target nodes and reuse Rectangle Reasoning Technique I with minor modifications.

Algorithm 4.1 shows the pseudo-code. It first treats all singletons as start and target node candidates (Lines 1 to 3) and then tries all combinations of them to find rectangle conflicts. If multiple rectangle conflicts are identified (see Example 4.2), it chooses the one of the highest priority type (i.e., cardinal > semi-cardinal > non-cardinal) and breaks ties in favor of the one with the largest rectangle area (Lines 11 and 12). It returns the pair of barrier constraints only if they block the current paths of the agents (Line 15), otherwise it would generate a child CT node whose paths and conflicts are exactly the same as those of the current CT node (see Example 4.3). We discuss details of the three functions on Lines 7,14, and 9 in Sections 4.2.2.1 to 4.2.2.3. The equations for computing the corner nodes (i.e., for the function on Line 8) are shown in Appendix C.1.

Example 4.2. When running Algorithm 4.1 for the vertex conflict $\langle a_1, a_2, (3,2), 2\rangle$ shown in Figure 4.4, Lines 2 and 3 assign one singleton $S_1 = ((1,2),0)$, $S_2 = ((2,1),0)$, and $G_1 = ((6,4),7)$ to N_1^S, N_2^S, and N_1^G, respectively, and two singletons $G_2 = ((5,5),7)$ and $G_2' = ((3,3),3)$ to N_2^G.

Algorithm 4.1: Rectangle reasoning for path segments.

```
Input: Semi/non-cardinal vertex conflict $\langle a_1, a_2, v, t\rangle$ at a CT node $N$ with two MDDs
       $MDD_1$ and $MDD_2$
1  foreach $i = 1, 2$ do                                   // Collect start and target node candidates.
2      $N_i^S \leftarrow$ singletons in $MDD_i$ no later than timestep $t$;
3      $N_i^G \leftarrow$ singletons in $MDD_i$ no earlier than timestep $t$;
4  $type' \leftarrow$ Not-Rectangle;
5  $area' \leftarrow 0$;
6  foreach $S_1 \in N_1^S, S_2 \in N_2^S, G_1 \in N_1^G, G_2 \in N_2^G$ do                 // Try all combinations.
7      if isRectangle($S_1, S_2, G_1, G_2$) then
8          $\{R_1, R_2, R_s, R_g\} \leftarrow$ getIntersection($S_1, S_2, G_1, G_2$);
9          $type \leftarrow$ classifyRectangle($R_1, R_2, R_g, S_1, S_2, G_1, G_2$);
10         $area \leftarrow |R_1.x - R_2.x| \times |R_1.y - R_2.y|$;
11         if $type' = Not\text{-}Rectangle \vee type > type' \vee (type = type' \wedge area > area')$ then
12             $type', area', R'_1, R'_2, R'_s, R'_g \leftarrow type, area, R_1, R_2, R_s, R_g$;
13 if $type' \neq Not\text{-}Rectangle$ then
14     $B_1, B_2 \leftarrow$generateBarriers($MDD_1, MDD_2, R'_1, R'_2, R'_g$);
15     if $B_1$ blocks $N.plan[a_1] \wedge B_2$ blocks $N.plan[a_2]$ then return $B_1$ and $B_2$;
16 return Not-Rectangle;
```

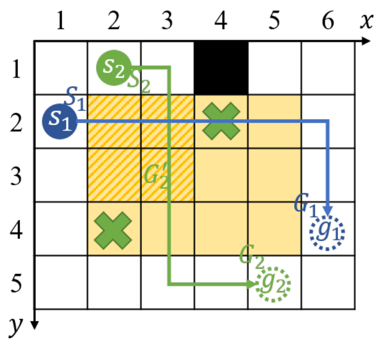


Figure 4.4: Example of deriving more than one rectangle conflict from a vertex conflict. Agent a_2 has two constraints that prohibit it from being at cells (2, 4) and (4, 2) at timestep 3. The two conflicting areas of the two rectangle conflicts are highlighted in yellow and in yellow with shadows, respectively.

Therefore, Lines 7-10 in Algorithm 4.1 find two rectangle conflicts, namely a cardinal rectangle conflict with the conflicting area highlighted in yellow (i.e., the rectangular area with corner cells (2, 2) and (5, 4)) and a semi-cardinal rectangle conflict with the conflicting area highlighted in yellow with shadows (i.e., the rectangular area with corner cells (2, 2) and (3, 3)). Line 11 in Algorithm 4.1 prefers the cardinal rectangle conflict. □

Example 4.3. Consider the MAPF instance shown in Figure 4.2b and assume that the paths for agent a_1 and a_2 are [(1, 2), (2, 2), (3, 2), (4, 2), (4, 3)] and [(2, 1), (3, 1), (4, 1), (4, 2), (5, 2), (5, 3), (5, 4)], respectively. Algorithm 4.1 (before Line 15) identifies the vertex conflict $\langle a_1, a_2, (4,2), 3 \rangle$ as a rectangle conflict with the conflicting area highlighted in yellow. The cells of the four corner nodes are shown in the figure. However, the resulting barrier constraint $B_2 = B(a_2, R_2, R_g)$ does not block the path of agent a_2. So, Algorithm 4.1 eventually discards this rectangle conflict. □

4.2.2.1 Identifying Rectangle Conflicts

We reuse the definition of rectangle conflicts from Definition 4.3 by replacing "paths" with "path segments". Since the start nodes of a rectangle conflict may not be at the same timestep (such as for the rectangle conflict in Figure 4.3b), the start and target nodes of a rectangle conflict have to satisfy not only Equations (4.1) to (4.4) but also

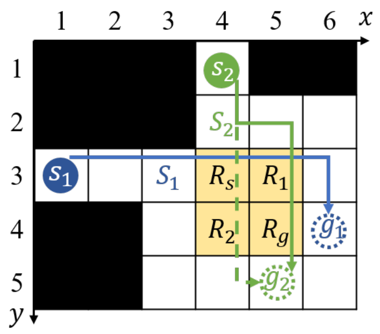
$$(S_1.x - S_2.x)(S_1.y - S_2.y)(S_1.x - G_1.x)(S_1.y - G_1.y) \leq 0. \tag{4.7}$$

This inequality guarantees that the start nodes are on different sides of the rectangle since, otherwise, adding barrier constraints might disallow a pair of paths that move both agents to the constrained border without waiting, such as in the example of Figure 4.3c.[3] We also require that $S_1 \neq S_2$ because, otherwise, the two agents have a cardinal vertex conflict at node S_1 (recall that S_1 and S_2 are singletons) that can be resolved with vertex constraints in a single branching step.

4.2.2.2 Resolving Rectangle Conflicts

When reasoning about entire paths, all paths of agent a_1 visit its start node S_1 as node S_1 is at its start vertex at timestep 0. However, when reasoning about path segments, only the shortest paths

[3]We do not check Equation (4.7) in Rectangle Reasoning Technique I because, when the start nodes are at the same timestep, situations like Figure 4.3c cannot occur.



(a) Two-agent MAPF instance, where agent a_2 follows the green solid arrow but waits at cell (4, 1) or cell (4, 2) for one timestep because of the constraints listed in (b).

Constraints:
$\langle a_2, (5,2), 2\rangle$,
$\langle a_2, (4,3), (5,3), 3\rangle$,
$\langle a_2, (4,3), (5,3), 4\rangle$,
$\langle a_2, (4,4), (5,4), 4\rangle$,
$\langle a_2, (4,4), (5,4), 5\rangle$,
$\langle a_2, (4,5), 4\rangle$,
$\langle a_2, (4,5), 5\rangle$.

(b) Constraints.

MDD_2:

$$\begin{tikzcd}
& ((4,1),0) \arrow[dl] \arrow[dr] & \\
((4,1),1) \arrow[dr] & & ((4,2),1) \arrow[dl] \\
& ((4,2),2) \arrow[d] & \\
& ((5,2),3) \arrow[d] & \\
& ((5,3),4) \arrow[d] & \\
& ((5,4),5) \arrow[d] & \\
& ((5,5),6) &
\end{tikzcd}$$

(c) Corresponding MDD for agent a_2.

Figure 4.5: Example where we cannot apply the original barrier constraints.

of agent a_1 are guaranteed to visit node S_1 as node S_1 is a singleton. Its non-shortest paths do not necessarily visit node S_1. In this case, using barrier constraints may block pairs of conflict-free paths and thus lose the completeness guarantee.

Example 4.4. Figure 4.5 provides a counterexample, where a CT node N has the set of constraints listed in Figure 4.5b. The constraints force agent a_2 to wait for at least one timestep before reaching its target vertex. It can either wait before entering the conflicting area, which leads to a conflict with agent a_1, or enter the conflicting area without waiting and wait later, which might avoid conflicts with agent a_1. However, all shortest paths (of length 6) of agent a_2 that satisfy the constraints in CT node N have to wait for one timestep before entering the conflicting area, see MDD_2 shown in Figure 4.5c. Therefore, node $S_2 = ((4,2),2)$ is a singleton, and agents a_1 and a_2 have a cardinal rectangle conflict. If this conflict is resolved using barrier constraints, then the pair of conflict-free paths where agent a_1 directly follows the blue arrow (which visits node $((5,3),4)$ constrained by $B(a_1, R_1, R_g)$) and agent a_2 follows the green dashed arrow but waits at cell $(4,4)$ for two timesteps (which visits node $((4,4),4)$ constrained by $B(a_2, R_2, R_g)$) satisfies neither of the child CT nodes of CT node N. Barrier constraints fail here because the constrained node $((4,4),4)$ is not in MDD_2, and thus agent a_2 could have a longer path that does not visit node S_2 but visits node $((4,4),4)$. □

Therefore, we redefine barrier constraints by considering only the border nodes that are in the MDD of the agent. That is, $B(a_i, R_i, R_g) = \{\langle a_i, (x,y), t\rangle \mid ((x,y),t) \in R_iR_g \cap MDD_i\}$ for $i = 1,2$.[4] When resolving a rectangle conflict for path segments, we generate two child CT nodes and add $B(a_1, R_1, R_g)$ to one of them and $B(a_2, R_2, R_g)$ to the other one.

4.2.2.3 Classifying Rectangle Conflicts

We reuse the method in Section 4.2.1.3 to classify rectangle conflicts.

4.2.2.4 Theoretical Analysis

We first present a property of MDDs.

Property 4.4. *Given a MDD MDD_i for agent a_i and a MDD node $(v,t) \in MDD_i$, for any path p for agent a_i that visits node (v,t), all nodes that path p visits before timestep t are also in MDD_i.*

Proof. Since $(v,t) \in MDD_i$, there exists a sub-path p' that moves agent a_i from node (v,t) to node $(g_i, length(N.plan[a_i]))$. So, a path that follows first the prefix of path p from node $(s_i, 0)$ to node (v,t) and then sub-path p' to node $(g_i, length(N.plan[a_i]))$ is a path for agent a_i of length $length(N.plan[a_i])$. That is, it is a shortest path for agent a_i, which indicates that all nodes visited by this path (including all nodes visited by path p before timestep t) are in MDD_i. □

We next present three properties of barrier constraints.

Property 4.5. *If Rectangle Reasoning Technique II finds a rectangle conflict between agents a_1 and a_2, then any path of agent a_i for $i = 1,2$ that visits a node constrained by $B(a_i, R_i, R_g)$ also visits its start node S_i.*

Proof. Let (v,t) be a node constrained by $B(a_i, R_i, R_g)$. By definition, $(v,t) \in MDD_i$. Since node S_i is a singleton of MDD_i and the timestep of S_i is no larger than t (due to Line 2 in Algorithm 4.1), from Property 4.4, any path of agent a_i that visits node (v,t) also visits its start node S_i. □

[4] In our implementation, a barrier constraint is encoded as a set of vertex constraints.

Property 4.6. *If Rectangle Reasoning Technique II identifies a rectangle conflict between agents a_1 and a_2, then any path of agent a_1 that visits a node constrained by $B(a_1, R_1, R_g)$ also visits a node on its entry border R_sR_2, and any path for agent a_2 that also visits a node constrained by $B(a_2, R_2, R_g)$ visits a node on its entry border R_sR_1.* □

Property 4.6 is straightforward to prove by reusing the proof of Property 4.2. Thus, we provide the proof only in Appendix C.2.

Property 4.7. *For all combinations of paths of agents a_1 and a_2 with a rectangle conflict found by Rectangle Reasoning Technique II, if one path violates $B(a_1, R_1, R_g)$ and the other path violates $B(a_2, R_2, R_g)$, then the two paths have one or more vertex conflicts within the rectangle.*

Proof. We apply the proof of Property 4.3 here by replacing Property 4.2 with Property 4.6. □

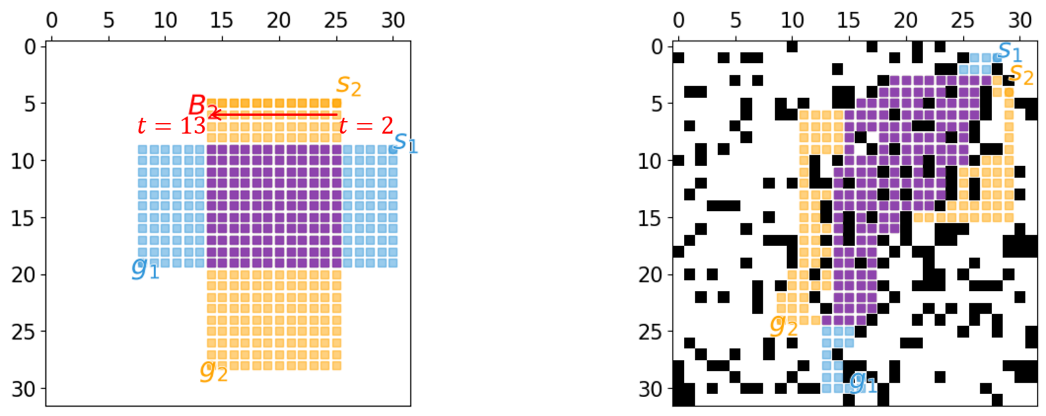
Property 4.7 implies that barrier constraints $B(a_1, R_1, R_g)$ and $B(a_2, R_2, R_g)$ are mutually disjunctive, and thus, based on Theorem 4.1 and the fact that $B(a_i, R_i, R_g)$ for $i = 1, 2$ blocks the current path for agent a_i, using them to split a CT node preserves the completeness and optimality of CBS.

Theorem 4.3. *Using Rectangle Reasoning Technique II preserves the completeness and optimality of CBS.* □

4.3 Generalized Rectangle Symmetry

Let us first look at two examples.

Example 4.5. Figure 4.6a shows a MAPF sub-instance with two agents on a 32×32 empty map with $dist(s_1, g_1) = 32$ and $dist(s_2, g_2) = 34$. Agent a_1 has no constraints, and thus all its shortest paths are Manhattan-optimal and of length 32. Agent a_2 has a barrier constraint B_2 that forces it to first take a wait action at one of the cells in the top yellow row and then follow a Manhattan-optimal path to its target vertex. Its shortest paths are thus of length 35. Due to this wait action, both agents



(a) Rectangular-shaped cardinal rectangle conflict. (b) Non-rectangular-shaped cardinal rectangle conflict.

Figure 4.6: Examples where the reasoning techniques in Section 4.2 fail to identify rectangle conflicts, reproduced from the MAPF benchmark [163]. The start and target vertices of the agents are shown in the figures. In (a), agent a_2 has a barrier constraint B_2 indicated by the red arrow (the timesteps of the leftmost and rightmost nodes blocked by B_2 are also shown in the figure), which forces agent a_2 to wait for one timestep. In both (a) and (b), the vertices of the MDD nodes of the MDDs of the two agents are highlighted in the corresponding colors. Purple cells represent the overlapping area. The timesteps when the agents reach every purple cell are the same for both agents.

reach every purple cell at the same timestep and thus have a conflict if they both visit the same purple cell following their shortest paths. Since the two agents need to cross each other to reach their target vertices, there is no way for them to reach their target vertices without visiting some common purple cell via their shortest paths. Therefore, the optimal resolution is for either agent a_1 to wait for one timestep (resulting in a path of length 33) or agent a_2 to wait for two timesteps or take a detour (resulting in a path of length 36).

This looks like a cardinal rectangle conflict described in Section 4.2. However, the only two singletons in MDD_2 are $(s_2, 0)$ and $(g_2, 35)$, which do not satisfy Equations (4.1) and (4.2) since no path of agent a_2 is Manhattan-optimal due to its wait action. Therefore, the rectangle reasoning techniques in Section 4.2 fail to identify it as a rectangle conflict, and, as a result, CBS needs to spend exponential time on solving it. □

Example 4.6. Figure 4.6b shows a 2-agent MAPF instance on a 32×32 map with randomly blocked cells. Both agents reach every purple cell at the same timestep if they follow their shortest paths and need to cross each other to reach their target vertices (or, formally, the line segments between their start and target vertices need to cross each other). Therefore, the optimal resolution is for one of the agents to wait for one timestep.

However, the rectangle reasoning techniques in Section 4.2 fail to identify this as a rectangle conflict because they cannot find a pair of singletons around the purple area of agent a_2 whose corresponding sub-path is Manhattan-optimal. The conflicting area here is not rectangular. □

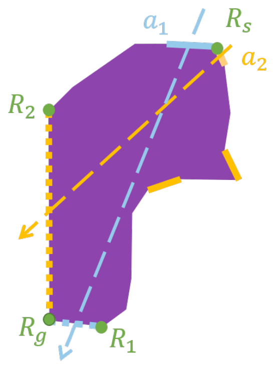
Example 4.5 behaves like a cardinal rectangle conflict, but there do not exist any appropriate singletons. Example 4.6 behaves like a cardinal rectangle conflict as well, but the conflicting area is not rectangular. They motivate us to define a more general cardinal rectangle conflict between two agents. These cardinal generalized rectangle conflicts have the following properties: (1) There is a purple area that both agents reach at the same timestep if they follow their shortest paths, and (2) the two agents have to cross each other inside the purple area. We further generalize the idea to generalized rectangle conflicts which also include semi- and non-cardinal generalized rectangle conflicts.

In Section 4.3.1, we present the high-level idea behind our generalized rectangle reasoning technique. Then, in Section 4.3.2, we present the algorithm in detail. We provide a proof sketch of the soundness of the proposed technique in Section 4.3.3 and formal proof in Appendix D. We empirically evaluate our generalized rectangle reasoning technique in Section 4.3.4 together with our rectangle reasoning techniques introduced in Section 4.2.

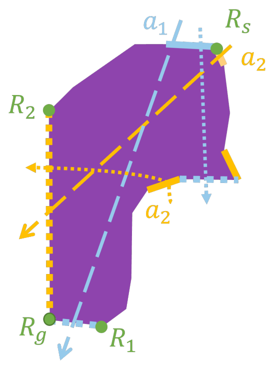
The generalized rectangle reasoning technique can be applied not only to four-neighbor grids but also other planar graphs, which covers most ways of representing 2D (or even 2.5D) environments for MAPF.

4.3.1 High-Level Idea

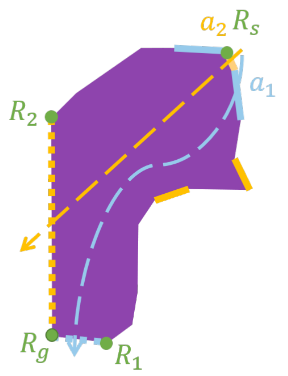
Consider the conflict in Figure 4.6b. Figure 4.7a shows an abstract illustration of it. Agent a_1 enters the purple area from (one of) the solid blue lines and leaves it from (one of) the dotted blue lines. Similarly, agent a_2 enters the purple area from one of the solid yellow lines and leaves it from one of the dotted yellow lines. If we scan the border of the purple area anticlockwise, we



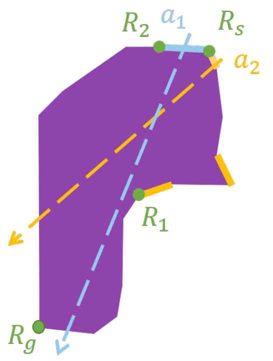
(a) Cardinal generalized rectangle conflict.



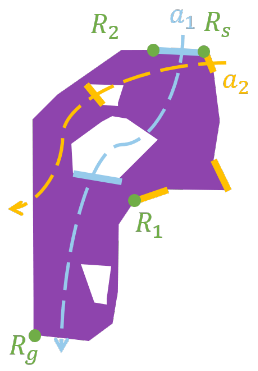
(b) Semi-cardinal generalized rectangle conflict.



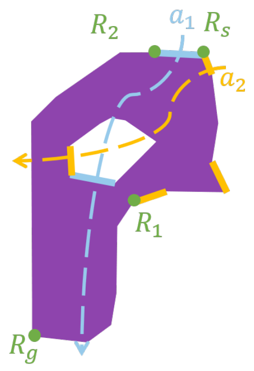
(c) Not generalized rectangle conflict.



(d) Generalized rectangle conflict.



(e) Generalized rectangle conflict with holes.



(f) No generalized rectangle conflict.

Figure 4.7: Illustrations of generalized rectangle conflicts. The purple area represents the conflicting area inside which both agents reach each vertex at the same timestep via their shortest paths. The solid lines represent where the agents enter the purple area via their shortest paths and the dotted lines represent where they leave the purple area via their shortest paths. The positions of R_1 and R_2 in Figures (a-c) are for illustration purposes and misleading. Figures (d-f) show their correct positions.

find a pattern of "solid blue lines $\rightarrow$ dotted yellow lines $\rightarrow$ dotted blue lines $\rightarrow$ solid yellow lines". So, from geometry, any line that connects a point on one of the solid blue lines with a point on one of the dotted blue lines without going outside the purple area must intersect with any line that connects a point on one of the solid yellow lines with a point on one of the dotted yellow lines without going outside the purple area. If the two agents follow such two lines, they must have a vertex conflict. Therefore, any path of agent a_1 that visits (one of) the dotted blue lines must conflict with any path of agent a_2 that visits (one of) the dotted yellow lines.

Following the idea in Section 4.2, we generate two barrier constraints $B(a_1,R_1,R_g)$ and $B(a_2,R_2,R_g)$, where the vertices of R_1, R_2 and R_g are marked in Figure 4.7a, and $B(a_i,R_i,R_g)$ for $i = 1,2$ is a set of vertex constraints that prohibits agent a_i from occupying all vertices along the border from R_i to R_g at the timestep when a_i would optimally reach the vertex. This pair of barrier constraints gives one of the agents priority within the purple area over the other agent and forces the other agent to leave it later or take a detour.

Figure 4.7b shows a slightly different example where agent a_1 can leave the purple area also from the dotted blue line on the right. Therefore, the two agents can traverse the purple area without conflicts, for instance, by following the dotted arrows. However, just like for Example 4.1, CBS may not find such a pair of conflict-free paths efficiently. And, in fact, this example is a semi-cardinal generalized rectangle conflict as we can use barrier constraints $B(a_1,R_1,R_g)$ and $B(a_2,R_2,R_g)$ to resolve it. This is so because, for the child CT node with constraint $B(a_1,R_1,R_g)$, CBS can find a path for agent a_1 of the same length, such as the path indicated by the dotted blue line, while, for the other child CT node, all shortest paths are blocked by $B(a_2,R_2,R_g)$, and thus CBS has to find a longer path for agent a_2.

Figure 4.7c shows an example where agent a_1 can enter the purple area also from the solid blue line on the right. However, this time, we cannot use barrier constraints $B(a_1,R_1,R_g)$ and $B(a_2,R_2,R_g)$ because there is a pair of conflict-free paths that violates both barrier constraints, indicated by the two arrows in the figure. Therefore, this example is no generalized rectangle conflict.

To sum up, how the solid lines of different colors distribute determines whether the conflict is a generalized rectangle conflict, and how the dotted lines of different colors distribute only affects the cardinality of the conflict. Therefore, when we identify generalized rectangle conflicts, we only focus on the solid lines, see Figure 4.7d. We denote the nodes on the border with the smallest and largest timesteps as R_s and R_g, respectively. Nodes R_s and R_g divide the border into two segments. If all solid blue lines belong only to one of the segments and all solid yellow lines belong only to the other segment, then the conflict is a generalized rectangle conflict. We denote the node on the solid blue and yellow lines that are furthest from node R_s (i.e., closest to node R_g) as R_2 and R_1, respectively. Then, we can prove that using the barrier constraints $B(a_1, R_1, R_g)$ and $B(a_2, R_2, R_g)$ to resolve this conflict preserves the completeness and optimality of CBS.

Now, let us consider the case when the purple area has holes. The holes can be caused by, for example, constraints. The key point is to exclude the cases when the lines can cross each other within the hole because the agents might cross the intersection point in the hole at different timesteps and thus have conflict-free paths. Therefore, we also draw blue and solid yellow lines on the border of each hole to indicate where the agents can enter the purple area from the hole. If every hole inside the purple area has solid lines of at most one color, such as in Figure 4.7e, then this is still a generalized rectangle conflict. Otherwise, as in Figure 4.7f, such a conflict is not a generalized rectangle conflict.

As for classifying conflicts, we simply check whether barrier constraint $B(a_i, R_i, R_g)$ for $i = 1, 2$ blocks all shortest paths of agent a_i by looking at MDD_i. The conflict is cardinal iff both barrier constraints block all shortest paths; it is semi-cardinal iff only one of them blocks all shortest paths; it is non-cardinal iff neither of them blocks all shortest paths.

4.3.2 Algorithm

We now provide a methodology for identifying, classifying, and resolving generalized rectangle conflicts. There are five key steps:

1. finding the generalized rectangle (i.e., the purple area in Figure 4.7),
2. scanning the border,
3. checking the holes,
4. generating the barrier constraints, and
5. classifying the conflict,

which correspond to the following five subsections, respectively. Given a semi- or non-cardinal vertex conflict between two agents, the generalized rectangle reasoning algorithm returns either a pair of barrier constraints or "Not-Rectangle".

4.3.2.1 Step 1: Finding the Generalized Rectangle

Definition 4.5 (Generalized Rectangle). *Given a vertex conflict $\langle a_1, a_2, v, t\rangle$, the* generalized rectangle *is a connected directed acyclic graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ such that*

1. *$\mathscr{G} \subseteq MDD_1 \cap MDD_2$,*
2. *$(v,t) \in \mathscr{V}$, and*
3. *for every node $(u,t_u) \in \mathscr{V}$, any shortest path of either agent that visits vertex u visits it only at timestep t_u.*

We use the term conflicting area *to denote the vertices (e.g., cells in four-neighbor grids) of the nodes in $\mathscr{V}$, which represent a connected area on the plane to which graph G is mapped.*

Condition 3 is important because it ensures that, if the shortest paths of agents a_1 and a_2 visit a common vertex in the conflicting area, they must have a vertex conflict. From Conditions 1 and 3, we know that $(u,t_u) \in \mathscr{V}$ only if $\forall i \in \{1,2\}\, \forall t'_u \in [0,t_u) \cup [t_u+1,+\infty)\, (u,t_u) \in MDD_i \wedge (u,t'_u) \notin MDD_i$.

Formally, to find a generalized rectangle, we first project the MDD nodes of the MDDs of both agents to the vertices in V. Let M_i for $i = 1,2$ be such a mapping, where $M_i[u]$ for $u \in V$ is a list of MDD nodes in MDD_i whose vertices are u. Then, we run a search starting from the conflicting vertex v to generate $\mathscr{G}$ whose nodes (u,t_u) are always the only nodes in both $M_1[u]$ and $M_2[u]$. If $\mathscr{V}$ is empty or contains only one node (which implies a cardinal vertex conflict), we terminate and report "Not-Rectangle".

During the search, we also collect the entry edges E_1 and E_2 for the conflicting area (corresponding to the solid blue and yellow lines in Figure 4.7).

Definition 4.6 (Entry Edge)**.** *The set of* entry edges E_i *for* $i = 1,2$ *is a set of directed MDD edges of* MDD_i *whose "from" node is not in* $\mathcal{V}$ *and whose "to" node is in* $\mathcal{V}$*.*

Since the start nodes $(s_1, 0)$ and $(s_2, 0)$ of agents a_1 and a_2 are different, they must be located outside of the conflicting area, and, thus, both E_1 and E_2 contain at least one entry edge.

4.3.2.2 Step 2: Scanning the Border

Let R_s and R_g denote the nodes with the smallest and largest timesteps on the border, respectively. Scan the border from node R_s to node R_g on both sides and check whether the entry edges of one agent are all on one side of R_sR_g and the entry edges of the other agent are all on the other side of R_sR_g. If not, we terminate and report "Not-Rectangle".

Recall that the underlying graph is a planar graph. So, we embed the graph into the plane and then scan the border clockwise and counterclockwise from R_s to R_g. During the scanning, we mark the "to" nodes of the last-seen entry edges of E_1 and E_2 as R_2 and R_1, respectively. We also remove every visited entry edge from E_1 or E_2 so that all remaining edges in E_1 and E_2 are entry edges on the borders of the holes, which will be used in the next step. For clarification, we use E_i^b to denote the removed edges from E_i, i.e., entry edges on the outer border of the conflicting area and $E_i^h = E_i \setminus E_i^b$ to denote the remaining edges in E_i for $i = 1,2$, i.e., entry edges for the holes.

4.3.2.3 Step 3: Checking the Holes

For each entry edge in E_1^h, we scan the border of its corresponding hole and check whether the "to" node of any edge in E_2^h is on the border. If so, then this hole contains entry edges of both agents, so we terminate and report "Not-Rectangle". If we succeed in examining every edge in E_1^h without terminating, then there is no hole in the conflicting area that contains an entry edge of both agents. We thus move to the next step.

4.3.2.4 Step 4: Generating the Barrier Constraints

We generate barrier constraints $B(a_1,R_1,R_g)$ and $B(a_2,R_2,R_g)$, where $B(a_i,R_i,R_g)$ for $i = 1,2$ is a set of vertex constraints that prohibits agent a_i from occupying all nodes along the border from R_i to R_g. All prohibited nodes are in the MDDs of the agents, so we do not need to worry about situations where, like in Example 4.4, the two agents might have conflict-free paths that visit the prohibited nodes. As on Line 15 in Algorithm 4.1, we check whether the generated barrier constraints block the current paths of both agents. If not, we terminate and report "Not-Rectangle".

4.3.2.5 Step 5: Classifying the Conflict

From Figures 4.7a and 4.7b, it seems that we can classify conflicts by checking whether the border segment R_iR_g covers all dotted lines of the color corresponding to agent a_i. However, this would be incorrect because the agent might have a shortest path that does not visit the purple area. Therefore, we run a search on the MDD of each agent and check whether the barrier constraint $B(a_i,R_i,R_g)$ for $i = 1,2$ blocks all paths in MDD_i from its start node to its target node, i.e., the nodes constrained by the barrier constraint form a cut of the MDD. The generalized rectangle conflict is cardinal iff both barrier constraints block all paths in the corresponding MDD; it is semi-cardinal iff only one of the barrier constraints blocks all paths in the corresponding MDD; and it is non-cardinal iff neither barrier constraint blocks all paths in the corresponding MDD.

4.3.3 Theoretical Analysis

Property 4.8. *For all combinations of paths of agents a_1 and a_2 with a generalized rectangle conflict, if one path violates $B(a_1,R_1,R_g)$ and the other path violates $B(a_2,R_2,R_g)$, then the two paths have one or more vertex conflicts within the generalized rectangle.*

Proof Sketch. We provide a proof sketch here and a formal proof in Appendix D:

1. All paths of agent a_i for $i = 1,2$ that visit a node constrained by $B(a_i,R_i,R_g)$ must traverse an entry edge in E_i^b.

Table 4.1: Benchmark details. We use 8 maps, each with 6 different numbers of agents. We have 25 instances for each map and each number of agents, yielding $8 \times 6 \times 25 = 1,200$ instances in total.

Map	Map name	Map size	#Empty cells	#Agents
`Random`	`random-32-32-20`	32×32	819	20, 30, ..., 70
`Empty`	`empty-32-32`	32×32	1,024	30, 50, ..., 130
`Warehouse`	`warehouse-10-20-10-2-1`	161×63	5,699	30, 50, ..., 130
`Game1`	`den520d`	256×257	28,178	40, 60, ..., 140
`Room`	`room-64-64-8`	64×64	3,232	15, 20, ..., 40
`Maze`	`maze-128-128-1`	128×128	8,191	3, 6, ..., 18
`City`	`Paris_1_256`	256×256	47,240	30, 60, ..., 180
`Game2`	`brc202d`	530×481	43,151	20, 30, ..., 70

2. Any sub-path from an entry edge in E_1^b to a node constrained by $B(a_1, R_1, R_g)$ must visit at least one common vertex with any sub-path from an entry edge in E_2^b to a node constrained by $B(a_2, R_2, R_g)$.

3. The common vertex must be inside the conflicting area, i.e., not inside one of the holes.

4. Following the two sub-paths, agents a_1 and a_2 must conflict at the common vertex in the conflicting area. $\square$

Property 4.8 implies that barrier constraints $B(a_1, R_1, R_g)$ and $B(a_2, R_2, R_g)$ are mutually disjunctive, and thus, based on Theorem 4.1 and the fact that $B(a_i, R_i, R_g)$ for $i = 1, 2$ blocks the current path for agent a_i, using them to split a CT node preserves the completeness and optimality of CBS.

Theorem 4.4. *Using the generalized rectangle reasoning technique preserves the completeness and optimality of CBS.* $\square$

4.3.4 Empirical Evaluation

In this and future subsections of this chapter, we evaluate the algorithms on eight maps of different sizes and structures from the MAPF benchmark suite [163]. We test six different numbers of agents per map. We use the "random" scenarios from the benchmark suite, yielding 25 instances for each

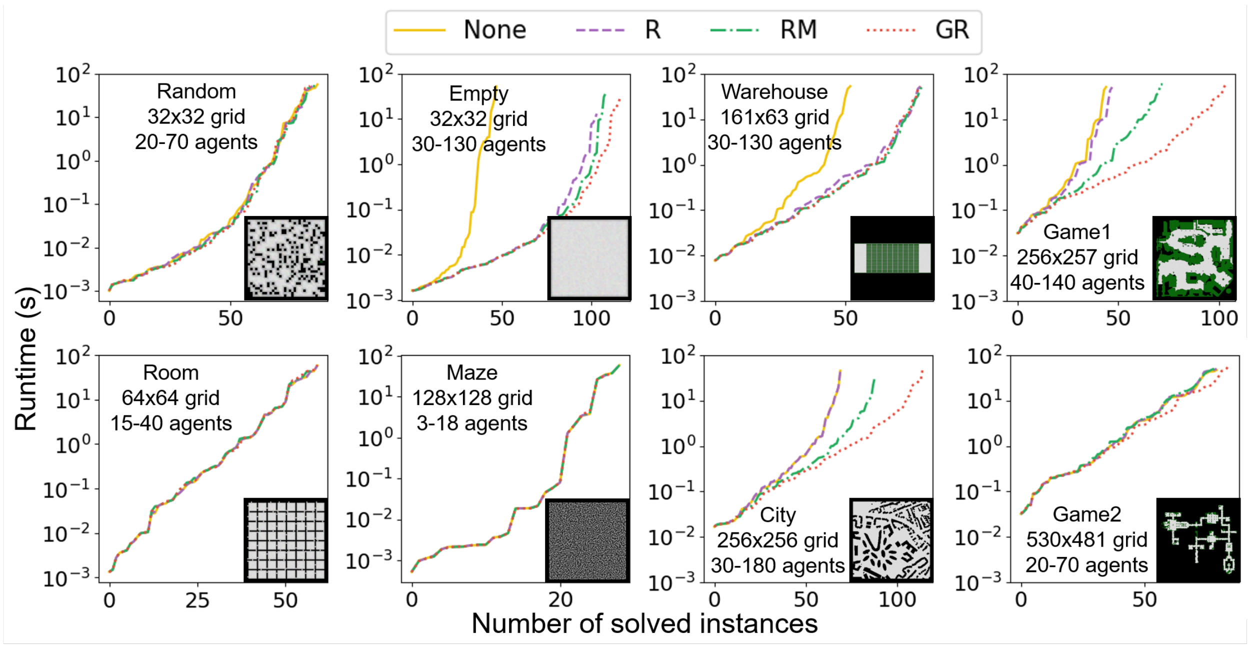


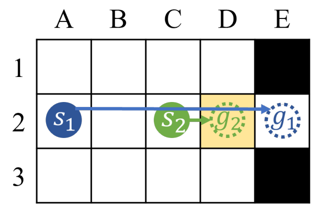
Figure 4.8: Runtime distributions of CBSH with different rectangle reasoning techniques. A point (x, y) in the figure indicates that there are x instances solved within y seconds.

map and each number of agents. Details of the benchmark instances are shown in Table 4.1, and a visualization of the maps is shown in Figure 4.8. The experiments are conducted on Ubuntu 20.04 LTS on an Intel Xeon 8260 CPU with a memory limit of 16 GB and a time limit of 1 minute.

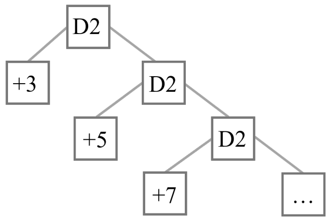
In this subsection, we compare CBSH (denoted **None**), CBSH with rectangle reasoning for entire paths (denoted **R**), CBSH with rectangle reasoning for path segments (denoted **RM**), and CBSH with generalized rectangle reasoning (denoted **GR**).[5] The results are reported in Figure 4.8.

As expected, the improvements due to our rectangle reasoning techniques depend on the structure of the maps. On maps with little open space, such as `Random`, `Room`, `Maze`, and `Game2`, rectangle reasoning techniques do not improve the performance. But fortunately, due to their small runtime overhead, they do not deteriorate the performance either. On the other maps with large open spaces, some or even all of the rectangle reasoning techniques speed up CBSH, and GR is always the best. Specifically, on map `Empty`, the shortest path of an agent (ignoring other agents)

[5]We demonstrate the symmetry reasoning techniques on top of CBSH instead of CBSH2 here (and in the following several subsections) because, for CBSH2, the symmetry reasoning techniques can be applied to both the main CBSH2 and the two-agent sub-MAPF solver CBSH, which makes it non-trivial to analyze the effectiveness of the techniques. Nevertheless, after we have presented all symmetry-reasoning techniques, we will show their effectiveness on top of CBSH2 in Section 4.8.



(a) Two-agent MAPF instance with a target conflict. Agent a_2 arrives at vertex D2 at timestep 1. Two timesteps later, agent a_1 visits the same vertex, leading to a vertex conflict at vertex D2 at timestep 3.



(b) CT generated by CBS when solving the two-agent MAPF instance in (a). Each left branch constrains agent a_2, and each right branch constrains agent a_1.

Figure 4.9: Example of a target conflict and the corresponding CT generated by CBS. In (b), each non-leaf CT node is marked with the vertex of the chosen conflict. The leaf CT node marked "+3" contains an optimal solution, whose sum of costs is the cost of the root CT node plus 3. The leaf CT nodes marked "+5" and "+7" contain suboptimal solutions, whose sums of costs are the cost of the root CT node plus 5 and 7, respectively. The leaf CT node marked "..." contains a plan with conflicts, whose sum of costs is the cost of the root CT node plus 3, and produces suboptimal solutions in its descendant CT nodes.

is always Manhattan-optimal, so R significantly speeds up CBSH, while RM and GR further speed it up, but only by a little bit. The performance on map `Warehouse` is similar, as the obstacles on this map are all of rectangular shape. Maps `Game1` and `City`, however, contain obstacles of various shapes, so the shortest path of an agent is not necessarily Manhattan-optimal, and the conflicting area is not necessarily of rectangular shape. Thus, R performs similarly to None, but GR significantly outperforms RM, which in turn significantly outperforms R.

4.4 Target Symmetry

A target symmetry occurs when one agent visits the target vertex of a second agent after the second agent has already arrived at it and stays there forever. We refer to the corresponding conflict as a *target conflict*.

Definition 4.7 (Target Conflict)**.** *Two agents are involved in a* target conflict *iff they have a vertex conflict that happens after one agent has arrived at its target vertex and stays there forever.*

Example 4.7. In Figure 4.9a, agent a_2 arrives at its target vertex D2 at timestep 1, but then a vertex conflict occurs with agent a_1 at vertex D2 at timestep 3. When CBS resolves this vertex conflict,

Table 4.2: Numbers of expanded CT nodes to resolve the target conflict shown in Figure 4.9a for different distances between vertices s_1 and g_2.

$dist(s_1, g_2)$	10	20	30	40	50
Number of expanded CT nodes for two-agent instances	10	20	30	40	50
Number of expanded CT nodes for four-agent instances	50	150	300	500	750

it generates two child CT nodes, as shown in Figure 4.9b. In the left child CT node, CBS adds a vertex constraint that prohibits agent a_2 from being at vertex D2 at timestep 3. The low-level search finds a new path [C2, C3, C3, C2, D2] for agent a_2, which does not conflict with agent a_1. The cost of this CT node is three larger than the cost of the root CT node. In the right child CT node, CBS adds a vertex constraint that prohibits agent a_1 from being at vertex D2 at timestep 3. Thus, agent a_1 arrives at vertex D2 at timestep 4, and the cost of this CT node is one larger than the cost of the root CT node. There are several alternative paths for agent a_1 where it waits at different vertices for the requisite timestep, e.g., path [A2, A2, B2, C2, D2, E2]. However, each such path produces a conflict with agent a_2 at vertex D2 at timestep 4. Although the left child CT node contains conflict-free paths, CBS has to split the right child CT nodes repeatedly to constrain agent a_1 (because it performs a best-first search) before eventually proving that the solution of the left child CT node is optimal. □

Target symmetry has the same pernicious characteristics as rectangle symmetry since, if undetected, it can explode the size of the CT and lead to unacceptable runtimes. Table 4.2 shows how many CT nodes CBS expands to resolve a target conflict of the type shown in Figure 4.9a for different distances between vertices s_1 and g_2. While the increase in CT nodes is linear in the distance, which may seem not too problematic, only one of the leaf CT nodes contains the optimal solution for the two agents. Later, when other conflicts occur, each leaf CT node might be further fruitlessly expanded. With two copies of the problem (resulting in four-agent instances), Table 4.2 shows already a quadratic increase in the number of CT nodes. For m-agent instances, the increases become exponential in m. Hence, we propose a target reasoning technique that efficiently detects and resolves all target symmetries. We introduce this technique in detail in the following four subsections and present its empirical performance in Section 4.4.5. Our target reasoning technique works for general graphs.

4.4.1 Identifying Target Conflicts

The detection of target conflicts is straightforward. For every vertex conflict, we compare the conflicting timestep with the path lengths of the agents and regard it as a target conflict iff the conflicting timestep is no smaller than the path length of one of the agents.

4.4.2 Resolving Target Conflicts

The key to resolving target conflicts is to reason about the path length of an agent. Suppose that agent a_1 visits the target vertex g_2 of agent a_2 at timestep t after agent a_2 has completed its path, i.e., $t \geq \mathit{length}(N.plan[a_2])$. We resolve this target conflict by branching on the path length l_2 of agent a_2 using the following two *length constraints*, one for each child CT node:

- $l_2 > t$, i.e., agent a_2 can complete its path only after timestep t, or
- $l_2 \leq t$, i.e., agent a_2 must arrive at vertex g_2 and stay there forever before or at timestep t, which also requires that any other agent cannot visit vertex g_2 at or after timestep t.

The first constraint $l_2 > t$ affects only the path of agent a_2, while the second constraint $l_2 \leq t$ could affect the paths of all agents.

The advantage of this branching method is immediate. In the first case, agent a_2 cannot finish before timestep $t+1$, so its path length increases from its current value $\mathit{length}(N.plan[a_2])$ to at least $t+1$. In the second case, agent a_1 is prohibited from being at vertex g_2 at or after timestep t. If agent a_1 has no alternative paths to its target vertex that do not use vertex g_2 at or after timestep t, then the CT node with this constraint has no solution and is thus pruned. If agent a_1 has alternative paths and the shortest one among them is longer than its current path, then its path length increases. We do not need to replan for agent a_2 since its current path is no longer than t. Nevertheless, we have to replan the paths for all other agents that visit vertex g_2 at or after timestep t. This is a very strong constraint as vertex g_2 can be viewed as an obstacle after timestep t for all agents except for agent a_2.

In order to handle the length constraints, we need the low-level search to take into account given bounds on the path length. This is fairly straightforward for given bounds $e \leq l_2 \leq u$ on the path length l_2 of agent a_2: If the low-level search reaches target vertex g_2 before timestep e, then it cannot terminate but must continue searching; if it reaches the target vertex between timesteps e and u (and the agent was not at the target vertex at the previous timestep), then it terminates and returns the corresponding path; if it reaches the target vertex after timestep u, then it terminates and prunes the corresponding CT node since the CT node has no solution. We require the agent to not be at the target vertex at the previous timestep because, otherwise, the agent could simply take its current path to the target vertex and wait there until timestep e is reached, which does not help to resolve the conflict.

For example, to resolve the target conflict in Figure 4.9a, we split the root CT node and add the length constraints $l_2 > 3$ and $l_2 \leq 3$. In the left child CT node, we replan the path of agent a_2 and find a new path [C2, C3, C3, C2, D2], which does not conflict with the path of agent a_1. In the right child CT node, agent a_1 cannot visit vertex D2 at or after timestep 3. We thus fail to find a path for it and prune the right child CT node. Therefore, the target symmetry is resolved in a single branching step.

4.4.3 Classifying Target Conflicts

Target conflicts are classified based on the vertex conflict at the target vertex: A target conflict is cardinal iff the corresponding vertex conflict is cardinal, and it is semi-cardinal iff the corresponding vertex conflict is semi-cardinal. It can never be non-cardinal because the cost of the child CT node with the additional length constraint $l_2 > t$ is always larger than the cost of the parent CT node. This is an approximate way of classifying target conflicts since the costs of both child CT nodes may increase when we branch on a semi-cardinal target conflict.

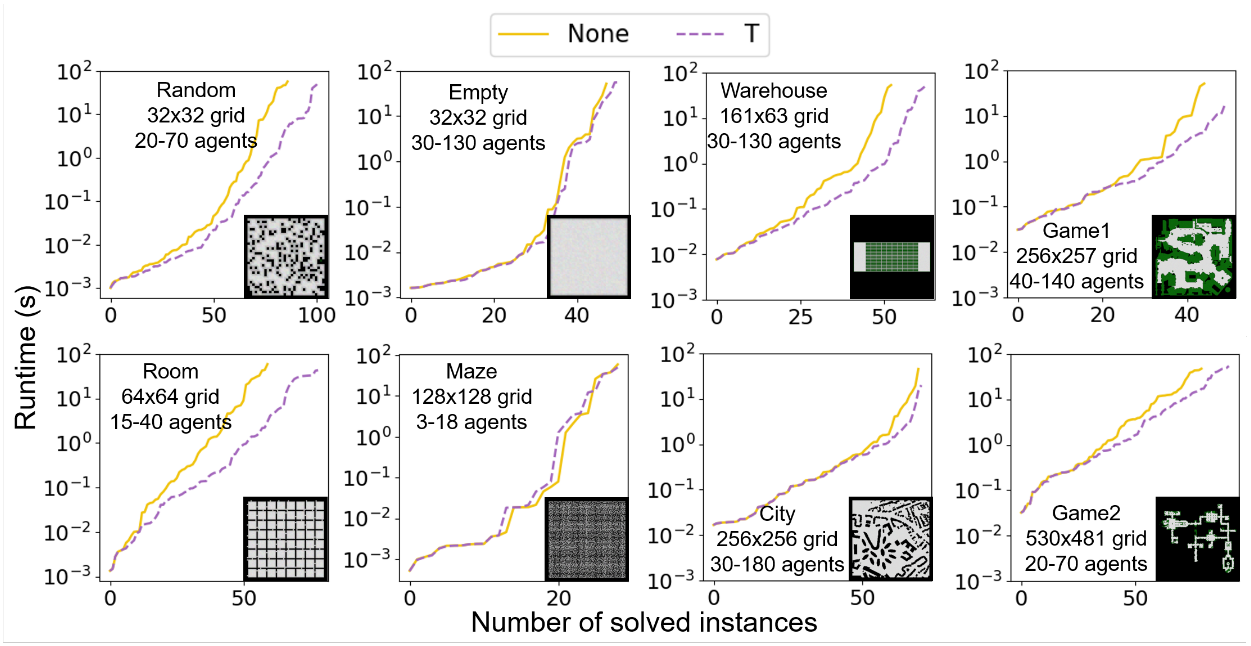


Figure 4.10: Runtime distributions of CBSH with and without target reasoning.

4.4.4 Theoretical Analysis

Proving the completeness and optimality of CBS when using length constraints for target conflicts is straightforward. Therefore, we omit the proof of the following theorem.

Theorem 4.5. *Resolving target conflicts with length constraints preserves the completeness and optimality of CBS.* □

4.4.5 Empirical Evaluation

In this subsection, we compare CBSH (denoted **None**) with CBSH with target reasoning (denoted **T**). Since we might need to replan paths for more than one agent for adding a length constraint $l_2 \leq t$, we cannot use the incremental method described in Section 3.1.1 for solving the Minimum Vertex Cover (MVC) problem for computing the CG heuristic. We thus use the following method instead. We partition G_D into its connected components and calculate the MVC for each component with a branch-and-bound algorithm that enumerates the possible vertex cover sets and prunes nodes using the best result so far. The MVC of G_D is the union of the MVCs of all components.

As shown in Figure 4.10, on all maps except for `Maze`, target reasoning speeds up CBSH, and the improvement is usually larger on maps with more obstacles. The performance on `Maze` is an exception due to the long runtime of the low-level space-time A* search when replanning results in extremely long or non-existing paths. On the one hand, the length constraint $l_i > t$ is powerful since it can substantially increase the path length of agent a_i. However, finding a long path is time-consuming for space-time A*. On the other hand, the length constraint $l_i \leq t$ is powerful since it prohibits all agents other than agent a_i from being at vertex g_i for all timesteps at and after timestep t. However, this might make it impossible for some agents to reach their target vertices. To realize that such a path does not exist, space-time A* has to enumerate all reachable pairs of a vertex and timestep, which is again time-consuming.

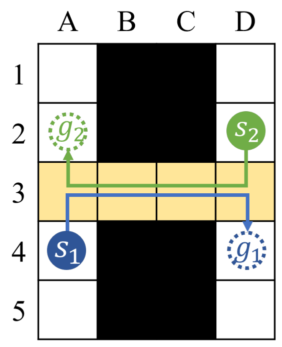
4.5 Corridor Symmetry

Definition 4.8 (Corridor)**.** *A* corridor $C = C_0 \cup \{e_1, e_2\}$ *of graph* $G = (V, E)$ *is a chain of connected vertices* $C_0 \subseteq V$*, each of degree 2, together with two endpoints* $\{e_1, e_2\} \in V$ *connected to* C_0*. Its* length *is defined as* $dist(e_1, e_2)$*. In this and the next subsections, we abuse the notion of* $dist(x, y)$ *for* $x \in C$ *and* $y \in C$ *and use it to represent the length of the shortest path between vertices* x *and* y *that uses only vertices inside the corridor even if it is not the shortest path between them.*

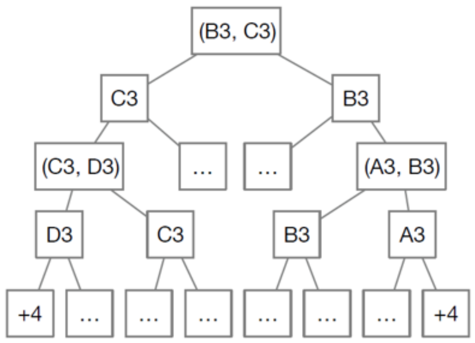
Figure 4.11a shows a corridor of length 3 made up of $C_0 = \{\text{B3}, \text{C3}\}$, $e_1 = \text{A3}$, and $e_2 = \text{D3}$. A corridor symmetry occurs when two agents attempt to traverse a corridor in opposite directions at the same time. We refer to the corresponding conflict as a *corridor conflict*.

Definition 4.9 (Corridor Conflict)**.** *Two agents are involved in a corridor conflict iff they traverse the same corridor in opposite directions and have one or more vertex or edge conflicts that occur inside the corridor.*

Example 4.8. In Figure 4.11a, CBS detects the edge conflict $\langle a_1, a_2, \text{B3}, \text{C3}, 3 \rangle$ and branches, thereby generating two child CT nodes. There are many shortest paths for each agent that avoid edge (B3, C3) at timestep 3 (e.g., path [A4, A3, B3, B3, C3, D3, D4] for agent a_1 and path [D2, D2,



(a) Two-agent MAPF instance with a corridor conflict. The shortest paths of agents a_1 and a_2 have an edge conflict inside the corridor at edge (B3, C3) at timestep 3.

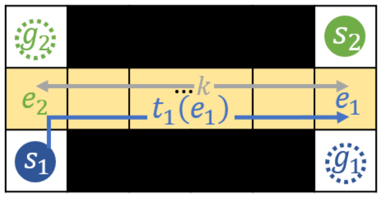


(b) CT generated by CBS when solving the two-agent MAPF instance in (a). Each left branch constrains agent a_2, and each right branch constrains agent a_1.

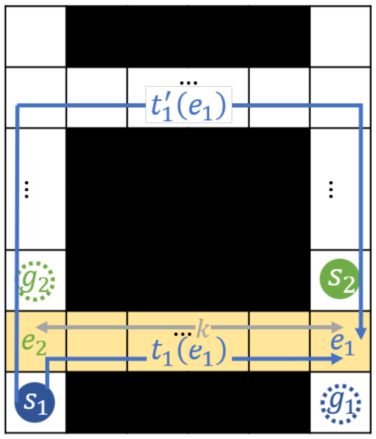
Figure 4.11: Example of a corridor conflict and the corresponding CT generated by CBS. In (b), each non-leaf CT node is marked with the vertex/edge of the chosen conflict. Each leaf CT node marked "+4" contains an optimal solution, whose sum of costs is the cost of the root CT node plus 4. Each leaf CT node marked "..." contains a plan with conflicts and produces suboptimal solutions in its descendant CT nodes.

D3, C3, B3, A3, A2] for agent a_2), all of which involve one wait action and differ only in where the wait action is taken. However, each of these single-wait paths remains in conflict with the path of the other agent. CBS has to branch at least four times to find conflict-free paths in such a situation and branch even more times to prove their optimality. Figure 4.11b shows the corresponding CT. Only two of the sixteen leaf CT nodes contain optimal solutions. □

This example highlights an especially pernicious characteristic of corridor symmetry: CBS may be forced to continue branching and exploring irrelevant and suboptimal resolutions of a corridor conflict to compute an optimal solution eventually. As the corridor length k increases, the number of expanded CT nodes grows exponentially as 2^{k+1} (because, when resolving a corridor conflict, the cost of a CT node at depth d is d plus the cost of the root CT node, and the cost of an optimal solution is k plus the cost of the root CT node). We therefore propose a new reasoning technique that identifies and resolves corridor conflicts efficiently. We present this technique in the following four subsections. We then extend it to handle several special corridor symmetries more efficiently and evaluate its empirical performance in the next section.



(a) Without bypasses.



(b) With bypasses.

Figure 4.12: Illustration of corridor conflicts with and without bypasses. The corridors are highlighted in yellow.

4.5.1 Identifying Corridor Conflicts

Detecting corridor conflicts is straightforward by checking every vertex and edge conflict. We find the corridor on-the-fly by checking whether the conflicting vertex (or an endpoint of the conflicting edge) is of degree 2. To find the endpoints of the corridor, we check the degree of each of the two adjacent vertices and repeat the procedure until we find either a vertex whose degree is not 2 or the start or target vertex of one of the two agents.

4.5.2 Resolving Corridor Conflicts

Consider a corridor C of length k with endpoints e_1 and e_2, see Figure 4.12. Assume that the path of agent a_1 traverses the corridor from vertex e_2 to vertex e_1 and the path of agent a_2 traverses the corridor from vertex e_1 to vertex e_2. They conflict with each other inside the corridor. Let $t_i(e_i)$ for $i = 1,2$ be the earliest timestep when agent a_i can reach its exit endpoint e_i.

We first assume that there are no *bypasses* (i.e., paths that move agent a_i for $i = 1,2$ from its start vertex s_i to its exit endpoint e_i without going through corridor C) for either agent (see Figure 4.12a). Therefore, one of the agents must wait until the other one has fully traversed the corridor. If we prioritize agent a_1 and let agent a_2 wait, then the earliest timestep when agent a_2 can start to traverse the corridor from vertex e_1 is $t_1(e_1)+1$. Therefore, the earliest timestep when agent a_2 can reach e_2 is $t_1(e_1)+1+k$. Similarly, if we prioritize agent a_2 and let agent a_1 wait, then the earliest timestep when agent a_1 can reach e_1 is $t_2(e_2)+1+k$. Therefore, any paths of agent a_1 that reach vertex e_1 before or at timestep $t_2(e_2)+k$ must conflict with any paths of agent a_2 that reach vertex e_2 before or at timestep $t_1(e_1)+k$.

Now, we consider bypasses (see Figure 4.12b). Assume that agent a_i for $i=1,2$ has bypasses to reach its exit endpoint e_i without traversing corridor C and the earliest timestep when it can reach vertex e_i using a bypass is $t'_i(e_i)$. If we prioritize agent a_1, then agent a_2 can either wait or use a bypass. Thus, the earliest timestep when agent a_2 can reach vertex e_2 is $\min(t'_2(e_2), t_1(e_1)+1+k)$. Similarly, if we prioritize agent a_2, then the earliest timestep when agent a_1 can reach vertex e_1 is $\min(t'_1(e_1), t_2(e_2)+1+k)$. Therefore, any paths of agent a_1 that reach vertex e_1 before or at timestep $\min(t'_1(e_1)-1, t_2(e_2)+k)$ must conflict with any paths of agent a_2 that reach vertex e_2 before or at timestep $\min(t'_2(e_2)-1, t_1(e_1)+k)$. In other words, the following two constraints are mutually disjunctive:

- $\langle a_1, e_1, [0, \min(t'_1(e_1)-1, t_2(e_2)+k)]\rangle$ and
- $\langle a_2, e_2, [0, \min(t'_2(e_2)-1, t_1(e_1)+k)]\rangle$,

where $\langle a_i, v, [t_{min}, t_{max}]\rangle$ is a *range constraint* [8] that prohibits agent a_i from being at vertex v at any timestep $t \in [t_{min}, t_{max}]$. Thus, to resolve a corridor conflict, we split the CT node N with two range constraints. We use state-time A* (recall Section 2.3.1) to compute $t_i(e_i)$ and $t'_i(e_i)$ for $i=1,2$. We cannot simply use the timesteps when the paths $N.plan[a_1]$ and $N.plan[a_2]$ visit vertices e_1 and e_2, respectively, for $t_1(e_1)$ and $t_2(e_2)$ because these paths minimize only the number of timesteps needed to reach the target vertices and do not necessarily minimize the number of timesteps needed to reach vertices e_1 and e_2, respectively.

For example, for the corridor conflict in Figure 4.11a, we calculate $k=3$, $t_1(\text{D3}) = t_2(\text{A3}) = 4$, and $t'_1(\text{D3}) = t'_2(\text{A3}) = +\infty$. Hence, to resolve this conflict, we split the root CT node and add the range constraints $\langle a_1, \text{D3}, [0,7]\rangle$ and $\langle a_2, \text{A3}, [0,7]\rangle$ to the child CT nodes. In the right (left) child CT node, we replan the path of agent a_1 (a_2) and find the new path [A4, A4, A4, A4, A4, A3, B3, C3, D3, D4] ([D2, D2, D2, D2, D2, D3, C3, B3, A3, A2]), that waits at its start vertex for 4 timesteps before moving to its target vertex. It waits at its start vertex rather than any vertex in the corridor because CBS breaks ties by preferring a path that has the fewest conflicts with the paths of other agents. Hence, the paths in both child CT nodes are conflict-free, and the corridor symmetry is resolved in a single branching step.

As for the rectangle reasoning techniques (or, more specifically, Line 15 in Algorithm 4.1), we use this branching method only when the range constraints block the paths of both agents in the current CT node, which ensures that the paths in both child CT nodes are different from the paths in the current CT node.

We add range constraints at the exit endpoints of the agents instead of their entry endpoints because there might be an optimal solution that violates both "entry-endpoint" range constraints. For instance, given the corridor conflict shown in Figure 4.11a, if we use "entry-endpoint" range constraints $\langle a_1, \text{A3}, [0,4] \rangle$ and $\langle a_2, \text{D3}, [0,4] \rangle$, then the pair of paths, [A4, A3, B3, C3, D3, D4] for agent a_1 and [D2, D3, C3, D3, D2, D3, C3, B3, A3, A2] for agent a_2, is an optimal solution that violates both "entry-endpoint" range constraints.

4.5.3 Classifying Corridor Conflicts

Similarly to target conflicts, we classify corridor conflicts based on the cardinality of the vertex/edge conflict inside the corridor. A corridor conflict is cardinal iff the corresponding vertex/edge conflict is cardinal; it is semi-cardinal iff the corresponding vertex/edge conflict is semi-cardinal; and it is non-cardinal iff the corresponding vertex/edge conflict is non-cardinal. This is an approximate way of classifying corridor conflicts. Figure 4.11a shows an example where, after branching on a non-cardinal corridor conflict in a CT node N, the costs of both resulting child CT nodes are larger than the cost of CT node N. Assume that CT node N has two constraints, each of which prohibits one of the agents from being at its target vertex at timestep 5, so both agents have to wait for one timestep and thus have paths of length 6. If agent a_1 waits at vertex D3 at timestep 5 and agent a_2 waits at vertex A3 at timestep 5, then they have the non-cardinal edge conflict $\langle a_1, a_2, \mathrm{B3}, \mathrm{C3}, 3\rangle$. As a result, the corridor conflict is classified as non-cardinal. However, as we saw above, when we use the range constraints $\langle a_1, \mathrm{D3}, [0,7]\rangle$ and $\langle a_2, \mathrm{A3}, [0,7]\rangle$ to resolve the corridor conflict, the costs of both child CT nodes are larger than the cost of CT node N.

We cannot use the differences between the upperbound timesteps of the range constraints and the timesteps of the agents when they reach their exit endpoints to predict the cost change of the child CT nodes because the agents may have paths to their target vertices without visiting their exit endpoints of the corridor.

4.5.4 Theoretical Analysis

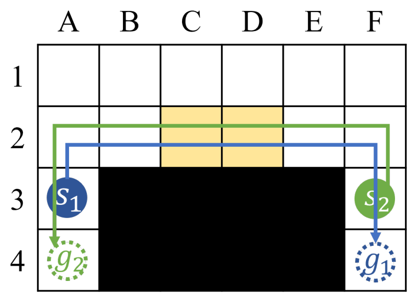
Property 4.9. *For all combinations of paths of agents a_1 and a_2 with a corridor conflict, if one path violates $\langle a_1, e_1, [0, \min(t'_1(e_1) - 1, t_2(e_2) + k)]\rangle$ and the other path violates $\langle a_2, e_2, [0, \min(t'_2(e_2) - 1, t_1(e_1) + k)]\rangle$, then the two paths have one or more vertex or edge conflicts inside the corridor.* □

Since we have already intuitively proved Property 4.9 when we introduce range constraints in Section 4.5.2, we move the formal proof to Appendix E, so as not to disrupt the flow of text too much. Property 4.9 implies that range constraints are mutually disjunctive, and thus, according to Theorem 4.1 and the fact that each range constraint blocks the current path of the agent on which it is imposed, using them to split a CT node preserves the completeness and optimality of CBS.

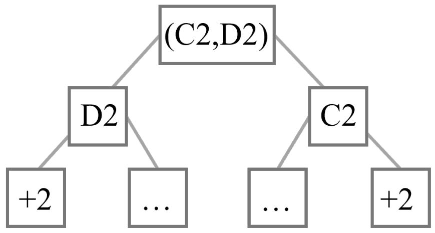
Theorem 4.6. *Resolving corridor conflicts with range constraints preserves the completeness and optimality of CBS.* □

4.6 Generalized Corridor Symmetry

The corridor reasoning technique in the previous section has some limitations when handling three special corridor symmetries, namely pseudo-corridor symmetries, corridor symmetries with start vertices inside the corridor, and corridor-target symmetries. In this section, we first discuss these special cases in detail in the following three subsections. We then present the framework of the



(a) Two-agent MAPF instance with a pseudo-corridor conflict. The shortest paths of agents a_1 and a_2 have an edge conflict at edge (C2, D2) at timestep 4.



(b) CT generated by CBS when solving the two-agent MAPF instance in (a). Each left branch constrains agent a_2, and each right branch constrains agent a_1.

Figure 4.13: Example of a pseudo-corridor conflict and the corresponding CT generated by CBS. In (b), each non-leaf CT node is marked with the vertex/edge of the chosen conflict. Each leaf CT node marked "+2" contains an optimal solution, whose sum of costs is the cost of the root CT node plus 2. Each leaf CT node marked "..." contains a plan with conflicts and eventually produces suboptimal solutions in its descendant CT nodes.

generalized corridor reasoning technique that can handle all types of corridor symmetries in Section 4.6.4. We finally show empirical results in Section 4.6.5.

4.6.1 Pseudo-Corridor Conflicts

Pseudo-corridor symmetry is a special corridor symmetry that behaves like a corridor conflict but occurs in a non-corridor region.

Example 4.9. In Figure 4.13a, CBS detects edge conflict $\langle a_1, a_2, \text{C2}, \text{D2}, 4\rangle$ and branches, thereby generating two child CT nodes. There are many shortest paths for each agent that avoid edge (C2, D2) at timestep 4 (e.g., path [A3, A2, B2, C2, C2, D2, E2, F2, F3, F4] for agent a_1 and path [F3, F2, E2, D2, D2, C2, B2, A2, A3, A4] for agent a_2), but they all involve one wait action and differ only in where the wait action is taken. However, each of these single-wait paths remains in conflict with the path of the other agent. CBS has to branch again to find conflict-free paths in such a situation. Figure 4.13b shows the corresponding CT. Only the left-most and right-most leaf CT nodes contain optimal solutions. □

Like corridor conflicts, a pseudo-corridor conflict occurs when (1) two agents move in opposite directions, (2) they have a vertex or edge conflict, and (3) adding one wait action to one of the agents before the timestep of the vertex or edge conflict, no matter where, leads to another edge or vertex conflict. In fact, a pseudo-corridor conflict can be viewed as a corridor conflict whose corridor is of length 1, i.e., consists of only two endpoints. Pseudo-corridor conflicts might seem to be less problematic than corridor conflicts, as the sizes of the CTs do not grow exponentially. However, they could occur more frequently as they are not restricted to maps that have corridors.

We reuse the corridor reasoning technique to resolve pseudo-corridor conflicts. That is, when we find a corridor conflict of length 1, we generate two range constraints

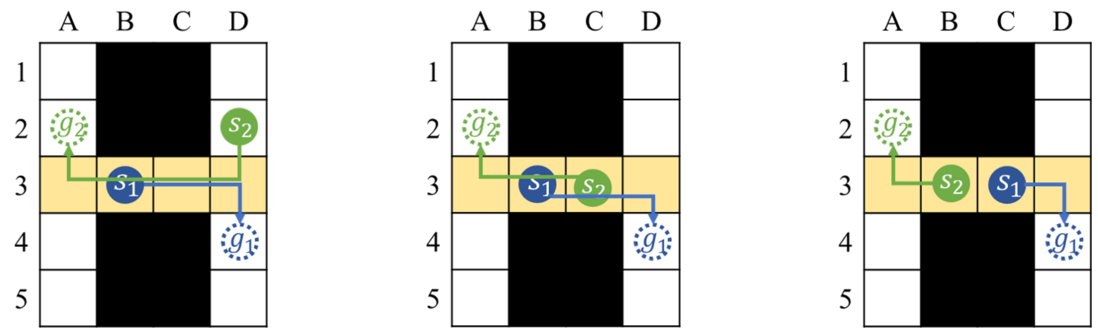
- $c_1 = \langle a_1, e_1, [0, \min(t'_1(e_1) - 1, t_2(e_2) + 1)] \rangle$ and
- $c_2 = \langle a_2, e_2, [0, \min(t'_2(e_2) - 1, t_1(e_1) + 1)] \rangle$,

where $t_i(e_i)$ for $i = 1, 2$ is the earliest timestep when agent a_i can reach endpoint e_i and $t'_i(e_i)$ for $i = 1, 2$ is the earliest timestep when agent a_i can reach endpoint e_i without using edge (e_1, e_2). All properties listed in Section 4.5.4 hold here. By reusing their proofs without changes, we can show that resolving a pseudo-corridor conflict with constraints c_1 and c_2 preserves the completeness and optimality of CBS.

In practice, we only use range constraints c_1 and c_2 to resolve the pseudo-corridor conflict at a CT node N if path $N.plan[a_1]$ violates range constraint c_1 and path $N.plan[a_2]$ violates range constraint c_2, and we are interested in only cardinal pseudo-corridor conflicts because semi-/non-cardinal pseudo-corridor conflicts are easy to resolve. A necessary but insufficient condition to ensure this is that, if the conflict between the two agents is a vertex conflict at timestep t, then the MDDs of both agents have only one MDD node at timesteps $t-1$, t, and $t+1$, and the MDD node of one agent at timestep $t-1$ is identical to the MDD node of the other agent at timestep $t+1$; or if the conflict is an edge conflict at timestep t, then the MDDs of both agents have only one MDD node at timesteps $t-1$ and t. Therefore, before we generate range constraints c_1 and c_2, we check the MDDs of both agents to eliminate some non-pseudo-corridor conflicts, as checking

Algorithm 4.2: Pseudo-corridor reasoning.

```
Input: Vertex conflict $c = \langle a_1, a_2, v, t \rangle$ or edge conflict $c = \langle a_1, a_2, v, u, t \rangle$ at a CT node $N$
       with two MDDs $MDD_1$ and $MDD_2$
1  if c is a vertex conflict $\wedge$ for $i = 1, 2$, $MDD_i$ has singletons at timesteps $t-1$, $t$, and $t+1$
     and the MDD node of $MDD_i$ at timestep $t-1$ is identical to the MDD node of $MDD_{3-i}$
     at timestep $t+1$ then
2      $e_1 \leftarrow v$;
3      $e_2 \leftarrow$ the vertex of the MDD node of $MDD_1$ at timestep $t-1$;
4  else if c is an edge conflict $\wedge$ for $i = 1, 2$, $MDD_i$ has singletons at timesteps $t-1$ and $t$
     then
5      $e_1 \leftarrow u$;
6      $e_2 \leftarrow v$;
7  else return Not-Corridor;
8  $c_1 \leftarrow \langle a_1, e_1, [0, \min\{t_1'(e_1) - 1, t_2(e_2) + 1\}] \rangle$;
9  $c_2 \leftarrow \langle a_2, e_2, [0, \min\{t_2'(e_2) - 1, t_1(e_1) + 1\}] \rangle$;
10 if $c_1$ blocks $N.plan[a_1] \wedge c_2$ blocks $N.plan[a_2]$ then return $c_1$ and $c_2$;
11 return Not-Corridor;
```



(a) Corridor conflict. (b) Corridor conflict. (c) No corridor conflict.

Figure 4.14: Examples of corridor conflicts with start vertices inside the corridor.

MDDs is substantially computationally cheaper than computing $t_i(e_i)$ and $t_i'(e_i)$ for generating range constraints. Algorithm 4.2 summarizes the pseudo-code for the pseudo-corridor reasoning technique. All pseudo-corridor conflicts returned by Algorithm 4.2 are cardinal.

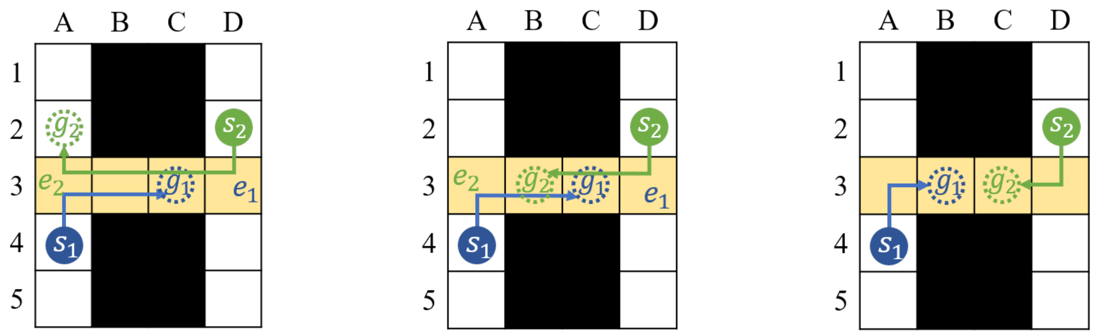
4.6.2 Corridor Conflicts with Start Vertices inside the Corridor

The corridor reasoning technique cannot resolve corridor conflicts efficiently when the start vertices of one or both agents are inside the corridor.

Example 4.10. Figure 4.14a shows the same example as Figure 4.11a except that the start vertex of agent a_1 is inside the corridor. If the two agents follow their individual shortest paths, they have an edge conflict at (C3, D3) at timestep 2. Thus, when we use the corridor reasoning technique described in Section 4.5.1, we find a corridor $C = \{\mathrm{B3}, \mathrm{C3}, \mathrm{D3}\}$ of length 2 and generate a pair of range constraints $\langle a_1, \mathrm{D3}, [0,5] \rangle$ and $\langle a_2, \mathrm{B3}, [0,4] \rangle$. However, when we generate the right child CT node with the first constraint, we cannot find a shortest path for agent a_1 that does not conflict with agent a_2. In fact, the shortest path for agent a_1 that does not conflict with agent a_2 is to first move to vertex A4, wait there until agent a_2 reaches vertex A3, and then traverse the corridor and reach the target vertex. □

This example shows that the previous corridor reasoning technique cannot resolve the corridor conflict in a single branching step because it stops extending the corridor after finding a start vertex. Figure 4.14b shows a similar example where the start vertices of both agents are inside the corridor. Therefore, in this subsection, we modify the corridor reasoning technique by allowing start vertices to be inside the corridor. Below are the details of the modification.

Identifying corridor conflicts We describe a method that identifies both basic corridor conflicts and corridor conflicts with start vertices inside the corridor. For every vertex and edge conflict, we first find the corridor on-the-fly by checking whether the conflicting vertex (or an endpoint of the conflicting edge) is of degree 2. To find the endpoints of the corridor, we check the degree of each of the two adjacent vertices and repeat the procedure until we find either a vertex whose degree is not 2 or the target vertex of one of the two agents. Then, we say that the two agents are involved in a corridor conflict iff they (1) leave the corridor from different endpoints (which ensures that the two agents traverse the corridor in opposite directions) and (2) have to cross each other inside the corridor (which will be formally defined by a MUSTCROSS(a_1, a_2, C) function in Algorithm 4.3 later). The second condition is to avoid cases like in Figure 4.14c. Although the paths of the two agents shown in Figure 4.14c do not conflict with each other, when considering constraints in the



(a) Corridor-target conflict. (b) Corridor-target conflict. (c) No corridor-target conflict.

Figure 4.15: Examples of corridor-target conflicts.

CT node, the shortest paths of the two agents may be longer than the paths shown in the figure and conflict inside the corridor. But we should not view it as a corridor conflict.

Resolving and classifying corridor conflicts They are the same as the original techniques shown in Sections 4.5.2 and 4.5.3.

Theoretical analysis All properties listed in Section 4.5.4 hold here. We can reuse their proofs without changes. Therefore, this modified technique preserves the completeness and optimality of CBS.

4.6.3 Corridor-Target Conflicts

Another interesting case occurs when the target vertex of an agent is inside the corridor.

Example 4.11. Figure 4.15a shows the same example as Figure 4.11a except that the target vertex of agent a_1 is inside the corridor. If the two agents follow their individual shortest paths, they have an edge conflict at edge (B3, C3) at timestep 3. Thus, when we use the corridor reasoning technique described in Section 4.5.1, we find a corridor $C = \{\text{A3}, \text{B3}, \text{C3}\}$ of length 2 and generate a pair of range constraints $\langle a_1, \text{C3}, [0,6] \rangle$ and $\langle a_2, \text{A3}, [0,5] \rangle$. In the right child CT node with the first constraint, agent a_1 waits until agent a_2 leaves the corridor and then starts to enter the corridor from vertex A3 at timestep 5. However, in the left child CT node with the second constraint, we

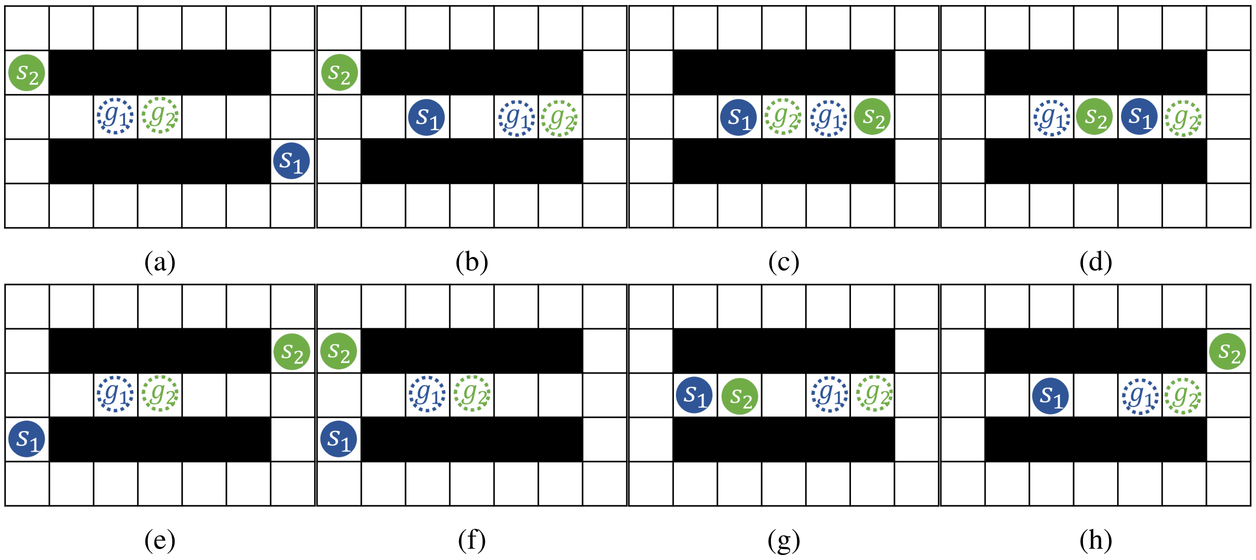


Figure 4.16: Examples of cases where the target vertices of agents a_1 and a_2 are inside corridor C. Only the cases shown in the first row are classified as corridor-target conflicts by Function MUSTCROSS(a_1,a_2,C).

cannot find a shortest path for agent a_2 that does not conflict with the path of agent a_1. The best resolution under this CT node is to first let agent a_1 travel through the corridor and leave vertex D3, then agent a_2 enter the corridor from vertex D3, and finally agent a_1 reenter the corridor from vertex D3. In other words, the paths of both agents have to be changed. □

This example shows that the previous corridor reasoning technique cannot resolve the corridor conflict in a single branching step because it stops extending the corridor after finding a target vertex. Therefore, we extend the reasoning technique for corridor conflicts with start vertices inside the corridor to one that also allows for target vertices inside the corridor. In particular, we refer to a corridor conflict with one or two target vertices inside the corridor as a *corridor-target conflict*.

4.6.3.1 Identifying Corridor Conflicts

We describe a method that identifies basic corridor conflicts, corridor conflicts with start vertices inside the corridor, and corridor conflicts with target vertices inside the corridor, i.e., corridor-target conflicts. For every vertex and edge conflict, we first find the corridor on-the-fly by checking

Algorithm 4.3: Identify generalized corridor conflicts.

```
1 Function MustCross($a_1, a_2, C$)
2     foreach $i = 1, 2$ do
3         if $s_i \in C$ then $b_i \leftarrow s_i$;
4         else $b_i \leftarrow$ GetEntryEndpoint($a_i, C$);
5         if $g_i \in C$ then $e_i \leftarrow g_i$;
6         else $e_i \leftarrow$ GetExitEndpoint($a_i, C$);
7     if $b_1 \neq b_2 \wedge e_1 \neq e_2 \wedge$ the direction of moving from $b_1$ to $b_2$ is opposite to the direction
        of moving from $e_1$ to $e_2$ then return false;
8     else return true;
```

whether the conflicting vertex (or an endpoint of the conflicting edge) is of degree 2. To find the endpoints of the corridor, we check the degree of each of the two adjacent vertices and repeat the procedure until we find a vertex whose degree is not 2. We say that the two agents are involved in a corridor conflict iff they have to cross each other inside the corridor. We remove the condition from Section 4.6.2 that requires the agents to move in opposite directions. This is so because, when the start and target vertices are inside the corridor, the two agents may move in the same direction but still have an unavoidable conflict, e.g., the conflict in Figure 4.16b.

We use a function MustCross(a_1, a_2, C) to determine whether agents a_1 and a_2 have to cross each other in corridor C (see Algorithm 4.3). We use vertex b_i for $i = 1, 2$ to denote the start vertex s_i of agent a_i if it is inside the corridor and the endpoint from where agent a_i enters the corridor otherwise. Similarly, we use vertex d_i for $i = 1, 2$ to denote the target vertex g_i of agent a_i if it is inside the corridor and the endpoint from where agent a_i leaves the corridor otherwise. Agents a_1 and a_2 must cross each other iff $b_1 \neq b_2$, $d_1 \neq d_2$, and the direction of moving from vertex b_1 to vertex b_2 is opposite to the direction of moving from vertex d_1 to vertex d_2. Figure 4.16 shows more examples.

4.6.3.2 Resolving Corridor-Target Conflicts

Unlike corridor conflicts with start vertices inside the corridor, that can be resolved by reusing the basic corridor reasoning technique, corridor-target conflicts require new techniques. We combine the corridor reasoning technique with the target reasoning technique to resolve corridor-target conflicts.

Case 1: Only one target vertex is inside the corridor. Without loss of generality, we assume that target vertex g_1 is inside the corridor and target vertex g_2 is not. We use Figure 4.15a as a running example, where the path of agent a_2 traverses the corridor from endpoint e_1 (i.e., vertex D3) to endpoint e_2 (i.e., vertex A3). However, agent a_2 might have bypasses that can reach vertex e_2 without traversing the corridor (omitted in Figure 4.15a). So, it can choose whether to use the corridor. If it uses the corridor, then agent a_1 has to use the corridor after agent a_2 because it must eventually wait at its target vertex inside the corridor forever. There are two cases for agent a_1 to use the corridor after agent a_2.

- If agent a_1 enters the corridor from endpoint e_2, then it has to let agent a_2 traverse the corridor first. So, the earliest timestep for it to enter the corridor from vertex e_2 is $\max\{t_1(e_2), t_2(e_2)+1\}$, and, therefore, the earliest timestep for it to reach its target vertex g_1 is $\max\{t_1(e_2), t_2(e_2)+1\} + \mathit{dist}(e_2, g_1)$ (recall that $t_i(x)$ for $i = 1, 2$ is the earliest timestep when agent a_i can reach vertex x).
- Similarly, if agent a_1 enters the corridor from endpoint e_1, then the earliest timestep for it to reach its target vertex g_1 is $\max\{t_1(e_1), t_2(e_1)+1\} + \mathit{dist}(e_1, g_1)$.

In other words, if agent a_1 reaches its target vertex at or before timestep

$$l = \min_{i=1,2}\{\max\{t_1(e_i) - 1, t_2(e_i)\} + \mathit{dist}(e_i, g_1)\}, \tag{4.8}$$

then agent a_2 cannot traverse the corridor without conflicting with a_1, i.e., the earliest timestep for it to reach endpoint e_2 is $t'_2(e_2)$ (i.e., using a bypass that does not traverse the corridor). Therefore, to resolve this corridor-target conflict, we generate two child CT nodes, each with one of the constraint sets $C_1 = \{l_1 > l\}$ and $C_2 = \{l_1 \leq l, \langle a_2, e_2, [0, t'_2(e_2) - 1]\rangle\}$.

Case 2: Both target vertices are inside the corridor. The reasoning is similar to Case 1. We use Figure 4.15b as a running example, where agent a_2 has to enter the corridor to reach its target vertex g_2 and can enter it from either endpoint e_1 (i.e., vertex D3) or endpoint e_2 (i.e, vertex A3). If agent a_2 enters the corridor from endpoint e_1, then it has to visit vertex g_1 before agent a_1 completes its path and waits at its target vertex g_1 forever.

- If agent a_1 enters the corridor from endpoint e_2, then it has to let agent a_2 traverse the corridor first. So, the earliest timestep for agent a_1 to enter the corridor from endpoint e_2 is $\max\{t_1(e_2), t_2(e_2) + 1\}$, and, therefore, the earliest timestep for agent a_1 to reach its target vertex g_1 is $\max\{t_1(e_2), t_2(e_2) + 1\} + dist(e_2, g_1)$.

- If agent a_1 enters the corridor from endpoint e_1, then the earliest timestep for agent a_1 to reach its target vertex g_1 is $\max\{t_1(e_1), t_2(e_1) + 1\} + dist(e_1, g_1)$.

In other words, if agent a_1 completes its path at or before timestep l (defined in Equation (4.8)), then agent a_2 cannot visit vertex g_1 without conflicting with agent a_1, i.e., the earliest timestep for agent a_2 to reach its target vertex g_2 is $t'_2(g_2)$, which represents the earliest timestep for agent a_2 to reach its target vertex g_2 via a bypass, i.e., a path that enters the corridor from vertex e_2. Therefore, to resolve this corridor-target conflict, we generate two child CT nodes, each with one of the constraint sets $C_1 = \{l_1 > l\}$ and $C_2 = \{l_1 \leq l, l_2 > t'_2(g_2) - 1\}$.

4.6.3.3 Classifying Corridor-Target Conflicts

We reuse the method in Section 4.5.3 to classify corridor-target conflicts.

Algorithm 4.4: Generalized corridor reasoning.

```
Input: Vertex conflict $c = \langle a_1, a_2, v, t\rangle$ or edge conflict $c = \langle a_1, a_2, v, u, t\rangle$ at a CT node $N$
       with two MDDs $MDD_1$ and $MDD_2$
1  Construct corridor $C = C_0 \cup \{e_1, e_2\}$ from vertex $v$ or edge $(v, u)$;
2  if $|C| = 2$ then                                   // Pseudo-corridor conflict; Algorithm 4.2
3      return PseudoCorridorReasoning($c$,$N$,$MDD_1$,$MDD_2$);
4  if MustCross($a_i$,$a_j$,$C$) = false then return Not-Corridor;            // Algorithm 4.3
5  if $g_1 \in C \wedge g_2 \in C$ then                  // Corridor-target conflict, Case 2
6      $l \leftarrow \min_{s=1,2}\{\max\{t_1(e_s) - 1, t_2(e_s)\} + dist(e_s, g_1)\}$;
7      $C_1 \leftarrow \{l_1 > l\}$;
8      $C_2 \leftarrow \{l_1 \leq l, l_2 > t'_2(g_2) - 1\}$;
9  else if $g_1 \in C \vee g_2 \in C$ then               // Corridor-target conflict, Case 1
10     WLOG, let $a_1$ be the agent whose target vertex is inside the corridor;
11     $l \leftarrow \min_{s=1,2}\{\max\{t_1(e_s) - 1, t_2(e_s)\} + dist(e_s, g_1)\}$;
12     $C_1 \leftarrow \{l_1 > l\}$;
13     $C_2 \leftarrow \{l_1 \leq l, \langle a_2, e_2, [0, t'_2(e_2) - 1]\rangle\}$;
14 else     // Basic corridor conflict or corridor conflict with start vertices inside the corridor
15     $C_1 \leftarrow \{\langle a_1, e_1, [0, \min\{t'_1(e_1) - 1, t_2(e_2) + dist(e_1, e_2)\}]\rangle\}$;
16     $C_2 \leftarrow \{\langle a_2, e_2, [0, \min\{t'_2(e_2) - 1, t_1(e_1) + dist(e_2, e_1)\}]\rangle\}$;
17 if $C_1$ blocks $N.plan[a_1] \wedge C_2$ blocks $N.plan[a_2]$ then return $C_1$ and $C_2$;
18 return Not-Corridor;
```

4.6.3.4 Theoretical Analysis

Property 4.10. *For all combinations of paths of agents a_1 and a_2 with a corridor-target conflict, if one path violates constraint set C_1 and the other path violates constraint set C_2, then the two paths have one or more vertex or edge conflicts inside the corridor.* □

Since we have already intuitively proved Property 4.10 when we introduced constraint sets C_1 and C_2 in Section 4.6.3.2, we move the formal proof to Appendix F. Property 4.10 implies that constraint sets C_1 and C_2 are mutually disjunctive, and thus, according to Theorem 4.1 and the fact that C_i for $i = 1, 2$ blocks the current path of a_i, using them to split a CT node preserves the completeness and optimality of CBS.

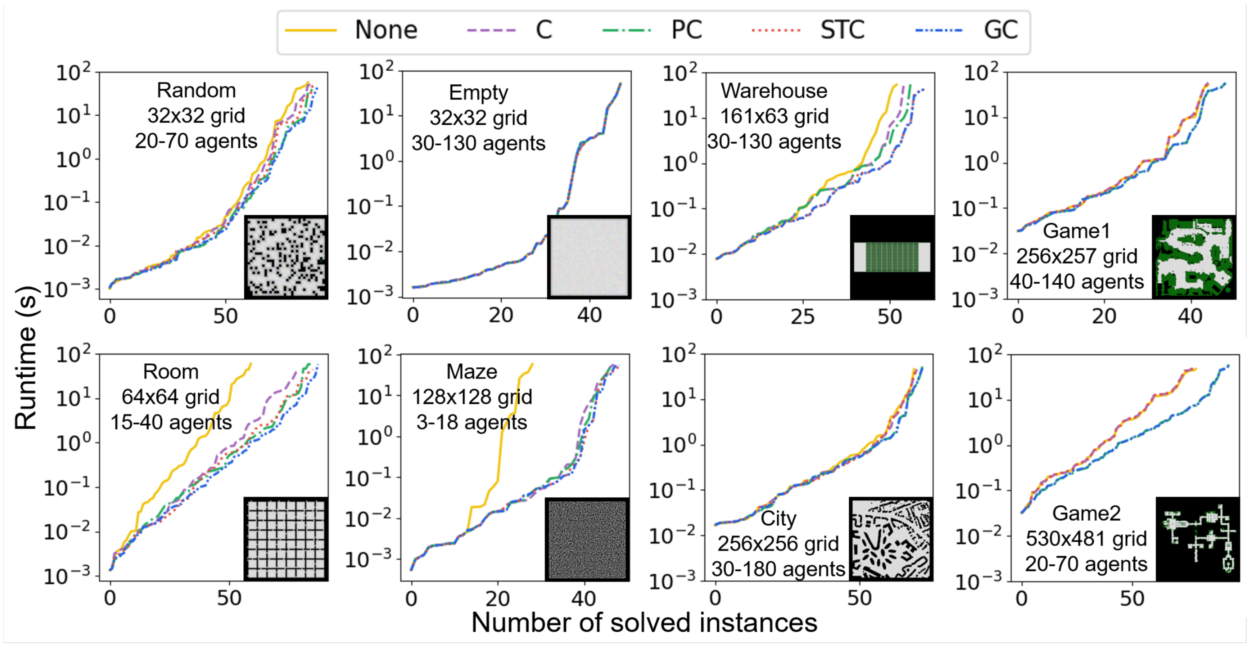


Figure 4.17: Runtime distributions of CBSH with different corridor reasoning techniques. In the city-map and two game-map figures, the yellow and purple lines are hidden by the red lines, and the green lines are hidden by the blue lines.

4.6.4 Summary on Generalized Corridor Symmetry

So far, we have discussed all types of generalized corridor conflicts separately, namely basic corridor conflicts, pseudo-corridor conflicts, corridor conflicts with start vertices inside the corridor, and corridor-target conflicts. Algorithm 4.4 shows the pseudo-code for generalized corridor reasoning, that integrates these reasoning procedures.

Combining the theoretical analysis for each type of generalized corridor conflicts, we obtain the following theorem.

Theorem 4.7. *Resolving generalized corridor conflicts with the constraint sets returned by Algorithm 4.4 preserves the completeness and optimality of CBS.* □

4.6.5 Empirical Evaluation

We compare all corridor reasoning techniques empirically and show the results in Figure 4.17. **None** represents CBSH, **C** represents CBSH with the basic corridor-reasoning technique described in Section 4.5, **PC** represents CBSH with the basic corridor-reasoning technique described in Section 4.5 plus the pseudo-corridor-reasoning technique described in Section 4.6.1, **STC** represents CBSH with the basic corridor-reasoning technique described in Section 4.5 plus the modification for handling start and target vertices described in Sections 4.6.2 and 4.6.3, and **GC** represents CBSH with generalized corridor-reasoning technique shown in Algorithm 4.4.

Map `Empty` contains no obstacles and thus no corridors. So, none of the corridor-reasoning techniques speeds up CBSH, but they do not slow down CBSH either. Maps `Game1`, `City`, and `Game2` have only few corridors, but they all have obstacles of various shapes, where pseudo-corridor reasoning can be useful. As a result, although C and STC do not improve the performance of CBSH, PC and GC do. Maps `Random`, `Warehouse`, `Room`, and `Maze` have many corridors, and, as a result, all corridor-reasoning techniques speed up CBSH. Among all maps, the improvements on map `Maze` are the largest. Among all corridor reasoning techniques, GC is always the best.

4.7 Symmetry Reasoning Framework

Until now, we have described and empirically evaluated each symmetry reasoning technique independently. In this section, we present the complete framework of our pairwise symmetry-reasoning techniques, namely how to identify different classes of symmetry conflicts and, when multiple conflicts exist, which conflict to resolve first. We then show some empirical results for combining all symmetry reasoning techniques and compare them with mutex propagation, a different symmetry reasoning technique.

4.7.1 Framework

During the expansion of a CT node, we run symmetry reasoning for each vertex and edge conflict c. Algorithm 4.5 shows the pseudo-code. We first run generalized corridor reasoning by calling Algorithm 4.4 [Line 1] and return the corresponding constraint sets C_1 and C_2 for resolving the conflict if it is a (generalized) corridor conflict [Line 2]. If conflict c is not a (generalized) corridor

Algorithm 4.5: Symmetry reasoning.

```
Input: Vertex conflict $c = \langle a_1, a_2, v, t \rangle$ or edge conflict $c = \langle a_1, a_2, v, u, t \rangle$ at a CT node $N$
       with two MDDs $MDD_1$ and $MDD_2$
1  $\{C_1, C_2\} \leftarrow$ GeneralizedCorridorReasoning($c, N, MDD_1, MDD_2$);
2  if $\{C_1, C_2\} \neq$ Not-Corridor then return $C_1$ and $C_2$;        // Corridor conflict
3  if $t \geq length(N.plan[a_1]) \vee t \geq length(N.plan[a_2])$ then
4      $\{C_1, C_2\} \leftarrow$ TargetReasoning($c$);
5      return $C_1$ and $C_2$;                                          // Target conflict
6  if $c$ is a semi-/non-cardinal vertex conflict then
7      $\{C_1, C_2\} \leftarrow$ GeneralizedRectangleReasoning($c, N, MDD_1, MDD_2$);
8      if $\{C_1, C_2\} \neq$ Not-Rectangle then return $C_1$ and $C_2$;  // Rectangle conflict
9  $\{C_1, C_2\} \leftarrow$ StandardCBSSplitting($c$);
10 return $C_1$ and $C_2$;                                              // Vertex/Edge conflict
```

conflict, we then check whether it is a target conflict by comparing the path lengths of the agents with the conflicting timestep t [Line 3]. If, say, the path of agent a_1 is no longer than t, then it is a target conflict, and we run target reasoning by calling function TargetReasoning(c) [Line 4] and return the corresponding constraint sets $C_1 = \{l_1 > t\}$ and $C_2 = \{l_1 \leq t\}$ [Line 5]. If conflict c is not a target conflict but a semi- or non-cardinal vertex conflict [Line 6], then we run generalized rectangle reasoning by calling the algorithm described in Section 4.3.2 [Line 7] and return the corresponding constraint sets if conflict c is a (generalized) rectangle conflict [Line 8]. If conflict c does not belong to any class of symmetry conflicts, then we use the standard CBS splitting method to generate the constraints [Line 9].

We run generalized corridor reasoning before target reasoning because, otherwise, target reasoning may identify some corridor-target conflicts as target conflicts and resolve them less efficiently. We run generalized rectangle reasoning after generalized corridor and target reasoning because this leads to the smallest runtime overhead. More specifically, if conflict c is a (generalized) rectangle conflict, then generalized corridor and target reasoning can quickly realize that conflict c is not a (generalized) corridor or target conflict because the graph structure around the conflicting vertex/edge and the conflicting timestep does not satisfy their requirements. On the other hand, if conflict *c* is a (generalized) corridor or target conflict, then generalized rectangle reasoning needs to manipulate the corresponding MDDs before it realizes that it is not a (generalized) rectangle conflict. Therefore, running generalized rectangle reasoning last leads to the smallest runtime overhead.

When choosing conflicts, we resolve cardinal conflicts first, then semi-cardinal conflicts, and last non-cardinal conflicts. The cardinality of symmetry conflicts are determined during the symmetry reasoning procedure, although we do not show it explicitly in Algorithm 4.5. When there are multiple conflicts of the same cardinality, we break ties using the motivation described in Section 2.3.2.1, i.e., in favor of conflicts that can increase the costs of the child CT nodes more. To be specific, we give target conflicts the highest priority because, when resolving a target conflict, the cost of at least one child CT node is at least one and often even larger than the cost of the parent CT node. (Generalized) corridor conflicts have the second highest priority because, when resolving a (generalized) corridor conflict, the costs of the child CT nodes can be more than one larger than the cost of the parent CT node. (Generalized) rectangle conflicts have the third highest priority because, when resolving a (generalized) rectangle conflict, the costs of the child CT nodes are typically at most one larger. Vertex and edge conflicts have the lowest priority because we prefer to resolve all symmetry conflicts first, and also, when resolving a vertex or edge conflict, the costs of the child CT nodes are typically at most one larger than the cost of the parent CT node.

4.7.2 Empirical Evaluation

In this subsection, we compare CBSH (denoted **None**), CBSH with the best variant of each of the reasoning technique, namely generalized rectangle reasoning (denoted **GR**), target reasoning (denoted **T**), and generalized corridor reasoning (denoted **GC**), and CBSH with their combination (denoted **GRTGC**, or **RTC** for short).

Runtimes and Success Rates Figure 4.18 presents the runtimes, and Figure 4.19 presents the *success rates*, i.e., the percentage of instances solved within the runtime limit of one minute. As

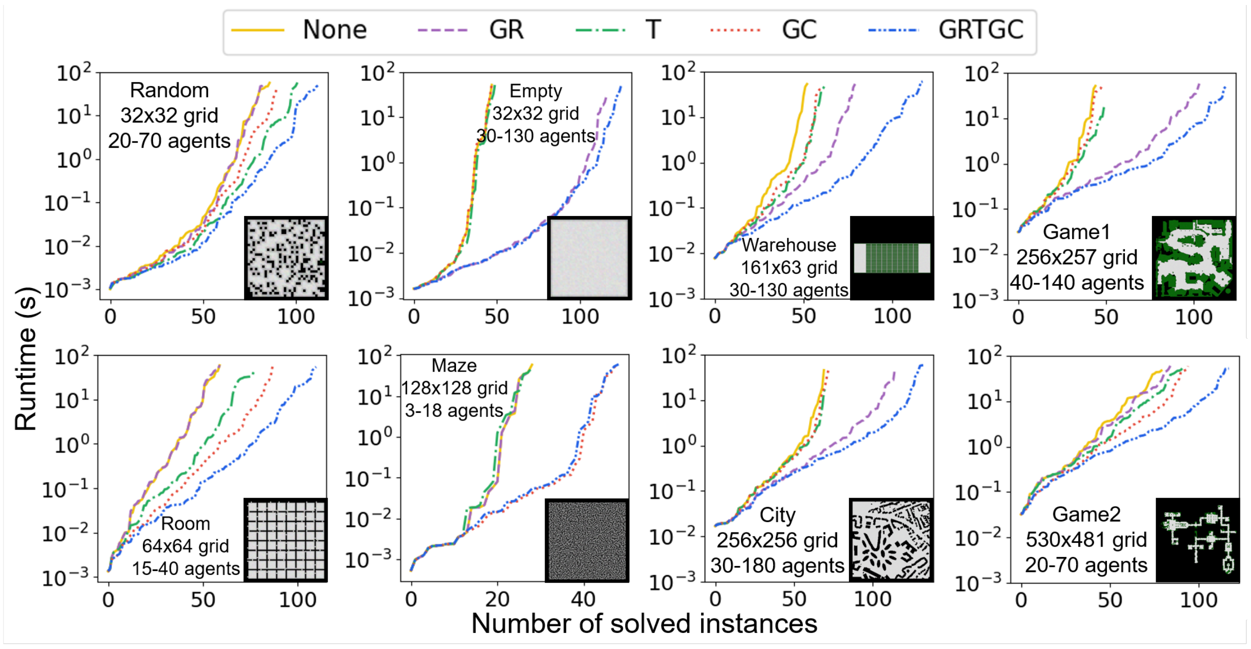


Figure 4.18: Runtime distributions of CBSH with different symmetry reasoning techniques.

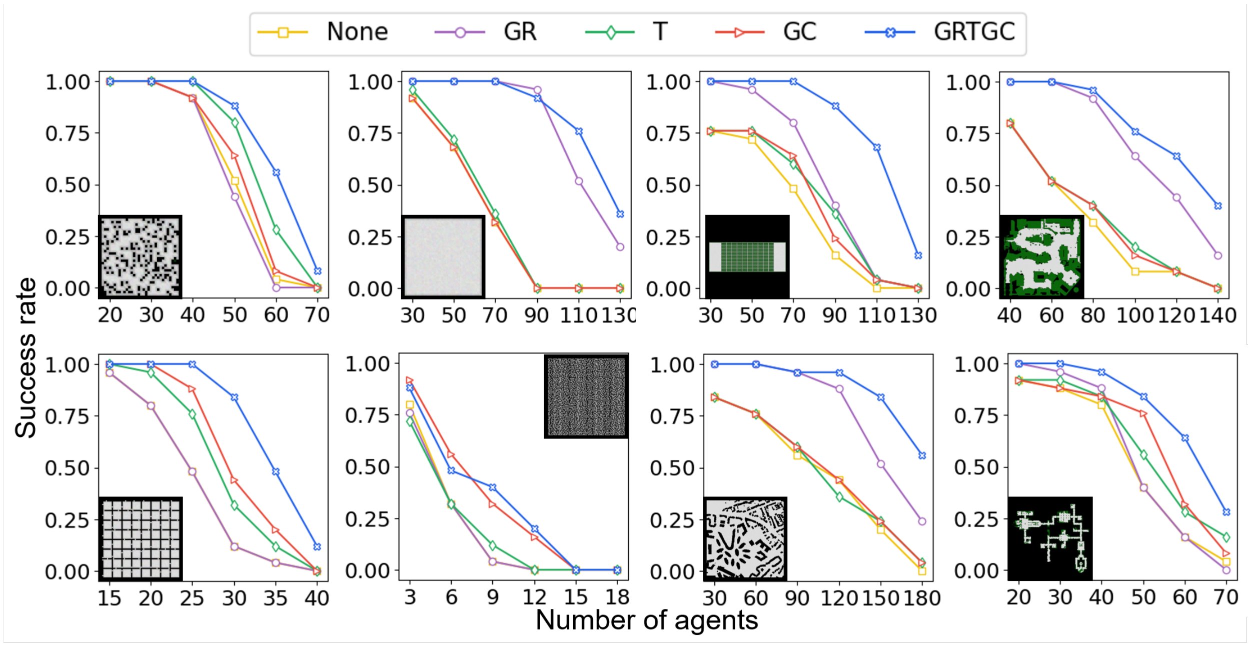


Figure 4.19: Success rates of CBSH with different pairwise symmetry reasoning techniques.

expected, all of GR, T, and GC can speed up CBSH, and the amount of their speedup depends on the structure of the maps. Their combination, i.e., RTC, is always (very close to) best. In Figure 4.19, we notice an interesting behavior on many maps, such as maps `Empty`, `Warehouse`,

Table 4.3: Scalability of None and RTC, i.e., the largest number of agents that each algorithm can solve with a success rate of 100%.

Map	None	RTC	Map	None	RTC	Map	None	RTC	Map	None	RTC
`Random`	35	47	`Empty`	18	82	`Warehouse`	17	84	`Game1`	5	67
`Room`	10	27	`Maze`	2	2	`City`	3	89	`Game2`	11	31

`Game1`, and `City`: the success rate improvements of the combination RTC are substantially larger than those of GR, T, and GC separately. This is so because, when a MAPF instance contains more than one class of symmetry conflicts, solving any class of symmetry conflicts with standard CBS splitting could result in unacceptable runtimes. Thus, CBSH with only one of the reasoning techniques does not solve many instances within the runtime limit, while CBSH with all techniques does.

Scalability To show the scalability of CBSH with and without our reasoning techniques, instead of using the instances described in Table 4.1, we run None and RTC on the same six maps with the number of agents increasing by one at a time, starting from 2. We report the largest number of agents that each algorithm can solve with a success rate of 100% in Table 4.3. We see that, except for map `Maze`, RTC dramatically improves the scalability of None, especially on large maps with lots of open space, such as maps `Game1` (with an improvement of 13 times) and `City` (with an improvement of 30 times).

Sizes of CTs Figure 4.20 compares the number of expanded CT nodes of None and RTC. It shows that our reasoning techniques can reduce the sizes of CTs by up to four orders of magnitude. Among the 890 instances solved by at least one of the algorithms, RTC performs worse than None on only 24 (= 2% of) instances and beats it on 782 (= 88% of) instances.

Runtime Overhead Table 4.4 reports the runtime overhead of generalized rectangle and corridor reasoning in RTC. The runtime overhead of generalized rectangle reasoning mainly stems from manipulating MDDs because it has to search the MDDs twice, once for finding the generalized rectangle and once for classifying the rectangle conflicts. However, both searches are

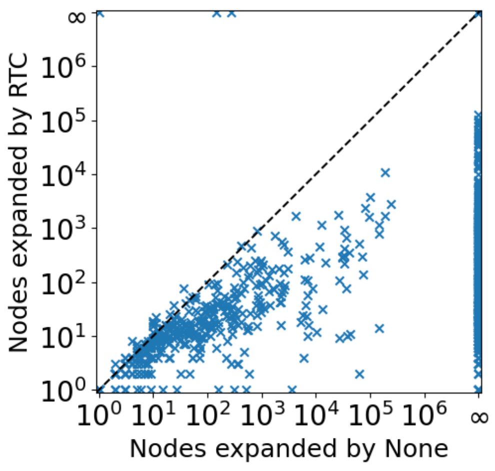


Figure 4.20: Numbers of expanded CT nodes of None and RTC. If an instance is not solved within the runtime limit of one minute, we set the number of its expanded CT nodes to infinity. Among the 1,200 instances, 310 instances are solved by neither algorithm; 418 instances are solved by RTC but not by None; and only 3 instances are solved by None but not by RTC. Among the 469 instances solved by both algorithms, RTC expands fewer CT nodes than None for 364 instances, the same number of CT nodes for 84 instances, and more CT nodes for only 21 instances.

Table 4.4: Percentages of runtimes of RTC spent on generalized rectangle and corridor reasoning. The runtime overhead of target reasoning is negligible and thus not reported here.

Map	Rectangle	Corridor	Map	Rectangle	Corridor
`Random`	3.62%	10.86%	`Empty`	5.79%	0.30%
`Warehouse`	1.26%	5.69%	`Game1`	1.73%	1.80%
`Room`	2.42%	30.12%	`Maze`	0.14%	0.57%
`City`	1.12%	0.98%	`Game2`	6.32%	8.52%

relatively fast, and, as a result, the overall runtime overhead of generalized rectangle reasoning is manageable, i.e., always less than 7% in Table 4.4. The runtime overhead of generalized corridor reasoning mainly stems from calculating $t_i(x)$ and $t'_i(x)$, as each of them, in our implementation, is a state-time A* search. We see that, on most maps, this overhead is small. But there are some maps, such as `Random` and `Room`, where the runtime overhead is more than 10%. Overall, the runtime overhead pays off in Figures 4.18 and 4.19 due to the effectiveness of the symmetry-breaking constraints for reducing the sizes of CTs.

Frequencies of Symmetries Table 4.5 reports how often RTC uses each reasoning technique to expand CT nodes, which also indicates how often different conflicts occur on different maps.

Table 4.5: Conflict distributions of RTC. “Rectangle”, “Target”, “Corridor”, and “Vertex/Edge” represent the percentages of CT nodes expanded by resolving generalized rectangle, target, generalized corridor, and vertex/edge conflicts, respectively.

Map	Expanded CT nodes	Rectangle	Target	Corridor	Vertex/Edge
`Random`	25,840	6.528%	54.391%	10.812%	28.269%
`Empty`	17,946	9.016%	61.856%	0.016%	29.112%
`Warehouse`	959	4.745%	55.579%	10.337%	29.339%
`Game1`	535	7.776%	50.851%	10.901%	30.472%
`Room`	8,848	3.443%	10.135%	55.036%	31.386%
`Maze`	30	0.000%	2.556%	44.315%	53.129%
`City`	401	6.183%	48.422%	5.364%	40.031%
`Game2`	345	2.400%	11.768%	67.998%	17.834%

Clearly, generalized rectangle conflicts are more frequent on maps with more open space. An extreme case is map `Maze`, where RTC does not branch on any rectangle conflicts as there is no open space on this map. Target conflicts happen frequently on all maps for two reasons: one is that we always choose to resolve target conflicts first, and the other is that the likelihood of a target conflict happening is high given the high density of the agents in our instances and regardless of the structures of the maps. The only exception is map `Maze` because most target conflicts are classified as corridor-target conflicts by generalized corridor reasoning there. Corridor conflicts are detected on all maps and frequent on maps with obstacles. Thanks to pseudo-corridor reasoning, we find many corridor conflicts not only on maps with many corridors, such as maps `Random`, `Warehouse`, `Room`, and `Maze`, but also on maps with few or even no corridors, such as maps `Empty`, `Game1`, `City`, and `Game2`. Generalized rectangle, target, and generalized corridor conflicts account for approximately 70% of conflicts used to expand CT nodes on many of the maps. Together with the efficiency of our reasoning techniques and the effectiveness of our symmetry-breaking constraints, this high frequency results in the gains that we see in Figures 4.18 to 4.20.

Two-Agent Analysis An interesting question about our symmetry reasoning techniques is whether the generalized rectangle, target, and generalized corridor reasoning techniques find all pairwise symmetries in MAPF. To answer this question, we design a two-agent experiment. Recall that CBSH2 solves a two-agent sub-MAPF instance for each pair of conflicting agents at each

Table 4.6: Numbers of expanded CT nodes of None and RTC when solving two-agent MAPF instances. The numbers in column $> n$ represent the percentages of instances that are solved by expanding more than n CT nodes.

Map	Agents	Algorithm	> 1	> 2	> 9	> 99	> 999
`Random`	100	None	13.577%	5.852%	0.754%	0.215%	0.055%
		RTC	1.748%	0.806%	0.428%	0.031%	0.014%
`Empty`	200	None	8.997%	8.262%	6.892%	5.583%	4.689%
		RTC	2.808%	0.588%	0.006%	0.001%	0.000%
`Warehouse`	200	None	20.896%	14.237%	1.049%	0.484%	0.297%
		RTC	0.948%	0.187%	0.029%	0.011%	0.011%
`Game1`	300	None	18.952%	4.477%	3.159%	2.926%	2.813%
		RTC	10.150%	0.502%	0.060%	0.050%	0.000%
`Room`	100	None	49.291%	28.169%	4.031%	0.007%	0.000%
		RTC	14.517%	3.283%	0.123%	0.003%	0.000%
`Maze`	20	None	96.886%	93.426%	69.550%	46.713%	16.609%
		RTC	6.484%	6.180%	1.418%	1.216%	0.405%
`City`	400	None	18.732%	7.338%	5.146%	3.531%	3.203%
		RTC	5.756%	0.189%	0.029%	0.029%	0.029%
`Game2`	150	None	38.282%	6.180%	0.116%	0.097%	0.093%
		RTC	18.930%	2.422%	0.024%	0.024%	0.000%

CT node to compute the WDG heuristic (see Section 3.3.3).[6] Here, we record the number of CT nodes expanded by None and RTC for solving such two-agent sub-MAPF instances and report the results in Table 4.6. Compared to None, RTC expands substantially fewer CT nodes for solving the two-agent sub-MAPF instances. Impressively, RTC solves up to 99% of two-agent sub-MAPF instances by expanding only a single CT node. Even in the worst case, it solves 81% of two-agent sub-MAPF instances by expanding only a single CT node. Except for map `Maze`, RTC expands 10 or more CT nodes for fewer than 0.5% of instances. As for map `Maze`, the percentage is less than 1.5%. Therefore, RTC identifies and efficiently eliminates most of the pairwise symmetries in MAPF.

Conflict Prioritization In order to show the effectiveness of our proposed conflict prioritization strategy (i.e., for conflicts of the same cardinality, we first choose target conflicts, then generalized corridor conflicts, generalized rectangle conflicts, and finally vertex and edge conflicts), we create

[6]In practice, CBSH2 does not do so for all pairs of agents, as it uses a memoization technique to avoid solving the same two-agent sub-MAPF instance at different CT nodes more than once.

Table 4.7: Numbers of instances solved by rRCT and RTC within one minute. The total number of instances for each map is 150 (i.e., 6 different numbers of agents with 25 instances for each number).

Map	rRTC	RTC	Map	rRTC	RTC	Map	rRTC	RTC	Map	rRTC	RTC
Random	97	113	Empty	116	126	Warehouse	90	118	Game1	114	119
Room	90	111	Maze	46	49	City	128	133	Game2	106	118

a strawman algorithm rRTC that chooses randomly among all conflicts of the same cardinality and compare it with RTC in Table 4.7. On all maps, RTC solves 3.9%-31.1% more instances than rRTC, which clearly shows that our fine-grained conflict prioritization strategy that sorts conflicts according to their cardinalities and then breaks ties according to their symmetry types is better than the existing conflict prioritization strategy [27] that sorts the conflicts according to their cardinalities only.

4.7.3 Empirical Comparison with Mutex Propagation

Our symmetry reasoning techniques are manually designed. In another line of research, we propose to use mutex propagation [208] to autonomously identify all cardinal symmetry conflicts and resolve them with a pair of vertex constraint sets. MDDs essentially capture the reachability information for single agents and thus resemble planning graphs in classical planning [25]. Therefore, we add mutex propagation on top of MDDs to capture the reachability information for pairs of agents. Two MDD nodes for two agents are *mutex* iff all pairs of their paths that visit the two MDD nodes are in conflict. So, two agents have a cardinal (symmetry) conflict iff the goal nodes of their MDDs are mutex. Given two agents with a cardinal (symmetry) conflict, we find two MDD node sets, each consisting of the MDD nodes of one agent that are mutex with the goal MDD node of the other agent, and use them to generate two constraint sets for branching in CBS. Therefore, the mutex propagation technique can automatically identify all cardinal symmetry conflicts and resolve them. See [208] for more details.

To compare our manually designed symmetry reasoning techniques and mutex propagation, we test four versions of CBSH: (1) CBSH with mutex propagation only (denoted **M**); (2) CBSH with

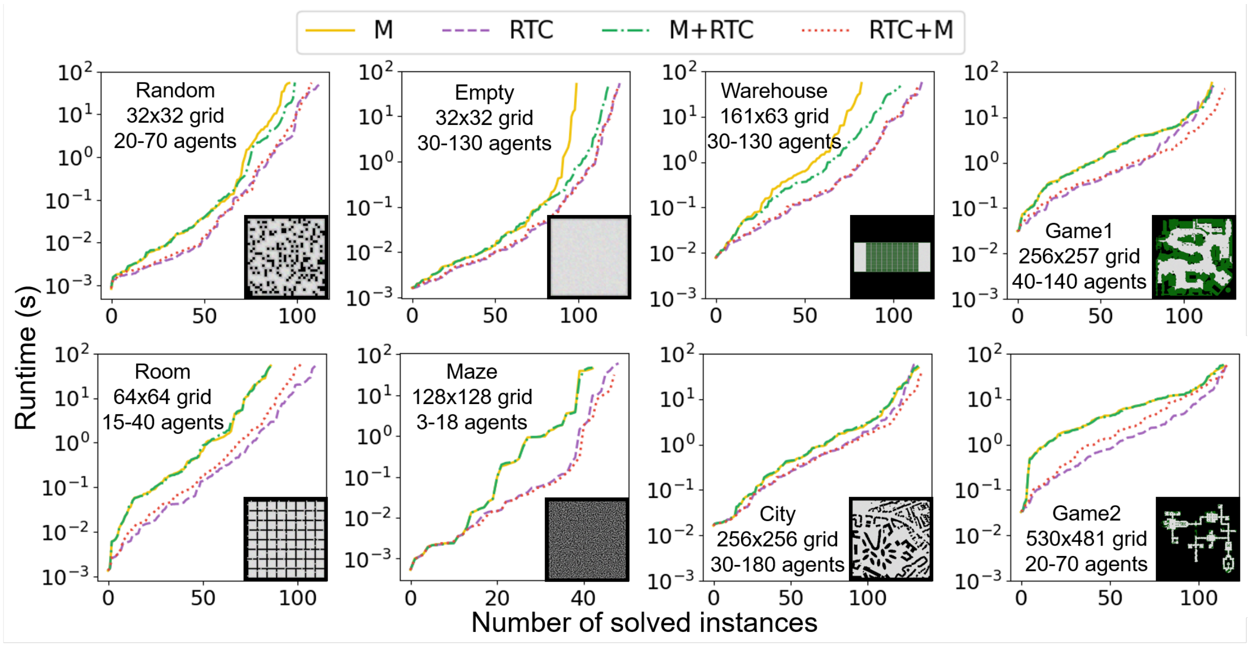


Figure 4.21: Runtime distributions of CBSH with our symmetry reasoning techniques and mutex propagation.

our symmetry reasoning techniques only (denoted **RTC**); (3) CBSH with both techniques where, for each vertex/edge conflict, we always perform mutex propagation first and then perform our symmetry reasoning only if mutex propagation fails to identify this conflict as a cardinal symmetry conflict (denoted **M+RTC**); and (4) CBSH with both techniques where, for each vertex/edge conflict, we always perform our symmetry reasoning first and then perform mutex propagation only if our symmetry reasoning fails to identify this conflict as a symmetry conflict (denoted **RTC+M**).

Figure 4.21 reports the runtime distributions of these four algorithms. First, our symmetry reasoning alone always performs better than mutex propagation alone. One of the reasons is that mutex propagation only reasons about cardinal symmetry conflicts but ignores semi- and non-cardinal symmetry conflicts. Therefore, when we apply our symmetry reasoning techniques after mutex propagation, M+RTC performs better than M in many cases. RTC always performs better than M+RTC for two reasons: Mutex propagation has a larger runtime overhead than our symmetry reasoning techniques and uses vertex constraint sets to resolve target and corridor-target conflicts, which are less effective than the length constraints that our symmetry reasoning techniques use. The performance of RTC and RTC+M is competitive. In some cases, RTC is slightly better than

RTC+M because RTC+M has a larger runtime overhead. In other cases, RTC is slightly worse than RTC+M because mutex propagation can identify some cardinal symmetry conflicts that our symmetry reasoning techniques fail to identify. The negligible improvement of RTC+M over RTC also implies that, although we developed our symmetry reasoning techniques by enumerating possible pairwise symmetries manually, RTC is able to identify most of the cardinal symmetry conflicts. In summary, our symmetry reasoning techniques are more effective than mutex propagation on the instances used. Their combination does not outperform our symmetry-reasoning techniques alone.

4.8 Combining Symmetry Breaking with the WDG Heuristic

CBSH2 uses CBSH to solve a two-agent sub-MAPF instance for each pair of agents in the original MAPF instance to generate informed heuristic guidance for the high-level search of CBS. We already showed in Table 4.6 that symmetry reasoning can significantly reduce the number of CT nodes expanded by CBSH when solving two-agent sub-MAPF instances — and thus reduce its runtime. Now, we show that symmetry reasoning can also reduce the number of CT nodes expanded by CBSH2 when solving the original MAPF instance — and thus reduce its runtime. In addition, we add the bypassing conflicts (see Section 2.3.2.2) to CBSH2, which resolves some semi- and non-cardinal conflicts without branching. We call the resulting algorithm CBSH2-RTC, which uses symmetry reasoning both in the main routine of CBSH2 and in the two-agent sub-MAPF solver CBSH of CBSH2, and show its pseudo-code in Algorithm 4.6. Compared to vanilla CBS shown in Algorithm 2.1, CBSH2-RTC contains four improvement techniques, namely prioritizing conflicts [Line 11], symmetry reasoning [Line 12], bypassing conflicts [Lines 23 and 26], and using the WDG heuristic [Lines 7 to 10].

Furthermore, we make two changes to the WDG heuristic. First, we need to modify the lazy heuristic technique used by the WDG heuristic slightly because, when adding a length constraint $l_2 \leq t$ to a CT node, we might need to replan paths for more than one agent, which may make the

Algorithm 4.6: CBSH2-RTC for solving MAPF optimally.

```
Input: MAPF instance $(G,A)$
1  Generate root CT node $R$;                    // Same as CBS on Lines 1 to 3 in Algorithm 2.1
2  OPEN $\leftarrow \{R\}$;
3  while OPEN $\neq \emptyset$ do
4      $N \leftarrow \arg\min_{N \in \text{OPEN}} f(N)$;      // Break ties by CT nodes with fewer conflicts
5      OPEN $\leftarrow$ OPEN $\setminus \{N\}$;
6      if $N.conflicts = \emptyset$ then return $N.plan$;
7      if the WDG heuristic for N has not yet been computed then
8          COMPUTEWDGHEURISTIC($N$);                  // WDG heuristic from Section 3.3
9          OPEN $\leftarrow$ OPEN $\cup \{N\}$;
10         continue;
11     CONFLICTPRIORITIZATION($N.conflicts$);  // Prioritizing conflicts from Section 2.3.2.1
12     SYMMETRYREASONING($N.conflicts$);       // Symmetry reasoning using Algorithm 4.5
13     $conflict \leftarrow$ a conflict in $N.conflicts$ with the highest priority;
14     Generate the two constraint sets $C_1$ and $C_2$ for resolving $conflict$;
15     $children \leftarrow \emptyset$;
16     for $i = 1,2$ do
17         $N' \leftarrow$ a copy of $N$;
18         $N'.constraints \leftarrow N.constraints \cup C_i$;
19         for $a_j \in A : N'.plan[a_j]$ violates $C_i$ do
20             $N'.plan[a_j] \leftarrow$ LOWLEVELSEARCH$(a_j, G, N')$;
21             if $N'.plan[a_j]$ does not exist then Go to Line 16;
22         $N'.conflicts \leftarrow$ all conflicts in $N'.plan$;
23         if $cost(N') = cost(N) \wedge |N'.conflicts| < |N.conflicts|$ then
24             $N.plan \leftarrow N'.plan$;
25             $N.conflicts \leftarrow N'.conflicts$;
26             Go to Line 6;                     // Bypassing conflicts from Section 2.3.2.2
27         $children \leftarrow children \cup \{N'\}$;
28     for $N' \in children$ do OPEN $\leftarrow$ OPEN $\cup \{N'\}$;
29 return "No Solution";
```

first item in Equation (3.1) inadmissible. Therefore, we delete the first item and use the following equation instead:

$$h_1(N') = \max\{cost(N) + h(N) - cost(N'), 0\}, \tag{4.9}$$

Second, when building the weighted pairwise dependency graph $G_{WD} = (V_D, E_D, W_D)$, we do not use the merging MDDs technique to determine whether an edge belongs to E_D. Instead, we run

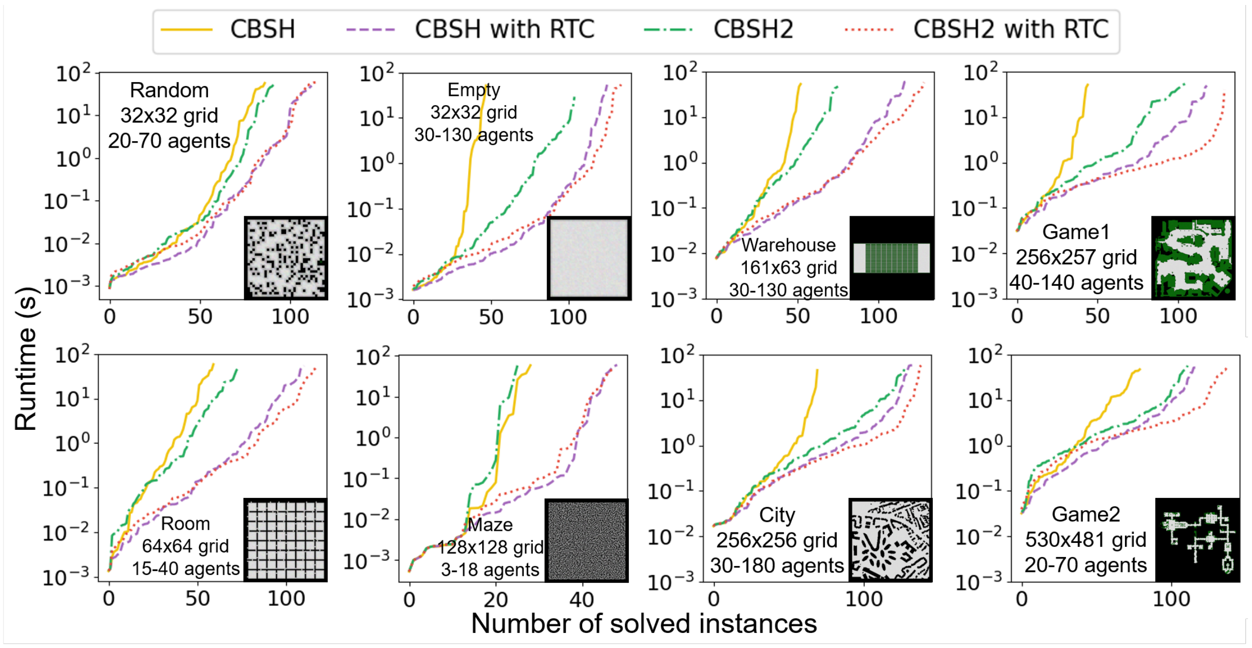


Figure 4.22: Runtime distributions of CBSH and CBSH2 with and without RTC.

CBSH-RTC for every pair of agents whose paths are in conflict to determine the existence of an edge and its weight simultaneously. This is so because CBSH-RTC, unlike CBSH, does not suffer from rectangle symmetry any longer and thus runs significantly faster than merging MDDs (as we show in Table 4.6, CBSH-RTC solves most of the two-agent sub-MAPF instances by expanding only one CT node).

4.8.1 Empirical Evaluation

Figure 4.22 shows the runtime distributions of CBSH and CBSH2 with and without symmetry reasoning. As expected, both RTC and CBSH2 outperform CBSH in most cases. In particular, RTC always performs better than CBSH2, which indicates that, although both symmetry reasoning and the heuristics used in CBSH2 reason about pairs of agents, RTC (that uses symmetry-breaking constraints to resolve symmetries directly) is more effective than CBSH2 (that relies on the heuristics to eliminate symmetries). Not surprisingly, CBSH2-RTC performs the best as it uses both symmetry-breaking constraints and informed heuristics.

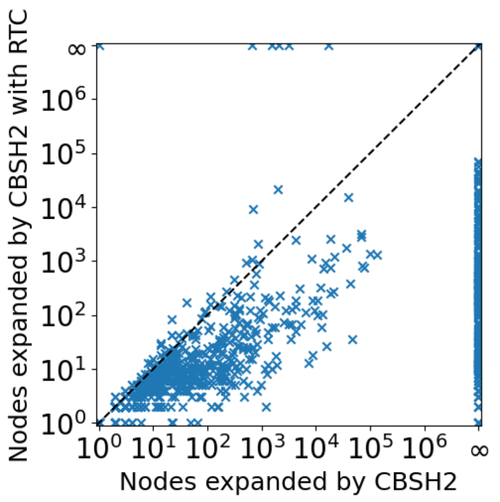


Figure 4.23: Numbers of CT nodes expanded by CBSH2 and CBSH2-RTC. If an instance is not solved within the runtime limit, we set its number of expanded nodes to infinity. Among the 1,200 instances, 239 instances are solved by neither algorithm; 241 instances are solved by CBSH2-RTC but not by CBSH2; and only 6 instances are solved by CBSH2 but not by CBSH2-RTC. Among the 714 instances solved by both algorithms, CBSH2-RTC expands fewer CT nodes than CBSH2 for 572 instances, the same number of CT nodes for 104 instances, and more CT nodes for only 38 instances.

In order to show that the gain of adding symmetry reasoning to CBSH2 is not just due to it speeding up CBSH when solving the two-agent sub-MAPF instances, we plot the number of CT nodes expanded by CBSH2 with and without symmetry reasoning in Figure 4.23. We see that symmetry reasoning can reduce the size of the CTs of CBSH2 by up to three orders of magnitude. Among the 961 instances solved by at least one of the algorithms, CBSH2 with RTC performs worse than CBSH2 only on 44 (= 5% of) instances and beats it on 676 (= 70% of) instances.

4.9 Summary

In this chapter, we provided evidence that symmetries are one of the reasons why MAPF is hard. We showed that symmetry conflicts arise extremely frequently in MAPF. Rectangle symmetry occurs when the paths of two agents must cross and have many equivalent ways of doing so. Generalized rectangle reasoning applies to planar graphs, which represent all MAPF instances on 2D maps in practice. Generalized corridor and target reasoning avoids symmetries resulting from multiple wait actions. Both of them apply to graphs in general and important for one of the main commercial uses of MAPF algorithms, namely routing robots in automated warehouses. We showed that our symmetry reasoning techniques scale up CBSH by up to thirty times and reduce its number of expanded CT nodes by up to four orders of magnitude. They significantly outperform both CBSH2 and CBSH with mutex propagation.

4.10 Extensions

As we introduced in Section 2.2.1, state-of-the-art optimal MAPF algorithms all search the conflict-resolution space and thus, not surprisingly, suffer from the issue of pairwise symmetries. Motivated by our work, researchers have developed a sequence of symmetry-reasoning techniques for speeding up BCP, a state-of-the-art optimal MAPF algorithm based on integer linear programming (introduced in Section 2.2.1.2) [96, 95, 97]. Our symmetry-reasoning techniques can be directly applied to Lazy CBS, a state-of-the-art optimal MAPF algorithm based on constraint programming (introduced in Section 2.2.1.2) and speed it up as well [95]. In Chapter 5, we will show that symmetry reasoning can also speed up bounded-suboptimal CBS variants.

Symmetries reduce the efficiency of optimal MAPF algorithms for not only classic MAPF problems but also generalized MAPF problems. Therefore, our symmetry-reasoning techniques can also significantly speed up CBS variants for generalized MAPF problems, such as k-robust MAPF [39], MAPF with precedence constraints [210], and MAPF with agents of different lengths [40].

Although mutex propagation does not outperform our symmetry-reasoning techniques in our experiments, it has the advantage that it can be easily adapted to many generalized MAPF problems and can easily detect new symmetries that our symmetry-reasoning techniques have not been designed to discover. For example, we have shown that mutex propagation can speed up the SAT-based optimal algorithm SAT-MDD for classic MAPF [173, 212], the CBS-based algorithm CBICS for MAPF with non-unit traversal times [187], and the CBS-based algorithm MC-CBS for MAPF with large agents [211].

Chapter 5

Speeding up Bounded-Suboptimal CBS via Inadmissible Heuristics

Many real-world applications of MAPF involve hundreds of agents but have only limited computing resources available for planning. Therefore, we study bounded-suboptimal MAPF algorithms in this chapter, that trade off solution quality for runtime. Bounded-suboptimal algorithms are algorithms that guarantee to find solutions with costs no more than a user-specified factor away from optimal. The current state-of-the-art bounded-suboptimal MAPF algorithms, just like the current state-of-the-art optimal MAPF algorithms, are CBS variants or employ ideas similar to CBS. Among all CBS variants, ECBS runs the fastest [16]. Its bounded suboptimality is achieved by replacing the best-first search on the high and low levels of CBS with focal search [134]. Focal search uses an admissible heuristic for bounding the solution cost and another heuristic for determining the distance of nodes to the goal nodes.

In this chapter, we first demonstrate that ECBS becomes inefficient if these heuristics are negatively correlated. We then propose a new bounded-suboptimal variant of CBS, called Explicit Estimation CBS (EECBS), to overcome this issue. EECBS replaces focal search with Explicit Estimation Search [174] on the high level and uses online learning [176] to learn an informed but potentially inadmissible heuristic to guide the high-level search. ECBS and EECBS differ from CBS only in the node-selection rules used in their high- and low-level searches. Hence, the ideas behind many improvements of CBS, such as prioritizing conflicts (see Section 2.3.2.1) and bypassing conflicts (see Section 2.3.2.2) as well as our proposed admissible heuristics and symmetry-reasoning techniques in Chapters 3 and 4, might improve ECBS and EECBS as well, and we show how they can be adapted to them. Finally, we empirically evaluate how each improvement affects the performance of ECBS and EECBS, finding that their combination is best. EECBS with the improvements runs significantly faster than ECBS as well as BCP-7 [95] and eMDD-SAT [172], two other state-of-the-art bounded-suboptimal MAPF algorithms.

This chapter closely follows [112].

5.1 Background: Enhanced CBS (ECBS)

Focal search is a bounded-suboptimal search algorithm based on A_ε^* [134]. It maintains two lists of nodes: OPEN and FOCAL. OPEN is the regular open list of A*, sorted according to an admissible cost function f.[1] Let $best_f$ be the node with the minimum f-value in OPEN and w be a user-specified suboptimality factor. FOCAL contains those nodes n in OPEN for which $f(n) \leq w \cdot f(best_f)$, sorted according to a function d that estimates the *distance-to-go*, i.e., the number of hops from node n to a goal node. Focal search always expands the node with the minimum d-value in FOCAL. Since $f(best_f)$ is a lower bound on the optimal solution cost c^*, focal search guarantees that the cost of the found solution is at most $w \cdot c^*$.

Enhanced CBS (ECBS) [16] is a bounded-suboptimal variant of CBS that uses focal search with the same suboptimality factor w on both the high and low levels. Given a CT node N, the low level of ECBS finds a bounded-suboptimal path for agent a_i that satisfies $N.constraints$ and minimizes the number of conflicts with the paths of other agents $\{N.plan[a_j] \mid a_j a_j \in A \setminus \{a_i\}\}$. It achieves this by using a focal search with $f(n)$ being the standard $f(n) = g(n) + h(n)$ of A* and $d(n)$ being the number of conflicts with the paths of other agents. When it finds a solution, it returns not only the path but also the minimum f-value $f^i_{\min}(N)$ in the low-level open list, which is a lower bound on the length of the shortest path for agent a_i. For clarity, we denote the length

[1] We say that a cost function is admissible iff it is provably a lower bound on the optimal cost and inadmissible otherwise. If $f = g + h$, then f being admissible is equivalent to h being admissible.

of the shortest path for agent a_i as $f^i_{opt}(N)$, although this value is in general unknown during the search. The low level of ECBS ensures that the found path $N.plan[a_i]$ and the returned lower bound $f^i_{\min}(N)$ satisfy

$$f^i_{\min}(N) \leq f^i_{opt}(N) \leq length(N.plan[a_i]) \leq w \cdot f^i_{\min}(N). \tag{5.1}$$

Unlike usual bounded-suboptimal searches, the focal search used on the low level of ECBS is to speed up the high-level search, instead of the low-level search itself, as it tries to reduce the number of conflicts that need to be resolved by the high-level search.

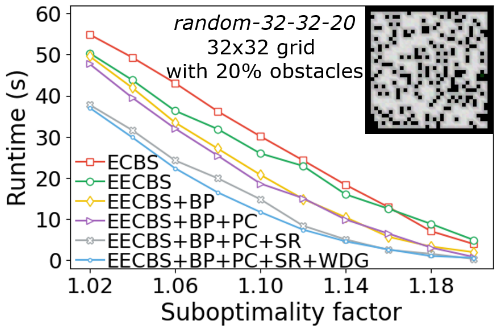
The high level of ECBS uses a modified focal search. OPEN is the regular open list of A*, which sorts its CT nodes N according to $lb(N) = \sum_{a_i \in A} f^i_{\min}(N)$, which is a lower bound on the minimum cost of the solutions below CT node N. From Inequality 5.1, we know that

$$lb(N) \leq cost(N) \leq w \cdot lb(N). \tag{5.2}$$

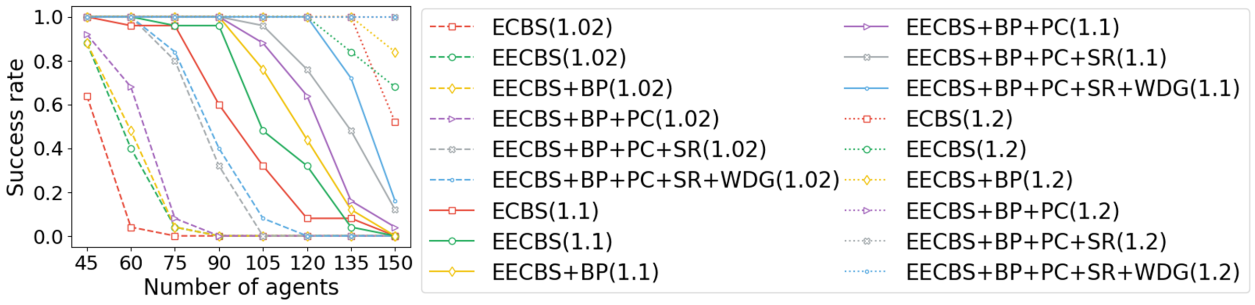
Let $best_{lb}$ be the CT node in OPEN with the minimum lb-value. FOCAL contains those CT nodes N in OPEN for which $cost(N) \leq w \cdot lb(best_{lb})$, sorted according to the number of conflicts $h_c(N)$ of the paths in $N.plan$, roughly indicating the distance-to-go for the high-level search, i.e., the number of splitting actions required to find a solution below CT node N. Since $lb(best_{lb})$ is a lower bound on the optimal sum of costs, the cost of any CT node in FOCAL is no larger than w times the optimal sum of costs. Thus, once a solution is found, its sum of costs is also no larger than w times the optimal sum of costs.

5.2 Explicit Estimation CBS (EECBS)

We first analyze the behavior of the high-level focal search of ECBS. We then present our new algorithm EECBS, which uses Explicit Estimation Search on the high level and online learning to estimate the sum of cost of the solution that can be found under each CT node.



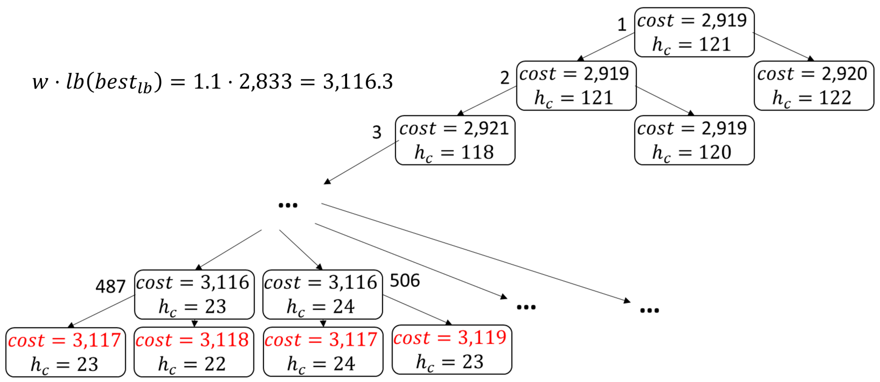
(a) Average runtimes of the algorithms over 200 instances. The runtime limit of one minute is included in the average for each instance not solved within the runtime limit.



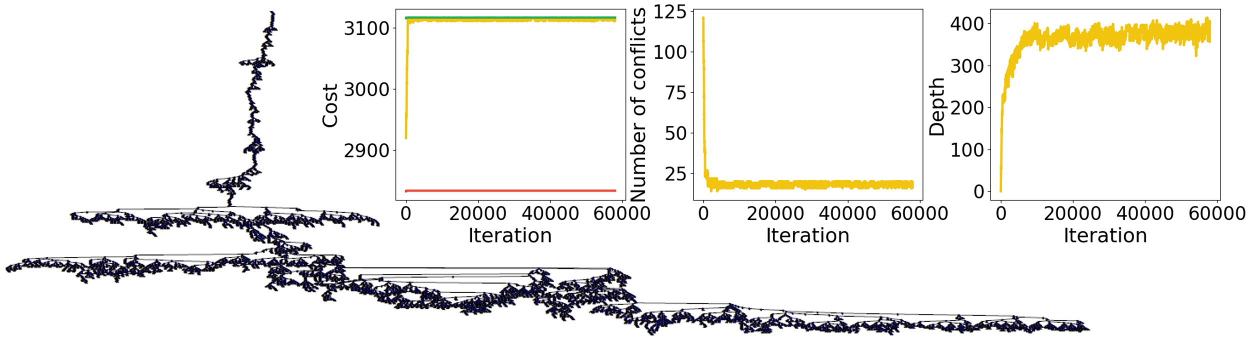
(b) Success rates (i.e., percentages of solved instances within the runtime limit of one minute) of the algorithms with $w = 1.02, 1.10$, and 1.20.

Figure 5.1: Performance of ECBS and EECBS with different improvement techniques. BP, PC, SR, and WDG are short for bypassing conflicts, prioritizing conflicts, symmetry reasoning, and using the WDG heuristic, respectively.

To evaluate the effectiveness of each technique that we introduce, we test it on 200 instances from the MAPF benchmarks [163] with a runtime limit of one minute per instance and report the reports in Figure 5.1. In particular, we use map `random-32-32-20`, a 32×32 four-neighbor grid with 20% randomly blocked cells, shown in Figure 5.1a, with the number of agents varying from 45 to 150 in increments of 15. We use the "random" scenarios from the benchmarks, yielding 25 instances for each number of agents. We vary the suboptimality factor from 1.02 to 1.20 in increments of 0.02. For now, we focus only on this random map, but later, in Section 5.4, we will evaluate our algorithms on additional maps.


$w \cdot lb(best_{lb}) = 1.1 \cdot 2{,}833 = 3{,}116.3$
1
$cost = 2{,}919$ $h_c = 121$
2
$cost = 2{,}919$ $h_c = 121$
$cost = 2{,}920$ $h_c = 122$
3
$cost = 2{,}921$ $h_c = 118$
$cost = 2{,}919$ $h_c = 120$
487
$cost = 3{,}116$ $h_c = 23$
$cost = 3{,}116$ $h_c = 24$
506
$cost = 3{,}117$ $h_c = 23$
$cost = 3{,}118$ $h_c = 22$
$cost = 3{,}117$ $h_c = 24$
$cost = 3{,}119$ $h_c = 23$

(a) CT of ECBS with suboptimality factor $w = 1.1$ after 506 iterations (i.e., after expanding 506 CT nodes).



(b) CT of ECBS after 5,000 iterations and three charts that plot the cost, the number of conflicts h_c, and the depth of the CT node selected for expansion at each iteration. The red and green lines in the first chart are the lower bound $lb(best_{lb})$ and the suboptimality bound $w \cdot lb(best_{lb})$ at each iteration, respectively.

Figure 5.2: Performance of ECBS on a hard MAPF instance.

5.2.1 Limitations of ECBS

Figure 5.2 shows the typical behavior of ECBS on a hard MAPF instance, revealing two drawbacks of its high-level focal search. The first drawback is that, when selecting CT nodes for expansion, ECBS considers only the distance-to-go and requires the cost of the selected CT node N to be within suboptimality bound $w \cdot lb(best_{lb})$ but ignores the fact that the cost of the solution below CT node N is likely to be larger than $cost(N)$ and thus could also be larger than the suboptimality bound. In the example of Figure 5.2a, ECBS first keeps expanding CT nodes roughly along a branch of the CT (i.e., CT nodes $1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots \rightarrow 487$). So, the cost of the CT node selected for expansion keeps increasing and its h_c-value keeps decreasing until ECBS expands a CT node N both of whose child CT nodes do not qualify for inclusion in FOCAL (i.e., CT node 487). As a result, ECBS then expands a neighboring CT node N' whose cost is slightly smaller than $cost(N)$

w		1.04	1.08	1.12	1.16	1.20
ECBS	Δlb	0.63	0.63	0.64	0.52	0.56
EECBS	Δlb	5.21	2.32	1.40	0.71	0.64
	Cleanup%	39.6%	14.4%	11.1%	0.7%	0.0%

Table 5.1: Lower-bound improvement Δlb, i.e., the value of $lb(best_{lb})$ when the algorithm terminates minus the lb-value of the root CT node. "Cleanup%" is the percentage of expanded CT nodes that are selected from CLEANUP.

and whose h_c value is slightly larger than $h_c(N)$. It keeps expanding CT nodes below CT node N', but, after several iterations, expands another CT node both of whose child CT nodes do not qualify for inclusion in FOCAL (i.e., CT node 506). This pattern repeats numerous times, as shown in the three charts in Figure 5.2b. As a result, ECBS is stuck in part of the CT and never gets a chance to explore other parts of the CT, as shown in the left diagram in Figure 5.2b. This thrashing behavior, in which the negative correlation of $h_c(N)$ and $cost(N)$ causes focal search to abandon the child CT nodes of expanded CT nodes repeatedly, was noticed by Thayer et al. [175].

The second drawback is that the lower bound $lb(best_{lb})$ of ECBS rarely increases, as shown by the flat red line in the first chart of Figure 5.2b. This is so because, when expanding a CT node N, ECBS resolves a conflict and adds new constraints to the generated child CT nodes. So, the lb-values of the child CT nodes tend to be equal to or larger than $lb(N)$ while the h_c-values of the child CT nodes tend to be smaller than $h_c(N)$. As a result, $best_{lb}$ tends to have a large h_c value. This results in $best_{lb}$ being expanded only when FOCAL is almost empty. Since there are many CT nodes of the same cost, ECBS rarely empties FOCAL, and thus its lower bound $lb(best_{lb})$ rarely increases. More statistics can be found in Table 5.1 (second row). Therefore, if the optimal sum of costs is not within the initial suboptimality bound $w \cdot lb(R)$ (where R is the root CT node), ECBS can have difficulty finding a solution within a reasonable amount of time. While this problematic behavior is similar to that in the first drawback, in that both involve a negative correlation of node values, it is subtly different as it involves the lower bound rather than the cost.

5.2.2 Explicit Estimation Search (EES)

Explicit Estimation Search (EES) [174] is a bounded-suboptimal search algorithm designed in part to overcome the poor behavior of focal search. It introduces a third function $\hat{f}$ that estimates, potentially inadmissibly, the cost of the best solution in the subtree below a given node. EES combines estimates of this solution cost and the distance-to-go to predict the expansion of which nodes will lead most quickly to a solution for the user-specified suboptimality factor. If the nodes of interest are not within the current suboptimality bound, then it expands the unexpanded node with the minimum f-value to increase the suboptimality bound. Formally, EES with a user-specified suboptimality factor w maintains three lists of nodes: CLEANUP, OPEN, and FOCAL.

- CLEANUP is the regular open list of A*, sorted according to an admissible cost function f. Let $best_f$ be the node with the minimum f-value in CLEANUP.
- OPEN is also another regular open list of A*, sorted according to a more informed but potentially inadmissible cost function $\hat{f}$. Let $best_{\hat{f}}$ be the node with the minimum $\hat{f}$-value in OPEN.
- FOCAL contains those nodes n in OPEN for which $\hat{f}(n) \leq w \cdot \hat{f}(best_{\hat{f}})$, sorted according to a distance-to-go function d. $\hat{f}(best_{\hat{f}})$ is an estimate of the cost of an optimal solution, so EES suspects that expanding the nodes in FOCAL can lead to solutions that are no more than a factor of w away from optimal. Let $best_d$ be the node with the minimum d-value in FOCAL.

When selecting nodes for expansion, EES first considers node $best_d$, as expanding nodes with nearby goal nodes should lead to a goal node fast. To guarantee bounded suboptimality, EES selects node $best_d$ for expansion only if $f(best_d) \leq w \cdot f(best_f)$. If node $best_d$ is not selected, EES next considers node $best_{\hat{f}}$, as it suspects that node $best_{\hat{f}}$ lies along a path to an optimal solution. To guarantee bounded suboptimality, EES selects node $best_{\hat{f}}$ for expansion only if $f(best_{\hat{f}}) \leq w \cdot f(best_f)$. If neither node $best_d$ nor node $best_{\hat{f}}$ is selected, EES selects node $best_f$, which can raise the suboptimality bound $w \cdot f(best_f)$, allowing EES to consider nodes $best_d$ or $best_{\hat{f}}$ in the next iteration.

5.2.3 Explicit Estimation CBS (EECBS)

We form EECBS by replacing focal search with EES on the high level of ECBS. Formally, the high level of EECBS maintains three lists of CT nodes: CLEANUP, OPEN, and FOCAL.

- CLEANUP is the regular open list of A*, sorted according to the lower bound function lb. $best_{lb}$ denotes the CT node with the smallest lb-value in CLEANUP.
- OPEN is another regular open list of A*, sorted according to a potentially inadmissible cost function $\hat{f}$, which estimates the minimum cost of the solutions below a CT node. We use $\hat{f}(N) = cost(N) + \hat{h}(N)$, where $\hat{h}(N)$ is the cost-to-go heuristic introduced in Section 5.2.4. $best_{\hat{f}}$ denotes the CT node with the smallest $\hat{f}$-value in OPEN.
- FOCAL contains those CT nodes N in OPEN for which $\hat{f}(N) \leq w \cdot \hat{f}(best_{\hat{f}})$, sorted according to the distance-to go function h_c. $best_{h_c}$ denotes the CT node with the smallest h_c-value in FOCAL.

EECBS selects CT nodes for expansion from these three lists using the SELECTNODE function:

1. if $cost(best_{h_c}) \leq w \cdot lb(best_{lb})$, then select $best_{h_c}$ (i.e., select from FOCAL);
2. else if $cost(best_{\hat{f}}) \leq w \cdot lb(best_{lb})$, then select $best_{\hat{f}}$ (i.e., select from OPEN);
3. else select $best_{lb}$ (i.e., select from CLEANUP).

Like EES, EECBS selects a CT node N for expansion only if its cost is within the current suboptimality bound, i.e.,

$$cost(N) \leq w \cdot lb(best_{lb}), \tag{5.3}$$

which guarantees bounded suboptimality. EECBS uses the same focal search as ECBS on its low level.

EECBS overcomes the first drawback from Section 5.2.1 by taking the potential cost increase below a CT node into consideration in Steps 1 and 2 of the SELECTNODE function and selecting a CT node N whose estimated cost $\hat{f}(N)$ is within the estimated suboptimality bound $w \cdot \hat{f}(best_{\hat{f}})$. It overcomes the second drawback by selecting the CT node with the minimum lb-value in Step 3 of the SELECTNODE function to raise the lower bound. Table 5.1 shows its empirical behavior in comparison with ECBS. Unlike for ECBS, whose lower-bound improvement is always around 0.6, the lower-bound improvement of EECBS increases as the suboptimality factor w decreases, as shown in Table 5.1(third row). The smaller w is, the less likely a solution is within the initial suboptimality bound and thus the more frequently EECBS selects CT nodes from CLEANUP, as shown in Table 5.1 (fourth row). Figure 5.1 compares the runtimes and success rates of ECBS and EECBS. As expected, EECBS (green lines) has smaller runtimes and larger success rates than ECBS (red lines). The improvement increases as w decreases.

5.2.4 Online Learning of the Cost-To-Go Heuristic

Our estimate of the minimum cost of the solutions below a given CT node N uses online learning since it does not require preprocessing and allows for instance-specific learning. Thayer et al. [176] present a method for learning the cost-to-go during search using the error experienced during node expansions. Consider a node n with an admissible cost-to-go heuristic h and a distance-to-go heuristic d. The error ε_d of the distance-to-go heuristic d, called *one-step distance error*, is defined as

$$\varepsilon_d(n) = d(bc(n)) - (d(n) - 1), \tag{5.4}$$

where $bc(n)$ is the *best child node* of node n, i.e., the child node with the smallest $\hat{f}$-value, breaking ties in favor of the child node with the smallest d-value. Similarly, the error ε_h of the cost function f, called *one-step cost error*, is defined as

$$\begin{aligned}
\varepsilon_h(n) &= f(bc(n)) - f(n) \\
&= g(bc(n)) + h(bc(n)) - g(n) - h(n) \\
&= h(bc(n)) - h(n) + (g(bc(n)) - g(n)) \\
&= h(bc(n)) - h(n) + c(n, bc(n)) \\
&= h(bc(n)) - ((h(n) - c(n, bc(n))), \qquad (5.5)
\end{aligned}$$

where $c(n, bc(n))$ is the cost of moving from node n to node $bc(n)$. These errors can be calculated after every node expansion. Thayer et al. [176] use a *global error model* that assumes that the distribution of one-step errors across the entire search space is uniform and can be estimated as an average of all observed one-step errors. Therefore, the search maintains a running average of the one-step errors observed so far, denoted by $\overline{\varepsilon}_d$ and $\overline{\varepsilon}_h$. Thayer et al. [176] prove that the true cost-to-go of node n can be approximated by

$$\hat{h}(n) = h(n) + \frac{d(n)}{1 - \overline{\varepsilon}_d(n)} \cdot \overline{\varepsilon}_h(n). \qquad (5.6)$$

We apply this method to EECBS. Since EECBS does not have an admissible cost-to-go heuristic h,[2] we define ε_d of CT node N as

$$\varepsilon_d(N) = h_c(bc(N)) - (h_c(N) - 1) \qquad (5.7)$$

and ε_h of CT node N as

$$\varepsilon_h(N) = cost(bc(N)) - cost(N). \qquad (5.8)$$

[2]The admissible heuristics introduced in Chapter 3 cannot be applied here because the plan of a CT node of EECBS consists of bounded-suboptimal paths (instead of optimal paths) whose sum of costs can be larger than the sum of costs of the optimal solution under the CT node.

Then,

$$\hat{h}(N) = \frac{h_c(N)}{1 - \overline{\varepsilon}_d(N)} \cdot \overline{\varepsilon}_h(N). \tag{5.9}$$

Thus, $\hat{h}(N)$ is linear in $h_c(N)$, indicating that, the larger the number of conflicts of a CT node, the higher the potential cost increments below the CT node could be.

Algorithm 5.1 shows the pseudo-code of the high-level search of EECBS. For now, we ignore Lines 4, 9-14, and 25-28 since they will be introduced in the next section. Compared to the high-level search of ECBS, EECBS changes the PUSHNODE and SELECTNODE functions [Lines Line 7,5,and 30] and adds an UPDATEONESTEPERRORS function at the end of each iteration to update the one-step errors [Line 31].

5.3 Bringing CBS Improvements to EECBS

We now show how we can incorporate recent CBS improvements into EECBS. We discuss these techniques one by one and, for each one, evaluate its effectiveness by adding it to the best version of EECBS so far and showing the resulting performance in Figure 5.1.

5.3.1 Bypassing Conflicts

Recall Section 2.3.2.2. In CBS, the paths of every CT node N are the shortest paths that satisfy $N.constraints$. However, in EECBS, the paths can be bounded-suboptimal. Therefore, we can resolve more conflicts with the bypassing conflicts (BP) technique in EECBS if we relax the conditions of accepting bypasses. Formally, when expanding a CT node N and generating its child CT nodes, EECBS replaces the paths of N with the paths of a child CT node N' and discards all generated child CT nodes iff

1. CT node N is not selected from CLEANUP,
2. the cost of every path of CT node N' is within the suboptimality bound of the corresponding agent in CT node N, i.e., $\forall a_i \in A\, length(N'.plan[a_i]) \leq w \cdot f^i_{\min}(N)$,

Algorithm 5.1: EECBS for solving MAPF bounded-suboptimally.

```
Input: MAPF instance $(G,A)$ and suboptimality bound $w$
1  Generate root CT node $R$ with an empty set of constraints;
2  for $a_i \in A$ do $R.plan[a_i] \leftarrow$ LowLevelSearch$(a_i, G, R, w)$;
3  $R.conflicts \leftarrow$ all conflicts in $R.plan$;
4  computeWDGHeuristic($R$);
5  pushNode($R$, OPEN, CLEANUP, FOCAL, $w$);
6  while OPEN $\neq \emptyset$ do
7      $N \leftarrow$ selectNode(OPEN, CLEANUP, FOCAL, w);
8      if $N.conflicts = \emptyset$ then return $N.plan$;
9      if $N$ is selected from CLEANUP $\wedge$ the WDG heuristic of $N$ has not yet been
         computed then
10         computeWDGHeuristic($N$);
11         pushNode($N$, OPEN, CLEANUP, FOCAL);
12         continue;
13     conflictPrioritization($N.conflicts$);
14     symmetryReasoning($N.conflicts$);
15     $conflict \leftarrow$ the conflict in $N.conflicts$ with the highest priority;
16     Generate the two constraint sets $C_1$ and $C_2$ for resolving $conflict$;
17     $children \leftarrow \emptyset$;
18     for $i = 1, 2$ do
19         $N' \leftarrow$ a copy of $N$;
20         $N'.constraints \leftarrow N.constraints \cup C_i$;
21         foreach $a_j \in A : N'.plan[a_j]$ violates $C_i$ do
22             $N'.plan[a_j] \leftarrow$ LowLevelSearch$(a_j, G, N', w)$;
23             if $N'.plan[a_j]$ does not exist then Go to Line 18;
24         $N'.conflicts \leftarrow$ all conflicts in $N'.plan$;
25         if $N$ is not selected from CLEANUP $\wedge \forall a_i \in A$ $length(N'.plan[a_i]) \leq$
             $w \cdot f^i_{\min}(N) \wedge cost(N') \leq w \cdot lb(best_{lb}) \wedge h_c(N') < h_c(N)$ then
26             $N.plan \leftarrow N'.plan$;
27             $N.conflicts \leftarrow N'.conflicts$;
28             Go to Line 8;
29         $children \leftarrow children \cup \{N'\}$;
30     for $N' \in children$ do pushNode($N'$, OPEN, CLEANUP, FOCAL, $w$);
31     updateOneStepErrors($N$, $children$);
32 return "No Solution";
```

3. the cost of CT node N' is within suboptimality bound $w \cdot lb(best_{lb})$, and

4. the number of conflicts decreases, i.e., $h_c(N') < h_c(N)$.

w	1.04	1.08	1.12	1.16	1.20
CBS bypassing	0.086	0.107	0.110	0.116	0.108
Relaxed bypassing	0.091	0.114	0.104	0.131	0.126

Table 5.2: Average numbers of accepted bypasses per expanded CT node.

The first condition avoids wasting time in applying the bypassing conflicts technique to CT nodes that are selected from CLEANUP because bypassing conflicts resolves conflicts without adding any constraints and thus does not change the lb-value of a CT node.[3] Therefore, it is not be helpful if the purpose of expanding a CT node N is to improve the lower bound, i.e., CT node N is selected from CLEANUP. The second and third conditions ensure that Equations (5.1) and (5.3) hold after replacing the paths, which guarantees bounded suboptimality. The last condition avoids deadlocks, as in the standard CBS bypassing conflicts technique. See Lines 25-28 of Algorithm 5.1. In addition, since bypassing conflicts resolves conflicts without adding any constraints, it does not change the lb value of a CT node.

Empirically, we compare the effectiveness of relaxed bypassing and CBS bypassing for EECBS and report the results in Table 5.2. Relaxed bypassing accepts more bypasses than CBS bypassing, and the difference increases as the suboptimality factor w increases. The yellow and green lines in Figure 5.1 show the performance of EECBS with and without relaxed bypassing. Relaxed bypassing improves the performance of EECBS for all values of w and all numbers of agents. In general, the improvement increases with w.

5.3.2 Prioritizing Conflicts

Recall Section 2.3.2.1. In CBS, the prioritizing conflicts technique (PC) tries to improve the costs of CT nodes faster since the cost of a CT node also serves as a lower bound on the minimum cost of the solutions below this CT node. In EECBS, however, the cost of a CT node is different from its lower bound. Therefore, we redefine cardinal, semi-cardinal, and non-cardinal conflicts. A conflict

[3]CT node N' has more constraints than CT node N, so the lb-value of CT node N' may be larger than the minimum cost of the solutions below CT node N and thus cannot be used as the lb-value of CT node N. Therefore, bypassing conflicts do not change the lb-value of a CT node.

is *cardinal* iff, when CBS uses the conflict to split CT node N and generates two child CT nodes N' and N'', where both $\sum_{a_i \in A} f^i_{opt}(N')$ and $\sum_{a_i \in A} f^i_{opt}(N'')$ are larger than $\sum_{a_i \in A} f^i_{opt}(N)$. The changes to the definitions of semi-cardinal and non-cardinal conflicts are similar. Like CBS, EECBS uses MDDs to classify conflicts and prioritizes conflicts before it chooses conflicts [Line 13]. Since the construction of MDDs incurs runtime overhead and not all cardinal conflicts are important to EECBS (because EECBS can resolve a cardinal conflict by finding bounded-suboptimal paths), EECBS classifies a conflict between agents a_i and a_j only when

1. the CT node N is selected from CLEANUP or
2. at least one of the path costs is equal to its lower bound, i.e., $length(N.plan[a_i]) = f^i_{\min}(N)$ or $length(N.plan[a_j]) = f^j_{\min}(N)$.

The prioritizing conflicts technique is applied in the first case because resolving cardinal conflicts tends to raise the lower bound in this case. It is applied in the second case because resolving cardinal conflicts has to increase the path lengths of the agents in this case (which makes the costs of the resulting child CT nodes closer to the sum of costs of the solutions below them). EECBS selects cardinal conflicts first, then semi-cardinal conflicts, non-cardinal conflicts, and finally unclassified conflicts. By comparing the purple and yellow lines in Figure 5.1, we see that the prioritizing conflicts technique improves the performance of EECBS.

5.3.3 Symmetry Reasoning

We adapt the symmetry reasoning (SR) techniques from Chapter 4 to EECBS with almost no changes. EECBS performs symmetry reasoning before it chooses conflicts [Line 14]. Rectangle symmetries occur only if the paths of both agents are the shortest because, otherwise, one of the agents can simply perform a wait action to avoid the rectangle symmetry. Therefore, for a given conflict between agents a_i and a_j at CT node N, we apply rectangle symmetry reasoning only when the paths of both agents are provably the shortest, i.e., $length(N.plan[a_i]) = f^i_{\min}(N)$ and $length(N.plan[a_j]) = f^j_{\min}(N)$. Corridor and target symmetries can occur even if the paths

w		1.04	1.08	1.12	1.16	1.20
No WDG	Δlb	36.1	23.4	12.9	7.4	3.9
	Cleanup%	84.0%	74.0%	56.6%	40.4%	11.6%
CBS WDG	Δlb	54.5	45.0	37.0	30.1	25.8
	Cleanup%	49.9%	46.5%	25.1%	17.8%	5.7%
	WDG time	87.9%	84.5%	80.1%	76.4%	65.3%
Adaptive WDG	Δlb	54.8	45.2	36.8	29.1	25.3
	Cleanup%	53.2%	44.0%	27.2%	14.4%	8.7%
	WDG time%	77.6%	62.6%	56.9%	43.2%	24.1%

Table 5.3: Lower-bound improvement Δlb, i.e., the minimum $(lb+h)$-value of the CT nodes in CLEANUP when EECBS terminates minus the lb-value of the root CT node. "Cleanup%" is the percentage of expanded CT nodes that are selected from CLEANUP. "WDG time%" is the percentage of runtime spent on computing the WDG heuristic.

of the agents are suboptimal. Therefore, we apply corridor and target symmetry reasoning to all conflicts. By comparing the grey and purple lines in Figure 5.1, we see that the symmetry reasoning technique significantly improves the performance of EECBS.

In our implementation, we added only the rectangle, target, and corridor reasoning techniques from Sections 4.2, 4.4, and 4.5 to EECBS. We did not implement the generalized rectangle and corridor reasoning techniques from Sections 4.3 and 4.6, which we leave for future work.

5.3.4 WDG Heuristic

Since a path of a CT node N in EECBS is not necessarily the shortest one and, for each agent a_i, $f^i_{\min}(N)$ could be smaller than $f^i_{opt}(N)$, we need to modify the WDG heuristic from Chapter 3 as follows. When EECBS computes the WDG heuristic for a CT node N, it builds a weighted dependency graph $G_{WD} = (V_D, E_D, W_D)$. Each vertex $i \in V_D$ corresponds to an agent a_i. The edge weight on each edge $(i, j) \in E_D$ is equal to the minimum sum of costs of the conflict-free paths for agents a_i and a_j that satisfy $N.constraints$ for their two-agent sub-MAPF problem minus $f^i_{opt}(N) + f^j_{opt}(N)$ (instead of $length(N.plan[a_i]) + length(N.plan[a_j])$). Like for CBS, we compute each edge weight by solving a two-agent sub-MAPF problem (with the constraints in $N.constraints$) using

CBS (or, more specifically, CBSH with symmetry reasoning[4]): $f_{opt}^i(N)$ and $f_{opt}^j(N)$ are equal to the costs of the paths of agents a_i and a_j of the root CT node of CBS, and the minimum sum of costs of the conflict-free paths for agents a_i and a_j is equal to the cost of the solution returned by CBS. Since computing the edge weights for all pairs of agents is time-consuming, we follow Section 4.8 by computing the weight of an edge (i, j) only when the paths $N.plan[a_i]$ and $N.plan[a_j]$ are in conflict (as the weight is more likely to be larger than 0 in this case). We delete the other edges and the vertices without incident edges. Let $h_{\text{WDG}}(N)$ be the value of the edge-weighted minimum vertex cover (defined in Definition 3.1) of G_{WD}. Reusing the reasoning in Section 3.3.1, we know that $\sum_{a_i \in A} f_{opt}^i(N) + h_{\text{WDG}}(N)$ is a lower bound on the minimum sum of costs of the solutions below CT node N. Since

$$\sum_{a_i \in A} f_{opt}^i(N) + h_{\text{WDG}}(N) = lb(N) + \sum_{a_i \in A} (f_{opt}^i(N) - f_{\min}^i(N)) + h_{\text{WDG}}(N) \tag{5.10}$$

$$\geq lb(N) + \sum_{i \in V} (f_{opt}^i(N) - f_{\min}^i(N_D)) + h_{\text{WDG}}(N), \tag{5.11}$$

$h(N) = \sum_{i \in V_D} (f_{opt}^i(N) - f_{\min}^i(N)) + h_{\text{WDG}}(N)$ is admissible, and we can thus use $lb(N) + h(N)$ to sort the CT nodes in CLEANUP and compute the lower bound.

Since computing the WDG heuristic for a CT node is time-consuming, we follow Section 3.4.1 and compute the WDG heuristic lazily [Lines 9-12]. But, instead of computing the WDG heuristic for all CT nodes, we only compute it for CT nodes selected from CLEANUP, as the purpose of computing the WDG heuristic is to improve the lower bound. In addition, we also compute the WDG heuristic for the root CT node [Line 4], as this can provide a higher lower bound to begin with.

Table 5.3 reports the lower-bound improvements of EECBS without the WDG heuristic technique (denoted as No WDG), with the WDG heuristic being computed for all CT nodes (denoted as CBS WDG), and with the WDG heuristic technique introduced above (denoted as Adaptive WDG).

[4]Like in Section 5.3.3, we used only the symmetry-reasoning techniques from Sections 4.2,4.4, and 4.5 in our implementation.

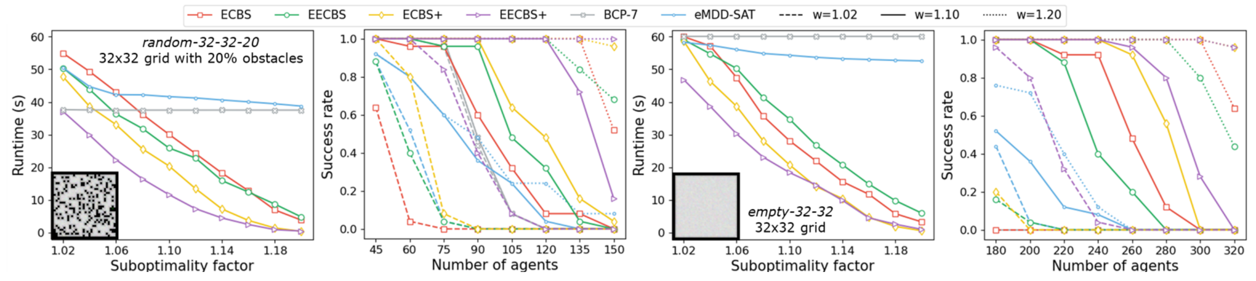
Compared to No WDG, Adaptive WDG selects CT nodes less frequently from CLEANUP but obtains a larger lower-bound improvement. Compared to CBS WDG, Adaptive WDG obtains similar lower-bound improvements but spends less time on computing the WDG heuristic. By comparing the blue and grey lines in Figure 5.1, we see that adaptive WDG improves the performance of EECBS for small and moderately large suboptimality factors.

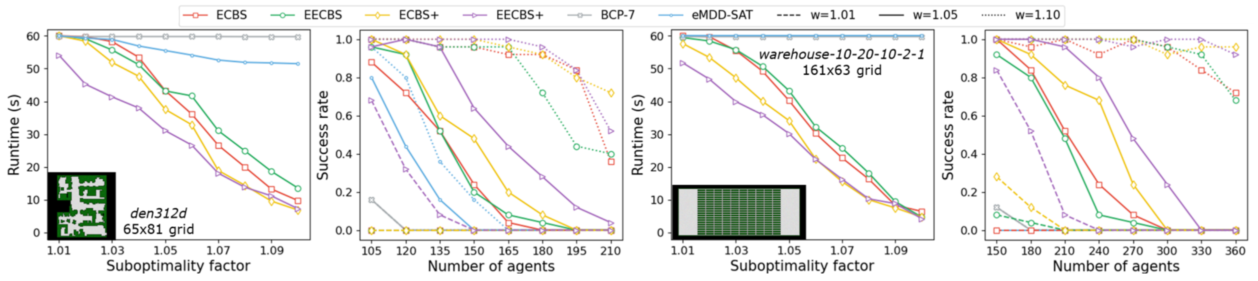
5.4 Empirical Evaluation

We evaluate EECBS on six maps of different sizes and structures from the MAPF benchmark suite [163] with eight different numbers of agents per map. We use the "random" scenarios, yielding 25 instances for each map and number of agents. We evaluate ten different values of w for each setting, and these values of w decrease as the map size increases. The experiments are conducted on Ubuntu 20.04 LTS on an Intel Xeon 8260 CPU with a memory limit of 16 GB and a runtime limit of one minute.

As shown in Figure 5.3, EECBS (green lines) outperforms ECBS (red lines) on some maps but has a similar or even slightly worse performance on other maps. The four improvements improve the performance of both ECBS and EECBS, but EECBS benefits more from them. As a result, EECBS+ (EECBS with all improvements, purple lines) significantly outperforms the other algorithms on all six maps in terms of both runtimes and success rates. When comparing the dashed lines in the success rate figures, we observe that, in many cases (e.g., for 60 agents on the random map and 180 agents on the empty map), EECBS+ is able to solve almost all instances within the runtime limit while ECBS solves only a few or even no instances. For a given success rate, EECBS+ is able to solve instances with up to twice the number of agents compared to ECBS (e.g., on map `den520d`).

We provide more details on the experiment on the random map in Figure 5.4. EECBS+ runs faster than ECBS in most cases and never fails to solve an instance that is solved by ECBS. In addition, the average solution costs (not shown in the figure) of ECBS and EECBS+ over the


(a) Small-sized maps with *w* ranging from 1.02 to 1.20.


(b) Medium-sizes maps with *w* ranging from 1.01 to 1.10.

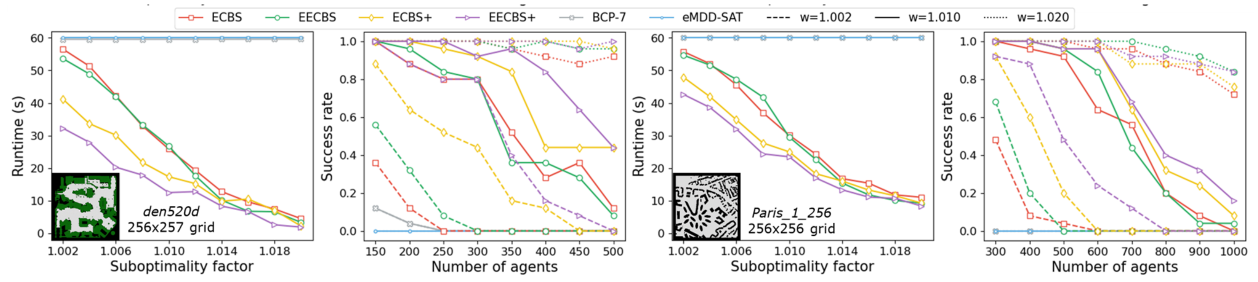

(c) Large-sized maps with *w* ranging from 1.002 to 1.020.

Figure 5.3: Performance of ECBS, EECBS, ECBS+ (ECBS with all improvements), EECBS+ (EECBS with all improvements), BCP-7, and eMDD-SAT. All results are presented in the same format as in Figure 5.1. The algorithms are indicated by the legend with the same color and the same marker style, while the suboptimality factors of the lines in the success rate figures are indicated by the legend with the same line style. Since some algorithms solve (almost) zero instances within the runtime limit, their lines overlap at the top of the runtime figures and the bottom of the success rate figures: the grey lines are hidden by the blue lines in the runtime figures of maps `warehouse-10-2-10-2-1`, `den520d`, and `Pairs_1_256`; the dashed red and green lines are hidden by the dashed yellow line in the success rate figure of map `den312d`; and many of the dashed/solid/dotted grey/blue lines are hidden by other lines at the bottom of many of the success rate figures.

1,081 instances solved by both algorithms are 1,967 and 1,958, respectively, indicating that the improvements used in EECBS+ do not sacrifice the solution quality.

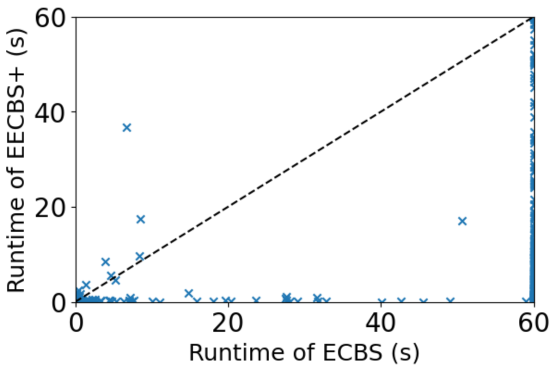


Figure 5.4: Runtimes of ECBS and EECBS+ on the random map with the number of agents m ranging from 45 to 150 and the suboptimality factor w ranging from 1.02 to 1.20. If an instance is not solved within the runtime limit of one minute, we set its runtime to one minute. Among the 2,000 instances, 475 instances are solved by neither algorithm, 444 instances are solved only by EECBS+, and 0 instances are solved only by ECBS.

We omit comparisons with the many search-based bounded-suboptimal MAPF algorithms that have already been shown to perform worse than ECBS [2, 16, 183]. However, there are two recent compilation-based bounded-suboptimal MAPF algorithms, namely BCP-7 [95] and eMDD-SAT [172] (introduced in Sections 2.2.1.2 and 2.2.2), that perform well. BCP-7 can outperform CBS when finding optimal solutions [95], and eMDD-SAT can outperform ECBS in some domains [172]. We therefore compare them with our algorithms. We modify the search of the ILP solver used by BCP-7 from a best-first search (which is more beneficial for finding optimal solutions) to a depth-first search with restarts (which is more beneficial for finding suboptimal solutions). As shown in Figure 5.3, BCP-7 and eMDD-SAT outperform ECBS on the two small maps when the suboptimality factor is small. But for larger suboptimality factors or larger maps, they perform worse than ECBS. EECBS+ performs better than them on all six maps.

5.5 Summary

In this chapter, we proposed a new bounded-suboptimal MAPF algorithm EECBS that uses online learning to estimate the cost of the solution below each high-level node and uses EES to select high-level nodes for expansion. We further improved it by adding bypassing conflicts, prioritizing conflicts, symmetry reasoning, and the WDG heuristic. With these improvements, EECBS+ significantly outperforms the state-of-the-art bounded-suboptimal MAPF algorithms ECBS, BCP-7, and eMDD-SAT. Within one minute, it is able to find solutions that are provably at most 2% worse than optimal for large MAPF instances with up to 1,000 agents, while, on the same map, state-of-the-art optimal algorithms can handle at most 200 agents [95]. We hope that the scalability of EECBS+ enables additional applications for bounded-suboptimal MAPF algorithms.

We used EECBS+ to represent EECBS with all improvements in this chapter in order to show the effectiveness of these improvements. In the other chapters, we refer to EECBS with all improvements simply as EECBS.

5.6 Extensions

This chapter focused on bounded-suboptimal MAPF algorithms that efficiently find near-optimal solutions with theoretical guarantees and thus used only small suboptimality factors. In Chapter 6, we will show that EECBS with large suboptimality factors is competitive with existing unbounded-suboptimal MAPF algorithms in terms of runtime (see Sections 6.1.4 and 6.2.4). Moreover, we will also show that EECBS can be further sped up by using a more efficient low-level search algorithm (see Section 6.2.2.2).

Although EECBS was published just a year ago, several pieces of work have used or extended EECBS to either further speed it up or use it as an efficient sub-MAPF solver. For example, flexible EECBS [36] speeds up EECBS by relaxing the bounded-suboptimality requirement that EECBS enforces on (single-agent) paths while preserving its (overall) bounded-suboptimal guarantee on solutions. Shard systems [101] partition the workspace into geographic regions and call EECBS to plan paths for the agents in each region.

Chapter 6

Improving MAPF Solutions via Large Neighborhood Search

Optimal and bounded-suboptimal algorithms guarantee that the costs of their solutions are at most a given (multiplicative) factor away from optimal. Sometimes, we are interested in good solutions without a guarantee on how good they are. Since providing optimality proofs is computationally expensive in MAPF, we focus in this chapter on using stochastic local search algorithms to find near-optimal solutions greedily (without guarantees) for challenging MAPF problems.

Large Neighborhood Search (LNS) [154] is a popular stochastic local search technique for improving the solution quality for combinatorial optimization problems. Starting from a given solution, LNS *destroys* part of the solution, called a *neighborhood*, and treats the remaining part of the solution as fixed. What results is a reduced and thus simpler form of the original problem to solve. One can use any desired approach from the literature to solve the reduced problem, assuming that it can take into account the fixed part of the solution. LNS *repairs* the solution and replaces the old solution with the repaired one if the repaired one is better. This procedure is repeated until some stopping criterion is met.

Although LNS is broadly used for solving optimization problems [80, 24, 159], we are unaware of any previous LNS approaches for MAPF. In this chapter, we propose two LNS-based MAPF algorithms for improving MAPF solutions in different ways:

- MAPF-LNS is an anytime MAPF algorithm. Starting from an initial MAPF solution that is quickly found by an existing MAPF algorithm from the literature, MAPF-LNS repeatedly selects a subset of agents and replans their paths to reduce their sum of costs until the runtime limit is reached. We show empirically that, compared to existing anytime MAPF algorithms, MAPF-LNS computes initial solutions fast, finds near-optimal solutions eventually, and scales to very large numbers of agents. MAPF-LNS improves the solution quality of existing suboptimal MAPF algorithms by up to 36 times.

- MAPF-LNS2 is an unbounded-suboptimal MAPF algorithm. Starting from a set of paths that contain conflicts, MAPF-LNS2 repeatedly selects a subset of colliding agents and replans their paths to reduce the number of conflicts until the paths become conflict-free. We show empirically that, compared to existing suboptimal MAPF algorithms, MAPF-LNS2 runs significantly faster while still providing near-optimal solutions in most cases. MAPF-LNS2 solves 80% of the random-scenario instances with the largest number of agents from the MAPF benchmark suite within a runtime limit of just five minutes, which, to our knowledge, has not been achieved by any existing MAPF algorithms.

We also show that MAPF-LNS and MAPF-LNS2 can be combined, and their combination achieves the best performance empirically in terms of scalability and solution quality.

In order to speed up MAPF-LNS2, we also propose an efficient single-agent pathfinding algorithm SIPPS based on SIPP [136] for finding a short path for an agent that avoids conflicts with a given set of paths and attempts to minimize the number of conflicts with another given set of paths. We show that SIPPS runs five times (or more) faster than space-time A* (introduced in Section 2.3.1), an A*-based algorithm widely used by many MAPF algorithms (such as all CBS-based and PP-based MAPF algorithms we have mentioned). Thus, we demonstrate that SIPPS can speed up not only MAPF-LNS2 but also many other MAPF algorithms, such as EECBS.

In this chapter, we refer to the difference between the length of a path p_i and the distance between its endpoints as the *delay* $delay(p_i) = length(p_i) - dist(s_i, g_i)$. Minimizing the sum of costs $\sum_{a_i \in A} length(p_i)$ of a solution P is equivalent to minimizing its *sum of delays* $\sum_{a_i \in A} delay(p_i)$.

Since we study challenging MAPF instances for which optimal solutions are usually unknown, unless explicitly stated otherwise, we refer to the (overestimated) suboptimality of a MAPF solution as $\sum_{a_i \in A} length(p_i) / \sum_{a_i \in A} dist(s_i, g_i)$.

This chapter closely follows [109, 114].

6.1 MAPF-LNS: Reducing the Cost of MAPF Solutions

MAPF algorithms can be categorized on a spectrum. At one end are (bounded-sub)optimal algorithms that can find high-quality MAPF solutions for small MAPF problems. At the other end are unbounded-suboptimal algorithms that can solve large MAPF problems but usually find low-quality MAPF solutions. In this section, we consider a third approach that combines the best of both worlds, namely anytime algorithms that quickly find an initial MAPF solution using efficient MAPF algorithms from the literature, even for large MAPF problems, and that subsequently improve the MAPF solution quality to near-optimal as time progresses by replanning subgroups of agents using LNS.

6.1.1 Background: Anytime MAPF Algorithms

Anytime behavior, i.e., generating an initial MAPF solution fast and improving it over time, is highly desirable in practice. Yet, there is little existing work on anytime MAPF algorithms: OA [162] and X* [182] achieve an anytime behavior by repeatedly calling joint-state A* to find optimal MAPF solutions for larger and larger sub-MAPF problems and are efficient only for non-congested instances. IMMI [194] repeatedly replans single-agent paths to reduce the makespan (instead of the sum of costs) of a MAPF solution over time. The optimal algorithm BCP [95] uses a branch-and-bound algorithm, which is anytime in theory, but rarely finds MAPF solutions much earlier than the optimal MAPF solution in practice.[1] *Anytime BCBS* [47] changes the high-level

[1] In Section 5.4, we run BCP on a given MAPF instance and report success if it finds a solution within the runtime limit that satisfies the bounded-suboptimality requirement. It turns out that, in most cases, BCP either fails to find any solution for the given instance within the runtime limit or finds a first solution whose cost is already very close to

focal search of the bounded-suboptimal algorithm Bounded CBS (BCBS) [16] to an anytime focal search. It starts with an infinite suboptimality bound on the sum of costs objective, which is then repeatedly tightened to $S-1$ whenever a new MAPF solution with sum of costs S is found. We compare our proposed algorithm against anytime BCBS empirically in Section 6.1.4 and report significant gains in scalability, runtime to the initial MAPF solution, and speed of improving the MAPF solution.

6.1.2 MAPF-LNS Framework

Given a MAPF instance, we first call a MAPF algorithm to find an initial MAPF solution P. Any non-optimal MAPF algorithm can be used here, including EECBS proposed in Chapter 5. Then, in each iteration, we select a subset of agents $A_s \subseteq A$, remove their paths $P_s^- = \{p_i \in P \mid a_i \in A_s\}$ from P, and replan new paths for them by calling a modified MAPF algorithm. The modified MAPF algorithm returns a set of paths P_s^+, one for each agent in A_s, that do not conflict with each other and with the paths in P. Most optimal (such as CBSH2-RTC proposed in Chapter 4), bounded-suboptimal (such as EECBS), and prioritized MAPF algorithms can be adapted for this purpose by treating the remaining paths in P as moving obstacles. We then compare the (old) path set P_s^- with the (new) path set P_s^+ and add the one with the smaller sum of costs to P. We repeat this procedure until we time out. We call the resulting algorithm MAPF-LNS.

6.1.3 Neighborhood Selection

The selection of good neighborhoods is critical to the success of LNS. For adaptive LNS (introduced in Section 6.1.3.4) to be most successful, the neighborhoods should be *orthogonal*, in the sense that they are formed very differently. Therefore, in this section, we define three different neighborhood selection methods for MAPF and combine them via adaptive LNS. We use a predefined neighborhood size N to specify the number of agents that are used to form each neighborhood.

optimal. This behavior is also supported by the observation that the success rate of BCP in Figure 5.3 seldom increases when the suboptimality factor increases.

Algorithm 6.1: Generate an agent-based neighborhood.

```
Input: MAPF instance $(G,A)$, neighborhood size $N$, MAPF solution $P$, and tabu list $tabuList$
1  $a_k \leftarrow \arg\max_{a_i \in A \setminus tabuList} delay(p_i)$;
2  $tabuList \leftarrow tabuList \cup \{a_k\}$;
3  if $|tabuList| = m \vee delay(p_k) = 0$ then $tabuList \leftarrow \emptyset$;
4  $A_s \leftarrow \{a_k\}$;
5  while $|A_s| < N$ do
6      $A_s \leftarrow$ RANDOMWALK$(G, a_k, P, A_s, N)$;
7      $a_k \leftarrow$ a random agent in $A_s$;
8  return $A_s$;
9  Function RANDOMWALK$(G, a_k, P, A_s, N)$
10     $(x,t) \leftarrow (p_k[t], t)$, where $t$ is a random timestep in $[0, length(p_k) - 1]$;
11     while $|A_s| < N$ do
12         $N_x \leftarrow \{v \in V \mid (x,v) \in E \cup \{(x,x)\} \wedge t + 1 + dist(v, g_k) < length(p_k)\}$;
13         if $N_x = \emptyset$ then break;
14         $y \leftarrow$ a random vertex in $N_x$;
15         $A_s \leftarrow A_s \cup \{a_i \in A \mid p_i[t+1] = y \vee (p_i[t] = y \wedge p_i[t+1] = x)\}$;
16         $(x,t) \leftarrow (y, t+1)$;
17     return $A_s$;
```

6.1.3.1 Agent-Based Neighborhood

The first neighborhood selection method is based on the agents and their paths. We want to select an agent whose path could be shorter if some other agents were not blocking its way, as replanning them together has a chance to reduce the overall sum of costs of their paths.

Algorithm 6.1 shows the pseudo-code. We first choose the agent a_k that is not in the *tabu list* (i.e., a globally maintained set, initially being empty, to avoid selecting the same agent repeatedly) and whose path has the largest delay [Line 1]. We update the tabu list by adding agent a_k to it [Line 2]. If the agents being delayed are all in the tabu list (i.e., either the tabu list contains all agents or the path of agent a_k has a delay of zero, indicating that the path of any agent that is not in the tabu list has a delay of zero as well), then we empty the tabu list [Line 3]. We then initialize the neighborhood A_s with agent a_k [Line 4] and let the agent perform a *restricted random walk* (details are introduced in the next paragraph) to collect the agents that prevent it from reaching its target vertex g_k earlier. These agents are added to the neighborhood A_s as well [Line 6]. We randomly select an agent in A_s [Line 7], which could be the same agent or a different agent, and repeat the procedure as long as fewer than N agents are in A_s [Line 5]. In the experiments, we iterate for at most ten iterations (not shown in the pseudo-code) to address the situation where the agent density is too low for us to collect N agents in A_s.

In function $\textsc{randomWalk}(G, a_k, P, A_s, N)$, agent a_k performs a restricted random walk, which allows it to take only the move or wait actions that could potentially lead to a path shorter than its current one, ignoring conflicts with the paths in P. Then, the agents that agent a_k conflicts with during the random walk are the ones that might prevent it from reaching its target vertex g_k earlier and are thus added to A_s. Formally, we first randomly choose a start *state* (i.e., a vertex-time pair) along the path of agent a_k for the random walk, i.e., a vertex x along path p_k at some timestep $t \in [0, length(p_k))$ [Line 10]. We then collect the set of possible vertices N_x of agent a_k at timestep $t+1$ that might be on paths to g_k (ignoring other agents) that are shorter than its current path p_k [Line 12]. More specifically, the length of a path of agent a_k that visits vertex v at timestep $t+1$ is at least $t+1+dist(v, g_k)$. Thus, if $t+1+dist(v, g_k) < length(p_k)$, then agent a_k might be able to reach vertex g_k via vertex v at timestep $t+1$ earlier than by following path p_k. We let the agent move to a random vertex y in N_x [Line 14] and add any agents to A_s whose paths conflict with this action [Line 15]. Lastly, we update the state of agent a_k [Line 16]. This procedure is repeated until we have collected N agents in A_s [Line 11] or vertex set N_x is empty [Line 13].

6.1.3.2 Map-Based Neighborhood

The second neighborhood selection method is based on the topology of the map (= graph). In particular, we are interested in the agents that visit the same *intersection vertex*, i.e., a vertex with a degree greater than two, because a different ordering of the agents traversing an intersection vertex could lead to better MAPF solutions. If there are not enough agents at one intersection vertex, we explore the map around the intersection vertex to find nearby intersection vertices and

Algorithm 6.2: Generate a map-based neighborhood.

```
    Input: MAPF instance $(G,A)$, neighborhood size $N$, and MAPF solution $P$
1   $V_I \leftarrow \{v \in V \mid degree(v) \geq 3\}$;
2   $x \leftarrow$ a random vertex in $V_I$;
3   $Q \leftarrow \{x\}$;                                            // $Q$ is a queue
4   $A_s \leftarrow \emptyset$;
5   while $|Q| > 0 \wedge |A_s| < N$ do                              // A breath-first search
6       $x \leftarrow Q.pop()$;
7       if $degree(x) \geq 3$ then $A_s \leftarrow$ GETINTERSECTIONAGENTS$(x,P,A_s)$;
8       for $y \in V : (x,y) \in E \wedge y$ has not been visited before do $Q.push(y)$;
9   return $A_s$;
10  Function GETINTERSECTIONAGENTS$(x,P,A_s)$ :
11      $T \leftarrow \max\{t \in \mathbb{N} \mid \exists p_i \in P : p_i[t] = x\}$; // $T$ is the last timestep when a path in P visits $x$
12      $t \leftarrow$ a random timestep in $[0,T]$;
13      $\Delta \leftarrow 0$;
14      while $|A_s| < N \wedge \Delta \leq \max\{t, T-t\}$ do
15          if $t + \Delta \leq T$ then $A_s \leftarrow A_s \cup \{a_i \in A \mid p_i[t+\Delta] = x\}$;
16          if $t - \Delta \geq 0$ then $A_s \leftarrow A_s \cup \{a_i \in A \mid p_i[t-\Delta] = x\}$;
17          $\Delta \leftarrow \Delta + 1$;
18      return $A_s$;
```

collect agents that visit them. This neighborhood is orthogonal to the agent-based neighborhood by design.

Algorithm 6.2 shows the pseudo-code. We begin by collecting all intersection vertices [Line 1] and picking a random one [Line 2]. We put this vertex into a queue [Line 3] and perform a breadth-first search from it. Every time when we pop a vertex x from the queue [Line 6], we examine whether it is an intersection vertex [Line 7]. If it is, then we add the agents that visit the vertex to the neighborhood A_s [Line 7] (details are introduced in the next paragraph). We then add the vertices adjacent to vertex x to the queue [Line 8]. This procedure is repeated until we have collected N agents in A_s or explored the entire map [Line 5].

We use function GETINTERSECTIONAGENTS(x,P,A_s) to add those agents to A_s whose paths in P visit vertex x. We first pick a random timestep t no later than the last timestep when an agent visits vertex x [Lines 11 and 12]. We then add agents to A_s by iteratively exploring, for increasing

Δ, which agents visit vertex x Δ timesteps before or after timestep t until we have collected N agents in A_s or explored all timesteps [Line 13 to 17].

6.1.3.3 Random Neighborhood

The third neighborhood selection method is to select N agents uniformly at random. Random neighborhoods are a good baseline used in many LNS approaches [52, 159]. Although this method appears to be naïve, it is surprisingly effective for congested MAPF instances, as we later show in Table 6.2.

6.1.3.4 Adaptive LNS (ALNS)

Adaptive LNS (ALNS) [146] is a strong variant of LNS as it adapts to what is working on the problem at hand. It makes use of multiple neighborhood selection methods by recording their relative success in improving the current solution and choosing the next neighborhood guided by the most promising method. Given the three methods above, we instantiate ALNS as described below.

We maintain a weight $w_i \geq 0$ for each neighborhood selection method i that represents the relative success of method i in reducing the cost of the current solution. In our experiments, we use $w_i = 1$ for all methods initially. Then, in each iteration, we select the method for generating the next neighborhood according to the weights. Specifically, we use the roulette wheel selection [70] for selecting a method. That is, we select method i with probability $w_i / \sum_j w_j$. We then use the selected method to generate a neighborhood and replan the paths of the agents in the neighborhood. After the new paths are found, we update the weight w_i of the selected method i according to how much the new paths improve the solution quality, namely w_i is set to

$$w_i = \gamma \cdot \max\{0, \sum_{p \in P_s^-} length(p) - \sum_{p \in P_s^+} length(p)\} + (1-\gamma) \cdot w_i, \tag{6.1}$$

where $\gamma \in [0,1]$ is a user-specified reaction factor that controls how quickly the weights react to the changes in the relative success in improving the current MAPF solution. We use $\gamma = 0.01$ in our experiments. The weights of the other methods remain the same.

6.1.4 Empirical Evaluation

We evaluate MAPF-LNS on six representative maps from the MAPF benchmarks [163], namely `empty-8-8` of size 8×8, `empty-32-32` of size 32×32, `random-32-32-20` of size 32×32 (denoted as `random`), `warehouse-10-20-10-2-1` of size 161×63 (denoted as `warehouse`), `ost003d` of size 194×194, and `den520d` of size 256×257. Illustrations of the maps are shown in Figure 6.2. We use the "random" scenarios from the MAPF benchmarks, yielding 25 instances for each map and each number of agents. If the number of agents that we want to test exceeds the number of agents in the benchmarks, we generate new instances with the start and target vertices being selected uniformly at random. The experiments are conducted on Ubuntu 20.04 LTS on an Intel Xeon 8260 CPU with a memory limit of 8 GB and a runtime limit of 60 seconds, except for Experiment 1 where the runtime limit is 10 seconds and Experiment 7 where the runtime limit is 600 seconds.

We use the CBS improvements introduced earlier for all CBS-based algorithms (i.e., CBS, EECBS, and BCBS) that we use as sub-solvers in anytime algorithms, including prioritizing conflicts (see Section 2.3.2.1), bypassing conflicts (see Section 2.3.2.2), using the WDG heuristic (see Chapter 3), and symmetry reasoning[2] (see Chapter 4). That is, CBS is actually CBSH2-RTC introduced in Algorithm 4.6; EECBS is the one with all improvements in Algorithm 5.1; and BCBS is a variant of Algorithm 5.1 that uses focal search instead of EES to search the CT. We use EECBS(x) to denote EECBS with suboptimality factor x.

Before examining the experimental results in detail, we show in Figure 6.1 the evolution of the sum of delays of the MAPF solution produced over time on the same instance. Traditional algorithms, like EECBS, return a single MAPF solution, shown as a point. Anytime algorithms

[2]Like in Section 5.3.3, we use only the symmetry-reasoning techniques from Sections 4.2, 4.4, and 4.5 in our implementation.

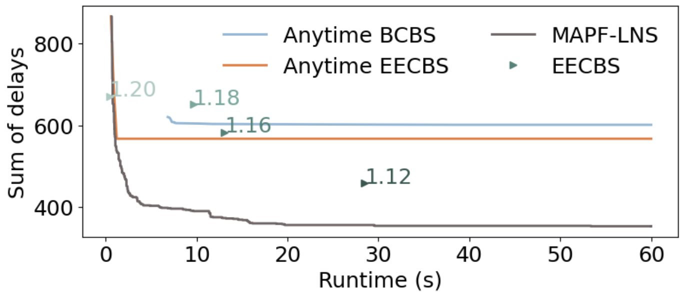


Figure 6.1: Evolution of the sum of delays over a minute for various algorithms on instance "random-32-32-20-random-1.scen" with 150 agents. The points for EECBS are labeled with the corresponding suboptimality factors w. EECBS with $w \leq 1.10$ failed to solve the instance within a minute.

improve the MAPF solution as time progresses, shown as continuous curves. More details are provided in Experiment 6. To judge an anytime algorithm, we use the *Area Under the Curve* (AUC) since it represents not only the sum of delays of the final MAPF solution but also how rapidly we approach it. We define AUC formally as the integral of the sum of delays, starting from the initial MAPF solution (since we only compare algorithms that start from the same initial MAPF solution) and ending when the runtime limit is reached.

Experiment 1: Algorithms for initial planning. We compare three representative non-optimal MAPF algorithms in different categories for creating the initial MAPF solutions, namely the bounded-suboptimal algorithm EECBS(2), the prioritized algorithm PP^R (see Section 2.2.3), and the rule-based algorithm PPS (see Section 2.2.3); all with a runtime limit of 10 seconds. If an algorithm terminates before 10 seconds without finding a MAPF solution, we restart it with a new random ordering of agents. As shown in Figure 6.2, no single algorithm dominates all other ones for all scenarios. Compared to the success rate, the quality of the initial MAPF solution is usually less critical. With a poor initial MAPF solution, MAPF-LNS may take longer to converge, but the improvement is rapid. For example, when we run MAPF-LNS on the `random` map with 100 agents (using the parameters in Experiment 5) with initial MAPF solutions from EECBS, PP, and PPS, their initial sums of delays are very different (namely 299, 468, and 2,224, respectively), but their final sums of delays are close (namely 138, 141, and 136, respectively). The initial solution

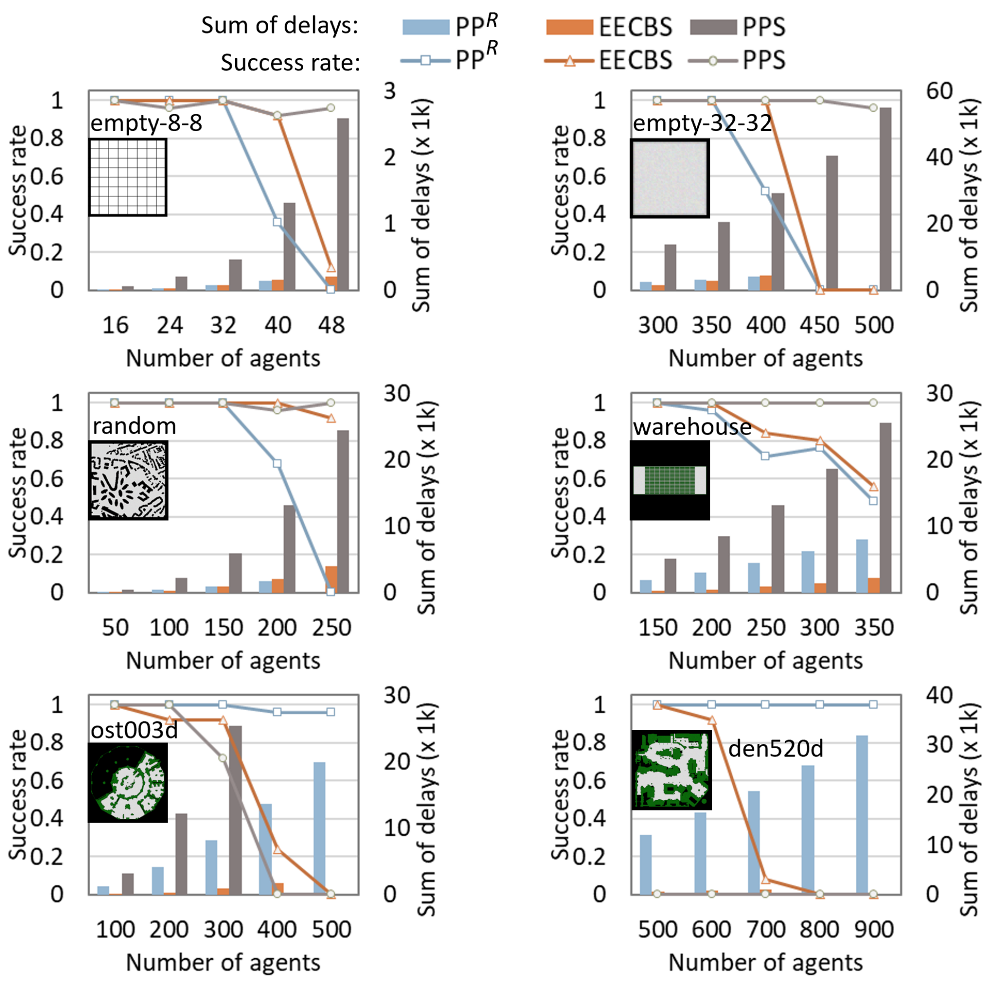


Figure 6.2: Success rates and sums of delays of various algorithms with a runtime limit of 10 seconds for finding initial MAPF solutions. The sum of delays is averaged over all instances solved by each algorithm. The bars of EECBS in the right bottom figure are hidden by the line of PPS. Some bars are missing because zero instances are solved.

quality may be more important for harder instances since it might then be harder for MAPF-LNS to improve the initial MAPF solutions. Hence, for each map and each number of agents, we use the algorithm with the highest success rate to find the initial MAPF solutions in our future experiments.

Experiment 2: Algorithms for replanning. We test various types of MAPF algorithms for replanning paths, namely the prioritized algorithm PP, the bounded-suboptimal algorithm EECBS(1.1), and the optimal algorithm CBS (which always finds optimal MAPF solutions with respect to the chosen subset of agents, i.e., is guaranteed to reach a local minimum). We use the agent-based neighborhood method with a neighborhood size of 4 to generate the neighborhoods.

	Initial algorithm	m	Iterations (x 1,000)			Final sum of delays			AUC	
			PP	EECBS	CBS	PP	EECBS	CBS	EECBS/PP	CBS/PP
empty-8-8	EECBS	16	1,644	257	275	3	3	3	1.00	1.03
		24	1,125	170	131	13	13	13	1.00	**0.98**
		32	1,013	125	98	30	32	31	1.04	1.03
	PPS	40	1,319	104	86	76	80	**71**	1.15	1.10
		48	770	32	20	1,067	**834**	**969**	**0.88**	**0.98**
empty-32-32	EECBS	300	68	36	22	437	450	**435**	1.28	1.31
		350	45	27	17	855	879	855	1.03	1.02
		400	29	18	11	1,616	1,614	**1,593**	1.01	1.02
	PPS	450	11	1	1	6,588	26,991	28,544	1.72	1.72
		500	3	1	1	37,013	47,055	47,233	1.13	1.13
random	EECBS	50	206	60	30	27	28	27	1.05	1.01
		100	71	32	16	143	147	**142**	1.03	**0.98**
		150	48	20	10	383	401	**382**	1.04	1.01
		200	24	13	6	871	889	878	1.03	1.03
	PPS	250	9	2	1	4,718	11,131	11,082	1.77	1.77
warehouse	EECBS	150	13	10	3	132	136	133	1.06	1.06
		200	7	5	2	268	291	276	1.10	1.10
	PPS	250	5	1	0.3	891	1,211	2,977	1.93	3.51
		300	3	1	0.1	1,774	3,903	10,510	1.85	2.74
		350	2	0.2	0.1	3,830	14,343	20,783	1.79	1.99
ost003d	PP^R	100	13	9	1	51	64	79	1.35	3.82
		200	8	4	0.4	333	495	1,150	1.80	3.92
		300	7	2	0.2	1,198	2,139	4,806	1.75	2.95
		400	5	1	0.1	3,337	7,217	10,344	1.78	2.11
		500	3	0.3	0.1	8,813	15,171	17,969	1.43	1.54
den520d	PP^R	500	5	1	0.1	1,788	6,422	9,116	2.53	3.07
		600	5	1	0.1	3,480	9,742	13,796	2.05	2.46
		700	5	0.3	0.1	5,980	15,743	18,678	1.88	2.04
		800	4	0.3	0.1	10,149	21,003	24,008	1.55	1.67
		900	4	0.4	0.1	15,275	27,133	30,371	1.40	1.49

Table 6.1: Performance of MAPF-LNS with various algorithms for replanning. The success rate for each map and each number of agents is the same as in Figure 6.2. Numbers in the *AUC* columns are the ratios of the average AUC of EECBS/CBS over the average AUC of PP. Numbers in bold correspond to the cases when EECBS/CBS has a smaller AUC/final sum of delays than PP.

Table 6.1 reports the resulting number of iterations, the sum of delays of the final MAPF solution, and the relative AUC with respect to PP. Overall, PP is significantly better, dominating CBS and

	m	Final Sum of delays				AUC		
		Random	Agent	Map	ALNS	Random/ALNS	Agent/ALNS	Map/ALNS
empty-8-8	16	3	3	3	3	1.00	1.18	1.02
	24	12	13	13	12	**0.99**	1.12	1.08
	32	30	30	33	29	1.04	1.04	1.14
	40	83	76	85	74	1.16	1.04	1.11
	48	480	1,051	583	434	1.10	2.33	1.30
empty-32-32	300	453	437	418	408	1.11	1.06	1.01
	350	853	852	794	772	1.10	1.08	1.02
	400	1,559	1,613	**1,407**	1,423	1.08	1.09	**0.99**
	450	4,626	7,774	5,618	4,469	**0.96**	1.31	1.18
	500	32,060	38,933	34,510	30,830	1.02	1.10	1.03
random	50	25	27	28	25	**0.99**	1.06	1.07
	100	**138**	143	145	139	1.02	1.04	1.05
	150	391	382	385	373	1.05	1.02	1.03
	200	881	870	864	838	1.04	1.03	1.02
	250	**3,388**	4,700	**3,843**	3,988	**0.90**	1.13	1.01
warehouse	150	138	134	146	130	1.11	1.03	1.19
	200	300	**280**	326	283	1.11	**0.98**	1.17
	250	1,535	884	1257	831	1.41	1.09	1.53
	300	2,706	**1,708**	2,851	1,736	1.27	1.13	1.38
	350	4,555	3,694	6,917	3,256	1.14	1.11	1.28
ost003d	100	59	51	211	44	2.59	1.00	5.10
	200	1,106	334	1,192	330	3.41	**0.96**	3.08
	300	3,964	**1,215**	2,985	1,227	2.46	**0.92**	1.81
	400	8,779	**3,289**	5,777	3,343	1.92	**0.96**	1.42
	500	15,386	**8,926**	11,947	9,207	1.40	**0.97**	1.18
den520d	500	8,451	1,752	5,288	1,661	2.99	**0.96**	2.13
	600	13,087	**3,415**	7,669	3,462	2.31	**0.94**	1.64
	700	17,364	**6,209**	11,024	6,597	1.82	**0.93**	1.37
	800	22,607	**9,882**	14,969	10,054	1.61	**0.95**	1.25
	900	28,342	**15,367**	19,956	15,746	1.41	**0.97**	1.15

Table 6.2: Performance of MAPF-LNS using LNS with various neighborhood selection methods with respect to MAPF-LNS using ALNS. Numbers in bold correspond to the cases where LNS with a single neighborhood selection method has a smaller AUC/final sum of delays than ALNS.

EECBS on 74% of the tested instances in terms of the AUC. This is so because PP runs significantly faster than the other two algorithms and thus can explore a substantially larger number of neighborhoods within the runtime limit.

Experiment 3: Neighborhood selection methods. We compare MAPF-LNS using LNS with each of the three neighborhood selection methods discussed in Section 6.1.3 against MAPF-LNS using ALNS with all three methods. We use the same setting as in Experiment 2 and PP to replan. As shown in Table 6.2, different methods perform differently on different maps with different numbers of agents, but ALNS is the best one overall. While not always superior, it is at least the second best in each scenario and never more than 10% worse than the best method in terms of AUC. An interesting result is that the naïve random neighborhood method outperforms the other methods on the random map with 250 agents. This is so because these instances are highly congested, so, even if we select agents randomly, we still find groups of agents that are coupled with each other, and some of the groups might not be covered by the other two handcrafted neighborhood selection methods.

Experiment 4: Neighborhood sizes. We examine different neighborhood sizes N by trying alternate sizes of 2, 8, and 16, comparing to the size of 4 used up to this point. We use ALNS with the same settings as in Experiment 3. In general, larger neighborhoods result in larger chances to find better MAPF solutions but require more time to replan, resulting in fewer iterations within the runtime limit. Table 6.3 shows that the neighborhood size makes a substantial difference, with larger neighborhood sizes being better for less congested instances.

Experiment 5: Solution quality. Table 6.4 shows the sum of delays of the initial and final MAPF solutions of MAPF-LNS using the same settings as in Experiment 4 and the best neighborhood sizes from Table 6.3. Within a minute, MAPF-LNS dramatically reduces the sum of delays by up to 36 times. Since it is too difficult to discover the optimal MAPF solutions for most instances, the suboptimality reported in Table 6.4 is an upper bound on the (actual) suboptimality of the final MAPF solution, namely $\sum_{a_i \in A} length(p_i) / \sum_{a_i \in A} dist(s_i, g_i)$. Overall, the suboptimality of MAPF-LNS is small. For the large maps (i.e., the `warehouse`, `ost003d`, and `den520d` maps), it is never worse than 14%, and almost certainly much better. For the small maps (i.e., the `empty` and `random` maps), it can grow large for large numbers of agents, but the upper bound on the suboptimality is

	m	Iterations (x 1,000)				AUC		
		N2	N4	N8	N16	N2/N4	N8/N4	N16/N4
empty-8-8	16	3,268	1,548	832	510	1.25	**0.97**	**0.97**
	24	2,708	1,455	800	451	1.39	0.89	**0.88**
	32	2,558	1,236	655	381	1.38	0.91	**0.87**
	40	2,727	1,593	809	431	1.71	**0.91**	1.84
	48	1,971	1,363	226	52	1.12	3.59	4.20
empty-32-32	300	172	87	33	17	1.18	**0.92**	0.99
	350	117	55	20	10	1.15	**0.98**	1.11
	400	71	33	13	6	1.09	**1.00**	1.17
	450	27	15	3	1	1.19	1.54	1.85
	500	5	4	1	0.4	1.04	1.13	1.19
random	50	451	257	146	81	1.11	0.96	**0.95**
	100	206	103	49	28	1.10	**0.94**	**0.94**
	150	137	58	24	14	1.11	**0.96**	**0.96**
	200	77	30	11	6	1.12	**0.98**	1.05
	250	35	12	4	2	**0.99**	1.29	1.57
warehouse	150	28	19	11	6	1.22	0.95	**0.89**
	200	14	9	5	2	1.13	0.94	**0.93**
	250	12	6	3	2	1.51	1.06	1.09
	300	8	4	2	1	1.27	1.14	1.12
	350	4	2	1	0.4	1.21	1.08	1.16
ost003d	100	28	19	10	5	2.27	**0.82**	0.87
	200	15	11	6	3	2.44	1.17	1.14
	300	11	7	3	1	1.81	1.00	**0.97**
	400	8	5	2	1	1.66	**0.90**	0.95
	500	5	3	1	0.4	1.31	**0.97**	1.02
den520d	500	10	7	3	1	2.05	**0.85**	0.91
	600	9	6	3	1	1.77	0.85	**0.83**
	700	8	5	2	1	1.53	0.83	**0.81**
	800	7	5	2	1	1.43	**0.83**	0.85
	900	6	4	2	1	1.31	0.91	**0.84**

Table 6.3: Performance of MAPF-LNS using neighborhood sizes of 2, 4, 8, and 16 (denoted as *N2*, *N4*, *N8*, and *N16*, respectively). Numbers in bold correspond to the neighborhood size with the smallest AUC. If no numbers are in bold, N4 has the smallest AUC.

highly misleading since, for these extremely congested instances, the sum of costs of the optimal MAPF solution (if we could find it) is probably much larger than $\sum_{a_i \in A} dist(s_i, g_i)$. When we use only instances for which we can find the optimal MAPF solutions, the (actual) suboptimality of MAPF-LNS is much smaller. Among the 750 (easier) instances used in Experiment 6, CBS solved 199 instances to optimality within 60 seconds. For these instances, MAPF-LNS finds optimal

	N	m	Sum of delays		Subopt-imality		N	m	Sum of delays		Subopt-imality
			Initial	Final					Initial	Final	
empty-8-8	16	16	7	3	≤1.03	empty-32-32	8	300	1,515	367	≤1.06
	16	24	33	11	≤1.09		8	350	2,740	743	≤1.10
	16	32	79	25	≤1.16		8	400	4,445	1,374	≤1.16
	8	40	1,314	63	≤1.30		4	450	40,513	5,121	≤1.54
	4	48	2,586	668	≤3.67		4	500	55,057	33,554	≤4.16
random	16	50	47	24	≤1.02	warehouse	16	150	261	124	≤1.01
	16	100	299	130	≤1.06		16	200	526	266	≤1.02
	16	150	914	346	≤1.10		8	250	13,199	635	≤1.03
	8	200	2,139	792	≤1.18		8	300	18,587	1,400	≤1.06
	4	250	24,455	3,390	≤1.60		4	350	25,539	3,979	≤1.14
ost003d	8	100	1,338	37	≤1.00	den520d	8	500	12,002	869	≤1.01
	4	200	4,103	346	≤1.01		8	600	16,424	2,034	≤1.02
	8	300	8,129	1,098	≤1.02		16	700	20,713	4,473	≤1.04
	8	400	13,634	2,427	≤1.04		8	800	25,885	7,408	≤1.05
	8	500	19,914	8,223	≤1.11		16	900	31,888	12,186	≤1.08

Table 6.4: Solution quality of MAPF-LNS.

MAPF solutions for 171 instances and <0.01%, <0.1%, and <1% suboptimal MAPF solutions for 175, 195, and 198 instances, respectively. The worst MAPF solution is 1.35% suboptimal.

Experiment 6: Alternative anytime algorithms. We compare MAPF-LNS against the state-of-the-art anytime algorithm Anytime BCBS. Anytime BCBS is based on the bounded-suboptimal algorithm BCBS, which is much slower than more recent bounded-suboptimal algorithms, such as EECBS. We therefore also created an anytime version of EECBS based on restarting its search. Anytime EECBS starts with an initial suboptimality factor of 2. Whenever it finds a MAPF solution with sum of costs S and a lower bound L on the optimal sum of costs, it updates the suboptimality factor to $1+0.99\times(S/L-1)$ and restarts the search. Since the value of $S/L-1$ is guaranteed to be at least 1% smaller after each iteration, it will converge to 0 after a finite number of iterations. That is, the MAPF solutions of Anytime EECBS are guaranteed to converge to optimal. MAPF-LNS uses EECBS(2) to generate an initial MAPF solution (i.e., the same initial MAPF solution as used by Anytime EECBS), ALNS with a neighborhood size of 16 to generate neighborhoods, and PP to replan. Since Anytime BCBS and Anytime EECBS cannot find MAPF solutions for many of the

	m	Success rate		Time to solution		Iterations		
		BCBS	MAPF-LNS	BCBS	MAPF-LNS	BCBS	EECBS	MAPF-LNS
empty-8-8	8	**1.00**	**1.00**	**0.0002**	**0.0002**	1	2	1,053k
	16	**1.00**	**1.00**	0.0012	**0.0005**	2	4	524k
	24	**1.00**	**1.00**	0.01	**0.0018**	8	11	462k
	32	**1.00**	**1.00**	0.05	**0.01**	7	11	377k
	40	0.60	**1.00**	0.87	**0.43**	5	4	401k
empty-32-32	100	**1.00**	**1.00**	0.05	**0.02**	10	7	111k
	200	**1.00**	**1.00**	4.21	**0.11**	15	7	55k
	300	0.72	**1.00**	16.96	**0.45**	8	5	18k
	400	0.00	**1.00**	-	**3.36**	-	4	7k
	500	0.00	**1.00**	-	**33.33**	-	1	1k
random	50	**1.00**	**1.00**	0.06	**0.02**	17	10	78k
	100	0.96	**1.00**	0.88	**0.10**	11	8	27k
	150	0.88	**1.00**	9.63	**0.42**	7	6	13k
	200	0.08	**1.00**	39.60	**1.50**	2	6	6k
	250	0.00	**1.00**	-	**5.00**	-	4	4k
warehouse	50	**1.00**	**1.00**	0.23	**0.03**	3	3	13k
	100	0.92	**1.00**	1.54	**0.12**	23	8	12k
	150	0.92	**1.00**	7.65	**0.90**	17	5	6k
	200	0.80	**1.00**	27.00	**2.86**	4	3	3k
	250	0.28	**1.00**	33.04	**4.67**	6	2	2k
ost003d	50	**1.00**	**1.00**	0.30	**0.06**	2	5	7k
	100	0.92	**1.00**	2.42	**0.22**	6	6	5k
	150	0.84	**1.00**	6.98	**1.48**	4	4	3k
	200	0.52	**1.00**	18.20	**1.86**	2	3	2k
	250	0.16	**1.00**	29.72	**2.88**	3	1	2k
den520d	100	0.92	**1.00**	0.69	**0.26**	2	4	5k
	200	0.68	**1.00**	4.35	**0.88**	5	5	3k
	300	0.44	**1.00**	12.39	**1.83**	3	3	2k
	400	0.08	**1.00**	20.98	**3.16**	1	2	2k
	500	0.00	**1.00**	-	**3.58**	-	1	1k

Table 6.5: Comparison of MAPF-LNS against Anytime BCBS and Anytime EECBS on easier MAPF instances. The numbers in the "Time to solution" (short for runtime to the initial MAPF solution) and "Iterations" columns are averaged over all instances solved by each algorithm. We omit the columns for Anytime EECBS when its values are always the same as the ones of MAPF-LNS. Numbers in bold correspond to the largest success rates or the smallest runtimes to the initial MAPF solution.

instances used in our previous experiments, we use instances with fewer agents than before in this experiment. Table 6.5 summarizes the success rates, runtimes to the initial MAPF solutions, and numbers of iterations, and Figure 6.3 summarizes the initial and final sums of delays. Since BCBS

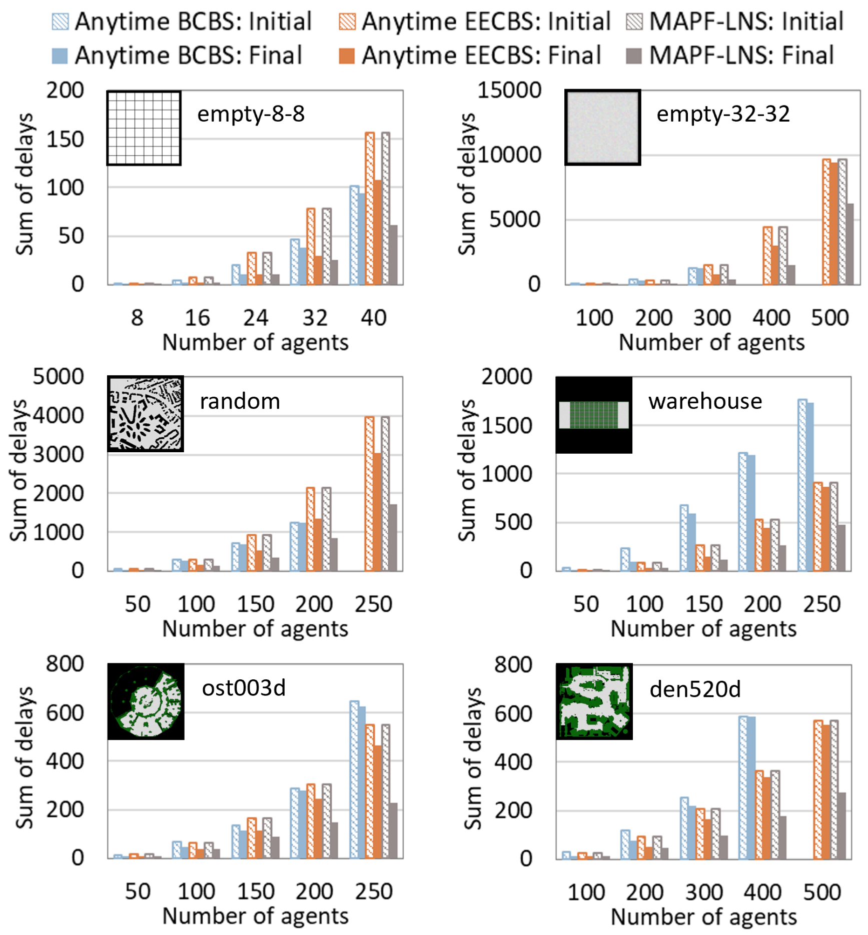


Figure 6.3: Initial and final sums of delays of MAPPF-LNS, Anytime BCBS, and Anytime EECBS, averaged over all instances solved by each algorithm. Some blue bars are missing because zero instances are solved.

is slower than EECBS, Anytime BCBS has lower success rates and longer runtimes to the initial MAPF solution than the other two algorithms. Anytime BCBS and Anytime EECBS are essentially multiple runs of a bounded-suboptimal search algorithm with decreasing suboptimality bounds, so they both result in longer and longer iterations and thus fail to improve the initial MAPF solutions substantially. MAPF-LNS, on the other hand, focuses on a few (ideally highly-coupled) agents in each iteration and thus can perform substantially more iterations and improve the initial MAPF solution rapidly. Although the solutions of Anytime BCBS/EECBS are guaranteed to converge to optimal in theory [47], this happens in practice only when the instances are very easy, such as some instances of the `random` and `warehouse` maps with 50 agents. MAPF-LNS does not have

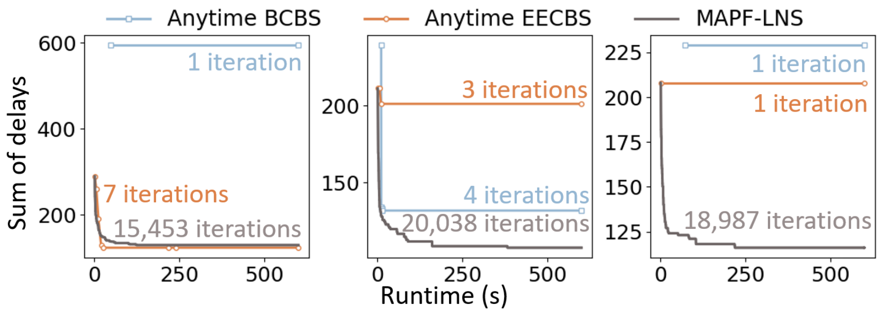


Figure 6.4: Evolution of the sum of delays over 10 minutes for the three anytime algorithms on the first three instances of map den520d with 300 agents that are solved by all of the algorithms. Each point on the Anytime BCBS/EECBS curves represents one iteration, except for the last point at 600 seconds. We omit the points on the MAPF-LNS curves as MAPF-LNS has too many iterations.

such guarantees, but its MAPF solutions also converge to optimal on most of those instances in practice. In addition, when facing harder instances that Anytime BCBS/EECBS cannot solve, MAPF-LNS can still solve them by using more efficient algorithms, such as PP and PPS, to find initial MAPF solutions and improve them over time.

Experiment 7: Longer runtime limits. To better understand the anytime behavior of the three algorithms used in Experiment 6, we repeat that experiment with a runtime limit of 600 seconds on map den520d. The final sums of delays of MAPF-LNS here are usually smaller (by at most 12%) than the final sum of delays of MAPF-LNS reported in Experiment 6, but the trends are the same (i.e., PP is the best replanning algorithm, and ALNS works better than LNS). Anytime BCBS and Anytime EECBS, on the other hand, run out of memory in many cases. Moreover, their final sums of delays and numbers of iterations do not change much, and their sum-of-delays graphs usually become flat after a few seconds (see Figure 6.4). Similar memory issues of CBS-based algorithms have been observed in [30], and similar convergence behavior of Anytime BCBS has been observed in [47].

6.2 MAPF-LNS2: Repairing Infeasible MAPF Solutions

Existing MAPF algorithms include optimal and bounded-suboptimal search algorithms (that are exponential-time), rule-based algorithms (that are usually polynomial-time and complete but not optimal), and prioritized algorithms (that run fast empirically but are neither complete nor optimal). When facing challenging MAPF instances, however, the first two types of algorithms suffer from either memory-outs or time-outs while the last type suffers from incompleteness. Although MAPF-LNS significantly improves the solution quality of non-optimal MAPF algorithms, it relies on existing MAPF algorithms to find initial MAPF solutions and thus cannot improve the runtimes or success rates of them.

One technique that can improve the chance of finding MAPF solutions is to restart the search with a new random seed [21, 48]. In this section, we propose a different way of improving the chance of finding MAPF solutions. Instead of giving up on the previous search effort and restarting the search from scratch, we make use of the infeasible set of paths produced by a MAPF algorithm and try to repair it via LNS.

6.2.1 MAPF-LNS2 Framework

We propose MAPF-LNS2, a version of LNS that can efficiently find a MAPF solution (instead of improving a given MAPF solution) for a MAPF instance. To begin with, MAPF-LNS2 calls a MAPF algorithm to solve the instance and obtain a plan that might be incomplete or have conflicts. For each agent that does not yet have a path, MAPF-LNS2 plans a path that minimizes the number of conflicts with the existing paths. Details of finding such paths are introduced in Section 6.2.2. For instance, for a CBS-based algorithm, each CT node contains a plan, so MAPF-LNS2 picks the plan with the minimum number of conflicts. For a prioritized algorithm, it fails when there is no path for an agent that avoids conflicts with the paths of all higher-priority agents. MAPF-LNS2 retains the already-planned paths and plans paths for the remaining agents that minimize the number of conflicts (instead of avoid conflicts) with the already-planned paths.

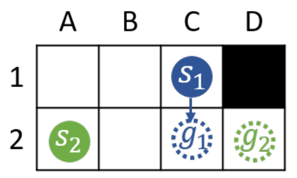
MAPF-LNS2 then repeats a repair procedure until the plan P becomes conflict-free. At each iteration, MAPF-LNS2 selects a subset of agents $A_s \subseteq A$ with a neighborhood selection method (see Section 6.2.3). We denote the paths of the agents in A_s as P^-. It then calls a modified MAPF algorithm to replan the paths of the agents in A_s so as to minimize the number of conflicts with each other and the paths in $P \setminus P^-$. Specifically, MAPF-LNS2 uses a modification of PP as the modified MAPF algorithm.[3] (Modified) PP assigns a random priority ordering to the agents in A_s and replans their paths one at a time according to the ordering. Each time, it calls a single-agent pathfinding algorithm (see Section 6.2.2) to find a path for an agent that minimizes the number of conflicts with the new paths of the higher-priority agents in A_s and the paths in $P \setminus P^-$. We denote the new paths of the agents in A_s as P^+. Finally, MAPF-LNS2 replaces the old plan P with the new plan $(P \setminus P^-) \cup P^+$ iff the *number of colliding pairs* (CP) of the paths in the new plan is no larger than that of the old plan.

6.2.2 Pathfinding with Dynamic Obstacles

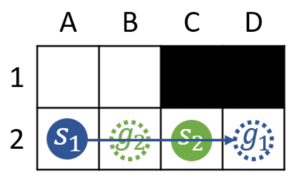
To make MAPF-LNS2 efficient, we need an efficient single-agent pathfinding algorithm to find a (short) path for an agent that minimizes the number of conflicts with a given set of paths. Here, we formulate a more general problem called Pathfinding with Mixed Dynamic Obstacles (PMDO).

Definition 6.1 (Pathfinding with Mixed Dynamic Obstacles)**.** *We call (v,t), (e,t), and $(v,[t,\infty))$ a vertex, edge, and target obstacle indicating that vertex $v \in V$, edge $e \in E$, and vertex $v \in V$ are occupied at timestep t, from timestep $t-1$ to timestep t, and at and after timestep t, respectively. Given a graph $G = (V,E)$, a start vertex $s \in V$, a target vertex $g \in V$, and two finite sets of obstacles $\mathcal{O}^h$ (called* hard obstacles*) and $\mathcal{O}^s$ (called* soft obstacles*), where each obstacle is either a vertex, edge, or target obstacle, our task is to find a path from vertex s to vertex g that does not conflict with any hard obstacles. We assume that vertex s at timestep 0 is not occupied by any hard obstacles, and vertex g at timestep ∞ is not occupied by any hard obstacles either, i.e., there exists a finite*

[3]We have adapted two other MAPF algorithms to replanning the paths, namely Greedy CBS [16] and PBS [123]. But both of them perform worse than PP empirically.



(a)



(b)

Figure 6.5: Examples when PP fails to find any MAPF solutions. Agent a_1 follows the arrow without waiting.

timestep after which no more hard obstacles occupy vertex g. The optimization objective is to minimize the number of soft conflicts*, i.e., conflicts between the found path and the soft obstacles, and break ties in favor of the shortest path.* □

The problem that the single-agent pathfinding solver in MAPF-LNS2 has to solve is a special case of PMDO with $\mathscr{O}^h = \emptyset$ and $\mathscr{O}^s = \{(v,t) \mid p[t] = v, p \in P', t \in [0, length(p) - 1]\} \cup \{((u,v),t) \mid p[t-1] = u \wedge p[t] = v \wedge u \neq v, p \in P', t \in [1, length(p) - 1]\} \cup \{(p[length(p)], [length(p), +\infty)) \mid p \in P'\}$, where P' consists of the new paths of the higher-priority agents in A_s and the paths in $P \setminus P^-$.

6.2.2.1 Space-Time A*

A straightforward algorithm for PMDO is space-time A*, which is used by many MAPF algorithms, such as ID [162] as well as all the CBS-based and prioritized MAPF algorithms that we have introduced so far. Space-time A* performs an A* search on a time-expanded graph. Each state in the graph is defined by a vertex v and a timestep t, representing the agent being at vertex v at timestep t. The agent can move from state (v,t) to state (v',t') iff $((v,v') \in E \vee v = v') \wedge t' = t + 1$ holds and the move action does not conflict with any obstacles in $\mathscr{O}^h$. In addition to the regular g-, h-, and f-values, each node in the search tree of space-time A* maintains a c-value, that represents the number of conflicts of the partial path from the root node to the current node with obstacles in $\mathscr{O}^s$. To solve PMDO optimally, space-time A* sorts the nodes in the open list in ascending order of their c-values, breaking ties in ascending order of their f-values.

While space-time A* can solve PMDO correctly, unfortunately, it cannot do so efficiently. Consider the instance shown in Figure 6.5a. If agent a_2 has to plan a path that minimizes the number of conflicts with the path of agent a_1, space-time A* has to expand all nodes whose c-values are zero before finding the optimal path, that has one conflict with a_1 at C2 at timestep 2 (because a_1 reaches C2 at timestep 1 and remains there forever). However, a potentially infinite number of nodes have zero conflicts as the time dimension is unbounded. So, space-time A* may not return a path in finite time. Although one can fix this issue by restricting space-time A* to generating states only with timesteps no greater than the maximum timestep of the obstacles $\max\{t \in \mathbb{N} \mid (v,t) \in \mathcal{O}^h \cup \mathcal{O}^s \vee (e,t) \in \mathcal{O}^h \cup \mathcal{O}^s \vee (v,[t,\infty)) \in \mathcal{O}^h \cup \mathcal{O}^s\}$ and switching to standard A* (without the time dimension) afterward [123], the number of nodes it has to expand can still be large. Similar performance issues have been observed in [45, 36] where ECBS and EECBS can run slower when their low levels use a larger suboptimality factor as a larger suboptimality factor results in more timesteps that space-time A* needs to consider.

6.2.2.2 SIPPS

Safe Interval Path Planning (SIPP) [136] is a fast variant of space-time A* that uses time intervals instead of timesteps to represent the time dimension of the problem. It performs an A* search on a time-interval graph. Each state in the graph is defined by a vertex and a safe (time) interval, representing that the vertex is not occupied by any hard obstacles at any timestep during the time interval. For each state with vertex v and safe interval $[a,b)$, SIPP always prefers the (partial) path that arrives at vertex v as early as possible within $[a,b)$ and then waits at vertex v if necessary, since this allows SIPP to prune paths that arrive at vertex v at a later timestep within $[a,b)$ without losing optimality. SIPP runs significantly faster than space-time A* empirically [136, 110], yet it cannot handle soft obstacles. We thus generalize SIPP to *Safe Interval Path Planning with Soft constraints* (SIPPS) for solving PMDO.[4]

[4]To the best of our knowledge, SCIPP [49] is the only existing SIPP variant that handles soft obstacles. However, it cannot solve PMDO as it cannot handle edge obstacles.

Safe intervals. A *safe interval* for a vertex is a contiguous period of time during which

- there are no hard vertex obstacles and no hard target obstacles at any timestep, and
- there is either
  - a soft vertex or soft target obstacle at every timestep or
  - no soft vertex obstacle and no soft target obstacle at any timestep.

We build a safe interval table $\mathscr{T}$, that maps each vertex $v \in V$ to a sequence of safe intervals $\mathscr{T}[v]$. To build $\mathscr{T}[v]$, we look at all hard and soft vertex and target obstacles at vertex v and divide interval $[0, \infty)$ into a minimum set of disjoint safe intervals in $\mathscr{T}[v]$ in chronological order. We do not consider edge obstacles here as they are handled elsewhere.

SIPPS nodes. A node n in the search tree of SIPPS consists of four elements, namely a vertex $n.v$, a safe interval $[n.low, n.high)$ where $n.low$ is also called the *earliest arrival time*, an index $n.id$ indicating that the safe interval is (a subset of) the id-th safe interval in $\mathscr{T}[n.v]$ (i.e., interval $\mathscr{T}[n.v][n.id]$), and a Boolean flag $n.is\_goal$ indicating whether the node is a goal node (set to *false* by default). The f-value of node n is the sum of its g-value and h-value, where the g-value is set to $n.low$ and the h-value is a lower bound on $dist(n.v, g)$. Each node n also maintains a c-value, which is (a lower bound on) the number of the soft conflicts of the partial path from the root node to node n, i.e.,

$$c(n) = c(n') + c_v + c_e, \tag{6.2}$$

where n' is the parent node of n, c_v is 1 if the safe interval of n contains soft vertex/target obstacles and 0 otherwise, and c_e is 1 if $((n.v, n'.v), n.low) \in \mathscr{O}^s$ and 0 otherwise. If n is the root node (i.e., n' does not exist), then $c(n) = c_v$. In principle, an agent may encounter more than one soft conflict if it

Algorithm 6.3: SIPPS for solving PMDO.

```
   Input: PMDO instance $(G = (V,E), s, g, \mathcal{O}^h, \mathcal{O}^s)$
1  $\mathcal{T} \leftarrow$ BUILDSAFEINTERVALTABLE$(V, \mathcal{O}^h, \mathcal{O}^s)$;
2  $root \leftarrow Node(s, \mathcal{T}[s][1], 1, false)$;    // 1 and $false$ indicate $id = 1$ and $is\_goal = false$
3  $T \leftarrow 0$;                                          // Lower bound on path length
4  if $\exists t \in \mathbb{N} : (g,t) \in \mathcal{O}^h$ then $T \leftarrow \max\{t \in \mathbb{N} \mid (g,t) \in \mathcal{O}^h\} + 1$;
5  Compute $g$-, $h$-, $f$-, and $c$-values of $root$;
6  OPEN $\leftarrow \{root\}$; CLOSED $\leftarrow \emptyset$;               // Initialize open and closed lists
7  while OPEN $\neq$ do
8      $n \leftarrow$ OPEN.$pop()$;                           // Node with the smallest c-value
9      if $n.is\_goal$ then return EXTRACTPATH$(n)$;
10     if $n.v = g \wedge n.low \geq T$ then
11         $c_{future} \leftarrow |\{(g,t) \in \mathcal{O}^s \mid t > n.low\}|$;
12         if $c_{future} = 0$ then return EXTRACTPATH$(n)$;
13         $n' \leftarrow$ a copy of $n$ with $is\_goal = true$;
14         $c(n') \leftarrow c(n') + c_{future}$;
15         INSERTNODE$(n', \text{OPEN}, \text{CLOSED})$;          // Algorithm 6.5
16     EXPANDNODE$(n, \text{OPEN}, \text{CLOSED}, \mathcal{T})$;          // Algorithm 6.4
17     CLOSED $\leftarrow$ CLOSED $\cup \{n\}$;
18 return "No Solution";
```

waits within a safe interval that contains soft vertex obstacles. We ignore such cases for efficiency.[5] More discussions can be found in the Theoretical analysis paragraph.

Main algorithm. Algorithm 6.3 shows the pseudo-code of SIPPS. To begin with, we build $\mathcal{T}$ [Line 1] and generate the root node with start vertex s, the first safe interval $\mathcal{T}[s][1]$ from $\mathcal{T}[s]$, index 1, and $is\_goal = false$ [Line 2]. T is a lower bound on the path length [Line 3]. If there are hard vertex obstacles at target vertex g, then T is set to one plus the maximum timestep of all hard vertex obstacles at vertex g [Line 4] because the agent cannot complete its path before all hard vertex obstacles at vertex g have disappeared. OPEN and CLOSED are regular open and closed lists, respectively [Line 6]. In order for SIPPS to find the path with the minimum number of soft conflicts (and break ties in favor of the shortest path), the nodes in OPEN are sorted in ascending

[5]If we consider such cases and define the c-value as the accurate number of the soft conflicts, then the key idea behind SIPP cannot be applied. That is, given a state with vertex v and safe interval $[a,b)$, the (partial) path that arrives at vertex v as early as possible within $[a,b)$ and then waits at vertex v if necessary can be suboptimal because waiting at vertex v may result in a larger c-value as opposed to waiting at the vertex before vertex v.

order of their c-values, breaking ties in ascending order of their f-values. At every iteration, SIPPS pops a node n from OPEN [Line 8] and return its corresponding path if it is a goal node [Line 9]. Function EXTRACTPATH(n) constructs a path by repeatedly moving to the parent node until the root node is found. The reversed sequence of vertices of the visited nodes is the sequence of vertices visited by the path, with their timesteps being the earliest arrival time of each node. If the difference between the timesteps of two adjacent vertices on the path is larger than one, we add wait actions in between accordingly so that the agent reaches the first vertex at the earliest arrival time of the corresponding node, waits there, and then moves to the second vertex at the earliest arrival time of the corresponding node. If the vertex of node n is target vertex g with $n.low \geq T$ [Lines 10 to 15], it can be a goal node, but its c-value does not consider the number of additional soft conflicts c_{future} that the agent encounters after timestep $n.low$ while staying at g forever. We thus terminate only if c_{future} is 0 and generate a goal node that considers c_{future} otherwise. Finally, we expand node n [Line 16] and insert it into CLOSED [Line 17].

Expanding nodes. When expanding a node n (see Algorithm 6.4), we first store all reachable vertex-index pairs from vertex $n.v$ at a timestep within interval $[n.low, n.high)$ in $\mathscr{I}$ [Lines 2 to 6]. A vertex-index pair (v, id) is reachable iff the agent can move to v at a timestep within $\mathscr{T}[v][id]$, i.e., $\mathscr{T}[v][id]$ overlaps with $[n.low+1, n.high+1)$, or wait at v from interval $[n.low, n.high)$ to interval $\mathscr{T}[v][id]$, i.e., $n.high = \mathscr{T}[v][id].low$. For each vertex-index pair $(v, id) \in \mathscr{I}$ [Line 7], we use $[low, high)$ to represent the corresponding interval [Line 8]. We update low to the earliest arrival time at vertex v within $[low, high)$ without colliding with any hard edge obstacles [Line 9]. We jump to the next iteration if low does not exist [Line 10]. We then find the earliest arrival time low' at vertex v within $[low, high)$ without colliding with any hard or soft edge obstacles [Line 11]. If low' exists and $low' > low$ [Lines 12 to 16], then the agent will conflict with a soft edge obstacle if it arrives at v during timesteps $[low, low')$ and will not if it arrives at timestep low' (and waits at v if necessary). Thus, we generate two child nodes, one with safe interval $[low, low')$ and one with safe interval $[low', high)$. The former child node has one more conflict than the latter one. If low' does not exist or $low' = low$ [Lines 17 to 19], then we generate one child node as usual (The

Algorithm 6.4: Expand a SIPPS node.

```
Function EXPANDNODE($n$, OPEN, CLOSED, $\mathscr{T}$)
    $\mathscr{I} \leftarrow \emptyset$;
    for $v : (n.v, v) \in E$ do
        $\mathscr{I} \leftarrow \mathscr{I} \cup \{(v, id) \mid \mathscr{T}[v][id] \cap [n.low+1, n.high+1) \neq \emptyset, id \in \mathbb{N}^+\}$;
    if $\exists id : \mathscr{T}[n.v][id].low = n.high$ then
        $\mathscr{I} \leftarrow \mathscr{I} \cup \{(n.v, id) \mid \mathscr{T}[n.v][id].low = n.high, id \in \mathbb{N}^+\}$;  // Indicates wait actions
    foreach $(v, id) \in \mathscr{I}$ do
        $[low, high) \leftarrow \mathscr{T}[v][id]$;
        $low \leftarrow \min\{t \in [low, high) \mid t-1 \in [n.low, n.high) \wedge ((v, n.v), t) \notin \mathscr{O}^h\}$;
        if $low$ does not exist then continue;
        $low' \leftarrow \min\{t \in [low, high) \mid t-1 \in [n.low, n.high) \wedge ((v, n.v), t) \notin \mathscr{O}^h \cup \mathscr{O}^s\}$;
        if $low'$ exists $\wedge$ $low' > low$ then
            $n_1 \leftarrow Node(v, [low, low'), id, false)$;
            INSERTNODE($n_1$, OPEN, CLOSED) ;                // Algorithm 6.5
            $n_2 \leftarrow Node(v, [low', high), id, false)$;
            INSERTNODE($n_2$, OPEN, CLOSED);                 // Algorithm 6.5
        else
            $n_3 \leftarrow Node(v, [low, high), id, false)$;
            INSERTNODE($n_3$, OPEN, CLOSED);                 // Algorithm 6.5
```

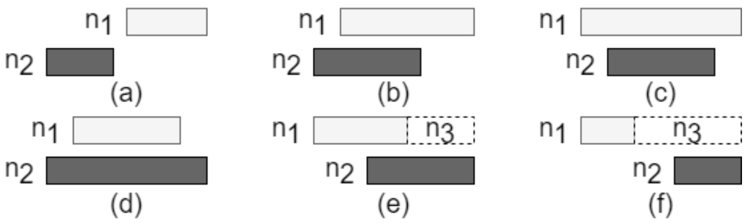


Figure 6.6: Illustration of all possible combinations of the relative positions of the safe intervals of two nodes n_1 and n_2 with the same identity. The timeline is from left to right. Without loss of generality, we assume that $c(n_1) < c(n_2)$ in (a), (b), and (d) and $c(n_1) \leq c(n_2)$ in (c), (e), and (f). We do not consider the cases where $c(n_1) = c(n_2)$ in (a), (b), and (d) because they are identical to the cases where $c(n_1) = c(n_2)$ in (f), (e), and (c), respectively.

case when low' does not exist results in one more conflict in the child node than the case when $low' = low$).

Inserting nodes. We say that two nodes n_1 and n_2 have the same *identity*, denoted as $n_1 \sim n_2$, iff $n_1.v = n_2.v$, $n_1.id = n_2.id$, and $n_1.is\_goal = n_2.is\_goal$. We say that n_1 *(weakly) dominates* n_2, denoted as $n_1 \succeq n_2$, iff $n_1 \sim n_2$, $[n_1.low, n_1.high) \supseteq [n_2.low, n_2.high)$, and $c(n_1) \leq c(n_2)$. We are interested in dominance because, if node n_1 dominates node n_2 (e.g., as in Figure 6.6(c)), we can prune n_2 without loss of completeness. Moreover, we know from Lines 7 to 19 in Algorithm 6.4 that a node n satisfies $n.high < \mathscr{T}[n.v][n.id].high$ iff it is generated on Line 13, i.e., there is a *twin node* n' with $n' \sim n$, $[n'.low, n'.high) = [n.high, \mathscr{T}[n.v][n.id].high)$, and $c(n') = c(n) - 1$. That is to say, if the situations in Figures 6.6(e) and (f) occur, although node n_1 does not dominate node n_2, there exists a twin node n_3 of node n_1 such that $n_1 \sim n_2 \sim n_3$, $[n_1.low, n_1.high) \cup [n_3.low, n_3.high) \supseteq [n_2.low, n_2.high)$, and $c(n_3) < c(n_1) \leq c(n_2)$. We can thus prune node n_2. Therefore, we generalize the definition of dominance as follow. We say that node n_1 *(weakly) dominates* node n_2, denoted as $n_1 \succeq n_2$, iff $n_1 \sim n_2$, $n_1.low \leq n_2.low$, and $c(n_1) \leq c(n_2)$. We can prune a node if it is dominated by another node. For situations when two nodes with the same identity have overlapping intervals but no dominance relationship (as in Figures 6.6(b) and (d)), the intersection of the two intervals would be explored twice if we expanded both nodes. We know that the node with the smaller lower bound always has the larger c-value (since, otherwise, the two nodes would have a dominance relationship), so we can shrink the interval of the node with the smaller lower bound by updating its upper bound to the lower bound of the the interval of the other node. This avoids the duplicate search effort without loss of completeness. The only unconsidered situation is the one shown in Figure 6.6(a), in which case we have to keep both nodes. In order to make SIPPS efficient, we use this pruning in SIPPS. As shown in Algorithm 6.5, we first compute the values of node n [Line 2] and collect all nodes in OPEN and CLOSED that have the same identity as node n [Line 3]. We need to compare with each such node so as to avoid duplicate search effort. Consider a node q [Line 4]. If it dominates node n [Line 5], then we do not need to generate node n and thus terminate [Line 6]. If it is dominated by node n [Line 7], then we do not need node q and thus remove it from OPEN and CLOSED [Line 8]. Otherwise, if the safe intervals of the two nodes overlap [Line 9], then we reset the upper bound of the interval with the smaller lower bound to the lower bound of the other interval [Lines 10 and 11].

Heuristics. To achieve high efficiency, most MAPF algorithms use the distance $dist(n.v, g)$ from vertex $n.v$ to vertex g as the h-value of node n when they plan paths for single agents, where the

Algorithm 6.5: Insert a SIPPS node.

```
Function INSERTNODE($n$, OPEN, CLOSED)
    Compute $g$-, $h$-, $f$-, and $c$-values of $n$;
    $\mathcal{N} \leftarrow \{q \in \text{OPEN} \cup \text{CLOSED} \mid q \sim n\}$;        // Nodes having same identity as n
    foreach $q \in \mathcal{N}$ do
        if $q.low \leq n.low \wedge c(q) \leq c(n)$ then                                  // $q \succeq n$
            return;                                                                        // No need to generate n
        else if $n.low \leq q.low \wedge c(n) \leq c(q)$ then                             // $n \succeq q$
            delete $q$ from OPEN and CLOSED;                                               // Prune q
        else if $n.low < q.high \wedge q.low < n.high$ then
            if $n.low < q.low$ then $n.high = q.low$;
            else $q.high = n.low$;
    insert $n$ into OPEN;
```

distances are computed during preprocessing. Such a heuristic is informed as long as the length of the optimal path p^* is not too much larger than $dist(s,g)$. Unfortunately, this is not always the case for PMDO for two reasons:

- $T = \max\{t \mid (g,t) \in \mathcal{O}^h\} + 1$ is a lower bound on $length(p^*)$ and can be substantially larger than $dist(s,g)$; and
- $T' = \max\{t \mid (g,t) \in \mathcal{O}^h \cup \mathcal{O}^s\} + 1$ is a lower bound on $length(p^*)$ when p^* has zero soft conflicts and can be substantially larger than $dist(s,g)$.

Therefore, we compute the h-value of a non-goal node n as

$$h(n) = \begin{cases} \max\{dist(n.v,g), T' - g(n)\}, & c(n) = 0 \\ \max\{dist(n.v,g), T - g(n)\}, & c(n) \geq 1, \end{cases} \tag{6.3}$$

The h-value of a goal node is, of course, 0.

Theoretical analysis. Below are two theorems for SIPPS. The proofs are omitted as they follow the proofs for SIPP.

Theorem 6.1. *SIPPS guarantees to return a path if one exists and "No Solution" otherwise.*

Theorem 6.2. *SIPPS guarantees to return a shortest path with zero soft conflicts if one exists.*

One limitation of SIPPS is that, if no zero-soft-conflict path exists, SIPPS may return a path that has more soft conflicts than the minimum because the c-value ignores the soft conflicts that occur when the agent waits within a safe interval that contains soft vertex obstacles. This approximation is acceptable since the minimization of the number of conflicts itself is an approximation of the CP minimization (i.e., minimization of the number of colliding pairs) used by MAPF-LNS2.[6] We have considered minimizing CP in SIPPS directly, but it is extremely inefficient as we have to keep track of the set of agents that the partial path from the root node to each node conflicts with, which substantially increases the search space.

Applications. Although SIPPS was designed for MAPF-LNS2, it can be used by a broad family of MAPF algorithms as PMDO is a problem that needs to be solved by many MAPF algorithms. Examples include the optimal MAPF algorithms ID [162] and CBS, the bounded-suboptimal MAPF algorithm ECBS, and the prioritized MAPF algorithm PBS [123] as well as their variants. With small changes to the priority function used by the open list of SIPPS (e.g., in CBS, prioritizing nodes with smaller f-values and breaking ties towards smaller c-values), SIPPS can speed up these MAPF algorithms while preserving their solution quality guarantees. Moreover, unlike space-time A*, SIPPS can also be applied in continuous-time settings. So, it can also speed up Continuous-Time CBS (CCBS) [5] and allows one to generalize CCBS to its suboptimal variants, e.g., Continuous-Time ECBS.

6.2.3 Neighborhood Selection

Following MAPF-LNS, we present three neighborhood selection methods and introduce adaptive LNS that intelligently combines these methods. Each neighborhood selection method derives from

[6]Empirically, we ran MAPF-LNS2 on the random map with 400 agents using the setup described in Section 6.2.4 and collected the results of 84,739 SIPPS runs. Among them, more than 95% of runs find the minimum-conflict paths, and 4% of runs find paths that contain only one more conflict than the minimum (where the minimum-conflict paths are found by space-time A*). Although space-time A* guarantees to find minimum-conflict paths, their CPs are occasionally larger than the CPs of the paths found by SIPPS.

Algorithm 6.6: Generate a conflict-based neighborhood.

```
Input: MAPF instance $(G,A)$, plan $P$, conflict graph $G_c$, and neighborhood size $N$
1  $v \leftarrow$ a random vertex in $\{v \in V_c \mid degree(v) > 0\}$;
2  $G'_c = (V'_c, E'_c) \leftarrow$ the largest connected component of $G_c$ that contains $v$;
3  if $|V'_c| \leq N$ then
4      $A_s \leftarrow \{a_v \in A \mid v \in V'_c\}$;
5      while $|A_s| < N$ do
6          $a_i \leftarrow$ a random agent in $A_s$;
7          $A_s \leftarrow A_s \cup \text{RANDOMWALK}(G, a_i, P)$;
8  else
9      $A_s \leftarrow \emptyset$;
10     while $|A_s| < N$ do
11         $A_s \leftarrow A_s \cup \{a_v\}$;
12         $v \leftarrow$ a random vertex in $\{u \in V'_c | (v,u) \in E'_c\}$;
13 return $A_s$;
14 Function RANDOMWALK$(G, a_k, P)$
15     $(x,t) \leftarrow (p_k[t], t)$, where $t$ is a random timestep in $[0, length(p_k) - 1]$;
16     while $t \leq \max\{length(p) \mid p \in P\}$ do
17         $y \leftarrow$ a random vertex in $\{u \in V \mid u = v \vee (v,u) \in E\}$;
18         $A_c \leftarrow \{a_i \in A \setminus \{a_k\} \mid p_i[t+1] = y \vee (p_i[t] = y \wedge p_i[t+1] = x)\}$;
19         if $A_c \neq \emptyset$ then return $A_c$;
20         $(x,t) \leftarrow (y, t+1)$;
21     return $\emptyset$;
```

a different motivation. Although there might be multiple implementations for each motivation, we present the one that works well for us and leave the exploration of other implementations for future work. We denote the current plan as P, the selected neighborhood as A_s, and the size of A_s as a predefined parameter N. $G_c = (V_c, E_c)$ is the *conflict graph*, where $V_c = \{i \mid a_i \in A\}$ and $E_c = \{(i,j) \mid p_i \in P$ conflicts with $p_j \in P\}$. We denote the degree of $i \in V_c$ as $degree(i)$.

6.2.3.1 Conflict-Based Neighborhood

A straightforward idea for generating neighborhoods that can potentially reduce CP is to select a subset of agents whose current paths conflict with each other. See Algorithm 6.6. To implement this idea, we first select a random vertex v from V_c with $degree(v) > 0$ [Line 1] and find the largest connected component $G'_c = (V'_c, E'_c)$ of conflict graph G_c that contains v [Line 2]. There are two cases:

- If $|V'_c| \leq N$ [Line 3], then we put all agents a_v with $v \in V'_c$ into A_s [Line 4] and repeatedly add additional agents that might conflict with some agents in A_s to A_s until $|A_s| = N$ [Lines 5 to 7]. At each iteration, we select a random agent from A_s [Line 6] and let it perform a random walk starting from a random vertex on its path [Line 15] and stop when it conflicts with another agent [Line 19], which is then added to A_s [Line 7]. In the experiments, we iterate for at most ten iterations (not shown in the pseudo-code) at which the random walk fails to find any conflicting agents (i.e., it returns an empty set on Line 21) to address the situation where the agent density is too low for us to collect N agents in A_s.

- Otherwise [Line 8], we select N vertices from V'_c via a random walk on G'_c starting at v and put the corresponding agents into A_s [Lines 9 to 12].

6.2.3.2 Failure-Based Neighborhood

The second idea for generating neighborhoods is to reason about why we failed to find conflict-free paths for some agents in the previous LNS iterations. Finding a path for an agent a_i that does not conflict with a given set of paths is an essential problem that is repeatedly solved in PP. Thus, previous work on PP has already studied this problem thoroughly [33]. Briefly speaking, there are two scenarios that result in failures, namely,

1. agent a_i is blocked by the agents from the given set of paths "sitting" at their target vertices surrounding agent a_i (see agent $a_1 = a_2$ in Figure 6.5a), i.e., all possible paths for agent a_i to reach its target vertex g_i are blocked by some target obstacles, and

2. agent a_i is "run over" by the given set of paths at (or around) its start vertex s_i during early timesteps (see agent $a_i = a_2$ in Figure 6.5b), i.e, the agent has no way to go.

Algorithm 6.7: Generate a failure-based neighborhood.

```
Input: MAPF instance $(G,A)$, plan $P$, conflict graph $G_c$, and neighborhood size $N$
1  Select an agent $a_i \in A$ with probability $\propto degree(i)$;
2  $A_s \leftarrow \{a_i\}$;
3  $A^s \leftarrow \{a_j \in A \mid p_j \in P$ visits $s_i\}$;
4  $A^g \leftarrow \{a_j \in A \mid p$ visits $g_j\}$;        // $p$ is the path from $s_i$ to $g_i$ that minimizes $|A^g|$
5  if $|A^s \cup A^g| \geq N-1$ then
6      if $|A^s| = 0$ then
7          $A' \leftarrow N-1$ random agents in $A^g$;
8          $A_s \leftarrow A_s \cup A'$;
9      else if $|A^g| \geq N-1$ then
10         $a_i \leftarrow$ the agent in $A^s$ that visits $s_i$ the earliest;
11         $A' \leftarrow N-2$ random agents in $A^g$;
12         $A_s \leftarrow A_s \cup \{a_i\} \cup A'$;
13     else
14         $A' \leftarrow$ the first $N-2-|A^g|$ agents in $A^s$ that visit $s_i$ the earliest;
15         $A_s \leftarrow A_s \cup A^g \cup A'$;
16 else if $|A^s \cup A^g| > 0$ then
17     $A_s \leftarrow A_s \cup A^s \cup A^g$;
18     while $|A_s| < N$ do
19         $a_i \leftarrow$ a random agent in $A_s$;
20         $a_j \leftarrow$ a random agent in $\{a_k \in A \mid p_i \in P$ visits $g_k\}$;
21         $A_s \leftarrow A_s \cup \{a_j\}$;
22 return $A_s$;
```

Therefore, the failure-based neighborhood focuses on an agent a_i that has conflicts and a set of agents whose paths visit vertex s_i or whose target vertices are on some path from vertex s_i to vertex g_i.

Formally, as shown in Algorithm 6.7, we first select an agent $a_i \in A$ with a probability proportional to $degree(i)$ (i.e., proportional to the number of agents that agent a_i conflicts with) [Line 1] and initialize A_s with it [Line 2]. We then collect two sets of agents $A^s = \{a_j \in A \mid p_j \in P \text{ visits } s_i\}$ [Line 3] and $A^g = \{a_j \in A \mid p \text{ visits } g_j\}$ [Line 4], where p is the path from s_i to g_i that minimizes $|A^g|$. There are three cases:

- If $|A^s \cup A^g| \geq N-1$ [Line 5], then we add $N-1$ agents to A_s using the following rule:
    - If $|A^s| = 0$ [Line 6], then we add $N-1$ random agents in A^g to A_s [Lines 7 and 8].

  - Otherwise, if $|A^g| \geq N-1$ [Line 9], then we add the agent in A^s that visits s_i the earliest and $N-2$ random agents in A^g to A_s [Lines 10 to 12].
  - Otherwise [Line 13], we add all agents in A^g and the first $N-1-|A^g|$ agents in A^s (from the sequence of agents in ascending order of the timesteps when their paths visit s_i) to A_s [Lines 14 and 15].

- Otherwise, if $|A^s \cup A^g| > 0$ [Line 16], then we add all agents in $A^s \cup A^g$ to A_s [Line 17] and then repeatedly add additional agents to A_s whose target vertices are visited by the paths of some agents in A_s until $|A_s| = N$ [Lines 18 to 21]. At each iteration, we select a random agent a_j from A_s and collect the agents whose target vertices are visited by $p_j \in P$. We select a random agent from the collected agents and add it to A_s. In the experiments, we terminate the while loop on Line 18 if the size of A_s remains the same after all agents in A_s have been selected (not shown in the pseudo-code) to address the situation where the agent density is too low for us to collect N agents in A_s.

- Otherwise, i.e., if $|A^s \cup A^g| = 0$, then we terminate and return $A_s = \{a_i\}$ [Line 22], because we are guaranteed to find a path for a_i that does not conflict with any other agents as a_i can sit at s_i until all other agents reach their target vertices and then move to g_i via path p.

This rule prefers agents in A^g slightly over agents in A^s because we find empirically that Scenario 1 occurs more frequently than Scenario 2 when PP fails.

6.2.3.3 Random Neighborhood

Generating neighborhoods randomly may sound naïve but has been shown to be extremely effective for many problems [52, 159]. Our third idea for generating neighborhoods therefore is to select N agents randomly, namely each a_i with a probability proportional to $degree(i) + 1$. We add one here in order to give the agents who do not conflict with others a chance to be selected.

6.2.3.4 Adaptive LNS (ALNS)

We use the same ALNS method as in Section 6.1.3.4. Formally, we maintain a weight w_i for each neighborhood selection method i that represents its relative success in reducing the CP. Initially, we set all w_i to 1. At each iteration, we select a method i with probability $w_i / \sum_j w_j$ to generate a neighborhood and replan the paths. After replanning, we set w_i to

$$w_i = \gamma \cdot \max\{0, c^- - c^+\} + (1 - \gamma) \cdot w_i, \tag{6.4}$$

where c^- and c^+ are the CPs of the plans before and after replanning, respectively, and $\gamma \in [0,1]$ is a user-specified reaction factor that controls how quickly the weights react to the changes in the relative success in reducing the CP. We use $\gamma = 0.1$ in our experiments. The weights for the other methods remain the same.

6.2.4 Empirical Evaluation

We compare MAPF-LNS2 against a representative set of scalable state-of-the-art MAPF algorithms, namely the bounded-suboptimal algorithm EECBS, the prioritized algorithms PP and PP^R, and the rule-based algorithm PPS. In addition, in order to show the effectiveness of SIPPS for speeding up MAPF algorithms other than MAPF-LNS2, we implement a variant of EECBS (denoted as EECBS*) that uses SIPPS instead of space-time A*. Both PP and PP^R use SIPP to plan paths for single agents (they do not use SIPPS as their underlying single-agent pathfinding problem does not have soft obstacles). Unless specified otherwise, MAPF-LNS2 uses PP to find the initial plans, ALNS to generate neighborhoods of size $N = 8$, and SIPPS to plan paths for single agents.

We use the random-scenario instances on all 33 maps from the MAPF benchmarks, yielding 25 instances per map and number of agents. 25 out of the 33 maps are shown in Figure 6.7, and the other 8 maps are 4 empty maps `empty-`i`-`i of size $i \times i$ for $i = 8, 16, 32, 48$ and 4 random maps `random-`i`-`i`-`j of size $i \times i$ with j% randomly blocked cells for $i = 32, 64$ and $j = 10, 20$. We conduct experiments on Amazon EC2 “m4.xlarge” instances with 16 GB of memory. Unless specified

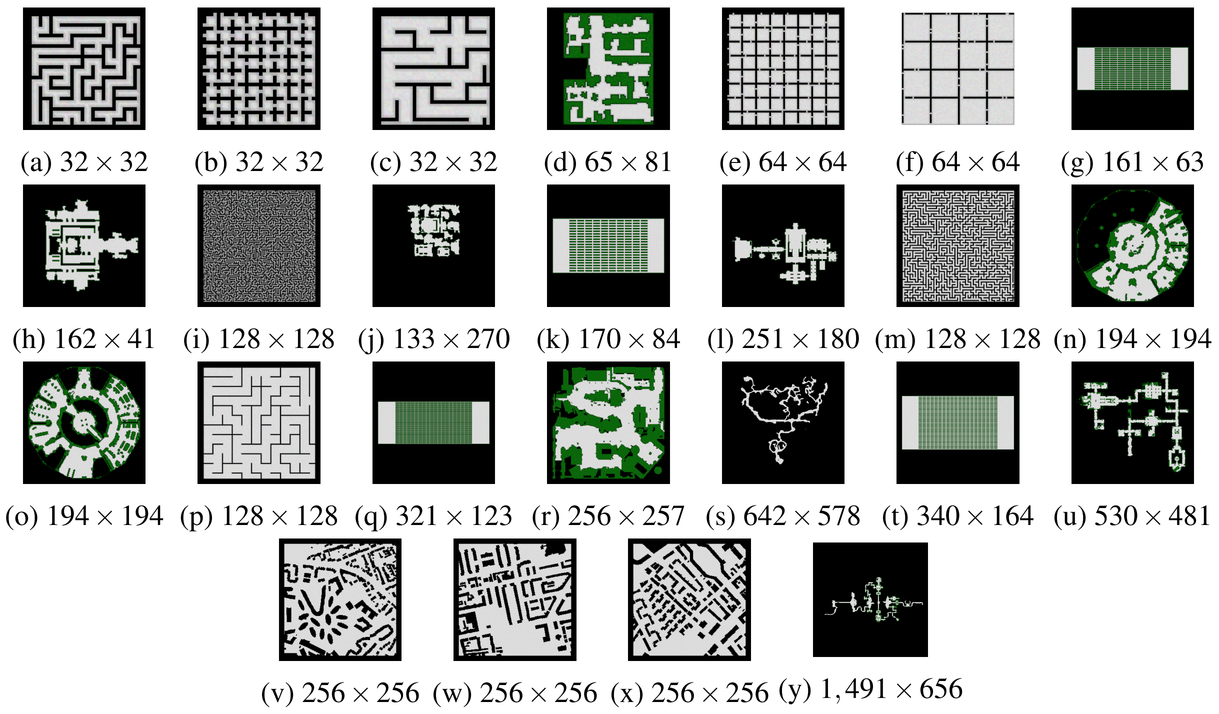


Figure 6.7: Illustrations of maps (a) `maze-32-32-2`, (b) `room-32-32-4`, (c) `maze-32-32-4`, (d) `den312d`, (e) `room-64-64-8`, (f) `room-64-64-16`, (g) `warehouse-10-20-10-2-1`, (h) `ht_chantry`, (i) `maze-128-128-1`, (j) `ht_mansion_n`, (k) `warehouse-10-20-10-2-2`, (l) `lt_gallowstemplar_n`, (m) `maze-128-128-2`, (n) `ost003d`, (o) `lak303d`, (p) `maze-128-128-10`, (q) `warehouse-20-40-10-2-1`, (r) `den520d`, (s) `w_woundedcoast`, (t) `warehouse-20-40-10-2-2`, (u) `brc202d`, (v) `Paris_1_256`, (w) `Berlin_1_256`, (x) `Boston_0_256`, and (y) `orz900d`. The caption of each map specifies its grid size.

m	Success rate		Runtime (s)		Runtime per call (ms)	
	Space-time A*	SIPPS	Space-time A*	SIPPS	Space-time A*	SIPPS
250	**1.00**	**1.00**	3.37	**0.64**	5.49 ± 17.19	1.11 ± 1.79
300	**1.00**	**1.00**	15.99	**2.67**	10.9 ± 28.72	1.94 ± 2.79
350	0.88	**1.00**	>68	**9.25**	15.83 ± 43.52	2.75 ± 3.72
400	0.68	**0.88**	>162	**>78**	15.28 ± 40.95	3.04 ± 4.23

Table 6.6: Performance of MAPF-LNS2 using different PMDO algorithms on the `random` map. "Runtime per call" is the average runtime of a single space-time A* or SIPPS search.

otherwise, the runtime limit is five minutes. We report results only on map `random-32-32-20` of size 32 × 32 (denoted as `random`) in Experiments 1-3 and map `warehouse-20-40-10-2-2` of size 340 × 164 (denoted as `warehouse`) in Experiment 5.

m	Success rate				Runtime (s)			
	Random	Failure	Conflict	ALNS	Random	Failure	Conflict	ALNS
250	**1.00**	**1.00**	**1.00**	**1.00**	0.80	**0.59**	0.79	0.64
300	**1.00**	**1.00**	**1.00**	**1.00**	13.13	3.41	3.09	**2.67**
350	**1.00**	0.96	**1.00**	**1.00**	32.57	>22	**9.11**	9.25
400	0.48	0.60	0.76	**0.88**	>192	>155	>128	>**78**

Table 6.7: Performance of MAPF-LNS2 using LNS with various neighborhood selection methods and MAPF-LNS2 using ALNS on the `random` map.

Experiment 1: PMDO algorithms. Table 6.6 compares MAPF-LNS2 with SIPPS against MAPF-LNS2 with space-time A* in terms of their *success rates* (i.e., percentages of instances solved within the runtime limit), average runtimes (with five minutes used for unsolved instances), and average runtimes per call with their standard deviations. SIPPS clearly dominates space-time A* with a speedup of more than five times. It is also more stable, e.g., the largest runtime per call for SIPPS is 51ms while that of space-time A* is 524ms (not shown in the table). This difference is even larger on larger maps.

Experiment 2: Neighborhood selection methods. Table 6.7 compares MAPF-LNS2 with ALNS against MAPF-LNS2 with the three individual neighborhood selection methods. As expected, ALNS performs the best as it combines the strengths of the other methods and is able to use a larger variety of neighborhoods. We also experiment with different neighborhood sizes $N = 4, 8, 16, 32$ but omit the results since they are similar to those reported in Section 6.1.4: There is no global winner, and larger neighborhoods increase the chance to find better MAPF solutions but require more time to replan, resulting in fewer iterations within the runtime limit.

Experiment 3: PP-based algorithms. We compare MAPF-LNS2 against other PP-based MAPF algorithms, namely PP and PP^R. MAPF-LNS2 can be viewed as a PP-based MAPF algorithm as it uses PP to both find initial plans and replan. As shown in Table 6.8, MAPF-LNS2 performs the best. It rapidly reduces the CP of the initial plan generated by PP and, as a result, substantially improves the success rate of PP. Its LNS framework is a more efficient approach than random restarts since MAPF-LNS2 requires significantly fewer runs of single-agent pathfinding than PP^R,

m	Success rate			Runtime (s)		#Single-agent runs		Initial
	PP	PP^R	MAPF-LNS2	PP^R	MAPF-LNS2	PP^R	MAPF-LNS2	CP
50	0.84	**1.00**	**1.00**	**0.01**	**0.01**	61	**53**	0.2
100	0.56	**1.00**	**1.00**	0.02	**0.01**	179	**105**	0.6
150	0.20	**1.00**	**1.00**	0.13	**0.05**	1,012	**170**	1
200	0.08	**1.00**	**1.00**	6.69	**0.14**	47,114	**262**	5
250	0.00	0.08	**1.00**	>288	**0.64**	-	**513**	20
300	0.00	0.00	**1.00**	>300	**2.67**	-	**1,285**	61
350	0.00	0.00	**1.00**	>300	**9.25**	-	**3,337**	155
400	0.00	0.00	**0.88**	>300	**>78**	-	-	316

Table 6.8: Comparison of MAPF-LNS2 against PP and PP^R on the `random` map. We omit the runtime of PP since it is equal to the runtime of PP^R and MAPF-LNS2 for any instance that it was able to solve. "#Single-agent runs" is the average number of times for which we run SIPP or SIPPS. We omit this result for PP because it is always equal to the number of agents m. "Initial CP" is the average CP of the initial plan.

which in turn results in significantly higher success rates and lower runtimes. Although MAPF-LNS2 failed to solve 3 instances with 400 agents, its final plans in these cases have only 1, 1, and 2 CPs (not shown in the table).

Experiment 4: State-of-the-art suboptimal MAPF algorithms. We compare MAPF-LNS2 against the state-of-the-art algorithms PP^R, PPS, and EECBS(5) as well as our EECBS*(5).[7] We use the instances in the random scenario on all 33 maps from the MAPF benchmarks with the largest number of agents, i.e., $m = \min\{0.5|V|, 1{,}000\}$ for each map. Figure 6.8 shows the runtime and solution quality (measured by an overestimated suboptimality) for each instance, and Figure 6.9 summarizes the success rates. MAPF-LNS2 solves more than 60% of the instances within a minute and 80% of the instances within five minutes. Its success rate is always the highest for all runtime limits. The instances that MAPF-LNS2 fails to solve are mostly on maps with lots of obstacles, such as the maze and room maps, and mostly not solved by the other algorithms either. Although PPS solves a few instances that are not solved by MAPF-LNS2, such as instances on map *room-32-32-4*, its solution quality is always substantially worse than that of MAPF-LNS2 (and the

[7]We pick 5 as the suboptimality factor because we intend to choose a large enough suboptimality factor such that, if EECBS fails to solve an instance that MAPF-LNS2 solves, it is due to the limited scalability of EECBS rather than it using a suboptimality factor that is too small.

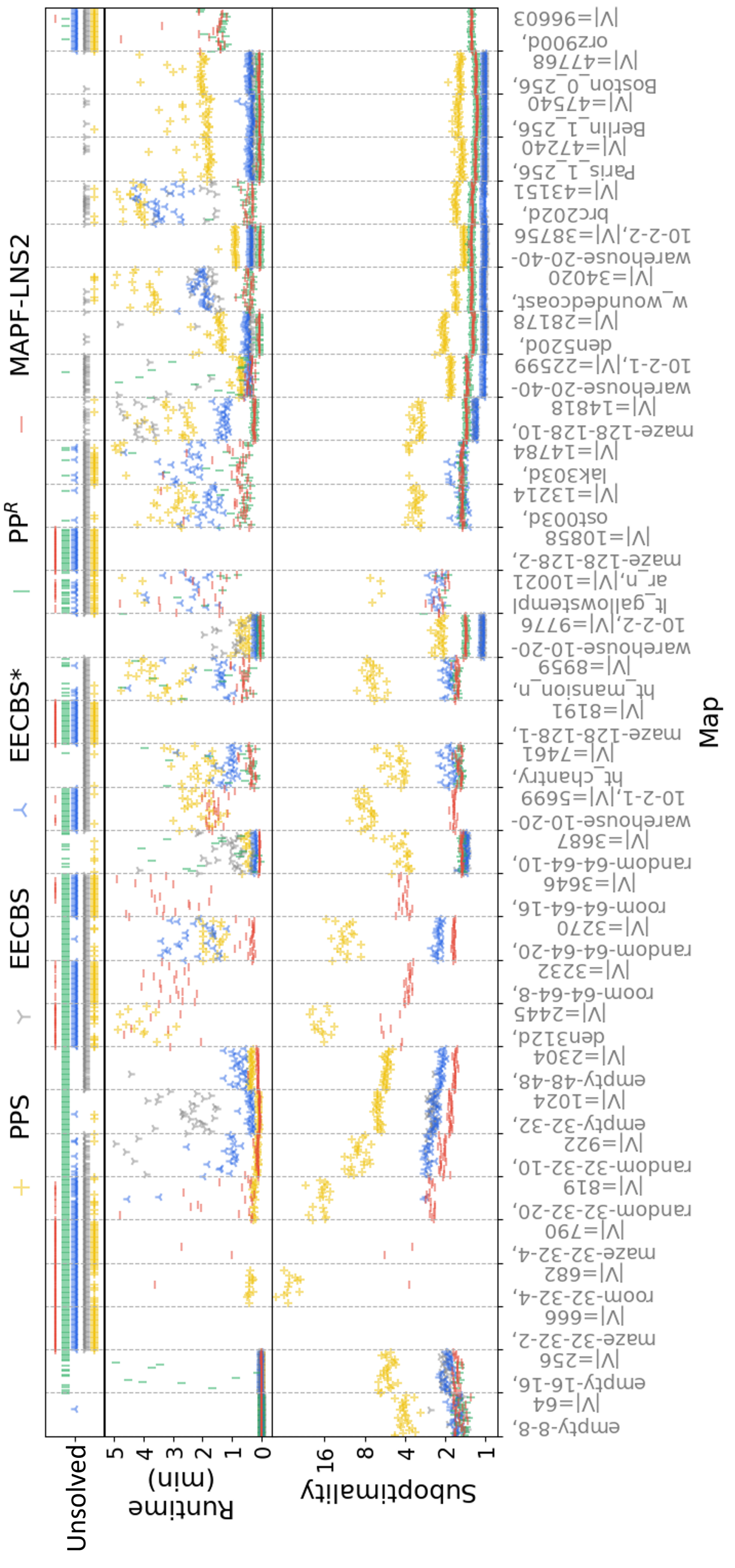


Figure 6.8: Runtimes and solution quality on all maps.

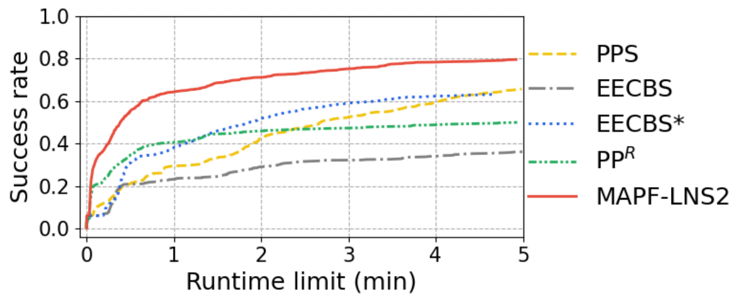


Figure 6.9: Success rates on all maps.

other algorithms). EECBS* finds MAPF solutions of slightly better quality than MAPF-LNS2 for some instances, yet its runtime is always larger. We did not use EECBS*/PPS to find initial plans for MAPF-LNS2 because, whenever PP finds MAPF solutions, it always finds them faster than EECBS*/PPS (as shown in Figure 6.8), and, whenever it fails to find them, MAPF-LNS2 repairs the plans rapidly and results in better success rates and runtimes than EECBS*/PPS eventually (as shown in Figure 6.8). In addition, the difference in the success rates and runtimes of EECBS and EECBS* clearly shows the advantage of SIPPS over space-time A*, especially on large maps, such as *ht_chantry*, *ost003d*, and *warehouse-20-40-10-2-1*. The success rate of EECBS* is almost twice of EECBS for a runtime limit of five minutes in Figure 6.9. The memory usage of PPS, EECBS, and EECBS* increases fast over time (as they generate longer and longer paths or larger and larger search frontiers), while that of PP^R and MAPF-LNS2 stays stable. Thus, PP^R and MAPF-LNS2 usually end up with a substantially smaller memory usage after five minutes than PPS, EECBS, and EECBS*.

Experiment 5: Longer runtime limits. We examine the effects of a longer runtime limit of an hour. Figure 6.10 shows that MAPF-LNS2 still performs the best. It plans conflict-free paths for 3,000 agents within a minute, 5,000 agents within five minutes, and 8,000 agents within a hour.

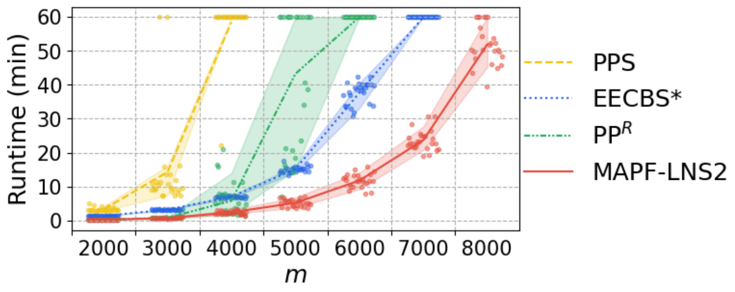


Figure 6.10: Runtimes on the `warehouse` map with a runtime limit of an hour. Each dot represents the runtime on one instance, with each line and filled area representing the mean and 0.1-quantile values over the 25 randomly generated instances for each number of agents. The runtime of each unsolved instance is set to an hour.

6.3 Combining MAPF-LNS and MAPF-LNS2

One limitation of the MAPF-LNS implementation in Section 6.1 is that no existing MAPF algorithm for finding initial solutions dominates the others. Thus, one must determine the MAPF algorithm for finding initial solutions manually. However, as shown in Section 6.2, MAPF-LNS2 significantly outperforms the existing algorithms in terms of runtimes on 30 out of 33 maps, i.e., all maps except for the 3 small maps with many blocked cells `room-32-32-4`, `random-32-32-20`, and `den312d`. Therefore, we can use MAPF-LNS2 to find initial MAPF solutions for MAPF-LNS. Algorithm 6.8 shows the pseudo-code. For simplicity, we still call the resulting algorithm MAPF-LNS2. MAPF-LNS2 contains three main steps:

1. finding an initial plan via PP [Lines 1 to 7],
2. repairing the plan via LNS if it has conflicts [Lines 8 to 19], and
3. reducing the solution cost via LNS until timeout [Lines 20 to 30].

Unlike MAPF-LNS in Section 6.1, that always uses space-time A* to plan single-agent paths, MAPF-LNS2 always uses SIPP or SIPPS as they run faster than space-time A*.

Algorithm 6.8: MAPF-LNS2 for solving MAPF suboptimally and in an anytime manner.

```
   Input: MAPF instance $(G,A)$
   /* PP for finding an initial plan                                          */
1  $success \leftarrow true$; $P \leftarrow \emptyset$;
2  for $a_i \in A$ do                          // Agents are selected in an random order
3      if $success = true$ then
4          $p_i \leftarrow SIPP(G,a_i,P)$;
5          if $p_i$ does not exist then $success \leftarrow false$;
6      if $success = false$ then $p_i \leftarrow SIPPS(G,a_i,P)$;
7      $P \leftarrow P \cup \{p_i\}$;
   /* LNS for repairing the plan to a solution                                */
8  if $success = false$ then
9      Initialize the weights $\vec{w}$ of the destroy heuristics for MAPF-LNS;
10     while $P$ is not conflict-free do
11         $A_s \leftarrow$ SELECTNEIGHBORHOOD$(G,A,P,\vec{w})$;     // by ALNS in Section 6.2.3
12         $P_s^- \leftarrow \{p_i \in P \mid a_i \in A_s\}$;
13         $P_s^+ \leftarrow \emptyset$;
14         for $a_i \in A_s$ do                // Agents are selected in an random order
15             $p_i \leftarrow SIPPS(G,a_i,P \setminus P_s^- \cup P_s^+)$;
16             $P_s^+ \leftarrow P_s^+ \cup \{p_i\}$;
17         $P' \leftarrow P \setminus P_s^- \cup P_s^+$;
18         if COLLIDINGPAIRS$(P') \leq$ COLLIDINGPAIRS$(P)$ then $P \leftarrow P'$;
19         Update $\vec{w}$;
   /* LNS for reducing the MAPF solution cost                                 */
20 Initialize the weights $\vec{w}$ of the destroy heuristics for MAPF-LNS2;
21 while not timeout do
22     $A_s \leftarrow$ SELECTNEIGHBORHOOD$(G,A,P,\vec{w})$;         // by ALNS in Section 6.1.3
23     $P_s^- \leftarrow \{p_i \in P \mid a_i \in A_s\}$;
24     $P_s^+ \leftarrow \emptyset$;
25     for $a_i \in A_s$ do                    // Agents are selected in an random order
26         $p_i \leftarrow SIPP(G,a_i,P \setminus P_s^- \cup P_s^+)$;
27         if $p_i$ does not exist then Go to Line 30;
28         $P_s^+ \leftarrow P_s^+ \cup \{p_i\}$;
29     if $\sum_{p \in P_s^+} length(p) \leq \sum_{p \in P_s^-} length(p)$ then $P \leftarrow P \setminus P_s^- \cup P_s^+$;
30     Update $\vec{w}$;
31 return $P$;
```

	m	Ins	Iterations	Average delay			Suboptimality	
				Initial	Final	Ratio	Initial	Final
empty-8-8	32	25	3,094,813	2.8	0.8	3.40	≤1.58	≤1.17
empty-16-16	128	25	550,723	7.0	2.5	2.77	≤1.64	≤1.23
room-32-32-4	341	1	1,326	63.0	43.1	1.46	≤3.50	≤2.71
maze-32-32-4	395	1	305	201.1	141.6	1.42	≤5.80	≤4.38
random-32-32-20	409	15	42,164	35.1	19.9	1.76	≤2.57	≤1.89
random-32-32-10	461	25	82,690	23.5	12.0	1.95	≤2.07	≤1.55
empty-32-32	512	25	79,050	17.4	8.7	2.00	≤1.82	≤1.41
empty-48-48	1,000	25	29,567	21.2	11.9	1.78	≤1.67	≤1.38
den312d	1,000	9	194	209.9	196.0	1.07	≤4.87	≤4.62
room-64-64-8	1,000	21	649	161.9	148.1	1.09	≤3.74	≤3.50
random-64-64-20	1,000	25	13,886	31.4	19.7	1.59	≤1.71	≤1.45
room-64-64-16	1,000	19	203	211.3	197.1	1.07	≤4.06	≤3.86
random-64-64-10	1,000	25	30,384	20.7	8.1	2.57	≤1.49	≤1.19
warehouse-10-20-10-2-1	1,000	23	3,187	56.9	31.7	1.79	≤1.71	≤1.40
ht_chantry	1,000	25	2,464	52.0	35.2	1.48	≤1.55	≤1.37
ht_mansion_n	1,000	25	1,488	64.3	48.3	1.33	≤1.62	≤1.47
warehouse-10-20-10-2-2	1,000	25	20,414	34.6	5.5	6.31	≤1.39	≤1.06
lt_gallowstemplar_n	1,000	18	164	131.2	125.3	1.05	≤2.17	≤2.12
ost003d	1,000	25	1,022	72.9	49.0	1.49	≤1.48	≤1.32
lak303d	1,000	25	794	89.4	66.0	1.35	≤1.48	≤1.36
maze-128-128-10	1,000	25	2,043	72.2	43.0	1.68	≤1.37	≤1.22
warehouse-20-40-10-2-1	1,000	25	5,829	58.3	13.2	4.42	≤1.35	≤1.08
den520d	1,000	25	6,795	38.1	7.0	5.46	≤1.22	≤1.04
w_woundedcoast	1,000	25	1,356	107.7	66.6	1.62	≤1.25	≤1.16
warehouse-20-40-10-2-2	1,000	25	12,725	46.6	0.4	110.02	≤1.26	≤1.00
brc202d	1,000	25	1,273	98.3	57.3	1.72	≤1.24	≤1.14
Paris_1_256	1,000	25	11,432	33.0	1.3	25.47	≤1.17	≤1.01
Berlin_1_256	1,000	25	10,853	25.7	1.6	15.78	≤1.14	≤1.01
Boston_0_256	1,000	25	8,421	32.4	3.2	10.05	≤1.17	≤1.02
orz900d	1,000	24	158	291.0	244.0	1.19	≤1.26	≤1.22

Table 6.9: Performance of MAPF-LNS and MAPF-LNS2 on all maps. Results on maps maze-32-32-2, maze-128-128-1, and maze-128-128-2 are omitted due to their 0% success rates. "Iterations" are the average number of iterations that we run MAPF-LNS for reducing the solution costs, i.e., the average number of times that Lines 22 to 30 in Algorithm 6.8 are executed. "Average delay" is the average sum of delays of the initial/final MAPF solution divided by the number of agents, and its ratio is the number in the "Initial" column divided by the number in the "Final" column.

6.3.1 Empirical Evaluation

We repeat Experiment 4 in Section 6.2 for this new version of MAPF-LNS2 and report the results in Table 6.9. This time, since we can solve more challenging MAPF instances than we have solved in Section 6.1 and use a longer runtime limit, we show more significant improvements in terms of the MAPF solution quality, i.e., we reduce the average delays of the MAPF solutions by up to 110 times.

Since Figure 6.8 shows that EECBS* sometimes finds solutions of better quality than MAPF-LNS2 (even though EECBS* has much lower success rates overall, as shown in Figure 6.9), we present a detailed comparison of the solution qualities of EECBS* versus MAPF-LNS2 in Figure 6.11. Although the quality of the initial MAPF solution of MAPF-LNS2 is worse than that of EECBS*(5) on many maps, the quality of its final MAPF solution is either similar or better on all maps except maps `w_woundedcoast` and `brc202d`. We also examine EECBS* with smaller suboptimality factors, namely $w = 1.1$ and $w = 1.5$. Compared to EECBS*(5), only on maps `empty-8-8`, `empty-16-16`, `random-64-64-10`, and `ost003d` is the solution quality significantly improved. On the other maps, EECBS* with smaller suboptimality factors either finds similar quality solutions or fails to find any solutions within the runtime limit. Among the four maps on which EECBS* with smaller suboptimality factors finds better solutions than EECBS*(5), only on map `ost003d` does EECBS* with smaller suboptimality factors find better solutions than MAPF-LNS2. Therefore, although MAPF-LNS2 does not have theoretical guarantees, it finds MAPF solutions of qualities that are at least as good as those of EECBS* in most cases no matter what suboptimality factors are used.

6.4 Summary

In this chapter, we provided first evidence that the use of Large Neighborhood Search (LNS) leads to very scalable and high-quality solutions to MAPF. Specifically:

- Our anytime MAPF algorithm MAPF-LNS significantly outperforms the existing anytime MAPF algorithm anytime BCBS in terms of success rates, runtimes to the first solutions, and speeds of improving the solutions. On easy instances that the optimal algorithm CBS can solve within a minute, MAPF-LNS finds solutions that are optimal in most cases and

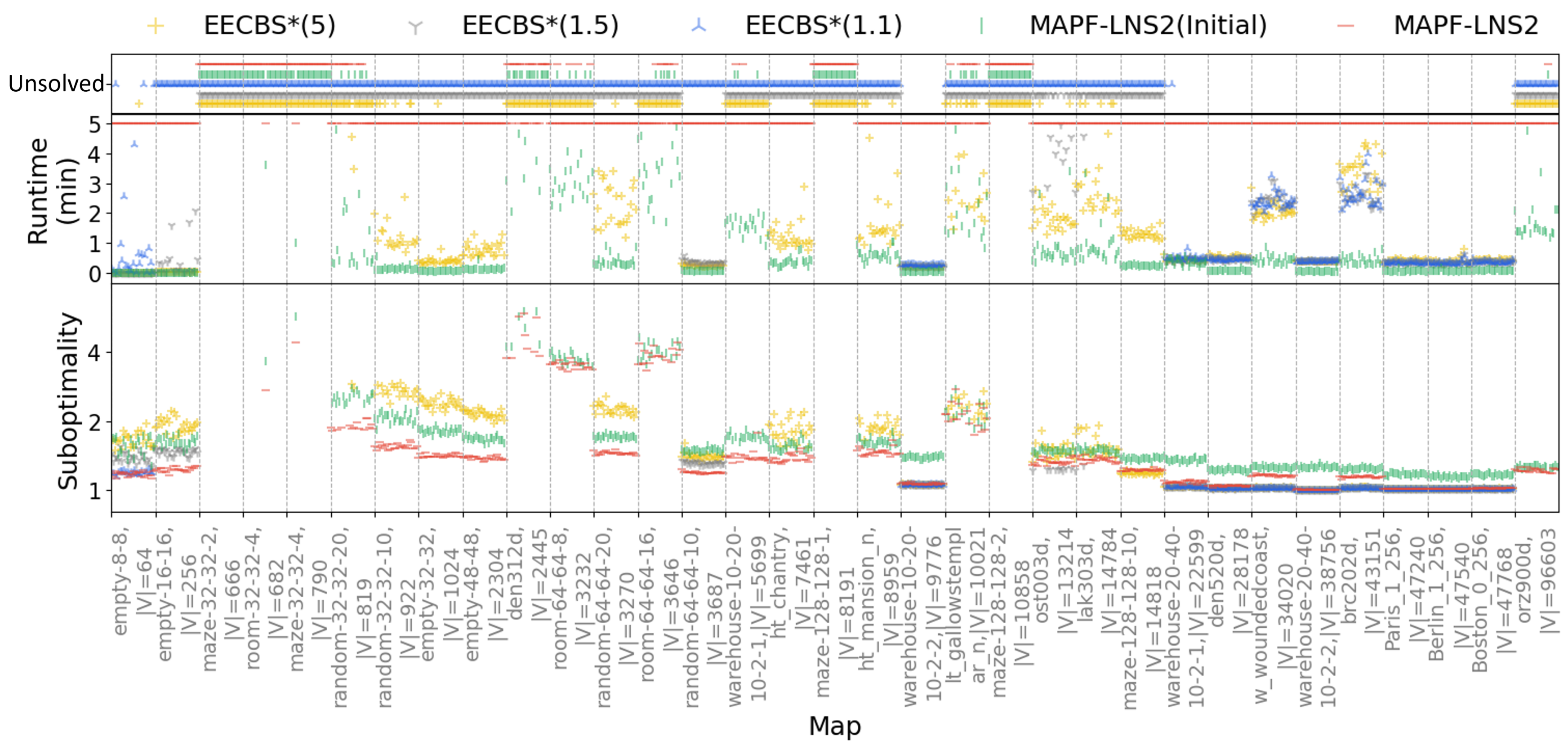


Figure 6.11: Runtimes and solution quality of MAPF-LNS2 and EECBS* on all maps. MAPF-LNS2(Initial) represents the results with respect to the first MAPF solution that MAPF-LNS2 finds, which is identical to MAPF-LNS2 in Figure 6.8.

within 1.35% of optimal in the worst case. On harder instances that the bounded-suboptimal algorithm EECBS can solve, MAPF-LNS rapidly improves the solution found by EECBS to near-optimal. On very challenging instances that only the unbounded-suboptimal algorithms PPS or PP^R can solve, MAPF-LNS rapidly reduces the sum of delays of the solution found by PPS or PP by up to 36 times.

- Our unbounded-suboptimal MAPF algorithm MAPF-LNS2 solves 80% of the most challenging MAPF-benchmark instances within a runtime limit of just five minutes, which significantly outperforms a variety of state-of-the-art MAPF algorithms, including EECBS, PP^R, and PPS. In addition, the single-agent path planner SIPPS used by MAPF-LNS2 runs five times (or more) faster than space-time A* and can be used to speed up a variety of MAPF algorithms. For example, it almost doubles the success rate of EECBS with a runtime limit of five minutes in our experiments.

- The combination of MAPF-LNS and MAPF-LNS2 leads to a strong anytime MAPF algorithm that solves 80% of the most challenging MAPF-benchmark instances within a runtime limit of five minutes and finds MAPF solutions whose costs are smaller, in most cases, than those of other non-optimal MAPF algorithms, including EECBS with different suboptimality factors, PP^R, and PPS.

6.5 Extensions

Just like CBS-based MAPF algorithms, LNS-based MAPF algorithms are flexible and can be easily generalized to different variants of MAPF problems. For example, we have applied MAPF-LNS to the 2020 Flatland Challenge, a NeurIPS competition about planning conflict-free paths for trains on rail networks [98]. We won both rounds of the competition, and MAPF-LNS was an essential algorithm in our software which helped to improve our score by 0.010 (= 3 times the difference in score to the team in second place) in Round 1 and 0.709 (= 61% of the difference in score to the team in second place) in Round 2. More details can be found in [110]. In the 2021 Flatland

Challenge, trains have different speeds and different departure and arrival time windows. We modified the neighborhood selection methods and objectives of MAPF-LNS to take these changes into account and won the competition again. In addition, we also showed that MAPF-LNS could be used to not only find conflict-free paths during planning but also reduce the costs of existing paths during execution. Applying MAPF-LNS during planning and execution improved our score by 0.205 and 0.264, respectively.

MAPF-LNS can be further improved by using machine learning to generate neighborhoods [87] and using simulated annealing to determine runtime limits [110].

Chapter 7

Conclusions and Future Work

Coordinating large teams of agents is a computationally challenging yet important problem for many applications. This dissertation builds the algorithmic foundations for solving one of the key multi-agent coordination problems, namely Multi-Agent Path Finding (MAPF), efficiently and effectively by exploiting the combinatorial structure of the MAPF problem and combining ideas from both artificial intelligence and operations research. We studied a variety of MAPF algorithms, including optimal, bounded-suboptimal, and unbounded-suboptimal MAPF algorithms:

- We studied optimal MAPF algorithms in Chapters 3 and 4. Since state-of-the-art optimal MAPF algorithms, such as CBS, have to perform a systematic search in the conflict-resolution space to prove optimality, we introduced two techniques to reduce the search effort, namely adding admissible heuristics, which prunes the search space that leads to costly solutions, and symmetry reasoning, which reduces the search space by eliminating symmetry conflicts. Specifically:
  - In Chapter 3, we developed the admissible heuristics CG, DG, and WDG to focus the search of CBS. Theoretically, the WDG heuristic is provably at least as informed as the DG heuristic, which in turn is provably at least as informed as the CG heuristic. Empirically, adding any one of thees three admissible heuristics to CBS can reduce its number of expanded CT nodes and its runtimes. The WDG heuristic performs the best, reducing its runtime by up to a factor of fifty.

  - In Chapter 4, we developed symmetry reasoning techniques, including generalized rectangle, target, and generalized corridor reasoning, to reduce the search space of optimal CBS. Every symmetry reasoning technique has different performances on different maps. Their combination performs the best. It scales up CBS by up to a factor of thirty in terms of number of agents and reduces its number of expanded CT nodes by up to four orders of magnitude.

- We studied bounded-suboptimal MAPF algorithms in Chapter 5. Like optimal MAPF algorithms, bounded-suboptimal MAPF algorithms need to perform a systematic search in the conflict-resolution space to prove bounded optimality, so the techniques that we developed for optimal MAPF solving in Chapters 3 and 4 can be applied here as well. Unlike optimal MAPF algorithms, bounded-suboptimal MAPF algorithms have the flexibility to greedily find MAPF solutions that are not optimal, so we developed a learned heuristic to focus the search of bounded-suboptimal CBS that trades off solution quality for runtime. The resulting algorithm EECBS significantly outperforms the state-of-the-art bounded-suboptimal MAPF algorithm ECBS and can, for example, find MAPF solutions that are provably at most 2% worse than optimal for large MAPF instances with up to 1,000 agents within just one minute. We showed in Chapter 6 that EECBS can be sped up further by replacing state-time A* with SIPPS on its low level.

- We studied unbounded-suboptimal MAPF algorithms in Chapter 6. Sometimes, we are interested in good solutions but not necessarily proof of how good the solutions are. We thus developed a first framework based on Large Neighborhood Search (LNS) for solving MAPF greedily with no theoretical guarantees. Our framework can significantly improve the solution quality of any non-optimal MAPF algorithms, including the bounded-suboptimal MAPF algorithm EECBS developed in Chapter 5, and scales significantly better than them. It solved 80% of the most challenging instances in the MAPF benchmark suite and found near-optimal solutions in most cases.

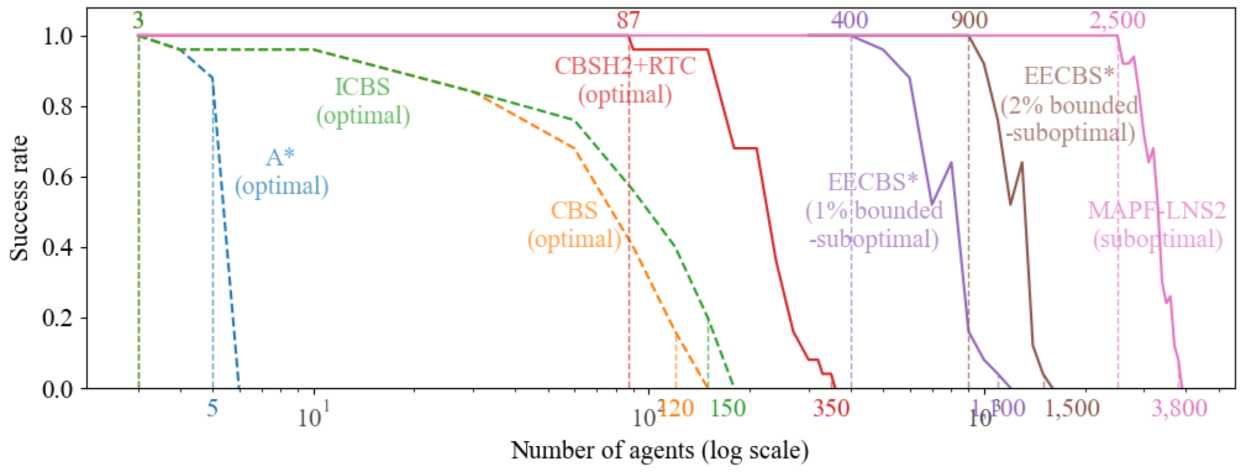


Figure 7.1: Success rates (= percentages of solved instances within one minute) on map `Paris_1_256`. Solid lines correspond to the MAPF algorithms introduced in this dissertation and dashed lines to existing MAPF algorithms. EECBS* is EECBS with SIPPS. The numbers shown on the top and bottom with colors indicate the largest numbers of agents that the corresponding MAPF algorithm can solve with a 100% success rate and a non-zero success rate, respectively. The solutions found by MAPF-LNS2 are, for example, 32% and 44% worse than optimal averaged over instances with 2,500 agents and 3,800 agents, respectively, where the optimal sum of cost is underestimated by $\sum_{a_i \in A} dist(s_i, g_i)$.

With these techniques, we have developed the state-of-the-art optimal MAPF algorithm CBSH2-RTC[1] (see Algorithm 4.6), the state-of-the-art bounded-suboptimal MAPF algorithm EECBS[2] (see Algorithm 5.1), and the state-of-the-art unbounded-suboptimal MAPF algorithm MAPF-LNS2[3] (see Algorithm 6.8), that scale to a few hundred agents, a thousand agents, and a few thousand agents, respectively. Figure 7.1 highlights the performance of these algorithms on map `Paris_1_256` (see Figure 6.7v) from the MAPF benchmark suite in comparison to the existing optimal MAPF algorithms A* (i.e., performing A* in the joint-state space of the agents), CBS [153] (see Algorithm 2.1), and ICBS [28].[4] We have also developed the state-of-the-art anytime MAPF algorithm MAPF-LNS,[5] that dominates the existing anytime MAPF algorithms in

[1]`https://github.com/Jiaoyang-Li/CBSH2-RTC`

[2]`https://github.com/Jiaoyang-Li/EECBS`

[3]`https://github.com/Jiaoyang-Li/MAPF-LNS2`

[4]We conducted experiments on Amazon EC2 "m4.xlarge" instances with 16 GB of memory and used instances from the "random" scenario from the MAPF benchmark suite, yielding 25 instances per number of agents. Since these instances contain only 1,000 pairs of start and target vertices, we generated start and target vertices uniformly at random when we needed more than 1,000 agents.

[5]`https://github.com/Jiaoyang-Li/MAPF-LNS`

terms of scalability, runtime to the initial solution, and speed of improving the solution. It rapidly reduces the solution cost of non-optimal MAPF algorithms by up to 36 times within just a minute and up to 110 times within five minutes (see Chapter 6).

The techniques introduced in this dissertation could be further enhanced to solve MAPF even more efficiently and effectively. Possible future work includes:

- **Reasoning about admissible heuristics and symmetries for more than two agents.** Our techniques for both computing admissible heuristics and breaking symmetries focus on pairs of agents: CG, DG, and WDG heuristics are computed based on pairwise dependency graphs of the agents, and generalized rectangle, target, and generalized corridor symmetries are studied between pairs of agents as well. The challenges of reasoning about groups of more than two agents are two-fold:

  - When we reason about two agents, there are $\binom{m}{2}$ pairs of agents, and this number can be significantly reduced by considering only pairs of agents whose paths are in conflict. However, when we reason about more than two agents, the number of groups of agents that we need to consider is significantly larger, and considering only the agents whose paths are in conflict is not sufficient any longer. Therefore, we need to design clever strategies to select groups of highly-coupled agents to perform such reasoning on.
  - For admissible heuristics, although we can use similar ideas for building dependency graphs and solving modified versions of the minimum vertex cover problem when reasoning about groups of more than two agents, the runtime overhead of both building the dependency graphs and solving the vertex cover problems becomes larger. For symmetry reasoning, it is unclear whether it is possible to design a binary branching strategy to resolve symmetry conflicts among more than two agents. We thus might have to develop new types of symmetry-breaking constraints.

  Nevertheless, there is some recent work that shows promise. Gange et al. [67] apply the nogood learning technique to CBS which allows them to discover groups of highly-coupled agents from the CT and obtain lower bounds on their sum of costs. Mogali et al. [126] project sub-MAPF problems to lower-dimensional “templates” and build a template database offline for groups of three agents, which one can use to compute lower bounds on their sum of costs via Lagrangian Relax-and-Cut. Han and Yu [75] focus on four-neighbor grids and build a MAPF solution database for agents within a small region, e.g., a 2×3 grid.

- **Developing more informed learned heuristics for bounded-suboptimal MAPF algorithms.** The focus of Chapter 5 was building the EECBS framework where one can deploy (potentially inadmissible) learned heuristics to find MAPF solutions with bounded-suboptimal guarantees. The learned heuristic that we use in EECBS is very simple and has potential for large improvements. There are many recent works on learning heuristics for A* search and its variants [202, 7, 160, 1] that show promise. In the context of MAPF, Huang et al. [86] use imitation learning to learn CT node selection rules and significantly speed up ECBS. It is thus interesting to design more advanced learned heuristics for EECBS.

- **Providing theoretical guarantees for LNS-based MAPF frameworks.** MAPF-LNS and MAPF-LNS2 exhibit good empirical performance but provide neither completeness nor optimality guarantees. Therefore, it is interesting future work to develop versions of MAPF-LNS that are bounded-suboptimal and/or complete. Possible approaches include:

  – To develop a version of MAPF-LNS that is bounded-suboptimal, we can use LNS to simultaneously find MAPF solutions of decreasing costs and provide increasing lower bounds on the cost of the optimal MAPF solution. MAPF-LNS in Chapter 6 can already find MAPF solutions of decreasing costs. To provide increasing lower bounds, we can repeatedly select subsets of agents, find optimal MAPF solutions for them, and use their optimal costs to underestimate the cost of the optimal solution for all agents via a generalized weighted dependency graph, just like how we compute the WDG heuristic.

  - To develop a version of MAPF-LNS that is complete, we can design adaptive methods for determining the sizes of the neighborhoods and the MAPF algorithms used for replanning.

In addition to the directions mentioned above, it is also interesting to study how these techniques apply to additional generalized MAPF problems.

Bibliography

[1] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1 (8):356–363, 2019.

[2] Faten Aljalaud and Nathan R. Sturtevant. Finding bounded suboptimal multi-agent path planning solutions using increasing cost tree search (extended abstract). In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 203–204, 2013.

[3] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006.

[4] Anton Andreychuk. Multi-agent path finding with kinematic constraints via conflict based search. In *Proceedings of the Russian Conference on Artificial Intelligence (RCAI)*, pages 29–45, 2020.

[5] Anton Andreychuk, Konstantin S. Yakovlev, Dor Atzmon, and Roni Stern. Multi-agent pathfinding with continuous time. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 39–45, 2019.

[6] Anton Andreychuk, Konstantin S. Yakovlev, Eli Boyarski, and Roni Stern. Improving continuous-time conflict based search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 11220–11227, 2021.

[7] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[8] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. Robust multi-agent path finding. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 2–9, 2018.

[9] Dor Atzmon, Amit Diei, and Daniel Rave. Multi-train path finding. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 125–129, 2019.

[10] Dor Atzmon, Roni Stern, Ariel Felner, Nathan R. Sturtevant, and Sven Koenig. Probabilistic robust multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 29–37, 2020.

[11] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. Robust multi-agent path finding and executing. *Journal of Artificial Intelligence Research*, 67:549–579, 2020.

[12] Dor Atzmon, Shahar Idan Freiman, Oscar Epshtein, Oran Shichman, and Ariel Felner. Conflict-free multi-agent meeting. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 16–24, 2021.

[13] Vincenzo Auletta, Angelo Monti, Mimmo Parente, and Pino Persiano. A linear-time algorithm for the feasibility of pebble motion on trees. *Algorithmica*, 23(3):223–245, 1999.

[14] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 73–87, 1999.

[15] Jacopo Banfi, Nicola Basilico, and Francesco Amigoni. Intractability of time-optimal multirobot path planning on 2D grid graphs with holes. *IEEE Robotics and Automation Letters*, 2(4):1941–1947, 2017.

[16] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 19–27, 2014.

[17] Roman Barták, Jiri Svancara, and Marek Vlk. A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 748–756, 2018.

[18] Matteo Bellusci, Nicola Basilico, and Francesco Amigoni. Multi-agent path finding in configurable environments. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 159–167, 2020.

[19] Gleb Belov, Wenbo Du, Maria Garcia de la Banda, Daniel Harabor, Sven Koenig, and Xinrui Wei. From multi-agent pathfinding to 3D pipe routing. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 11–19, 2020.

[20] Belaid Benhamou. Study of symmetry in constraint satisfaction problems. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming (CP)*, pages 246–254, 1994.

[21] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Optimizing schedules for prioritized path planning of multi-robot systems. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 271–276, 2001.

[22] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous Systems*, 41(2-3):89–99, 2002.

[23] Alexander Berndt, Niels Van Duijkeren, Luigi Palmieri, and Tamas Keviczky. A feedback scheme to reorder a multi-agent execution schedule by persistently optimizing a switchable action dependency graph. In *ICAPS Workshop on Distributed and Multi-Agent Planning (DMAP)*, pages 1–9, 2020.

[24] Gustav Björdal, Pierre Flener, Justin Pearson, Peter J. Stuckey, and Guido Tack. Solving satisfaction problems using large-neighbourhood search. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 55–71, 2020.

[25] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[26] Dragan Bosnacki and Mark Scheffer. Partial order reduction and symmetry with multiple representatives. In *Proceedings of the International Symposium on NASA Formal Methods (NFM)*, pages 97–111, 2015.

[27] Eli Boyarski, Ariel Felner, Guni Sharon, and Roni Stern. Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 47–51, 2015.

[28] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 740–746, 2015.

[29] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 740–746, 2015.

[30] Eli Boyarski, Ariel Felner, Daniel Harabor, Peter J. Stuckey, Liron Cohen, Jiaoyang Li, and Sven Koenig. Iterative-deepening conflict-based search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4084–4090, 2020.

[31] Eli Boyarski, Ariel Felner, Pierre Le Bodic, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. f-aware conflict prioritization & improved heuristics for conflict-based search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 12241–12248, 2021.

[32] Kyle Brown, Oriana Peltzer, Martin A. Sehr, Mac Schwager, and Mykel J. Kochenderfer. Optimal sequential task assignment and path finding for multi-agent robotic assembly planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 441–447, 2020.

[33] Michal Cáp, Peter Novák, Alexander Kleiner, and Martin Selecký. Prioritized planning algorithms for trajectory coordination of multiple mobile robots. *IEEE Transactions on Automation Science and Engineering*, 12(3):835–849, 2015.

[34] Michal Cáp, Jean Gregoire, and Emilio Frazzoli. Provably safe and deadlock-free execution of multi-robot plans under delaying disturbances. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5113–5118, 2016.

[35] Shao-Hung Chan, Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Graeme Gange, Liron Cohen, and Sven Koenig. Nested ECBS for bounded-suboptimal multi-agent path finding. In *IJCAI Workshop on Multi-Agent Path Finding (WoMAPF)*, 2020.

[36] Shao-Hung Chan, Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. Flex distribution for bounded-suboptimal multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 9313–9322, 2022.

[37] Jingkai Chen, Jiaoyang Li, Chuchu Fan, and Brian C. Williams. Scalable and safe multi-agent motion planning with nonlinear dynamics and bounded disturbances. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 11237–11245, 2021.

[38] Jingkai Chen, Jiaoyang Li, Yijiang Huang, Caelan Reed Garrett, Dawei Sun, Chuchu Fan, Andreas G. Hofmann, Caitlin Mueller, Sven Koenig, and Brian C. Williams. Cooperative task and motion planning for multi-arm assembly systems. *CoRR*, abs/2203.02475, 2022.

[39] Zhe Chen, Daniel Harabor, Jiaoyang Li, and Peter J. Stuckey. Symmetry breaking for k-robust multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 12267–12274, 2021.

[40] Zhe Chen, Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. Multi-train path finding revisited. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 38–46, 2022.

[41] Peng Cheng, Emilio Frazzoli, and Steven M. LaValle. Improving the performance of sampling-based motion planning with symmetry-based gap reduction. *IEEE Transactions on Robotics*, 24(2):488–494, 2008.

[42] Shushman Choudhury, Jayesh K. Gupta, Mykel J. Kochenderfer, Dorsa Sadigh, and Jeannette Bohg. Dynamic multi-robot task allocation under uncertainty and temporal constraints. In *Proceedings of the Conference on Robotics: Science and Systems (RSS)*, 2020.

[43] Shushman Choudhury, Kiril Solovey, Mykel J. Kochenderfer, and Marco Pavone. Efficient large-scale multi-drone delivery using transit networks. *Journal of Artificial Intelligence Research*, 70:757–788, 2021.

[44] David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3): 115–137, 2006.

[45] Liron Cohen. *Efficient Bounded-Suboptimal Multi-Agent Path Finding and Motion Planning via Improvements to Focal Search*. PhD thesis, University of Southern California, 2020.

[46] Liron Cohen, Tansel Uras, T. K. Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. Improved solvers for bounded-suboptimal multi-agent path finding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3067–3074, 2016.

[47] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernández, Ariel Felner, T. K. Satish Kumar, and Sven Koenig. Anytime focal search with applications. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1434–1441, 2018.

[48] Liron Cohen, Glenn Wagner, David M. Chan, Howie Choset, Nathan R. Sturtevant, Sven Koenig, and T. K. Satish Kumar. Rapid randomized restarts for multi-agent path finding solvers. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 148–152, 2018.

[49] Liron Cohen, Tansel Uras, T. K. Satish Kumar, and Sven Koenig. Optimal and bounded-suboptimal multi-agent motion planning. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 44–51, 2019.

[50] Adem Coskun and Jason M. O'Kane. Online plan repair in multi-robot coordination with disturbances. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pages 3333–3339, 2019.

[51] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 148–159, 1996.

[52] Emrah Demir, Tolga Bektas, and Gilbert Laporte. An adaptive large neighborhood search heuristic for the pollution-routing problem. *European Journal of Operational Research*, 223 (2):346–359, 2012.

[53] Carmel Domshlak, Michael Katz, and Alexander Shleyfman. Enhanced symmetry breaking in cost-optimal planning as forward search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 343–347, 2012.

[54] Carmel Domshlak, Michael Katz, and Alexander Shleyfman. Symmetry breaking: Satisficing planning and landmark heuristics. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 298–302. AAAI, 2013.

[55] Rodney G. Downey and Micheal R. Fellows. Parameterized computational feasibility. In *Feasible Mathematics II*, pages 219–244, 1995.

[56] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31:591–656, 2008.

[57] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods System Design*, 9(1/2):105–131, 1996.

[58] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 290–296, 2013.

[59] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. *Algorithmica*, 2 (1-4):477, 1987.

[60] Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[61] Ariel Felner, Roni Stern, Solomon Eyal Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan R. Sturtevant, Glenn Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 29–37, 2017.

[62] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, T. K. Satish Kumar, and Sven Koenig. Adding heuristics to conflict-based search for multi-agent pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 83–87, 2018.

[63] Cornelia Ferner, Glenn Wagner, and Howie Choset. ODrM* optimal multirobot path planning in low dimensional search spaces. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3854–3859, 2013.

[64] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 462–476, 2002.

[65] Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 956–961, 1999.

[66] Maria Fox and Derek Long. Extending the exploitation of symmetries in planning. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 83–91, 2002.

[67] Graeme Gange, Daniel Harabor, and Peter J. Stuckey. Lazy CBS: Implicit conflict-based search using lazy clause generation. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 155–162, 2019.

[68] Martin Gebser, Philipp Obermeier, Thomas Otto, Torsten Schaub, Orkunt Sabuncu, Van Nguyen, and Tran Cao Son. Experimenting with robotic intra-logistics domains. *Theory and Practice of Logic Programming*, 18(3-4):502–519, 2018.

[69] Daniel Gnad, Álvaro Torralba, Alexander Shleyfman, and Jörg Hoffmann. Symmetry breaking in star-topology decoupled search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 125–134, 2017.

[70] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

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

[72] Rodrigo N. Gómez, Carlos Hernández, and Jorge A. Baier. A compact answer set programming encoding of multi-agent pathfinding. *IEEE Access*, 9:26886–26901, 2021.

[73] Nir Greshler, Ofir Gordon, Oren Salzman, and Nahum Shimkin. Cooperative multi-agent path finding: Beyond path planning and collision avoidance. In *Proceedings of the International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 20–28, 2021.

[74] Naveed Haghani, Jiaoyang Li, Sven Koenig, Gautam Kunapuli, Claudio Contardo, Amelia Regan, and Julian Yarkony. Multi-robot routing with time windows: A column generation approach. *CoRR*, abs/2103.08835, 2021.

[75] Shuai D. Han and Jingjin Yu. DDM: Fast near-optimal multi-robot path planning using diversified-path and optimal sub-problem solution database heuristics. *IEEE Robotics and Automation Letters*, 5(2):1350–1357, 2020.

[76] Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1114–1119, 2011.

[77] Christian Henkel and Marc Toussaint. Optimized directed roadmap graph for multi-agent path finding using stochastic gradient descent. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, pages 776–783, 2020.

[78] Christian Henkel, Jannik Abbenseth, and Marc Toussaint. An optimal algorithm to solve the combined task allocation and path finding problem. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4140–4146, 2019.

[79] Florence Ho, Ana Salta, Rúben Geraldes, Artur Goncalves, Marc Cavazza, and Helmut Prendinger. Multi-agent path finding for UAV traffic management. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 131–139, 2019.

[80] Khoi D. Hoang, Ferdinando Fioretto, William Yeoh, Enrico Pontelli, and Roie Zivan. A large neighboring search schema for multi-agent optimization. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 688–706, 2018.

[81] Wolfgang Hönig, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 477–485, 2016.

[82] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. Conflict-based search with optimal task assignment. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 757–765, 2018.

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

[84] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. Persistent and robust execution of MAPF schedules in warehouses. *IEEE Robotics and Automation Letters*, 4(2):1125–1131, 2019.

[85] Shuli Hu, Daniel Harabor, Graeme Gange, Peter J. Stuckey, and Nathan R. Sturtevant. Multi-agent path finding with temporal jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 169–173, 2022.

[86] Taoan Huang, Bistra Dilkina, and Sven Koenig. Learning node-selection strategies in bounded-suboptimal conflict-based search for multi-agent path finding. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 611–619, 2021.

[87] Taoan Huang, Jiaoyang Li, Sven Koenig, and Bistra Dilkina. Anytime multi-agent path finding via machine learning-guided large neighborhood search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 9368–9376, 2022.

[88] Taoan Huang, Vikas Shivashankar, Michael Caldara, Joseph W. Durham, Jiaoyang Li, Bistra Dilkina, and Sven Koenig. Deadline-aware multi-agent tour planning. In *IJCAI Workshop on Heuristic Search in Industry (HSI)*, 2022.

[89] M. Renee Jansen and Nathan R. Sturtevant. Direction maps for cooperative pathfinding. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pages 185–190, 2008.

[90] 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 (ICAPS)*, pages 161–165, 2020.

[91] Michael Katz and Carmel Domshlak. Optimal additive composition of abstraction-based admissible heuristics. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 174–181, 2008.

[92] Justin Kottinger, Shaull Almagor, and Morteza Lahijanian. Conflict-based search for explainable multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 692–700, 2022.

[93] Justin Kottinger, Shaull Almagor, and Morteza Lahijanian. Conflict-based search for multi-robot motion planning with kinodynamic constraints. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page (in print), 2022.

[94] Ngai Meng Kou, Cheng Peng, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Idle time optimization for target assignment and path finding in sortation centers. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 9925–9932, 2020.

[95] Edward Lam and Pierre Le Bodic. New valid inequalities in branch-and-cut-and-price for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 184–192, 2020.

[96] Edward Lam, Pierre Le Bodic, Daniel Damir Harabor, and Peter J. Stuckey. Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1289–1296, 2019.

[97] Edward Lam, Pierre Le Bodic, Daniel Harabor, and Peter J. Stuckey. Branch-and-cut-and-price for multi-agent path finding. *Computers and Operations Research*, 144:105809, 2022.

[98] Florian Laurent, Manuel Schneider, Christian Scheller, Jeremy D. Watson, Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Konstantin Makhnev, Oleg Svidchenko, Vladimir Egorov, Dmitry Ivanov, Aleksei Shpilman, Evgenija Spirovska, Oliver Tanevski, Aleksandar Nikov, Ramon Grunder, David Galevski, Jakov Mitrovski, Guillaume Sartoretti, Zhiyao Luo, Mehul Damani, Nilabha Bhattacharya, Shivam Agarwal, Adrian Egli, Erik Nygren, and Sharada P. Mohanty. Flatland competition 2020: MAPF and MARL for efficient train coordination on a grid world. In *Proceedings of Machine Learning Research (PMLR)*, pages 275–301, 2020.

[99] Yat-Chiu Law and Jimmy Ho-Man Lee. Symmetry breaking constraints for value symmetries in constraint satisfaction. *Constraints*, 11(2-3):221–267, 2006.

[100] Duong Le and Erion Plaku. Cooperative, dynamics-based, and abstraction-guided multi-robot motion planning. *Journal of Artificial Intelligence Research*, 63:361–390, 2018.

[101] Christopher Leet, Jiaoyang Li, and Sven Koenig. Shard systems: Scalable, robust and persistent multi-agent path finding with performance guarantees. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 9386–9395, 2022.

[102] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 442–449, 2019.

[103] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Ariel Felner, Hang Ma, and Sven Koenig. Disjoint splitting for conflict-based search for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 279–283, 2019.

[104] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. Symmetry-breaking constraints for grid-based multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 6087–6095, 2019.

[105] Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding for large agents. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 7627–7634, 2019.

[106] Jiaoyang Li, Han Zhang, Mimi Gong, Zi Liang, Weizi Liu, Zhongyi Tong, Liangchen Yi, Robert Morris, Corina Pasareanu, and Sven Koenig. Scheduling and airport taxiway path planning under uncertainty. In *Proceedings of the AIAA Aviation Forum*, 2019.

[107] Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. New techniques for pairwise symmetry breaking in multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 193–201, 2020.

[108] Jiaoyang Li, Kexuan Sun, Hang Ma, Ariel Felner, T. K. Satish Kumar, and Sven Koenig. Moving agents in formation in congested environments. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 726–734, 2020.

[109] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. Anytime multi-agent path finding via large neighborhood search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4127–4135, 2021.

[110] Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chen, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. Scalable rail planning and replanning: Winning the 2020 flatland challenge. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 477–485, 2021.

[111] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, Graeme Gange, and Sven Koenig. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence*, 301:103574, 2021.

[112] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. EECBS: Bounded-suboptimal search for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 12353–12362, 2021.

[113] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 11272–11281, 2021.

[114] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. MAPF-LNS2: Repairing multi-agent path finding via large neighborhood search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 10256–10265, 2022.

[115] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. Task and path planning for multi-agent pickup and delivery. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1152–1160, 2019.

[116] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 294–300, 2011.

[117] Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1144–1152, 2016.

[118] Hang Ma, Craig A. Tovey, Guni Sharon, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 3166–3173, 2016.

[119] Hang Ma, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding with delay probabilities. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 3605–3612, 2017.

[120] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 837–845, 2017.

[121] Hang Ma, Jingxing Yang, Liron Cohen, T. K. Kumar, and Sven Koenig. Feasibility study: Moving non-homogeneous teams in congested video game environments. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AI-IDE)*, pages 270–272, 2017.

[122] Hang Ma, Glenn Wagner, Ariel Felner, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding with deadlines. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 417–423, 2018.

[123] Hang Ma, Daniel Harabor, Peter J. Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 7643–7650, 2019.

[124] Ellips Masehian and Azadeh Hassan Nejad. Solvability of multi robot motion planning problems on trees. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5936–5941, 2009.

[125] Christopher Mears, Maria J. García de la Banda, Mark Wallace, and Bart Demoen. A novel approach for detecting symmetries in CSP models. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, pages 158–172, 2008.

[126] Jayanth Krishna Mogali, Willem-Jan van Hoeve, and Stephen F. Smith. Template matching and decision diagrams for multi-agent path finding. In *Proceedings of the International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pages 347–363, 2020.

[127] James Motes, Read Sandström, Hannah Lee, Shawna L. Thomas, and Nancy M. Amato. Multi-robot task and motion planning with subtask dependencies. *IEEE Robotics and Automation Letters*, 5(2):3338–3345, 2020.

[128] Aniello Murano, Giuseppe Perelli, and Sasha Rubin. Multi-agent path planning in known dynamic environments. In *Proceedings of the International Conference on Principles and Practice of Multi-Agent Systems (PRIMA)*, pages 218–231, 2015.

[129] Bernhard Nebel. On the computational complexity of multi-agent pathfinding on directed graphs. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 212–216, 2020.

[130] Van Nguyen, Philipp Obermeier, Tran Cao Son, Torsten Schaub, and William Yeoh. Generalized target assignment and path finding using answer set programming. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1216–1223, 2017.

[131] Ayano Okoso, Keisuke Otaki, and Tomoki Nishi. Multi-agent path finding with priority for cooperative automated valet parking. In *Proceedings of the IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 2135–2140, 2019.

[132] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. Priority inheritance with backtracking for iterative multi-agent path finding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 535–542, 2019.

[133] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. winPIBT: Expanded prioritized algorithm for iterative multi-agent path finding. In *AAAI Workshop on Multi-Agent Path Finding (WoMAPF)*, 2020.

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

[135] Oriana Peltzer, Kyle Brown, Mac Schwager, Mykel J. Kochenderfer, and Martin A. Sehr. STT-CBS: A conflict-based search algorithm for multi-agent path finding with stochastic travel times. *CoRR*, abs/2004.08025, 2020.

[136] Mike Phillips and Maxim Likhachev. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 5628–5635, 2011.

[137] Nir Pochter, Aviv Zohar, and Jeffrey S. Rosenschein. Exploiting problem symmetries in state-based planners. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1004–1009, 2011.

[138] Florian Pommerening, Gabriele Röger, and Malte Helmert. Getting the most out of pattern databases for classical planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2357–2364, 2013.

[139] Florian Pommerening, Gabriele Röger, Malte Helmert, and Blai Bonet. Heuristics for cost-optimal classical planning based on linear programming. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4303–4309, 2015.

[140] Jean-Francois Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of International Symposium on the Methodologies for Intelligent Systems (ISMIS)*, pages 350–361, 1993.

[141] Jean-Francois Puget. Symmetry breaking using stabilizers. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 585–599, 2003.

[142] Jean-Francois Puget. Automatic detection of variable and value symmetries. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 475–489, 2005.

[143] Arthur Queffelec, Ocan Sankur, and François Schwarzentruber. Conflict-based search for connected multi-agent path finding. *CoRR*, abs/2006.03280, 2020.

[144] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. Multi-objective conflict-based search for multi-agent path finding. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 8786–8791, 2021.

[145] Gabriele Röger, Silvan Sievers, and Michael Katz. Symmetry-based task reduction for relaxed reachability analysis. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 208–217, 2018.

[146] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[147] Malcolm R. K. Ryan. Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research*, 31:497–542, 2008.

[148] Malcolm R. K. Ryan. Constraint-based multi-robot path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 922–928, 2010.

[149] Qandeel Sajid, Ryan Luna, and Kostas E. Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 2012.

[150] Meinolf Sellmann and Pascal Van Hentenryck. Structural symmetry breaking. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 298–303, 2005.

[151] Tomer Shahar, Shashank Shekhar, Dor Atzmon, Abdallah Saffidine, Brendan Juba, and Roni Stern. Safe multi-agent pathfinding with time uncertainty. *Journal of Artificial Intelligence Research*, 70:923–954, 2021.

[152] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013.

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

[154] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1520, pages 417–431, 1998.

[155] Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. Heuristics and symmetries in classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 3371–3377, 2015.

[156] Devon Sigurdson, Vadim Bulitko, William Yeoh, Carlos Hernández, and Sven Koenig. Multi-agent pathfinding with real-time heuristic search. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[157] David Silver. Cooperative Pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pages 117–122, 2005.

[158] Juan Irving Solis Vidana, James Motes, Read Sandstrom, and Nancy Amato. Representation-optimal multi-robot motion planning using conflict-based search. *IEEE Robotics and Automation Letters*, 54(7):111–118, 2021.

[159] Jialin Song, ravi lanka, Yisong Yue, and Bistra Dilkina. A general large neighborhood search framework for solving integer programs. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 20012–20023, 2020.

[160] Markus Spies, Marco Todescato, Hannes Becker, Patrick Kesper, Nicolai Waniek, and Meng Guo. Bounded suboptimal search with learned heuristics for multi-agent systems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 2387–2394, 2019.

[161] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 173–178, 2010.

[162] Trevor Scott Standley and Richard E. Korf. Complete algorithms for cooperative pathfinding problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 668–673, 2011.

[163] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 151–159, 2019.

[164] Charlie Street, Bruno Lacerda, Manuel Mühlig, and Nick Hawes. Multi-robot planning under uncertainty with congestion-aware models. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1314–1322, 2020.

[165] Nathan R. Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

[166] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3613–3619, 2009.

[167] Pavel Surynek. Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1177–1183, 2019.

[168] Pavel Surynek. Lazy compilation of variants of multi-robot path planning with satisfiability modulo theory (SMT) approach. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3282–3287, 2019.

[169] Pavel Surynek. Bounded sub-optimal multi-robot path planning using satisfiability modulo theory (SMT) approach. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11631–11637, 2020.

[170] Pavel Surynek. Multi-goal multi-agent path finding via decoupled and integrated goal vertex ordering. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 12409–12417, 2021.

[171] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 810–818, 2016.

[172] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Sub-optimal sat-based approach to multi-agent path-finding problem. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 90–105, 2018.

[173] Pavel Surynek, Jiaoyang Li, Han Zhang, T. K. Satish Kumar, and Sven Koenig. Mutex propagation for SAT-based multi-agent path finding. In *Proceedings of the International Conference on Principles and Practice of Multi-Agent Systems (PRIMA)*, 2020.

[174] Jordan T. Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 674–679, 2011.

[175] Jordan T. Thayer, Wheeler Ruml, and Jeff Kreis. Using distance estimates in heuristic search: A re-evaluation. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 2009.

[176] Jordan T. Thayer, Austin J. Dionne, and Wheeler Ruml. Learning inadmissible heuristics during search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 250–257, 2011.

[177] Shyni Thomas, Dipti Deodhare, and M. Narasimha Murty. Extended conflict-based search for the convoy movement problem. *IEEE Intelligent Systems*, 30(6):60–70, 2015.

[178] David Tolpin, Tal Beja, Solomon Eyal Shimony, Ariel Felner, and Erez Karpas. Toward rational deployment of multiple heuristics in A*. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 674–680, 2013.

[179] Jur van den Berg and Mark H. Overmars. Prioritized motion planning for multiple robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 430–435, 2005.

[180] Jur van den Berg, Jack Snoeyink, Ming C. Lin, and Dinesh Manocha. Centralized path planning for multiple robots: Optimal decoupling into sequential plans. In *Proceedings of the Conference on Robotics: Science and Systems (RSS)*, 2009.

[181] Sumanth Varambally, Jiaoyang Li, and Sven Koenig. Which MAPF model works best for automated warehousing? In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 190–198, 2022.

[182] Kyle Vedder and Joydeep Biswas. X*: Anytime multi-agent path finding for sparse domains using window-based iterative repairs. *Artificial Intelligence*, 291:103417, 2021.

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

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

[185] Glenn Wagner and Howie Choset. Path planning for multiple agents under uncertainty. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 577–585, 2017.

[186] Thayne T. Walker, Nathan R. Sturtevant, and Ariel Felner. Generalized and sub-optimal bipartite constraints for conflict-based search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 7277–7284, 2020.

[187] Thayne T. Walker, Nathan R. Sturtevant, Han Zhang, Jiaoyang Li, Ariel Felner, and T. K. Satish Kumar. Conflict-based increasing cost search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 385–395, 2021.

[188] Toby Walsh. General symmetry breaking constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 650–664, 2006.

[189] Qian Wan, Chonglin Gu, Sankui Sun, Mengxia Chen, Hejiao Huang, and Xiaohua Jia. Lifelong multi-agent path finding in A dynamic environment. In *Proceedings of the International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 875–882, 2018.

[190] Hanlin Wang and Michael Rubenstein. Walk, stop, count, and swap: Decentralized multiagent path finding with theoretical guarantees. *IEEE Robotics and Automation Letters*, 5 (2):1119–1126, 2020.

[191] Jiangxing Wang, Jiaoyang Li, Hang Ma, Sven Koenig, and T. K. Satish Kumar. A new constraint satisfaction perspective on multi-agent path finding. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 2253–2255, 2019.

[192] Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 380–387, 2008.

[193] Ko-Hsin Cindy Wang and Adi Botea. MAPP: A scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research*, 42:55–90, 2011.

[194] Wenjie Wang and Wooi Boon Goh. An iterative approach for makespan-minimized multi-agent path planning in discrete space. *Autonomous Agents and Multi-Agent Systems*, 29(3): 335–363, 2015.

[195] Charles W. Warren. Multiple robot path coordination using artificial potential fields. In *Proceedings of the IEEE International Conference on Robotics and Automation (IRCA)*, pages 500–505, 1990.

[196] Martin Wehrle, Malte Helmert, Alexander Shleyfman, and Michael Katz. Integrating partial order reduction and symmetry elimination for cost-optimal classical planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1712–1718, 2015.

[197] Licheng Wen, Yong Liu, and Hongliang Li. CL-MAPF: Multi-agent path finding for car-like robots with kinematic and spatio-temporal constraints. *Robotics and Autonomous Systems*, 150:103997, 2022.

[198] Wenying Wu, Subhrajit Bhattacharya, and Amanda Prorok. Multi-robot path deconfliction through prioritization by path prospects. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 9809–9815, 2020.

[199] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of co-operative, autonomous vehicles in warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1752–1760, 2007.

[200] Hong Xu, Kexuan Sun, Sven Koenig, and S. T. Satish Kumar. A warning propagation-based linear-time-and-space algorithm for the minimum vertex cover problem on giant graphs. In *Proceedings of the International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pages 567–584, 2018.

[201] Fan Yang, Joseph C. Culberson, Robert Holte, Uzi Zahavi, and Ariel Felner. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research*, 32: 631–662, 2008.

[202] Sung Wook Yoon, Alan Fern, and Robert Givan. Learning heuristic functions from relaxed plans. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 162–171, 2006.

[203] Jingjin Yu. Intractability of optimal multirobot path planning on planar graphs. *IEEE Robotics and Automation Letters*, 1(1):33–40, 2016.

[204] Jingjin Yu. Constant factor time optimal multi-robot routing on high-dimensional grids. In *Proceedings of the Conference on Robotics: Science and Systems (RSS)*, 2018.

[205] Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1444–1449, 2013.

[206] Jingjin Yu and Steven M. LaValle. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, 32(5):1163–1177, 2016.

[207] Jingjin Yu and Daniela Rus. Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms. In *Proceedings of the International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, pages 729–746, 2014.

[208] Han Zhang, Jiaoyang Li, Pavel Surynek, Sven Koenig, and T. K. Satish Kumar. Multi-agent path finding with mutex propagation. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 323–332, 2020.

[209] Han Zhang, Mingze Yao, Ziang Liu, Jiaoyang Li, Lucas Terr, Shao-Hung Chan, T. K. Satish Kumar, and Sven Koenig. A hierarchical approach to multi-agent path finding. In *ICAPS Workshop on Hierarchical Planning (HPLAN)*, 2021.

[210] Han Zhang, Jingkai Chen, Jiaoyang Li, Brian C. Williams, and Sven Koenig. Multi-agent path finding for precedence-constrained goal sequences. In *Proceedings of the International Joint Conference on Autonomous Agents and Mult-Agent Systems (AAMAS)*, pages 1464–1472, 2022.

[211] Han Zhang, Yutong Li, Jiaoyang Li, S. K. Satish Kumar, and Sven Koenig. Mutex propagation in multi-agent path finding for large agents. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 249–253, 2022.

[212] Han Zhang, Pavel Surynek, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding with mutex propagation. *Artificial Intelligence*, page 103766, 2022.

[213] Shuyang Zhang, Jiaoyang Li, Taoan Huang, Sven Koenig, and Bistra Dilkina. Learning a priority ordering for prioritized planning in multi-agent path finding. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 208–216, 2022.

[214] Yi Zheng, Srivatsan Ravi, Sven Koenig Erik Kline, and T. K. Satish Kumar. Conflict-based search for the virtual network embedding problem. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 423–433, 2022.

[215] Xinyi Zhong, Jiaoyang Li, Sven Koenig, and Hang Ma. Optimal and bounded-suboptimal multi-goal task assignment and path finding. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 10731–10737, 2022.

Appendices

A Proof for the CG Heuristic

In order to help explain the proof of Theorem 3.1, we define another type of MDDs, called *extended MDDs*, that ignores constraints, i.e., its MDDs include MDD nodes prohibited by constraints. All MDDs that we discuss in this section are extended MDDs.

Definition A.1 (Extended MDDs)**.** *An extended MDD MDD_i^{μ} for agent a_i is a directed acyclic graph that consists of* all *paths of agent a_i from its start vertex s_i to it target vertex g_i that are at most μ timesteps long. In particular, MDD_i^{μ} is empty iff $\mu < dist(s_i, g_i)$. We say that a MDD node $(x,t) \in MDD_i^{\mu}$ iff MDD_i^{μ} has vertex x at level t, i.e., there is a path from vertex s_i at timestep 0 to vertex x at timestep t and then to vertex g_i by timestep μ.*

Extended MDDs have the following two properties.

Property A.1. $(x,t) \in MDD_i^{\mu} \iff dist(s_i,x) \leq t \wedge dist(x,g_i) \leq \mu - t.$

Proof. The property follows from Definition A.1. □

Property A.2. *For any path p of agent a_i, if $(p[t],t) \notin MDD_i^{\mu}$, then $(p[t'],t') \notin MDD_i^{\mu}$ for all $t' \geq t$.*

Proof. This property holds because, by contradiction, if $(p[t'],t') \in MDD_i^{\mu}$ for some timestep $t' \geq t$, then there is a sub-path p' in MDD_i^{μ} from vertex $p[t']$ at timestep t' to vertex g_i at timestep μ. The path that follows a prefix of path p from vertex s_i at timestep 0 to vertex $p[t']$ at timestep t' and then follows sub-path p' to vertex g_i at timestep μ visits vertex $p[t]$ at timestep t, i.e., $(p[t],t) \in MDD_i^{\mu}$. □

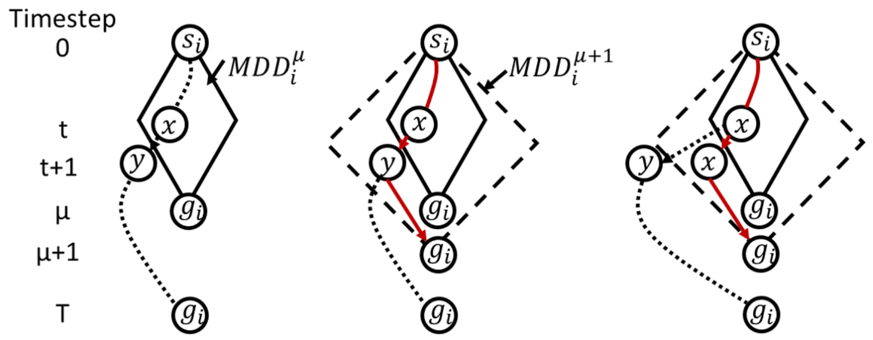


Figure 7.2: Illustration of constructing a path of length $\mu+1$ for agent a_i. The solid and dashed diamonds include all MDD nodes in MDD_i^μ and $MDD_i^{\mu+1}$, respectively.

Property A.2 implies that, once a path *leaves* an extended MDD, it will never revisit this extended MDD.

We say that MDD_i^μ includes all MDD nodes prohibited by a set of constraints C iff MDD_i^μ has vertex v at level t for any vertex constraint $\langle a_i, v, t\rangle \in C$ and edge (u, v) from level $t-1$ to level t for any edge constraint $\langle a_i, u, v, t\rangle \in C$.

Lemma A.1. *Let C be a set of constraints and μ be a sufficiently large integer such that MDD_i^μ is not empty and includes all nodes prohibited by C. If agent a_i has a path that satisfies the constraints in C, then agent a_i has a path no longer than $\mu+1$ that satisfies the constraints in C.*

Proof. By assumption, agent a_i has a path p that satisfies the constraints in C. If $length(p) \leq \mu+1$, then we are done. Otherwise, as shown in Figure 7.2, we will construct a path of length $\mu+1$ by letting agent a_i (1) first follow the prefix of p to the border of MDD_i^μ, (2) then cross the border to a MDD node in $MDD_i^{\mu+1}$, and (3) finally follow a path in $MDD_i^{\mu+1}$ to vertex g_i at timestep $\mu+1$. This new path satisfies C because p satisfies C and any MDD nodes or edges outside of MDD_i^μ also satisfy C. The existence of such a path is proved as follows.

The first MDD node $(s_i, 0)$ visited by path p is in MDD_i^μ, and the last MDD node $(g_i, length(p))$ visited by path p is not in MDD_i^μ (because $length(p) > \mu$). So, there exists a timestep $t < length(p)$ such that $(p[t], t) \in MDD_i^\mu$ and $(p[t+1], t+1) \notin MDD_i^\mu$. Let x and y represent $p[t]$ and $p[t+1]$, respectively (Figure 7.2(left)). We distinguish two cases and show how to construct a path of length $\mu+1$ that satisfies the constraints in C for each case.

- Case 1: $(y, t+1) \in MDD_i^{\mu+1}$ (Figure 7.2(middle)). There exists a sub-path p' in $MDD_i^{\mu+1}$ from vertex y at timestep $t+1$ to vertex g_i at timestep $\mu+1$. By applying $(y, t+1) \notin MDD_i^{\mu}$ to Property A.2, we know that sub-path p' does not visit any MDD node in MDD_i^{μ} and thus does not violate any constraints. Therefore, a path that follows a prefix of path p from vertex s_i at timestep 0 to vertex y at timestep $t+1$ and then follows sub-path p' to vertex g_i at timestep $\mu+1$ is a path of length $\mu+1$ that satisfies the constraints in C.

- Case 2: $(y, t+1) \notin MDD_i^{\mu+1}$ (Figure 7.2(right)). Since MDD node $(y, t+1)$ is visited by path p, we know that $dist(s_i, y) \leq t+1$. Then, from Property A.1, it holds that $dist(y, g_i) > (\mu+1) - (t+1)$, which leads to $dist(y, g_i) \geq \mu - t + 1$. Rearranging the terms yields

$$\begin{aligned} \mu &\leq t + dist(y, g_i) - 1 && \text{(A.1)} \\ &\leq t + dist(y, x) + dist(x, g_i) - 1 && \text{(A.2)} \\ &\leq t + dist(x, g_i) && \text{(A.3)} \\ &\leq \mu. && \text{(A.4)} \end{aligned}$$

Inequality (A.2) is based on the the triangle inequality; Inequality (A.3) follows from $dist(y, x) \leq 1$; and Inequality (A.4) is obtained by applying $(x, t) \in MDD_i^{\mu}$ to Property A.1. Comparing the first and last lines of these inequalities shows that all lines are actually equal. Specifically, we know from the last line that

$$t + dist(x, g_i) = \mu. \tag{A.5}$$

Since MDD node (x, t) is visited by path p, we know that $dist(s_i, x) \leq t$. From Equation (A.5) and Property A.1, we know that $(x, t+1) \notin MDD_i^{\mu}$ and $(x, t+1) \in MDD_i^{\mu+1}$. Hence, there is a sub-path p'' in $MDD_i^{\mu+1}$ from vertex x at timestep $t+1$ to vertex g_i at timestep $\mu+1$. From Property A.2, we know that sub-path p'' does not visit any MDD node in MDD_i^{μ} and thus does not violate any constraints in C. Therefore, a path that follows a prefix of path

p from vertex s_i at timestep 0 to vertex x at timestep t, waits for one timestep, and then follows sub-path p'' to vertex g_i at timestep $\mu+1$ is a path of length $\mu+1$ that satisfies the constraints in C.

Therefore, the lemma holds. □

Finally, we prove Theorem 3.1 based on Lemma A.1.

Theorem 3.1. *Suppose that CBS chooses to resolve a conflict between agents a_i and a_j at timestep t at a CT node N and successfully generates two child CT nodes N_1 (with an additional constraint imposed on a_i) and N_2 (with an additional constraint imposed on a_j). If the conflict occurs after one of the agents, say a_i, completes its path, i.e., $t \geq length(N.plan[a_i])$, then $cost(N_1) = cost(N) + t + 1 - length(N.plan[a_i])$ and $cost(N_2) \in \{cost(N), cost(N)+1\}$. Otherwise (i.e., the conflict occurs before both agents have completed their paths), $cost(N_1), cost(N_2) \in \{cost(N), cost(N)+1\}$.*

Proof. Without loss of generality, we focus only on child CT node N_1. Every constraint in $N.constraints$ imposed on agent a_i was generated due to a vertex or edge conflict that occurred on one of the old paths of agent a_i, i.e., paths of agent a_i in the plans of any ancestor CT nodes of CT node N. The length of any old path of agent a_i is no longer than $length(N.plan[a_i])$. Therefore, $MDD_i^{length(N.plan[a_i])}$ includes all MDD nodes prohibited by $N.constraints$.

If the chosen conflict occurs before agent a_i completes its path (i.e., $t < length(N.plan[a_i])$), then $MDD_i^{length(N.plan[a_i])}$ includes all MDD nodes prohibited by $N_1.constraints$. From Lemma A.1, we know that agent a_i has a path no longer than $length(N.plan[a_i]) + 1$ that satisfies the constraints in $N_1.constraints$. So, the length of the shortest path of agent a_i in CT node N_1 is at most $length(N.plan[a_i]) + 1$, and thus $cost(N_1) \in \{cost(N), cost(N)+1\}$.

If the chosen conflict occurs after agent a_i completes its path (i.e., $t \geq length(N.plan[a_i])$), then the chosen conflict is $\langle a_i, a_j, g_i, t\rangle$, and the additional constraint added to CT node N_1 is $\langle a_i, g_i, t\rangle$. MDD_i^t includes all nodes prohibited by $N_1.constraints$, and thus, by Lemma A.1, agent a_i has a path no longer than $t+1$ that satisfies $N_1.constraints$. On the other hand, since agent a_i is prohibited from being at vertex g_i at timestep t, its shortest path is of length at least $t+1$. Therefore, the length of the shortest path of agent a_i in CT node N_1 is $t+1$. So, $cost(N_1) = cost(N) + t + 1 - length(N.plan[a_i])$. □

B Proof for the WDG Heuristic

Property 3.4. *The WDG heuristic strictly dominates the greedy WDG heuristic.*

Proof. By definition, the WDG heuristic h_{WDG} is equal to the solution value of the following integer linear program:

$$\begin{aligned} \min & \sum_{v_i \in V_D} x_i & & \\ \text{s.t.} \quad & x_i + x_j \geq \Delta_{ij} & & \forall (v_i, v_j) \in E_D \\ & x_i \in \mathbb{N} & & \forall v_i \in V_D. \end{aligned} \tag{B.1}$$

The solution value h_1 of the linear programming relaxation of Problem B.1 (i.e., replacing $x_i \in \mathbb{N}$ with $x_i \geq 0$ in Problem B.1) satisfies $h_1 \leq h_{WDG}$. The dual problem of this linear programming relaxation is

$$\begin{aligned} \max & \sum_{(v_i,v_j) \in E_D} \Delta_{ij} y_{ij} & & \\ \text{s.t.} \quad & \sum_{(v_i,v_j) \in E_D} y_{ij} \leq 1 & & \forall v_i \in V_D \\ & 0 \leq y_{ij} \leq 1 & & \forall (v_i, v_j) \in E_D \end{aligned} \tag{B.2}$$

By the weak duality theorem, we know that the solution value h_2 of Problem B.2 satisfies $h_2 \leq h_1$. if we turn Problem B.2 into an integer linear program by replacing $0 \leq y_{ij} \leq 1$ with $y_{ij} \in \{0, 1\}$, then the resulting solution value h_3 satisfies $h_3 \leq h_2$. This integer version of Problem B.2 is identical to the weighted maximum problem, a problem that finds the matching that maximizes the sum of the weights of the selected edges (i.e., $y_{ij} = 1$ indicates a selected edge, and $y_{ij} = 0$ indicates an unselected edge). Since the greedy WDG heuristic h'_{WDG} is the value of a maximal matching, it satisfies $h'_{WDG} \leq h_3$. Therefore, $h'_{WDG} \leq h_3 \leq h_2 \leq h_1 \leq h_{WDG}$. $\square$

C Supplement for the Rectangle Reasoning Techniques

We first present the equations for calculating the four corners R_s, R_g, R_1, and R_2 from the start and target nodes S_1, S_2, G_1, and G_2 and then provide proofs of the properties and theorems for the rectangle reasoning techniques.

C.1 Calculating Corner Nodes

Recall the definition of the four corners in Definition 4.4. We analyze all combinations of the relative locations of start and target nodes and come up with the following way of calculating the locations of R_s and R_g:

$$R_s.x = \begin{cases} S_1.x, & S_1.x = G_1.x \\ \max\{S_1.x, S_2.x\}, & S_1.x < G_1.x \\ \min\{S_1.x, S_2.x\}, & S_1.x > G_1.x \end{cases} \tag{C.1}$$

$$R_g.x = \begin{cases} G_1.x, & S_1.x = G_1.x \\ \min\{G_1.x, G_2.x\}, & S_1.x < G_1.x \\ \max\{G_1.x, G_2.x\}, & S_1.x > G_1.x. \end{cases} \tag{C.2}$$

We calculate $R_s.y$ and $R_g.y$ by replacing all "x" with "y" in Equations (C.1) and (C.2). We calculate the locations of R_1 and R_2 as follows:

- If $(S_1.x - S_2.x)(S_2.x - R_g.x) \geq 0$, then

$$R_1.x = R_g.x \tag{C.3}$$

$$R_1.y = S_1.y \tag{C.4}$$

$$R_2.x = S_2.x \tag{C.5}$$

$$R_2.y = R_g.y. \tag{C.6}$$

- Otherwise,

$$R_1.x = S_1.x \tag{C.7}$$

$$R_1.y = R_g.y \tag{C.8}$$

$$R_2.x = R_g.x \tag{C.9}$$

$$R_2.y = S_2.y. \tag{C.10}$$

Finally, we calculate the timesteps of all corner nodes R_i for $i = 1, 2, s, g$ as follows:

$$R_i.t = S_1.t + |S_1.x - R_i.x| + |S_1.y - R_i.y|. \tag{C.11}$$

When we use Rectangle Reasoning Technique II, the aforementioned method can miscalculate R_1 and R_2 when $S_1.x = S_2.x$, such as for the rectangle conflict in Figure 4.3b. Instead, we calculate R_1 and R_2 with the following method when $S_1.x = S_2.x$:

- If $(S_1.y - S_2.y)(S_2.y - R_g) \leq 0$, then

$$R_1.x = R_g.x \tag{C.12}$$

$$R_1.y = S_1.y \tag{C.13}$$

$$R_2.x = S_2.x \tag{C.14}$$

$$R_2.y = R_g.y. \tag{C.15}$$

- Otherwise,

$$R_1.x = S_1.x \tag{C.16}$$

$$R_1.y = R_g.y \tag{C.17}$$

$$R_2.x = R_g.x \tag{C.18}$$

$$R_2.y = S_2.y. \tag{C.19}$$

C.2 Proof for the Rectangle Reasoning Techniques

Property 4.2. *If Rectangle Reasoning Technique I identifies a rectangle conflict between agents a_1 and a_2, then any path of agent a_1 that visits a node on its exit border R_1R_g also visits a node on its entry border R_sR_2, and any path of agent a_2 that visits a node on its exit border R_2R_g also visits a node on its entry border R_sR_1.*

Proof. We assume that the vertex conflict between agents a_1 and a_2 is at node C. We also assume that $S_1.x \leq G_1.x$ and $S_1.y \leq G_2.y$ without loss of generality (because MAPF is invariant under rotations of axes). From Equations (4.3) and (4.4), we know that $S_2.x \leq G_2.x$ and $S_2.y \leq G_2.y$.[6] Since Equations (4.1) and (4.2) ensure that the path of each agent a_i for $i = 1,2$ from its start node

[6]Note that, when $S_1.x = G_1.x$, it is possible that $S_2.x > G_2.x$ according to Equation (4.3). In this case, we flip the x-axis so that $S_1.x \leq G_1.x$ and $S_2.x \leq G_2.x$ both hold. We proceed the y-axis similarly.

to its target node is within its S_i-G_i rectangle, we know that node C is within the intersection of the S_1-G_1 and S_2-G_2 rectangles, i.e., the cell of node C is within the conflicting area:

$$\max\{S_1.x, S_2.x\} \leq C.x \leq \min\{G_1.x, G_2.x\} \tag{C.20}$$

$$\max\{S_1.y, S_2.y\} \leq C.y \leq \min\{G_1.y, G_2.y\}. \tag{C.21}$$

Then, by Property 4.1, we know

$$(C.x - S_1.x) + (C.y - S_1.y) = (C.x - S_2.x) + (C.y - S_2.y), \tag{C.22}$$

which can be simplified to

$$S_1.x + S_1.y = S_2.x + S_2.y. \tag{C.23}$$

We assume that $S_1.x \geq S_2.x$ without loss of generality (because MAPF is invariant under swaps of the indexes of agents), which implies that $S_1.y \leq S_2.y$. According to the definition of the four corners of the rectangle in Definition 4.4 (or the equations in Section C.1), we have

$$R_s.x = S_1.x \tag{C.24}$$

$$R_s.y = S_2.y \tag{C.25}$$

$$R_g.x = \min\{G_1.x, G_2.x\} \geq S_1.x \tag{C.26}$$

$$R_g.y = \min\{G_1.y, G_2.y\} \geq S_2.y \tag{C.27}$$

$$R_1.x = S_1.x \tag{C.28}$$

$$R_1.y = R_g.y \tag{C.29}$$

$$R_2.x = R_g.x \tag{C.30}$$

$$R_2.y = S_2.y. \tag{C.31}$$

Thus,

$$S_1.x \leq S_2.x = R_s.x = R_2.x \leq R_g.x = R_1.x \tag{C.32}$$

$$S_2.y \leq S_1.y = R_s.y = R_1.y \leq R_g.y = R_2.y. \tag{C.33}$$

Consequently, the relative locations of the start, target, and rectangle corner nodes are exactly the same as the ones given in Figure 4.2. Since the S_1-R_g and R_s-R_g rectangles are of the same width (i.e., $|S_1.x - R_g.x| = |R_s.x - R_g.x| = |R_1.x - R_g.x|$) and any sub-path p_1 from node S_1 to a node on border R_1R_g is Manhattan-optimal, sub-path p_1 visits a node on border R_sR_2. Similarly, since the S_2-R_g and R_s-R_g rectangles are of the same length (i.e., $|S_2.y - R_g.y| = |R_s.y - R_g.y| = |R_2.y - R_g.y|$) and any sub-path from node S_2 to a node on border R_2R_g is Manhattan-optimal, any sub-path p_2 visits a node on border R_sR_1. Therefore, the property holds. $\square$

Property 4.6. *If Rectangle Reasoning Technique II identifies a rectangle conflict between agents a_1 and a_2, then any path of agent a_1 that visits a node constrained by $B(a_1, R_1, R_g)$ also visits a node on its entry border R_sR_2, and any path for agent a_2 that also visits a node constrained by $B(a_2, R_2, R_g)$ visits a node on its entry border R_sR_1.*

Proof. According to Property 4.5, we need to prove that any path of agent a_1 from its start node S_1 to one of the nodes constrained by $B(a_1, R_1, R_g)$ visits a node on the entry border R_sR_2 and any path of agent a_2 from its start node S_2 to one of the nodes constrained by $B(a_2, R_2, R_g)$ visits a node on the entry border R_sR_1, which can be done by the proof of Property 4.2 after replacing Equation (C.23) by Equation (4.7). $\square$

D Proof for the Generalized Rectangle Reasoning Technique

For a given generalized rectangle $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, we use $\mathscr{V}' = \{u | (u,t) \in \mathscr{V}\}$ to denote the vertices in the conflicting area.

Lemma D.2. *Any path of agent a_i for $i = 1,2$ that visits a node in $\mathscr{V}$ also traverses an entry edge in E_i.*

Proof. Consider an arbitrary path p of agent a_i that visits a node in $\mathscr{V}$. Since path p starts from the node $(s_i, 0)$ that is not in $\mathscr{V}$, there exists an edge $e = ((p[t],t),(p[t+1],t+1))$ such that $(p[t],t) \notin \mathscr{V}$ and $(p[t+1],t+1) \in \mathscr{V}$. Since $(p[t+1],t+1) \in \mathscr{V}$, node $(p[t+1],t+1)$ is in MDD_i. By Property 4.4, node $(p[t],t)$ is also in MDD_i. By the definition of the entry edges in Definition 4.6, it holds that $e \in E_i$. Therefore, the lemma holds. □

Lemma D.3. *Any path of agent a_i for $i = 1,2$ that visits a node in $\mathscr{V}$ also traverses an entry edge in E_i^b.*

Proof. According to Lemma D.2 and the fact that $E_i = E_i^b \cup E_i^h$, we only need to prove that any path of agent a_i that visits an entry edge in E_i^h also traverses an entry edge in E_i^b. Consider an arbitrary path p of agent a_i that traverses an entry edge $e = ((p[t],t),(p[t+1],t+1)) \in E_i^h$. Since vertex s_i is outside of the conflicting area while vertex u is in a hole, geometry requires that path p visits at least one vertex in $\mathscr{V}'$. We use τ to denote the earliest timestep when path p visits a vertex in $\mathscr{V}'$ (indicating that $\tau < t$, $p[\tau-1] \notin \mathscr{V}'$, and $p[\tau] \in \mathscr{V}'$) and $(p[\tau],\tau')$ to denote the corresponding node in $\mathscr{V}$. By Definition 4.5, node $(p[\tau],\tau')$ is the only MDD node in MDD_i at vertex $p[\tau]$. By Property 4.4 and $(p[t+1],t+1) \in MDD_i$ (because $e \in E_i^h$), all nodes before timestep $t+1$ visited by path p, including node $(p[\tau],\tau)$, are in MDD_i. So, $\tau = \tau'$. Thus, $(p[\tau-1],\tau-1) \notin \mathscr{V}$, $(p[\tau],\tau) \in \mathscr{V}$, and both of them are in MDD_i. Therefore, edge $e' = ((p[\tau-1],\tau-1),(p[\tau],\tau))$ is an entry edge in E_i^b, and the lemma holds. □

Property 4.8. *For all combinations of paths of agents a_1 and a_2 with a generalized rectangle conflict, if one path violates $B(a_1,R_1,R_g)$ and the other path violates $B(a_2,R_2,R_g)$, then the two paths have one or more vertex conflicts within the generalized rectangle.*

Proof. Since all nodes prohibited by $B(a_i,R_i,R_g)$ for $i = 1,2$ are in $\mathscr{V}$, from Lemma D.3, any path of agent a_i for $i = 1,2$ that visits a node prohibited by $B(a_i,R_i,R_g)$ traverses an entry edge in E_i^b. The four nodes R_s, R_g, R_1, and R_2 partition the border of the generalized rectangle $\mathscr{G}$ into four segments R_sR_2, R_2R_g, R_gR_1, and R_1R_s, denoted Seg_1, Seg_2, Seg_3, and Seg_4, respectively. The endpoint nodes of all entry edges in E_1^b are on Seg_1, and the endpoint nodes of all entry edges in E_2^b are on segment Seg_4. The nodes prohibited by $B(a_1, R_1, R_g)$ are on segment Seg_3, and the nodes prohibited by $B(a_2, R_2, R_g)$ are on segment Seg_2. Therefore, we only need to prove that any path p_1 of agent a_1 that visits a node on Seg_1 and a node on segment Seg_3 conflicts with any path p_2 of agent a_2 that visits a node on Seg_4 and a node on segment Seg_2. According to geometry, paths p_1 and p_2 cross each other, i.e., visit at least one common vertex u. According to Section 4.3.2.3, vertex u is not in one of the holes, i.e., $u \in \mathscr{V}'$. Let node (u, t_u) be the corresponding node in $\mathscr{V}$. Then, both paths p_1 and p_2 visit node (u, t_u), i.e., they conflict at vertex u at timestep t_u. Therefore, the property holds. □

E Proof for the Corridor Reasoning Technique

Property 4.9. *For all combinations of paths of agents a_1 and a_2 with a corridor conflict, if one path violates $\langle a_1, e_1, [0, \min(t_1'(e_1) - 1, t_2(e_2) + k)] \rangle$ and the other path violates $\langle a_2, e_2, [0, \min(t_2'(e_2) - 1, t_1(e_1) + k)] \rangle$, then the two paths have one or more vertex or edge conflicts inside the corridor.*

Proof. Let path p_1 be an arbitrary path of agent a_1 that visits vertex e_1 at timestep $\tau_1 \in [0, \min(t_1'(e_1) - 1, t_2(e_2) + k)]$ and path p_2 be an arbitrary path of agent a_2 that visits vertex e_2 at timestep $\tau_2 \in [0, \min(t_2'(e_2) - 1, t_1(e_1) + k)]$. We need to prove that paths p_1 and p_2 have one or more vertex or edge conflicts inside the corridor.

Since $\tau_1 \leq \min(t_1'(e_1) - 1, t_2(e_2) + k) \leq t_1'(e_1) - 1 < t_1'(e_1)$ (recall that $t_1'(e_1)$ is the earliest timestep when agent a_1 can reach vertex e_1 without using corridor C), path p_1 traverses the corridor. Similarly, path p_2 traverses the corridor as well.

Since $\tau_1 \leq \min(t_1'(e_1) - 1, t_2(e_2) + k) \leq t_2(e_2) + k$ (recall that $k = dist(e_1, e_2)$), the latest timestep when path p_1 visits vertex e_2 is no larger than timestep $t_2(e_2)$. $t_2(e_2)$ is the earliest timestep when path p_2 can visit vertex e_2, so path p_1 visits vertex e_2 before path p_2. Similarly, path p_2 visits vertex e_1 before path p_1. Thus, paths p_1 and p_2 have a conflict in the corridor between vertices e_1 and e_2. Therefore, the property holds. □

F Proof for the Corridor-Target Reasoning Technique

Property 4.10. *For all combinations of paths of agents a_1 and a_2 with a corridor-target conflict, if one path violates constraint set C_1 and the other path violates constraint set C_2, then the two paths have one or more vertex or edge conflicts inside the corridor.*

Proof. Since any path of agent a_1 cannot violate the length constraints $l_1 > l$ and $l_1 \leq l$ simultaneously, we only need to consider the case where a path of agent a_1 violates $l_1 > l$ and a path of agent a_2 violates $\langle a_2, e_2, [0, t'_2(e_2) - 1] \rangle$ (Case 1) or $l_2 > t'_2(g_2)$ (Case 2).

Case 1 We first consider the case where the target vertex of agent a_2 is outside of the corridor. Let path p_1 be an arbitrary path of agent a_1 that is no longer than l and path p_2 be an arbitrary path of agent a_2 that visits vertex e_2 at timestep $\tau_2 \in [0, t'_2(e_2) - 1]$. We need to prove that paths p_1 and p_2 have one or more vertex or edge conflicts inside the corridor. Since $\tau_2 \leq t'_2(e_2) - 1 < t'_2(e_2)$ (recall that $t'_2(e_2)$ is the earliest timestep when agent a_2 can reach vertex e_2 without using the corridor between vertices e_1 and e_2), path p_2 traverses the corridor. Since the target vertex of agent a_1 is inside the corridor, path p_1 eventually enters the corridor via endpoints e_1 or e_2 without leaving it again. Assume that path p_1 enters the corridor via endpoint e_i for $i = 1,2$ at timestep τ_1 (without leaving it again). Then,

$$\begin{aligned}
\tau_1 &\leq \mathit{length}(p_1) - \mathit{dist}(e_i, g_1) && \text{(F.1)}\\
&\leq l - \mathit{dist}(e_i, g_1) && \text{(F.2)}\\
&= \min_{j=1,2}\{\max\{t_1(e_j) - 1, t_2(e_j)\} + \mathit{dist}(e_j, g_1)\} - \mathit{dist}(e_i, g_1) && \text{(F.3)}\\
&\leq (\max\{t_1(e_i) - 1, t_2(e_i)\} + \mathit{dist}(e_i, g_1)) - \mathit{dist}(e_i, g_1) && \text{(F.4)}\\
&= \max\{t_1(e_i) - 1, t_2(e_i)\} && \text{(F.5)}\\
&\leq \max\{\tau_1 - 1, t_2(e_i)\} && \text{(F.6)}\\
&= t_2(e_i), && \text{(F.7)}
\end{aligned}$$

Inequality (F.1) holds because agent a_1 needs at least $\mathit{dist}(e_i, g_1)$ timesteps to move from vertex e_i to vertex g_1, i.e., $\mathit{length}(p_1) - \tau_1 \geq \mathit{dist}(e_i, g_1)$. Inequality (F.2) holds because of the assumption that $\mathit{length}(p_1) \leq l$. Equation (F.3) holds because of Equation (4.8). Inequality (F.4) holds because $\min\{x, y\} \leq x$ holds for any x and y. Equation (F.5) rearranges the terms. Inequality (F.6) holds because of the definition of $t_1(e_i)$. Equation (F.7) holds because, otherwise, from Inequality (F.1), we would have $\tau_1 \leq \tau_1 - 1$. These inequalities result in

$$\tau_1 \leq t_2(e_i), \tag{F.8}$$

which indicates that path p_1 enters the corridor via endpoint e_i (without leaving it again) no later than path p_2 visits endpoint e_i. Since path p_2 traverses the corridor, paths p_1 and p_2 have one or more vertex or edge conflicts inside the corridor.

Case 2 We then consider the case where the target vertices of both agents are inside the corridor. Let path p_1 be an arbitrary path of agent a_1 that is no longer than l and path p_2 be an arbitrary path of agent a_2 that is no longer than $t_2'(g_2) - 1$. We need to prove that paths p_1 and p_2 have one or more vertex or edge conflicts inside the corridor. Since $length(p_2) \leq t'_2(g_2) - 1 < t'_2(g_2)$ (recall that $t'_2(g_2)$ is the earliest timestep when agent a_2 can reach its target vertex g_2 via endpoint e_2), path p_2 reaches its target vertex g_2 via endpoint e_1, i.e., path p_2 reaches its target vertex g_2 via vertex g_1. Since the target vertex of a_1 is inside the corridor, path p_1 eventually enters the corridor via endpoint e_1 or e_2 without leaving it again. (1) If path p_1 enters the corridor via endpoint e_2, then path p_1 reaches its target vertex g_2 via vertex g_1. So, paths p_1 and p_2 have one or more vertex or edge conflicts inside the corridor. (2) If path p_1 enters the corridor via endpoint e_1, say at timestep τ_1, then, according to Inequality (F.8), we know that $\tau_1 \leq t_2(e_1)$, which indicates that path p_1 enters the corridor via endpoint e_1 (without leaving it again) no later than path p_2 visits endpoint e_1 when it traverses the corridor. Therefore, paths p_1 and p_2 have one or more vertex or edge conflicts inside the corridor.

Therefore, the property holds. $\square$