Prioritised Planning: Completeness, Optimality, and Complexity

JONATHAN MORAG*, Ben-Gurion University of the Negev, Israel
YUE ZHANG, Monash University, Australia
DANIEL KOYFMAN, Ben-Gurion University of the Negev, Israel
ZHE CHEN, Monash University, Australia
ARIEL FELNER, Ben-Gurion University of the Negev, Israel
DANIEL HARABOR, Monash University, Australia
RONI STERN, Ben-Gurion University of the Negev, Israel

Prioritised Planning (PP) is a popular approach for multi-agent and multi-robot navigation. In PP, collision-free paths are computed for one agent at a time, following a total order over the agents, called a priority ordering. Many MAPF algorithms follow this approach or use it in some way, including several state-of-the-art MAPF algorithms, although it is known that PP is neither complete nor optimal. In this work, we characterise the space of problems a PP algorithm can solve, and define the search problem of identifying whether a given MAPF problem is in that space. We call this search problem Prioritised MAPF (P-MAPF) and investigate its computational complexity, showing that it is generally NP-hard. Then, we develop a novel efficient search algorithm called Path and Priority Search (PaPS), which solves P-MAPF, providing guarantees of completeness and optimality. We next observe that PP algorithms operate with two primary degrees of freedom – the choice of priority ordering, and the choice of individual paths for agents. Accordingly, we further divide P-MAPF into two planning problems corresponding to the two degrees of freedom. We call them Priority-Function Constrained MAPF (PFC-MAPF), where the path choice is fixed while the priority ordering is not, and Priority Constrained MAPF (PC-MAPF), where the priority ordering is fixed while the path choice is not. We analyse these problems as well, and show how PaPS can be easily adapted to create algorithms that solve these problems optimally. We experiment with our algorithms in a range of settings, including comparisons with existing PP baselines. Our results show how the different degrees of freedom of PP-based algorithms affect their behaviour, and provide the first-known results for solution-quality optimality for PP-based algorithms on a popular MAPF benchmark set. The latter can be used as a lower bound for any PP algorithm.

JAIR Track: Multi-Agent Path Finding

JAIR Associate Editor: Erez Karpas

JAIR Reference Format:
Jonathan Morag, Yue Zhang, Daniel Koyfman, Zhe Chen, Ariel Felner, Daniel Harabor, and Roni Stern. 2025. Prioritised Planning: Completeness, Optimality, and Complexity. *Journal of Artificial Intelligence Research* 84, Article 21 (November 2025), 47 pages. DOI: 10.1613/jair.1.19358

*Corresponding Author.

Authors' Contact Information: Jonathan Morag, ORCID: 0000-0002-3778-0018, moragj@post.bgu.ac.il, Ben-Gurion University of the Negev, Beer-Sheva, Israel; Yue Zhang, ORCID: 0009-0004-6091-7213, Yue.Zhang@monash.edu, Monash University, Melbourne , Australia; Daniel Koyfman, ORCID: 0009-0004-6137-8602, koyfdan@post.bgu.ac.il, Ben-Gurion University of the Negev, Beer-Sheva, Israel; Zhe Chen, ORCID: 0000-0002-0425-3131, zhe.chen@monash.edu, Monash University, Melbourne , Australia; Ariel Felner, ORCID: 0000-0003-0065-2757, felner@bgu.ac.il, Ben-Gurion University of the Negev, Beer-Sheva, Israel; Daniel Harabor, ORCID: 0000-0001-6828-7712, Daniel.Harabor@monash.edu, Monash University, Melbourne , Australia; Roni Stern, ORCID: 0000-0003-0043-8179, roni.stern@gmail.com, Ben-Gurion University of the Negev, Beer-Sheva, Israel.

DOI: 10.1613/jair.1.19358

1 Introduction

Multi-Agent Path Finding (MAPF) is the problem of planning the movement of multiple agents in a shared environment, such that they do not collide with static obstacles in the environment and with each other (Stern et al. 2019). This problem manifests in applications such as automated warehouses (Li, Tinka, et al. 2021; Ma, Li, et al. 2017; Morag, Stern, et al. 2023; Varambally et al. 2022). Solving MAPF is NP-hard (Nebel 2020) for directed graphs and can be done in polynomial time in the special case of MAPF on undirected graphs (Daniel Kornhauser 1984). The problem graph is typically unweighted. In general, it is preferred that agents reach their targets as quickly as possible. Accordingly, the quality of a solution is usually measured by considering the sum of the lengths of all paths, known as the *Sum-of-Costs* (SOC). Finding solutions to MAPF problems with optimal (lowest) SOC is also NP-hard, even on undirected graphs (Yu and LaValle 2013).

Academics and practitioners alike have been seeking ways to mitigate the computational complexity of MAPF. One commonly used approach to do so is Prioritised Planning (PP) (Erdmann and Lozano-Pérez 1986). In PP for MAPF, paths are found for each agent, one at a time, following a given priority order $\mathcal{P} = \{a_1 \prec a_2 \prec \ldots \prec a_k\}$. When planning for an agent, we aim for it to reach its target as quickly as possible, while avoiding the paths previously planned for the higher-priority agents. Many MAPF algorithms follow this approach, including algorithms that are applied in Robotics and AI (Li, Chen, Harabor, P. Stuckey, et al. 2021; Li, Chen, Harabor, P. J. Stuckey, et al. 2022; Ma, Harabor, et al. 2019; Silver 2005; Van Den Berg and Overmars 2005). PP algorithms are usually fast and scalable (Li, Chen, Harabor, P. Stuckey, et al. 2021), since a path for each agent is only planned once, and the quality of the returned solution is often close to optimal (Ma, Harabor, et al. 2019). Nevertheless, PP algorithms have two main drawbacks: (i) they are incomplete, i.e., they may fail to find any solution to a MAPF problem instance that is solvable; and (ii) they are suboptimal, i.e., the solutions they return have no quality guarantees. This raises the question that is the focus of this work: **how to characterise the problems that can be solved by a PP algorithm and the quality of the solutions that it can return.**

To characterise the space of problems that can be solved by a PP algorithm and the space of solutions they may return, we define a PP algorithm according to the following two key design choices in all PP algorithms:

> **Choice of Priority Ordering.** That is, the order of agents according to which a PP algorithm will plan paths. The number of such orderings is factorial in the number of agents.
> **Choice of Individual Paths.** When planning a path for an agent a_i, which of the individually optimal paths that avoid the higher-priority agents' paths to choose from. The number of such paths depends on the topology of the problem graph G, the source and target locations s_i and t_i, and the paths of previous agents, but they are often numerous (Li, Harabor, P. J. Stuckey, Ma, et al. 2019).

Correspondingly, we define a *PP algorithm* as a pairing of a priority ordering $\mathcal{P}$ and a path-function $\mathcal{F}$, where the former specifies a total order over the agents and the latter is a function that chooses which specific optimal path to return for a given agent (given the higher-priority agents' paths). The choice of which priority ordering $\mathcal{P}$ is known to have a significant effect on the odds of finding a solution and the quality of that solution. Accordingly, previous works have mainly focused on finding "good" orderings (Bennewitz et al. 2002; Ma, Harabor, et al. 2019; Zhang et al. 2022). We show, both theoretically and empirically, that the choices of both $\mathcal{P}$ and $\mathcal{F}$ are important. That is to say, some pairing of $\mathcal{P}$ and $\mathcal{F}$ may lead PP to fail to find a solution, or to find a low-quality (high-cost) solution.

This is illustrated in Figure 1. In this example, the source vertex and target vertex of each agent a_i are labelled s_i and t_i, respectively. Vertices v_1, v_2, v_3 are additional vertices in the graph without any special association. Consider a priority ordering $\mathcal{P} = \{a_1 \prec a_2 \prec a_3\}$. Clearly, a_1 plans its shortest path from its source s_1 to its target t_1. Then, when it is a_2's turn to plan, it will use some path-function $\mathcal{F}$ to select among several equivalent-cost paths to reach its target t_2 while avoiding the path of a_1. To avoid conflicting with a_1, a_2 will have to move into s_3, v_3, or t_3, and also perform one wait action. This still leaves a_2 with multiple paths (of cost 5 for a_2) to choose from. Let

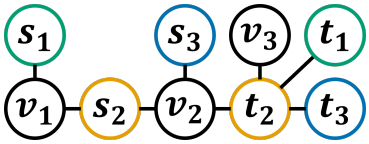

Fig. 1. A MAPF problem with three agents that may be solved by PP under certain conditions. Vertices s_i and t_i indicate the source and target positions of agent i.

us focus on three of these possible paths. The first path, $[s_2, s_2, v_2, t_2, v_3, t_2]$, will result in an optimal (minimal) SOC for all agents (which is 13). The second path, $[s_2, v_2, t_2, v_3, v_3, t_2]$, causes additional waiting (+1) for agent a_3 (when a_2 passes in v_2), resulting in a sub-optimal solution (a total SOC of 14). The third path, $[s_2, v_2, s_3, s_3, v_2, t_2]$, produces a deadlock failure, since a_3 is not able to move away from s_3, resulting in an unavoidable collision. Thus, the choice of individual paths is clearly crucial for the success of PP in this example. Alternatively, using a different priority ordering would change the set of paths that the agents can choose from, possibly increasing the cost of a solution or resulting in a failure. For example, any ordering where a_2 has the highest priority will result in a deadlock, as a_2 would occupy t_2 indefinitely, blocking both other agents from reaching their targets.

Then, we define the *Prioritised MAPF* (P-MAPF) problem, which is the problem of solving a MAPF problem with any PP algorithm using any choice of $\mathcal{P}$ and $\mathcal{F}$. Any solution returned for P-MAPF is a solution that could be returned by a PP algorithm for some choice of $\mathcal{P}$ and $\mathcal{F}$ and vice versa. We analyse P-MAPF theoretically, showing that computing feasible P-MAPF solutions is NP-hard. We then propose a new algorithm, *Path and Priority Search* (PaPS), which computes optimal solutions to P-MAPF problems and returns failure if no such solution exists. PaPS is the first prioritised algorithm to have these guarantees.

We then decompose P-MAPF into two separate problems, along the lines of PP's two degrees of freedom: The first problem is called *Priority Constrained MAPF* (PC-MAPF), and is the problem of finding a solution that is prioritised with respect to a specific, given, priority ordering $\mathcal{P}$. We show PaPS can easily be adapted into an optimal and complete algorithm for PC-MAPF, which we call Priority Constrained Search (PCS). We also propose a simple and sub-optimal algorithm, *Prioritised Planning with Randomised A** (PPR*), which uses randomisation to sample the space of feasible PC-MAPF solutions. The second problem is called *Priority-Function Constrained MAPF* (PFC-MAPF), and is the problem of finding a solution that is prioritised with respect to any ordering, but where $\mathcal{F}$ is given as input. Here too, we show PaPS can easily be adapted into an optimal and complete algorithm, this time for PFC-MAPF. We call this algorithm Priority-Function Constrained Search (PFCS). We further show that these problems are also NP-hard.

In an empirical analysis, we compare our optimal and sub-optimal algorithms with each other, as well as with existing baselines, in terms of solution quality, runtime, and success rate. We show that our proposed methods can succeed where a PP algorithm fails, and we report first optimal solutions for a wide range of P-MAPF problems from recent literature. We also use a sampling-based approach to approximate a P-MAPF oracle that scales to more agents than PaPS. We use this to show the potential of better path choice in PP and inspire future works that would leverage it.

The rest of this paper is structured as follows. In section 2, we give background on MAPF and relevant algorithms, techniques, and data structures. Section 3 introduces and analyses P-MAPF. In section 4 we describe PaPS and its properties. Section 5 introduces PFC-MAPF and PC-MAPF[1], as well as algorithms for solving these problems. Section 6 presents an experimental evaluation of new and existing algorithms. In section 7 we give

[1]Our preliminary work on PC-MAPF was published in the Proceedings of the 17th International Symposium on Combinatorial Search (2024).

our conclusions and directions for future work, followed by acknowledgements in section 8. Finally, appendix A includes long complexity proofs which we skip in the main body of the paper, and appendix B includes extended results for our experiments.

2 Background and Definitions

In this work, we consider the classic Multi-Agent Path Finding (**MAPF**) model from Stern et al. (2019). The input of a MAPF problem is a directed graph $G = (V, E)$ and a set of k agents. Each agent a_i has an associated source location $s_i \in V$ and target location $t_i \in V$, given as a source vector s and the target vector t, respectively. Time is discretised into time steps. At each time step x an agent at vertex $v \in V$ can move to an adjacent vertex $u \in V$ in the graph (provided there exists an appropriate edge $(u, v) \in E$) or wait at its current location. A *path* is a sequence of moves and waits that transition an agent a_i from s_i to t_i, and its *cost* is the number of time steps required to traverse it. A *conflict* occurs when two agents a_i and a_j attempt to occupy the same vertex $v \in V$ at the same time x, called *vertex conflict* (denoted as $\langle a_i, a_j, v, x \rangle$), or traverse the same edge $(u, v) \in E$ (or the inverse edge (v, u)) at the same time x, called *edge conflict* (denoted as $\langle a_i, a_j, u, v, x \rangle$). A *solution* π to the MAPF problem is a set of k paths, one for each agent. We say that a solution is *valid* (equivalently, *feasible*) if the paths are conflict-free. If a solution does not meet the definition above in any way, such as by containing conflicts or only mapping some of the agents to paths, we call it *invalid* (equivalently, *infeasible*). The cost of a solution is defined as the *Sum-of-Costs* (SOC), which is the sum of the path cost of each agent; i.e., $\sum_{i=1}^{k} cost(\pi_i)$. An alternative, less commonly used cost function is *Makespan*. This function is simply the maximal path cost, and disregards all paths with a lower cost; i.e. $\max_{i=1}^{k} cost(\pi_i)$. We will only focus on SOC in this paper. A solution is called *optimal* if it has the lowest SOC of all valid solutions to a given problem.

2.1 Heuristic Search

Heuristic Search is a common technique for quickly finding solutions to problems with large (even infinite) state spaces. A fundamental and widely used algorithm in this field is A^* (Hart et al. 1968). The following is a standard formulation of A*, though many variations exist. A* searches a *search tree*, where node n contains a *state* as described by the problem definition, and a reference to the node's parent node. The *root* of this tree is a node n_I initialised with the initial state I of the problem. A* efficiently explores the search tree to find a sequence of state transitions, from I in n_I, to a desired goal state G, in a goal node n_G. In some problems, multiple goal states exist, in which case G is implicitly defined by a function that tests if a given state is considered a goal state. A *cost function*, commonly denoted $g(n)$, assigns a cost to the search node. $g(n)$ must be monotonic and non-decreasing, so the cost of a child node is never lower than that of its parent. A *heuristic function*, commonly denoted $h(n)$, estimates the minimal additional cost that must be added on top of $g(n)$ to eventually transition to a goal node n_G. In other words, $h(n)$ is an estimate of $g(n_G) - g(n)$, where $g(n_G)$ has minimal cost among goal nodes reachable from n. A* performs a *best-first search* on the tree in the following manner: First, the root node is generated and inserted into a priority queue called *OPEN*. Then, we iteratively remove (pop) from *OPEN* the node n with a minimal $f(n) = g(n) + h(n)$. After removing the node, we *expand* it. A process in which we *generate* child nodes, one for each state that we may transition to from the current node's state, as defined by the problem formulation. We then insert these nodes into *OPEN* and continue to the next iteration. The search stops when we pop a goal node. We can then follow the chain of parent state references in these nodes, and return the sequence of state transitions from the initial state to the goal state. In some problems, the sequence of transitions can be discarded, as only the goal state is desired.

To guarantee that the goal state that A* finds is optimal, i.e., has minimal cost, the heuristic function $h(n)$ must be *admissible*. An admissible heuristic function is one that never overestimates the remaining cost to get to a goal state. In other words, if $h(n)$ is admissible, then for any node n, $h(n) \leq g(n_G) - g(n)$, where $g(n_G)$ has minimal

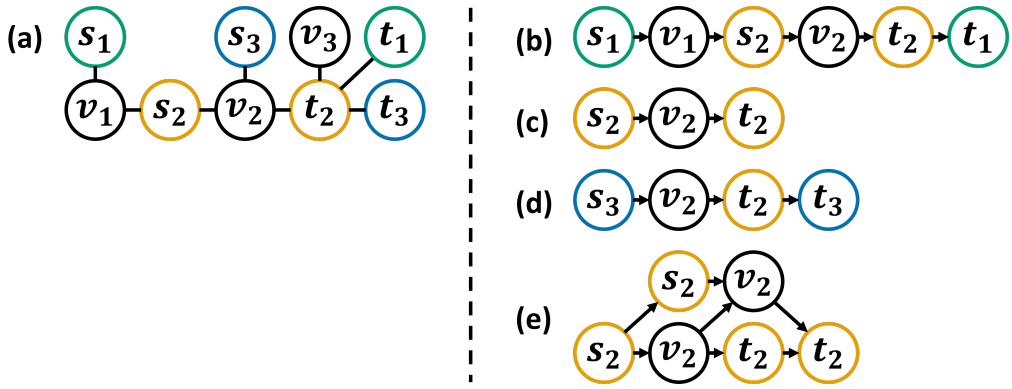

Fig. 2. **(a)** A P-MAPF problem with three agents (repeated from Figure 1 for convenience). Vertices s_i and t_i indicate the source and target positions of agent i. **(b-d)** The minimal MDD of each agent. **(e)** MDD of a_2 with +1 depth.

cost among goal nodes reachable from n. Under certain conditions, A* is also optimally efficient (Dechter and Pearl 1988).

2.2 Path Finding for Individual Agents

Algorithms that solve MAPF often explicitly compute a shortest path for an individual agent a_i from its source s_i to its target t_i, while also avoiding constraints (often arising from paths of other agents). A common way to achieve this, is by running an A* search on a *time-expanded* version of the problem graph (Silver 2005). A state in this search is a tuple $\langle v, x \rangle$ of vertex and time, and its neighbours are $\{\langle u, x+1 \rangle \mid (v, u) \in E \vee u = v\}$. If a neighbour state violates a constraint, it is discarded. The initial state (called a *root*) is the agent's source location and time $\langle s_i, 0 \rangle$. A goal state is a state $\langle v, x \rangle$ where $v = t_i$ and no constraints on v prevent the agent from waiting there indefinitely after time x. We refer to this algorithm as **Temporal A***.

Multi-Valued Decision Diagrams (MDDs) are used by algorithms that need to reason over the set of all equivalent paths of a given cost z (Sharon, Stern, Goldenberg, et al. 2013). An MDD for a given agent, subject to constraints, is a directed acyclic graph. It has a source node, corresponding to the agent's source location, and a sink node, corresponding to the agent's target location. Internal nodes of the MDD are the set of all graph vertices that appear on a path of cost z (called the *depth* of the MDD) from source to sink that satisfies the given constraints. An MDD that has a minimal depth while satisfying the agent's constraints is called *optimal* or *minimal*. In Figure 2, (b-d) shows the minimal (assuming no constraints) MDDs for the three agents in the problem shown in Figure 2(a). Figure 2(e) shows an MDD for agent a_2, with a depth of 3 instead of 2. Note how only one node has to be used for v_2 at depth 2, even though two possible paths pass through it. To build an optimal MDD, a single-agent search is invoked to find paths for the given agent subject to the given constraints. Once all optimal paths are found, each location v reached at time step x on any optimal path is added as an MDD node v at depth x. We define the Cartesian product "$\times$" of a set of MDDs $M = \{m_1, m_2, \ldots, m_{|M|}\}$ as the set of all unique sets of paths (equivalently, solutions) that can be formed by taking one path from each MDD ($\times(M) = m_1 \times m_2 \times \cdots \times m_{|M|}$).

2.3 MAPF Algorithms

Approaches for solving MAPF can be roughly divided into *optimal* or *sub-optimal* algorithms, and into *complete* or *incomplete* algorithms. Optimal MAPF algorithms guarantee to return a solution with minimal cost, whereas

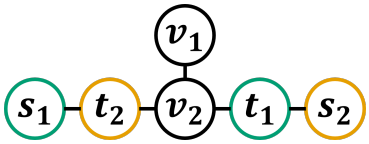

Fig. 3. A MAPF problem that is solvable, but for which PP could never find a solution (Čáp, Novák, et al. 2015). Vertices s_i and t_i indicate the source and target positions of agent i.

sub-optimal algorithms do not. Complete MAPF algorithms guarantee to find a solution if one exists, and return *no-solution* if one does not exist. Incomplete algorithms might not return a solution even when one exists. Alternatively, an algorithm may be *solution-complete*, meaning it guarantees to find a solution if one exists, but might continue running indefinitely when no solution exists. An algorithm is *sound* if it only returns valid (feasible) solutions. Typically in MAPF, only sound algorithms are considered.

Conflict Based Search (CBS) (Sharon, Stern, Felner, and N. R. Sturtevant 2015) is an optimal and solution-complete MAPF algorithm of particular relevance for this paper. CBS is a two-level search algorithm. In the **low-level search**, CBS invokes Temporal A* to find an optimal path for each agent that satisfies all its constraints from the high-level search while ignoring other agents' planned paths. The **high-level search** of CBS performs a best-first search on a binary search tree, known as the *Constraint Tree* (CT). Each node in the CT contains a set of paths, one path for each agent. Put together, these paths form a solution, though it may contain conflicts. During the search, CBS maintains a priority queue (*OPEN*) of unexplored CT nodes, which prioritises nodes with the lowest solution cost (SOC) for expansion. The root node of the CT is created by invoking the low-level search without constraints, once for each agent, creating a set of individually optimal paths that may conflict with each other. The root node is then inserted into *OPEN*. Next, CBS pops a CT node from *OPEN*, detects conflicts in the current solution and selects one conflict $\langle a_i, a_j, v, x \rangle$ between a pair of agents (a_i, a_j) (assume a vertex conflict for simplicity). To resolve the conflicts, CBS expands the node and generates two child nodes. Each child node is assigned one new constraint, either $\neg \langle a_i, v, x \rangle$ or $\neg \langle a_j, v, x \rangle$, prohibiting one of the two agents involved in the conflict from using the vertex at the time of the conflict. Each child node then runs a low-level search with the addition of the new constraint to compute a new path for the constrained agent. These nodes are then inserted into the priority queue. This process iterates until a node with no conflicts is popped from *OPEN*.

Prioritised Planning (PP) (Erdmann and Lozano-Pérez 1986; Silver 2005) is a general problem-solving technique popularly used to tackle MAPF and other similar coordination problems. PP assumes a total order over the agents, and then proceeds to compute paths for the agents, one by one, in the specified order. For each agent in its turn, PP computes an individually optimal path that avoids the paths of preceding (equivalently, higher-priority) agents. Note that it does not matter whether the order is given as input or revealed on the fly, so long as each agent knows the paths of the agents that preceded it in the order. PP does not provide any completeness or optimality/sub-optimality guarantees. Among all MAPF problems that have feasible solutions, only a subset is solvable using PP. Moreover, when PP fails, it is unknown whether a feasible solution exists that satisfies the given priority order, or any other order (Ma, Harabor, et al. 2019). The example in Figure 3 shows a solvable MAPF problem instance that PP nevertheless can not solve. In this instance, there are two agents, a_1 and a_2, starting at s_1 and s_2 respectively, and going to t_1 and t_2 respectively. To solve this instance, the agents must swap their positions relative to each other in the corridor – i.e., a_1 must be right of a_2. Clearly, in any valid solution, one of the agents will have to move up into v_1, while the other will have to wait in place to allow the first agent to reach v_1 without colliding at v_2. PP will never find such a solution. Instead, the first agent will plan to move to its target without stopping, trapping the other agent in a dead-end and causing an unavoidable

collision. One way to view this deficiency is to say that in some cases, agents must exchange priorities on the fly, as they advance, rather than have static priorities.

When planning for a single agent, PP may discover that there exist multiple equivalent-cost paths to the target. The decision of which path to choose is handled by tie-breaking during search, often arbitrarily, such that one path is selected.

2.4 MAPF Variations

One advantage of PP is that its simplicity allows it to easily be extended to support non-standard definitions of MAPF. While this work focuses on the classic formulation of MAPF, many variations on the problem have been suggested. The following is a non-comprehensive list of MAPF variations:

- **Weighted Graphs.** In the classic MAPF formulation, the problem graph is unweighted, and agents move "instantly" between vertices, in the span of one time step. Some works considered weighted graphs, where traversing an edge may take multiple time steps (Barták et al. 2018).
- **Continuous Time.** Instead of using discrete time steps, some formulations use continuous time to allow a more expressive state description (Andreychuk et al. 2022; Walker 2022).
- **Kinematics.** Unlike classic MAPF, where any traversal of an edge takes the same amount of time, the kinematics of agent movements, such as acceleration and deceleration, are sometimes considered (Hönig et al. 2016). This is typically combined with continuous time.
- **Agent Shape.** Agents typically occupy a single vertex at a time. Previous works considered agents that have physical shapes such as circles, spheres, hexagons, etc. in metric space (Li, Surynek, et al. 2019; Walker 2022), or agents that occupy an arbitrary continuous set of vertices (Atzmon, Zax, et al. 2020).
- **Online and Lifelong.** A common extension of MAPF describes a problem that continues over time, potentially indefinitely. Here, new tasks or new agents are revealed over time, while existing tasks are eventually completed or existing agents leave the problem (Li, Tinka, et al. 2021; Morag, Felner, et al. 2022; Švancara et al. 2019). These formulations are often further extended by giving agents pickup-and-delivery tasks where they must visit one location to pick up a package, and then another location to deliver it (Ma, Li, et al. 2017). In these problems, *throughput*, in the form of the number of tasks completed over time, is measured instead of SOC.
- **Robustness.** Various strategies were introduced for prevention of, or recovery from, failures in planning (Morag, Stern, et al. 2023) and execution (Atzmon, Stern, Felner, N. R. Sturtevant, et al. 2020; Atzmon, Stern, Felner, Wagner, et al. 2020; Ma, Kumar, et al. 2017; Shahar et al. 2021) of MAPF solutions.

3 Prioritised MAPF

In this work, we aim to characterise the space of MAPF problems that a PP algorithm can solve. As a preliminary, we formalise what a *PP algorithm* is. We defined a PP algorithm by a pair $\langle \mathcal{P}, \mathcal{F} \rangle$. $\mathcal{P}$, referred to as the *priority order*, is a total order over the set of agents, i.e., a relation where for any pair of agents a_i and a_j, either $a_i \prec a_j$ or $a_j \prec a_i$. $\mathcal{F}$, referred to as the *path-function*, is a function that accepts as input a pair of vertices $v_s, v_t \in V$ and a set of constraints (usually, the paths of higher-priority agents), and outputs a shortest path from v_s to v_t that avoids the given constraints. A PP algorithm $\langle \mathcal{P}, \mathcal{F} \rangle$ iterates over the agents according to the priority order ($\mathcal{P}$) and finds a path for each agent using $\mathcal{F}$.

To characterise the space of MAPF problems that a PP algorithm can solve, we also define the following concepts. A MAPF solution π is *priority-constrained* w.r.t. $\mathcal{P}$ if for every agent a_i, the cost of the plan $\pi(a_i)$ is *smaller than or equal to* the cost of any path for a_i that avoids conflicts with the paths in π of the higher-priority agents. This means that the paths assigned in π for agents whose priority is lower than a_i did not prevent

assigning a_i with a lower cost path.[2] A MAPF solution is called *prioritised* if it is priority-constrained w.r.t. at least one priority order over the agents.

Theorem 1. *A MAPF problem is solvable by a PP algorithm iff there exists a prioritised solution for this problem.*

Proof. (PP $\rightarrow$ Prioritised) PP plans agents one by one, from highest priority to lowest priority, as specified by some $\mathcal{P}$. Each agent a_j takes an individually optimal path but must avoid the previously planned paths of higher-priority agents $\{a_i \mid a_i \prec a_j\}$. Removing the path of a_j cannot improve the arrival time of any a_i, since the path of each a_i is a shortest path computed independently of a_j. Thus, the path of each a_i and a_j is valid, and the solution as a whole is prioritised.

(Prioritised $\rightarrow$ PP) By definition, there exists an ordering $\mathcal{P}$ where the paths of any two agents $a_i, a_j \mid i \prec j \in \mathcal{P}$ are such that the path of a_i is independent of any constraints introduced by a_j. Therefore, there exists a path function $\mathcal{F}$ for which this path is returned for a_i when that agent is planned by PP, according to $\mathcal{P}$. □

A direct corollary of Theorem 1 is that if a MAPF problem does not have a prioritised solution, then PP cannot solve this problem and vice versa. Next, we consider the challenge of identifying whether a given MAPF problem has a prioritised solution, i.e., whether it can be solved by a PP algorithm. We call the problem of finding a prioritised solution for a MAPF problem, the Prioritised MAPF (P-MAPF) problem.

3.1 Properties of P-MAPF

An algorithm is called *P-MAPF-complete* if, given a P-MAPF problem, it guarantees to find a prioritised solution if one exists, or return *no solution* if no such solution exists.

Theorem 2. *The decision problem of whether a prioritised solution exists is* NP*-hard on directed graphs.*

The proof for Theorem 2 is in appendix A.1.

Specific sub-classes of MAPF problem instances might be devised, where the equivalent P-MAPF problem instances are all solvable. A prominent example is *well-formed* instances (Čáp, Novák, et al. 2015; Čáp, Vokřínek, et al. 2015), where the sources and targets of all agents are positioned such that each agent has at least one path that does not visit the source or target location of any other agent. Given such an instance, we can pair any ordering with a path-function that explicitly prohibits (via additional constraints) each agent from visiting any of the source and target locations of all other agents. The resulting PP algorithm is guaranteed to produce a solution, though the added constraints may result in longer paths. An intuitive explanation of this guarantee is that each agent can, in the worst case, stay at its source location until all higher-priority agents have finished moving into their target locations, and then move into its own target location. Of course, not all MAPF instances are well-formed.

Seeing as how some MAPF instances are not solvable with PP, it would be desirable to know, given a MAPF instance, if it can be solved by PP. If the instance can not be solved using PP, then attempting to do so would be a fruitless endeavour, and we could instead use some other MAPF algorithm. However, Theorem 2 implies that given an arbitrary MAPF problem instance, discovering if it is possible to solve it using PP is a problem in NP-hard. Therefore, we likely can not easily make such recommendations for arbitrary MAPF instances.

Beyond the question of whether a problem can be solved by a PP algorithm, one may also ask about the quality, i.e., the cost of the solution a PP algorithm may return. It is easy to see that the lowest-cost solution that can be returned by a PP algorithm is a solution to the corresponding P-MAPF problem, that has the smallest cost among all prioritised solutions. Such a solution is referred to as a P-MAPF-optimal solution.

Theorem 3. *The optimisation problem of P-MAPF is* NP*-hard on undirected graphs.*

[2] Ma, Harabor, et al. (2019) referred to a priority-constrained solution as a solution that is *consistent* with the priority order.

The proof for Theorem 3 is in appendix A.2.

It is easy to see that for every P-MAPF problem there exists an analogous MAPF problem where the input is identical, but the returned solution may not be prioritised. Hence, for any P-MAPF-optimal solution, there exists a corresponding MAPF solution with an equal or smaller cost. Similarly, if a MAPF problem is unsolvable, then the corresponding P-MAPF problem is also unsolvable. However, the reverse is not true; i.e., for an unsolvable P-MAPF problem there may exist a valid MAPF solution.

A straightforward approach for optimal P-MAPF is to alternate between enumerating all options for which agent should be next in the ordering, and enumerating all optimal constrained paths for the current agent in the ordering. We can do so using a search tree, which would start with an empty root node. The root would then split on the choice of agent. Since the root is empty, all agents in the problem are valid options for being next (first) in the ordering, so we create k child "agent nodes", one for each agent. Each of these nodes would then split on the choice of path for the newly added agent, creating a child "path node" for each possible path that is optimal (shortest) while avoiding the paths of previous agents. We will then split each of these nodes on the choice of the next agent, which may be any agent that has not yet appeared in the branch. Thus, each node will have $k - 1$ child nodes. This process will continue in each branch of the tree, until all agents have been added and a path has been chosen for each. The resulting tree would have a depth of $2k$, with alternating layers of agent nodes and path nodes.

A trivial algorithm for P-MAPF would therefore be to search this tree by expanding nodes in a best-first order (smallest SOC). We call this algorithm *Exhaustive Prioritised Planning* (EPP). EPP guarantees to eventually find a P-MAPF-optimal solution and eventually terminate if no such solution exists. The main disadvantage is a large branching factor, as the number of ways to order k agents is $k!$, and even within a single ordering, many individually-optimal paths can exist for each agent.

Note that Priority-Based Search (PBS) (Ma, Harabor, et al. 2019), a popular prioritised MAPF algorithm, does not search this tree, and is neither optimal nor complete for P-MAPF. PBS only searches over the space of (partial) orderings, without systematically exploring the space of path-functions. Additionally, PBS uses a depth-first search strategy on its search tree. Even if we were to run PBS with a best-first search, it would not guarantee optimality (on PBS's search space), as its use of partial orderings creates negative-cost edges.

In the next section, we introduce a new and computationally efficient optimal algorithm for P-MAPF called *Path and Priority Search* (PaPS).

4 The Path and Priority Search Algorithm (PaPS)

Path and Priority Search (PaPS) is a novel search algorithm that solves P-MAPF problems. The primary insight behind PaPS is to use MDDs in place of paths. This allows PaPS to efficiently reason about sets of all shortest paths of an agent (subject to constraints), rather than inefficiently enumerate them.

PaPS is a two-level search algorithm. The *high-level* of PaPS explores a novel search tree that gradually adds agents while branching over possible agent orderings (similarly to EPP) and over conflicts between MDDs of agents. The *low-level* of PaPS finds MDDs for individual agents, resolving conflicts by avoiding constraints imposed by the high-level search. PaPS is similar in broad strokes to CBS (Sharon, Stern, Felner, and N. R. Sturtevant 2015), but differs from it in several important ways:

(1) PaPS stores a set of paths for each agent (in an MDD) in every search node, while CBS stores only a single path per agent.
(2) A node in the PaPS high-level search tree may include paths of only a subset of the agents. The root of the search tree does not include any path, and PaPS gradually adds agents (and their paths) as the search tree is generated.

Algorithm 1 PaPS

Input: a problem instance $\langle G, k, s, t, \rangle$, a heuristic function H
1: $OPEN \leftarrow$ priority-queue
2: $root \leftarrow$ an empty PT node
3: $OPEN.insert(root)$
4: **while** $OPEN$ not empty **do**
5: $n \leftarrow OPEN.pop()$
6: **if** $|n.M| = k$ and $n.M$ is conflict free **then**
7: **return** solution from $n.M$
8: **else if** $n.M$ is conflict free **then**
9: $ExpandOnAgents(n)$
10: **else**
11: $ExpandOnConflict(n)$
12: return no-solution

(3) PaPS has two expand operations. The first is a binary split similar to that of CBS. The second creates one child node for each agent not yet added in the expanded node.
(4) To resolve a conflict, PaPS employs a novel type of constraint that favours high-priority agents, and it never generates nodes where the cost of these agents increases.

Next, we describe PaPS in full detail.

4.1 High-level Search

At the high-level, PaPS conducts a best-first search over the *Priority Tree* (PT); which is a search tree where each node n has the following fields:

- $n.\mathcal{P}r$ - a total ordering over a subset of the agents.
- $n.c$ - The identity of the lowest priority agent according to $n.\mathcal{P}r$. We refer to this agent as the *current agent* of n, and denote it by a_c.
- $n.M$ - a mapping of the agents in $n.\mathcal{P}r$ to MDDs, where each agent is mapped to a single MDD.
- $n.constraints$ - a set of constraints relevant for node n.

We say that a search node n in the PT contains *disjoint MDDs* if no two MDDs in $n.M$ contain the same node (vertex and time) or edge (pair of vertices and time). When $n.M$ is disjoint, each agent mapped to an MDD in $n.M$ can reach its target without any collisions by following any path in its MDD. In other words, the Cartesian product $\times(n.M)$ is a set of conflict-free sets of paths. If n contains disjoint MDDs, then we consider n to be conflict-free. Otherwise, we consider n to have a conflict.

Algorithm 1 shows the pseudo code for the high-level search. We use a priority queue, *OPEN*, to determine the order of node expansions. To initialise the search, we generate an empty (all fields set to *null* or $\emptyset$) root node and insert it to *OPEN* (lines 2, 3). Nodes in *OPEN* are ordered according to $f(n) = g(n) + h(n)$, where $g(n)$ is the sum of depths of all MDDs in $n.M$ and $h(n)$ is a heuristic function that estimates the remaining solution cost. We describe two such heuristics in Section 4.5. Let n denote the most recently popped node. If n is conflict-free and covers all agents, then it is a goal node, and the search is done (line 6). This means each agent can reach its target without any collisions by following any path in its MDD. The Cartesian product of all MDDs in the node $\times(n.M)$ is thus a set of feasible solutions to the PaPS problem instance. We can now choose any arbitrary solution from $\times(M)$ and return it (line 7). If n does not meet the criteria of a goal node, PaPS continues the search using the *generate* and *expand* functions described next.

Algorithm 2 ExpandOnAgents

Input: parent node p
1: $updatedConstraints \leftarrow p.constraints$
2: Add all critical resources of $p.M(c)$ to $updatedConstraints$
3: **for** $i \in \{1, \dots, k\} \mid (i \prec p.c) \notin p.Pr$ **do**
4: $\quad Pr \leftarrow p.Pr \cup \{p.c \prec i\}$
5: $\quad Generate(i$, a copy of $p.M$, a copy of $updatedConstraints, Pr)$

4.1.1 Expand. There are two types of *expand* operations in PaPS: ExpandOnAgents (Algorithm 1, line 9) and ExpandOnConflict (Algorithm 1, line 11). The former occurs when all MDDs in the expanded node are disjoint, i.e., there is no conflict between them. The latter occurs when a conflict is detected between the current MDDs. We describe both below.

ExpandOnAgents (Algorithm 2). At this point, we can conclude that we are done with the current agent $p.c$, as it does not conflict with agents that have a higher priority. This brings us to the main difference between the way CBS and PaPS resolve conflicts. To preserve the priority, in PaPS we do not allow a higher-priority agent to increase its cost. This is enforced as follows. *Critical resources* are single nodes or edges of the MDD, the removal of which would introduce a cut – i.e., make it so there is no path from the source to the sink node of the MDD. In particular, the agent's target at any time past the depth of the MDD (the MDD's sink) is a critical resource, since agents stay at their targets indefinitely.

Before we can add any new agents, we add all critical resources of the MDD of the current agent $p.M(c)$ to the constraints (line 2). This ensures that a conflict with a critical resource of a high-priority agent will never occur in the PT. Thus, no lower-priority (upcoming) agents will be able to cut this MDD. We can then move on to addressing agents that we have not yet planned for. We expand p by adding a new agent to the priority ordering, assigning it the lowest priority among current agents (line 4). To ensure we cover all possible orderings, we branch and create a separate child node for each agent that is not already in $p.Pr$ (line 3).

ExpandOnConflict (Algorithm 3). The MDDs $p.M$ inside a parent node p may have an internal conflict. This conflict is a tuple comprising two agents and a contested resource; e.g., a vertex conflict $\langle a_{hi}, a_c, v, x\rangle$ (line 1). One agent is always the current agent, a_c; the other agent is always a higher-priority agent $a_{hi} \prec_{\mathcal{P}r} a_c$. The conflicting resource (vertex or edge) appears in the MDDs of both. There may exist multiple conflicts between the current agent and multiple higher-priority agents, in which case we pick the conflict with the earliest time. To resolve the conflict we call the *ExpandOnConflict* procedure (Algorithm 1, line 11). This procedure branches at the parent node p and produces two child nodes. To resolve the current conflict, in each child node we will first directly constrain a_{hi}, and then later indirectly constrain a_c. The **left child** will have a *positive constraint* $\langle a_{hi}, v, x\rangle$, which says a_{hi} must occupy vertex v at time x; i.e., $MDD_{hi}[t] = \{v\}$ (line 3). We enforce this by using the *Restrict* function (line 7) to create a sub-graph of the MDD $M(hi)$ that only contains paths that satisfy the constraint. This is done by taking the sub-graph of the MDD that is rooted at (v, x) and connecting it to all paths from the root that lead to (v, x) (by searching back from (v, x) to the MDD's root). Other paths are deleted from the MDD. The **right child** will have a *negative constraint* $\neg \langle a_{hi}, v, x\rangle$, which says that a_{hi} must not occupy vertex v at time x; i.e., $v \notin M(hi)[x]$ (line 4). Here too we use *Restrict*, which in this case will remove v from $M(hi)$. Then, as a result, any vertex that is not pointing to, or is not pointed to by, any other vertex is iteratively removed. The procedure of splitting by creating a positive and a negative constraint on the same agent is known as *disjoint splitting* (Li, Harabor, P. J. Stuckey, Felner, et al. 2019) for CBS.

During the *restrict* operation, if new critical resources are identified in the updated $M(a_{hi})$, then we pre-emptively constrain all lower-priority agents from conflicting with these critical resources (line 8, 9). Note that $M(a_c)$ might currently be violating some of these new constraints. This will be handled in the *generate* function.

Algorithm 3 ExpandOnConflict

Input: parent node p
1: $conf \leftarrow$ earliest conflict in $p.M$
2: $a_{hi} \leftarrow conf.a_{hi}$
3: $pos \leftarrow \langle a_{hi}, conf.v, conf.x \rangle$
4: $neg \leftarrow \neg \langle a_{hi}, conf.v, conf.x \rangle$
5: **for** $r \in \{pos, neg\}$ **do**
6: $M \leftarrow$ a copy of $p.M$
7: $M(hi).Restrict(r)$
8: $constraints \leftarrow$ all new critical resources of $M(hi)$ as constraints
9: $constraints \leftarrow constraints \cup p.constraints$
10: $Generate(p.c, M, constraints, p.Pr)$

Algorithm 4 Generate

Input: index of current agent c, mapping of MDDs M, constraints set *constraints*, ordering prefix Pr
1: $n \leftarrow \langle Pr, c, M, constraints \rangle$ // initialize a new node
2: **if** $n.M(c) = null \vee n.M(c)$ violates any constraint in $n.constraints$ **then**
3: $n.M(c) \leftarrow lowLevel(a_c, constraints)$
4: **if** $n.M(c) =$ no-solution **then**
5: Return
6: $OPEN.insert(n)$

Generate (Algorithm 4). This function is invoked as a consequence of expanding (splitting) a parent node p (Alg. 2 line 5, Alg. 3 line 10). It first initialises a new PT node with the given current agent, MDDs, constraints, and ordering prefix (line 1). Next, if $M(c)$ (the MDD of the current agent) has not been created yet (when arriving from *ExpandOnAgents*), or if $M(c)$ violates some constraints (might happen when arriving from *ExpandOnConflict*), then *Generate* calls the low-level search to build $M(c)$ while satisfying the constraints (line 3). Note that in the latter case ($M(c)$ already existed), the new $M(c)$ might have a larger depth (and thus cost) than the MDD it replaced. If the low-level search finds that no valid MDD exists given the current constraints, this node is pruned (lines 4-5). Finally, we add the new node to *OPEN* (line 6).

4.2 Low-Level Search

The low-level search for PaPS generates MDDs. This search is required to return the MDD representing all shortest paths for an agent to reach its target while obeying given constraints. This can be done using A* to search the space of vertices and time, while avoiding constraints by not generating nodes that do not satisfy them. This is a typical way to plan paths in MAPF. However, instead of stopping as soon as a goal node is found, as is sufficient when searching for a single path, we must continue searching until all search nodes with $f(n) = C^*$ (optimal cost) are expanded. During this search, we keep track of all parents of each A* node, to maintain all equivalent length paths to each node. We can then construct the MDD using this information.

For the sake of efficiency, when the agent already has an MDD, but the set of constraints imposed on this agent has changed since it was built, we may try to amend this MDD rather than build one from scratch. In this case, we remove from the MDD any path that violates one of the new constraints. If this procedure removes all paths from source to sink in the MDD, we search for a new MDD as described above. Otherwise, we can use the amended MDD, as it now obeys all current constraints. The amended MDD would have the same depth (cost) as the original MDD, while a new MDD will have a higher depth (cost).

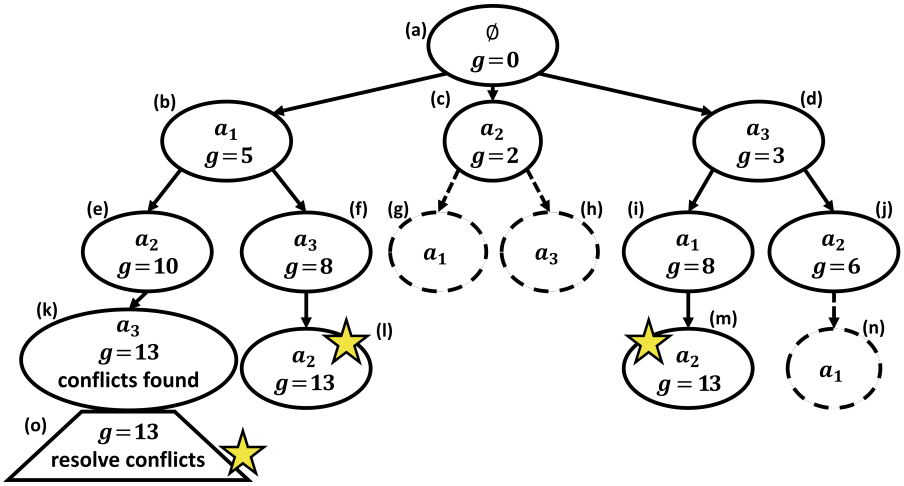

Fig. 4. The PT for the problem in Figure 1, excluding expansions on conflicts.

4.3 PaPS Example

Figures 4 and 5, when put together, show the PT for the problem in Figure 1. We shall first discuss Figure 4, which gives a high-level view of how the PT contains all viable orderings of the agents, and how PaPS searches them. Then we will examine Figure 5, which shows in detail how conflicts are resolved.

The PT, illustrated in Figure 4, starts with an empty root node (a). In each node, we write the newly added agent and the cost (g) of the node. Generated nodes have a solid border, while invalid nodes that were not generated have a dashed border. Goal nodes are marked with a star. The root node is inserted into *OPEN*, and immediately popped and expanded. This generates three nodes (b-d), for each possible choice of highest-priority agent. We insert these into *OPEN*, pop (c) since it has the lowest cost ($g = 2$), and expand it. Both child nodes (g) and (h) are immediately discarded, as the low-level detects there is no possible path for either agent to reach its target given the constraints imposed by a_2. We then expand node (d), adding either a_1 or a_2 as the next agent in the priority order after a_3. We continue expanding nodes in order of minimal cost – (b), (j), (i), (f), and (e). Finally, we end up with three nodes with a cost of $g = 13$: (l) and (m) are both conflict-free and therefore possible goal nodes. If we choose to expand one of them, we will then finish the search. Conversely, (k) contains a conflict, so finding a solution with the ordering $a_1 \prec a_2 \prec a_3$ will require further work. We abstract this procedure away in subtree (o), and explain how this is done in the following example.

Figure 5 focuses on a subtree of the PT for the problem in Figure 1, where the ordering is $\{a_1 \prec a_2 \prec a_3\}$. The figure only shows new or changed MDDs and new constraints in each node, and we limit our focus to vertex constraints. We also indicate the chosen conflict for nodes with conflicts. We start by generating node (b) as a child node of the root (a), by adding agent a_1. $M(1)$, the minimal (and unconstrained) MDD for a_1 is added. All vertices in $M(1)$ are critical resources, as removing any one of them would introduce a cut to the MDD. Thus, negative constraints on all lower-priority agents are now added to *constraints*, preventing any conflicts with these vertices. Next, we are done with $M(1)$, and there are no conflicts since there is only one MDD, so we expand this node, generating child node (e), where agent a_2 is added, creating the ordering prefix $\{a_1 \prec a_2\}$. Next, we add $M(2)$, the minimal MDD for a_2 that obeys the current constraints. We check for conflicts with $M(1)$ and find that there are none, so we are done with $M(2)$. Therefore, we add constraints for its only critical resource, $\langle a_2, t_2, 5+\rangle$. Next, we generate node (k), creating the ordering $\{a_1 \prec a_2 \prec a_3\}$. We then build $M(3)$ while obeying the current

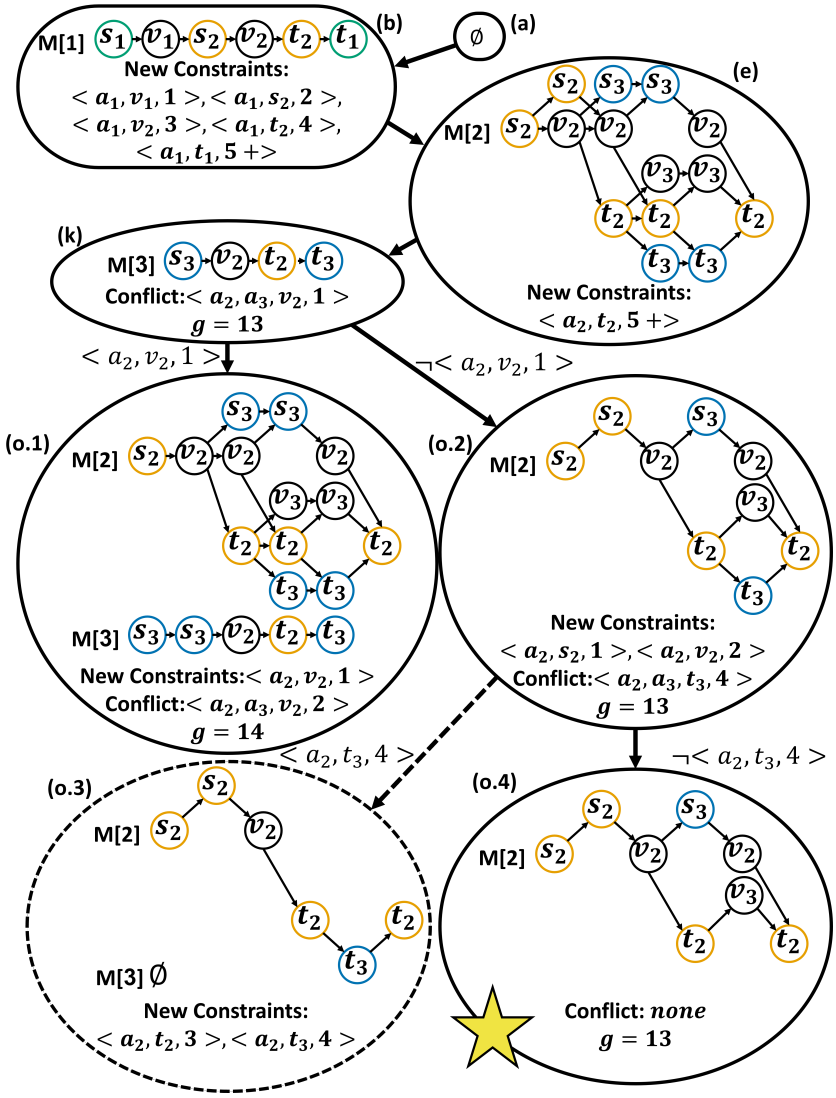

Fig. 5. A part of the PT for the problem in Figure 1, focusing on the ordering $\{a_1 \prec a_2 \prec a_3\}$.

constraints, and detect that it has multiple conflicts with $M(2)$ (not shown). We choose the conflict $\langle a_2, a_3, v_2, 1\rangle$ (other choices are possible). We expand the node (k), generating two child nodes. The left child (o.1) receives the positive constraint $\langle a_2, v_2, 1\rangle$. This constraint is applied to $M(2)$, removing any path that does not include vertex v_2 at time 1. This introduces a new critical resource, so we add the constraint $\langle a_2, v_2, 1\rangle$, and update $M(3)$ to accommodate it, resulting in a cost (g) of 14. We insert (o.1) into *OPEN*. The right child (o.2) receives the negative constraint $\neg\langle a_2, v_2, 1\rangle$. $M(2)$ is restricted accordingly, new critical resources are detected, and their corresponding constraints are added. $M(3)$ remains unchanged since the new constraints do not affect any resource that it uses. (o.2) is inserted to *OPEN*, and immediately popped, as its cost is 13, making it the current minimum. We expand

(o.2), generating (o.3) and (o.4), but (o.3) is pruned since the constraints it adds result in a deadlock where it is impossible for a_3 to reach its target. Next, (o.4) is popped from *OPEN*, and since it contains no conflicts (the MDDs are disjoint), it is considered a goal node. We return the path $[s_1, v_1, s_2, v_2, t_2, t_1]$ for a_1, $[s_3, v_2, t_2, t_3]$ for a_3, and arbitrarily choose between the paths $[s_2, s_2, v_2, s_3, v_2, t_2]$ and $[s_2, s_2, v_2, t_2, t_3, t_2]$ for a_2.

Note that had the problem contained some other agent a_4, (o.4) would not have been a goal node yet. Instead, we would have next expanded (o.4) by adding $a_3 \prec a_4$, building an MDD $M(4)$, checking for conflicts, etc. Additionally, note that node (o.1) may still be expanded, and indeed the subtree rooted in it contains a solution. However, that solution is sub-optimal.

4.4 PaPS Properties

Lemma 1. *The PT is finite.*

Proof. Starting from any node in the PT, putting more constraints on the MDD $M(i)$ of a higher-priority agent $a_i \prec a_c$ will eventually (down the branch) either result in no MDD for a_c (a dead-end), or no conflicts between $M(i)$ and $M(c)$. This is because $M(i)$ will eventually degenerate into a single path (a list rather than a directed acyclic graph). The number of constraints we might put on a_i before it becomes a single path is finite, limited by the initial size of $M(i)$. This translates to a set of constraints (stemming from all parts of $M(i)$ being critical resources) on $M(c)$, preventing conflicts with any component of $M(i)$. Crucially, this set includes a constraint on t_i that starts when a_i arrives at t_i and lasts indefinitely, preventing $M(c)$ from conflicting with it. This allows us to detect when a_c is indefinitely prevented from reaching t_c because a_i is blocking t_i. Thus, this set of constraints will either force us to find an MDD $M(c)$ that does not conflict with $M(i)$, or make it impossible for a_c to reach its target.

Extending this logic to all higher-priority agents, we see that we can only add a finite number of constraints in any branch of the tree, and that adding these constraints either leads to a dead-end, or to a disjoint set of MDDs, at which point we can add another agent to the ordering. The number of agents is also finite, so eventually, as we traverse down the PT, there will be no more agents to add. Thus, all traversals down the PT will eventually reach either a dead-end or a goal node, making the PT finite. □

Theorem 4. *PaPS is sound and complete for P-MAPF*

Proof. A solution is *permitted* by a PT node n if its paths obey all constraints in $n.constraints$. A solution is *represented* by a node if it is included in the Cartesian product of all MDDs in the node.

Notice that if for any arbitrary P-MAPF problem instance, (i) the set of all goal nodes in the PT together permit all valid solutions to the P-MAPF instance, and (ii), none of them represents an invalid solution, then PaPS is sound and complete for P-MAPF. The following proof by induction will prove as much by showing that both properties are maintained whenever we split nodes while building the PT. The basic idea of the proof is that nodes only represent prioritised solutions, goal nodes only represent valid solutions, and that the tree starts by permitting all prioritised solutions and never discards any as it branches. The induction is split into two: The first induction proves that adding new agents (*ExpandOnAgents*) to a node that has no conflicts maintains both properties. The second induction proves that splitting by adding constraints to resolve a conflict (*ExpandOnConflict*) maintains both properties.

Induction 1:

> **Base Case.** When the root node is generated, there are no MDDs, and thus the constraints set is empty. Therefore, all valid solutions are permitted at this point (i). Additionally, since no solution is explicitly represented yet, all represented solutions are valid (and accordingly, prioritised) (ii).

Induction Hypothesis. Assume a parent PT node p that contains $|p.M|$ disjoint MDDs. Also assume that all solutions it represents are prioritised solutions for the set of $|p.M|$ agents that correspond to the MDDs, though not necessarily for the entire problem, as it may contain more agents.
Induction Step. When splitting p on agents, in each child node n, we:

1. Add constraints corresponding to the critical resources of the MDD $p.M(p.c)$.

(i) Some of the solutions permitted by p are not permitted by n. However, none of the lost solutions are prioritised, as they would all involve a lower-priority agent causing a higher-priority agent ($p.c$) to increase its cost.

2. Add a new agent $n.c$ to the ordering prefix $n.\mathcal{P}r$ as the new current (lowest-priority thus far) agent, and we create an MDD $n.M(n.c)$ for it.

(ii) The new MDD $n.M(n.c)$ is built to have the minimal possible depth while avoiding existing constraints, all of which are critical resources of the existing MDD of some higher-priority agent. Thus, all represented solutions are prioritised (though not necessarily valid), as the cost of the new agent's paths can not be reduced.

In the case that with the addition of the new agent $n.c$, the node has come to contain all agents, and that the MDD of the new agent does not conflict with those of existing agents, then all represented solutions are valid. That is because they have a path for each agent, are prioritised, and contain no conflicts. In such a case, we have generated a goal node.

Note that this step will not always lead us back to the hypothesis above. Instead, when the MDD of the new agent $n.c$ conflicts with MDDs of existing agents, we will instead arrive at the base case of induction 2.

Induction 2:

Base Case. When a new agent $p.c$ is added, properties (i) and (ii) are maintained according to the first induction. A special case of adding a new agent, which forms the base case for this induction, is when the MDD of the new agent has conflicts with one of the previous MDDs. Since this is just a special case, both properties also hold here.
Induction Hypothesis. Assume a parent PT node p that contains $|p.M|-1$ disjoint MDDs and also a $|p.M|$th MDD, $p.M(c)$, which has conflicts with some of the higher-priority MDDs.
Induction Step. (i) When the MDD of a higher-priority agent (not $p.c$) is constrained, no prioritised solutions are lost. Observe the set of paths covered by the MDD before constraining it – any path either uses the constrained resource, or does not use it. Therefore, the union of the sets of paths represented by the positively (left child node) and negatively (right child node) constrained versions of the MDD is equal to the set of paths represented by the original MDD. Thus, any path permitted in a parent node is also permitted in one of its child nodes.
(ii) After the current MDD ($M(c)$) is updated to obey the new constraints (avoid the new critical resources), all solutions in the new set of solutions represented by this node are prioritised. This is because the updated MDD is built to contain all paths that have a minimal depth given the current constraints (just as in the induction step of induction 1).
Each child node n we generate at this step may now either have conflicts between c and higher-priority agents, or not have any. If a conflict exists, we return to the hypothesis step of this induction. If a conflict does not exist, we return to the hypothesis step of induction 1, or, if n already contains all agents, then n is a goal node, and all solutions it represents are prioritised and conflict-free.

Thus, both the addition of new MDDs and the addition of new constraints maintain the property that all represented solutions are prioritised, and never eliminate any prioritised solutions from the PT. In particular, goal nodes represent prioritised solutions that are also valid. Any algorithm that systematically explores the PT

and only returns goal nodes will be sound and complete for P-MAPF. In the worst case, PaPS will eventually explore the entire PT, which is finite (Lemma 1). Thus, PaPS is sound and complete for P-MAPF. □

Theorem 5. *PaPS with an admissible heuristic is P-MAPF-optimal*

Proof. First, let us examine the optimality of the search without a heuristic function.

The cost function ($g(n)$) is monotonically non-decreasing: When expanding on a conflict, we constrain the MDD of one of the higher-priority agents; however, the depth of that MDD may not decrease. If new critical resources arise as a result, the current MDD may only increase in depth, as it was already minimal given the parent's constraints. When expanding on agents, we add a new MDD for a new agent that was not covered by the parent node. This action only increases the cost of the child node. Thus, we see that by always expanding the node n with the minimal $g(n)$ from *OPEN* in a best-first-search manner, the first goal node we expand will have minimal cost.

Finally, adding an admissible heuristic to such a cost function maintains the optimality guarantee (Hart et al. 1968). □

4.5 Heuristic Functions

We propose two heuristic functions for the high level of PaPS.

H1 is a simple heuristic that sums the lengths of the shortest path (while ignoring constraints) of each agent not yet in the node.

$$H1 = \sum_{i>|M|} cost(minPath(i))$$

This function can be computed in a preprocessing phase, stored in memory, and then quickly retrieved.

Lemma 2. *H1 is admissible: H1 sums the minimal depths of the MDDs of agents not represented in the node, so the real cost accrued when adding each agent may not be smaller.*

H2 uses a single-agent search algorithm to find the shortest path (while considering constraints but ignoring other agents) for each agent not yet in the node.

$$H2 = \sum_{i>|M|} cost(minPath(i, constraints))$$

These single-agent searches may be done efficiently using modern algorithms such as SIPP (Phillips and Likhachev 2011) or JPST (Hu et al. 2022). We used SIPP in our implementation. Another advantage of this heuristic is that it can detect cases where, given the current constraints in the node, it is already impossible to find a path for one of the upcoming agents. In this case, we prune the node. H2 is more informed than H1, but requires more online computations.

Lemma 3. *H2 is admissible: H2 finds the minimal depth that an MDD for an agent may have given current constraints. Adding more constraints may only increase this cost. Thus, when each agent is eventually added, the cost of its MDD will not be smaller than the cost found by H2.*

4.6 Pruning Equivalent States

Detecting independence between agents is a popular approach for easing the computational difficulty of MAPF (Sharon, Stern, Felner, and N. Sturtevant 2012; Sharon, Stern, Felner, and N. R. Sturtevant 2015; Standley 2010). Existing approaches usually assume independence optimistically, but must continuously re-examine this assumption as they solve the problem. This is because concluding that two (or more) agents are entirely independent in the problem, and will never interfere with each other, is very difficult. Let us call these supposedly

independent agents a_1 and a_2. Some third agent, a_3 may always induce either a_1, a_2, or both, to change their paths, causing a_1 and a_2 to conflict and become dependent.

Fortunately, in P-MAPF, we can make stronger observations regarding independence. The feature of P-MAPF that enables this is that lower-priority agents have a very limited capacity to influence higher-priority agents, as the latter will never increase their cost. Next, we will show how to use this characteristic to detect and prune states in the PT which are functionally equivalent.

We use a form of *Partial Order Reduction* (Godefroid 1996) in PaPS, which only applies to a node n that has only disjoint MDDs; i.e., a node that would be expanded using Algorithm 2. If some other node n' has the same set of MDDs ($n.M = n'.M$), then we can conclude they are equivalent – meaning the sub-trees rooted at each node are identical. Thus, one of the nodes can be discarded. Note that two agents may never be assigned the same MDD (even in different PT nodes), since the source location of each agent, and thus the source node of any MDD assigned to it, is unique. Importantly, this pruning can be done even when the nodes contain different priority orderings ($n.\mathcal{P}r \neq n'.\mathcal{P}r$), so long as the MDDs are identical. We can do this, since PaPS never considers the ordering between the higher-priority MDDs in a node, and would thus generate an identical subtree under each of these nodes.

This process can also be viewed as merging two total orderings into one partial ordering which adequately represents both. Note that we can only say these two total orderings are equivalent *subject to* their respective constraints that were applied to resolve conflicts, as those created the MDDs which we use to check for equivalency. Formally, $n.M = n'.M \implies (n \equiv n')$

As an example, consider a case where the root node is expanded, and among its child nodes exists a node n_1 where a_1 has the highest priority, and a node n_2 where a_2 has the highest priority. The MDDs that they were each assigned just so happen to be disjoint (perhaps because the agents are very distant from each other). In this case, if we expand n_1 by adding a_2, or we expand n_2 by adding a_1, we will have the same MDDs, just ordered differently. All agents added later will conflict with or avoid these MDDs with no regard to the internal ordering between a_1 and a_2, so these are duplicate nodes. In this case, a potentially large subtree (either the child of n_1 or the child of n_2) can be pruned.

More involved cases may occur when MDDs conflict. For example, we might arrive at the ordering prefix $a_1 \prec a_2$, and see that the MDDs of a_1 and a_2 conflict. We will then split the node and add constraints to resolve their conflicts. Then, once the conflicts are resolved, a_3, which happens to be independent of a_1 and a_2, will be added without any conflicts. Let us call this node n. In another branch of the tree, a_3 will be given the highest priority, then a_1 will be added next without any conflicts, and then a_2 will be added, conflicting with a_1. Here too, we will split to resolve the conflicts between the MDDs of a_1 and a_2. The conflicts will be resolved in the same manner that led to node n, since we assume for the purpose of this example that a_3 is completely independent of a_1 and a_2. Thus, we will reach a node n', which contains the same set of MDDs as node n, and we will prune it.

We note that such duplicates do not happen often in dense problems, but can happen in sparse environments.

5 Studying PP with Random Restarts

Recall that in basic PP a single priority ordering $\mathcal{P}$ is used, as well as a single path-function $\mathcal{F}$. In many practical implementations, PP is executed multiple times. A popular implementation fixes the path-function $\mathcal{F}$, and PP is run multiple times with different priority orders $\mathcal{P}$. This is known as *PP with Random Restarts* (PP-RR) (Bennewitz et al. 2002). In this section, we characterise the space of MAPF problems PP-RR can solve. We also characterise the space of MAPF problems that can be solved with the less-studied complementing algorithm, in which the priority order is fixed and the path function is varied.

Figure 6 illustrates the relationship between these solution spaces. The outside white area is the solution-space of a given MAPF problem instance. The blue and yellow inner circles will be explained below. The union of these

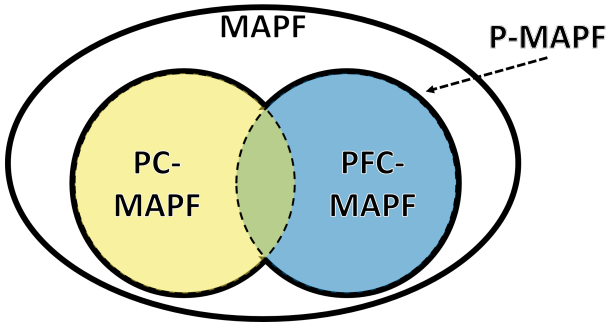

Fig. 6. Relations of the solution spaces MAPF, P-MAPF, PC-MAPF, and PFC-MAPF.

circles is the P-MAPF solution space. The intersection of the two inner circles (in green) is the solution returned by a single PP algorithm defined by a single pair of $\langle \mathcal{P}, \mathcal{F} \rangle$.

5.1 Path-Function Constrained MAPF

The Path-Function Constrained MAPF (PFC-MAPF) problem (Figure 6, blue circle) is defined by a tuple $\langle G, k, s, t, \mathcal{F} \rangle$ where G is a graph; k is the number of agents; s and t are the source and target vectors, mapping each agent to its initial and final location in G.

We call a solution π *path-function constrained* w.r.t. a path-function $\mathcal{F}$ if all paths match the mapping in $\mathcal{F}$; i.e., we could recreate π by ordering the agents in some order $\mathcal{P}$, and then iterate over the agents according to that order, deriving a path π_i for an agent a_i using $\mathcal{F}$ and adding that path (as constraints) to the input of $\mathcal{F}$ for all subsequent agents. Formally:

$$\exists \mathcal{P} \, \forall \, \pi_i \in \pi \mid \pi_i = \mathcal{F}(s_i, t_i, \{\pi_j \in \pi | i \prec j \in \mathcal{P}\})$$

A solution π to the PFC-MAPF problem is a *prioritised* MAPF solution that is *path-function constrained* w.r.t. the given function $\mathcal{F}$.

An algorithm is called *path-function-complete* if, given a PFC-MAPF problem, it guarantees to find a path-function-constrained solution if one exists or return *no solution* if no such solution exists.

Theorem 6. *The decision problem of whether a PFC-MAPF solution exists is* NP*-hard on directed graphs.*

The proof for Theorem 6 is in appendix A.3.

A solution to a PFC-MAPF problem is called *path-function-optimal* if it has the smallest cost among all path-function-constrained solutions.

Theorem 7. *The optimisation problem of PFC-MAPF is* NP*-hard on undirected graphs.*

The proof for Theorem 7 is in appendix A.4.

Observation 1. *The optimisation problem of PFC-MAPF is* NP-hard *on directed graphs.*

As the solution space of PFC-MAPF is a subset of the solution space of P-MAPF, it naturally follows from Theorem 1 that the solutions to PFC-MAPF problems are equivalent to the solutions that may be computed by PP with a given path-function $\mathcal{F}$ to the equivalent MAPF problems.

5.2 Priority Constrained MAPF

The Priority Constrained MAPF[3] (PC-MAPF) problem (Figure 6, yellow circle) is defined by a tuple $\langle G, k, s, t, \mathcal{P} \rangle$ where G is a graph; k is the number of agents; s and t are the source and target vectors, mapping each agent to its initial and final location in G; $\mathcal{P}$ is a total order (priority) over the set of agents. Note that here, unlike in PFC-MAPF, the path-function is not given in advance.

A solution π to the PC-MAPF problem is a set of paths, one for each agent, that is also *priority-constrained* w.r.t. to $\mathcal{P}$ (which was given as input). A solution to a PC-MAPF problem is called *priority-optimal* w.r.t. $\mathcal{P}$ if it has the smallest cost among all priority-constrained solutions w.r.t. $\mathcal{P}$. An algorithm is called *priority-complete* w.r.t $\mathcal{P}$ if, given a PC-MAPF problem, it guarantees to find a priority-constrained solution if one exists or return *no solution* if no such solution exists.

As the solution space of PC-MAPF is a subset of the solution space of P-MAPF, it naturally follows from Theorem 1 that the solutions to PC-MAPF problems are equivalent to the solutions that may be computed by PP with a given priority ordering $\mathcal{P}$ to the equivalent MAPF problems.

Theorem 8. *The decision problem of whether a priority-constrained solution exists is* NP*-hard on undirected graphs.*

In appendix A.5 we prove theorem 8, stating that deciding whether a solution to a PC-MAPF problem instance exists is NP-hard. This puts PC-MAPF and MAPF in the same complexity class. Moreover, PC-MAPF is potentially harder than MAPF on undirected graphs, as the decision problem for those is polynomial for MAPF (Daniel Kornhauser 1984).

Observation 2. *The decision problem of PC-MAPF in directed graphs and the corresponding optimisation problems both in directed and undirected graphs are* NP-hard.

5.3 Finding Constrained MAPF Solutions

To find optimal solutions to either PC-MAPF or PFC-MAPF, we propose Priority Constrained Search (PCS) and Path-function constrained Search (PFCS), which are adaptations of PaPS to solve PC-MAPF and PFC-MAPF, respectively. PCS receives as input a PC-MAPF problem $\langle G, k, s, t, \mathcal{P} \rangle$ and returns a priority-optimal solution for it. PCS operates identically to PaPS, with one alteration – when splitting on agents, PCS only generates a single child node, where the added agent is the next agent according to $\mathcal{P}$. PFCS receives as input a PFC-MAPF problem $\langle G, k, s, t, \mathcal{F} \rangle$ and returns a path-function-optimal solution for it. PFCS operates identically to PaPS, with one alteration – the low-level search generates a single optimal path using $\mathcal{F}$, instead of an MDD of all optimal paths. A single path is a special case of an MDD, so this path can be handled by the rest of the algorithm in the same manner any arbitrary MDD is handled. The same heuristic functions used in PaPS may also be used in PCS and PFCS.

Observe that PCS and PFCS search the same search space as PaPS, but restrict it to eliminate solutions *iff* they are not valid given their problem inputs (priority ordering, or path-function, respectively). Thus, it is clear to see that given that PaPS is P-MAPF complete and optimal, so too are PCS and PFCS priority complete and optimal, and path-function complete and optimal, respectively.

In Figure 5, we focused on a subtree of the PT restricted to a single ordering. Therefore, that example is also as an example of PCS on the same problem, when given the ordering $\mathcal{P} = \{a_1 \prec a_2 \prec a_3\}$.

Solving PFC-MAPF suboptimally can be done using PP-RR (which uses a fixed path-function $\mathcal{F}$ but randomises the priority ordering $\mathcal{P}$ of agents every iteration). Clearly, PP-RR cannot be used to solve PC-MAPF suboptimally, since the priority order is fixed. As an alternative, we fixed the priority ordering $\mathcal{P}$ but randomised $\mathcal{F}$ in order to choose between different paths with equal costs. We call this simple and scalable algorithm *Prioritised Planning*

[3] Our preliminary work on this problem was published in the Proceedings of the 17th International Symposium on Combinatorial Search (2024).

Table 1. Summary of how the algorithms relate to the choice of ordering and path-function. Note that the left column relates to the PFC-MAPF problem, the top row relates to the PC-MAPF problem while the entire table relates to P-MAPF

		Path-Function			
		Fixed	**Sub-Optimal**	**Optimal**	
Ordering	**Fixed**	PP	PPR*	PCS	**PC-MAPF**
	Sub-Optimal	PP-RR		-	
	Optimal	PFCS	-	PaPS, EPP	
		PFC-MAPF			**P-MAPF**

with Randomised A* (PPR*). The main advantage of PPR* is that given $\mathcal{P}$, PPR* repeatedly samples from the space of possible priority-constrained solutions, while PP considers only one. Both PP and PPR* are incomplete, and neither provides any guarantees on the quality of returned solutions.

6 Experimental Results

Table 1 summarises how the different algorithms in this paper relate to the choice of ordering $\mathcal{P}$ and path-function $\mathcal{F}$. "Fixed" means $\mathcal{P}$ and/or $\mathcal{F}$ are given as input. "Sub-Optimal" refers to algorithms that search for a good $\mathcal{P}$ or $\mathcal{F}$ and return the best solution they discover within the allotted planning time, providing no guarantees on its quality. "Optimal" relates to algorithms that systematically find the best $\mathcal{P}$ and/or $\mathcal{F}$. The combination of fixed $\mathcal{P}$ and fixed $\mathcal{F}$ is a single execution of the basic PP algorithm. We also marked which algorithm maps to which of the problems we defined in this paper, using the same colouring as in Figure 6. We address the empty brackets of this table in the discussion (section 7).

We implemented our optimal algorithms: PaPS, PCS, and PFCS with H1 and H2, as well as our suboptimal algorithms: PPR*, PP-RR, and PP. Our code and data are available online at: www.github.com/J-morag/MAPF/releases/tag/25.JAIR.PP. The first iteration for PPR* and PP-RR was made to run identically to PP. We also created an algorithm that randomises both the ordering (like PP-RR) and the path-function (like PPR*), but found that its performance in practice was very similar to that of PP-RR. We experimented using a grid-based MAPF benchmark from Stern et al. (2019). For each map, we used 25 different scenarios, each with 8 MAPF problem instances. In each instance, we varied the number of agents from 5 to 40, increasing by 5. We converted the MAPF instances into P-MAPF instances for PaPS, into PFC-MAPF instances for PFCS and PP-RR, and into PC-MAPF instances for PCS and PPR*. For PC-MAPF instances, we used the order that agents appear in the benchmark as the instance's priority ordering. For PFC-MAPF instances, we implemented path-functions as follows: We assign A* nodes with identifier numbers, and when deciding which of a set of equal nodes (same $f()$) to expand next, we choose the one with the minimal identifier. We derive these numbers from a deterministic pseudo-random number generator. This number generator is reset every time A* is used, and resetting it with a different seed produces a different path-function. We present results on a diverse set of maps from the benchmark. We ran our experiments on Linux virtual environments, on a cluster of AMD EPYC 7763 CPUs, with 16GB RAM each. We used Open Java Runtime 18 (JRE 18) to run our code.

6.1 Coverage and Runtime

Figure 7 (additional results in appendix B) shows the runtime of each algorithm (y-axis) as a factor of the number of successes (x-axis), where for each algorithm (separately) we ordered the instances in increasing order of their runtime. Thus, when examining a specific point on an algorithm's line, we can say "the algorithm would solve this many instances if its runtime were limited to this many seconds". We count a success if either the algorithm found a solution, or it proved that the problem is unsolvable. Naturally, proving an instance is unsolvable only

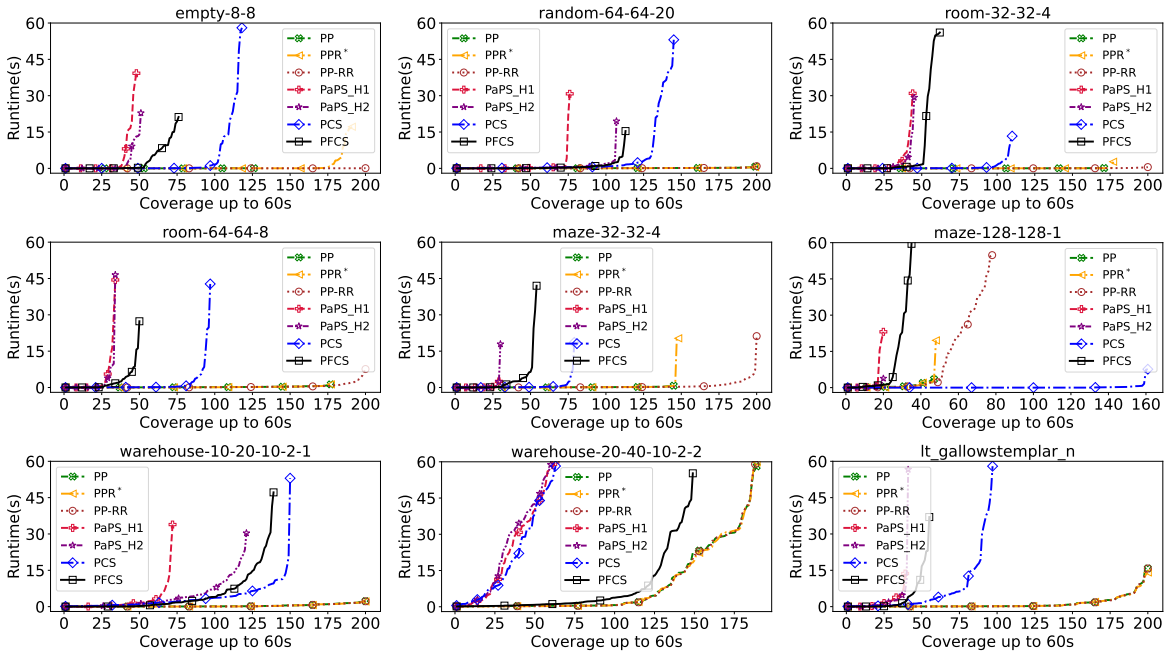

Fig. 7. Max runtime per instance, as a factor of coverage.

applies to the complete algorithms, and each of those considers a different problem as unsolvable – P-MAPF problems for PaPS, PC-MAPF problems for PCS, and PFC-MAPF problems for PFCS. For PPR* and PP-RR, the runtime until the first solution was found is used on the y-axis. This is because they always use all the allowed runtime to attempt to improve the solution.

Comparing PaPS with the two proposed heuristics, PaPS-H2 often solves more problems in less time (The H2 curve is below the H1 curve). For example, in map warehouse-10-20-10-2-1, PaPS-H2 achieves 121 successes, whereas PaPS-H1 only succeeded on 72 problems. For PCS and PFCS we always used H2. In most maps, the difference was smaller, but it was never better to use H1. Therefore, we only use H2 below.

We next compare the different suboptimal algorithms. As could be expected, PP, PP-RR, and PPR* take the same amount of runtime to find the first solution. Consequently, their lines mostly overlap in the plots (at the bottom). However, in cases where PP fails to find a solution, PPR* and PP-RR continue to run and often eventually find a solution. For example, in maze-32-32-4, PP solved 126 instances, PPR* solved 191 instances, and PP-RR solved 200 instances.

Clearly, algorithms with stronger guarantees had fewer successes and required more time. PaPS had the least successes, followed by PFCS and PCS, with the suboptimal algorithms achieving significantly more successes. In maps where many of the instances were unsolvable (with the given ordering), PCS was able to succeed much more often. For example, this happened in maze-128-128-1, which is a maze map comprised of many corridors that are one vertex wide. The reason is that in 66% of this map's instances, PCS was able to exhaust the search space and prove that those instances were unsolvable.

Table 2 shows how many of the PC-MAPF instances were proved unsolvable by PCS. Note that it is possible that more instances are unsolvable, but PCS was unable to prove so within the runtime limit. In all other maps, PCS did not prove any instances were unsolvable. Therefore, PCS had partial success in proving unsolvability. By

Table 2. Number of PC-MAPF instances proven by PCS to be unsolvable with the default priority ordering.

empty-8-8	maze-128-128-1	maze-128-128-2	maze-32-32-2	maze-32-32-4	room-32-32-4	room-64-64-8
3 (2%)	132 (66%)	3 (2%)	18 (9%)	22 (11%)	15 (8%)	23 (12%)

contrast, PFCS and PaPS are also theoretically capable of proving instances are unsolvable under their respective problem definitions, but neither was able to do so within the runtime limit in our experiments. Thus, they had no success in proving unsolvability. In the worst case, PFCS and PaPS need to go over $K!$ priority orderings in order to prove unsolvability, and the time limit did not allow that.

6.2 Solution Costs

We compare the solution costs of PaPS, PCS, PFCS, PP, PPR* and PP-RR in Figure 8 (additional results in appendix B). For heuristics, we use H2, as the previous experiment suggests it is never detrimental to use it. The x-axis is the *best known lower-bound* on the cost of each **MAPF** problem, extracted from an online tracker (Shen et al. 2023). This is usually the optimal MAPF cost. The MAPF bound is also a lower bound on the cost of all our other problems. Among different problems on the same map, a larger bound generally implies more agents, longer paths, and more interference between agents, and thus implies a more difficult problem. The y-axis shows the ratio between the SOC of a solution and the best-known bound on the cost (as given by the tracker). If an algorithm failed to solve an instance, it simply has no marker that corresponds to that instance.

To make the figures easier to read, we merge every 10 solutions of each algorithm, by showing a single marker corresponding to their average SOC and average best bound. In comparing PP and PaPS, it is clear to see that PP often finds solutions with a significantly higher cost. The trend was a clear increase in the ratio of the costs of their solutions as the number of agents increased. For example, in empty-8-8, on instances solved by both PaPS and PP, the average ratio of their costs was 1.04 with 5 or 10 agents, 1.07 with 15 agents, and 1.13 with 20 agents. PaPS usually found solutions with costs equal to, or very close to, those of optimal solutions to the equivalent MAPF problems. Of course, these solutions are all optimal P-MAPF solutions. This demonstrates that PP has the potential to find optimal MAPF solutions, even though in its basic form of a single run, it often does not. Nevertheless, our experiment is limited by the limited scaling of PaPS. We shall revisit this point in section 6.3.

Both PPR* and PP-RR provided a balance of solution quality and scalability. PPR* was able to find solutions with lower SOC than those found by PP, though often not as low as those found by PCS. Similarly, PP-RR was able to find solutions with significantly lower SOC than those found by PP, though often not as low as those found by PFCS. PP-RR consistently found better solutions than PPR*, but the differences were limited. For example, in room-64-64-8, PPR* solutions had on average 2.3% higher SOC. These results imply that the choice of ordering is more important than the choice of path-function for finding high-quality (low cost) solutions. However, it seems the gap between PPR* and PP-RR, given the same amount of runtime, is usually not large.

6.3 P-MAPF-Optimal Approximation for Numerous Agents

Our results showed that PP often produces poor quality solutions, or fails to find them altogether. We saw that choosing better individual paths, either through sampling with PPR*, or through optimal search using PCS, can greatly improve the quality of solutions, even without varying the prioritisation. We further saw that the commonly used PP-RR can find even higher quality solutions given the same amount of runtime, and that optimal PFC-MAPF solutions found by PFCS were close in cost to optimal MAPF solutions. This raises the question – how much could we improve PP-RR by always using the optimal path-function for each ordering? Answering this question is difficult in practice because it is difficult to find the exact optimal path-function. When the number of

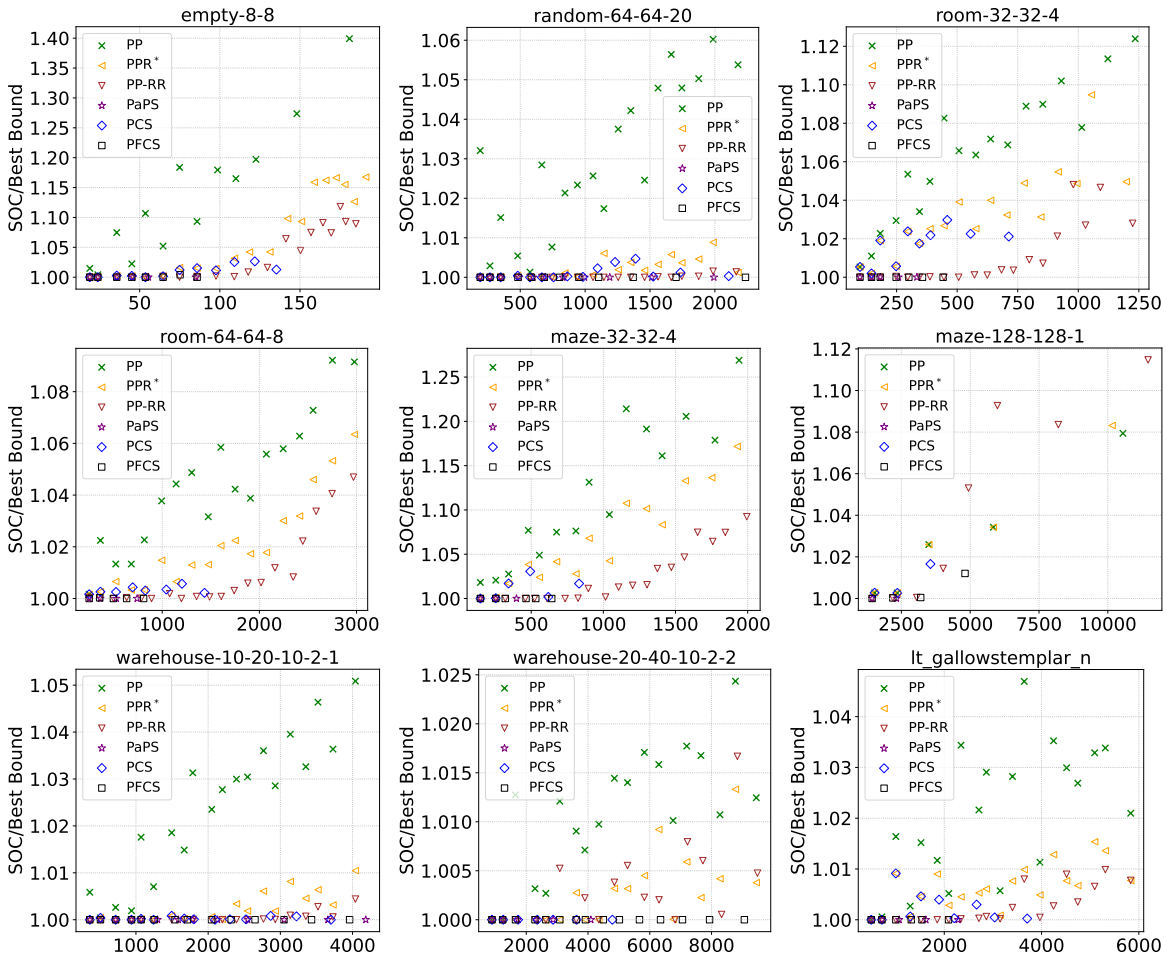

Fig. 8. The cost of solutions, relative to the best known lower bound on the cost of optimal solutions to the equivalent MAPF problems.

agents is low, most algorithms perform well enough, and the differences in solution quality are small. The more interesting results will be those of higher numbers of agents. However, our relevant optimal solvers, PCS and PaPS, only scale to about 10 to 40 agents. To bridge this gap, we propose an approximation of optimal path choice. We run PP-RR with a budget of 150 orderings and no time limit, and we attempt to solve each ordering 50 times, using different random path-functions. We shall call this algorithm PP-RR-50. We compare the solutions found in this manner with those found by PP-RR (always using the same path-function, attempting each ordering just once). We only include instances where both solvers had found a solution starting from the first iteration, and we only use instances that were solved starting from the first ordering. We use the highest number of agents available in the benchmark and solved successfully, up to 100 agents.

We plot the results of this experiment on the left column of Figure 9, where the horizontal axis is the number of attempted orderings, and the vertical axis is the mean ratio between the SOC of a solution and the best-known bound on the cost. The shading around each line is the standard error of the mean. We only include instances

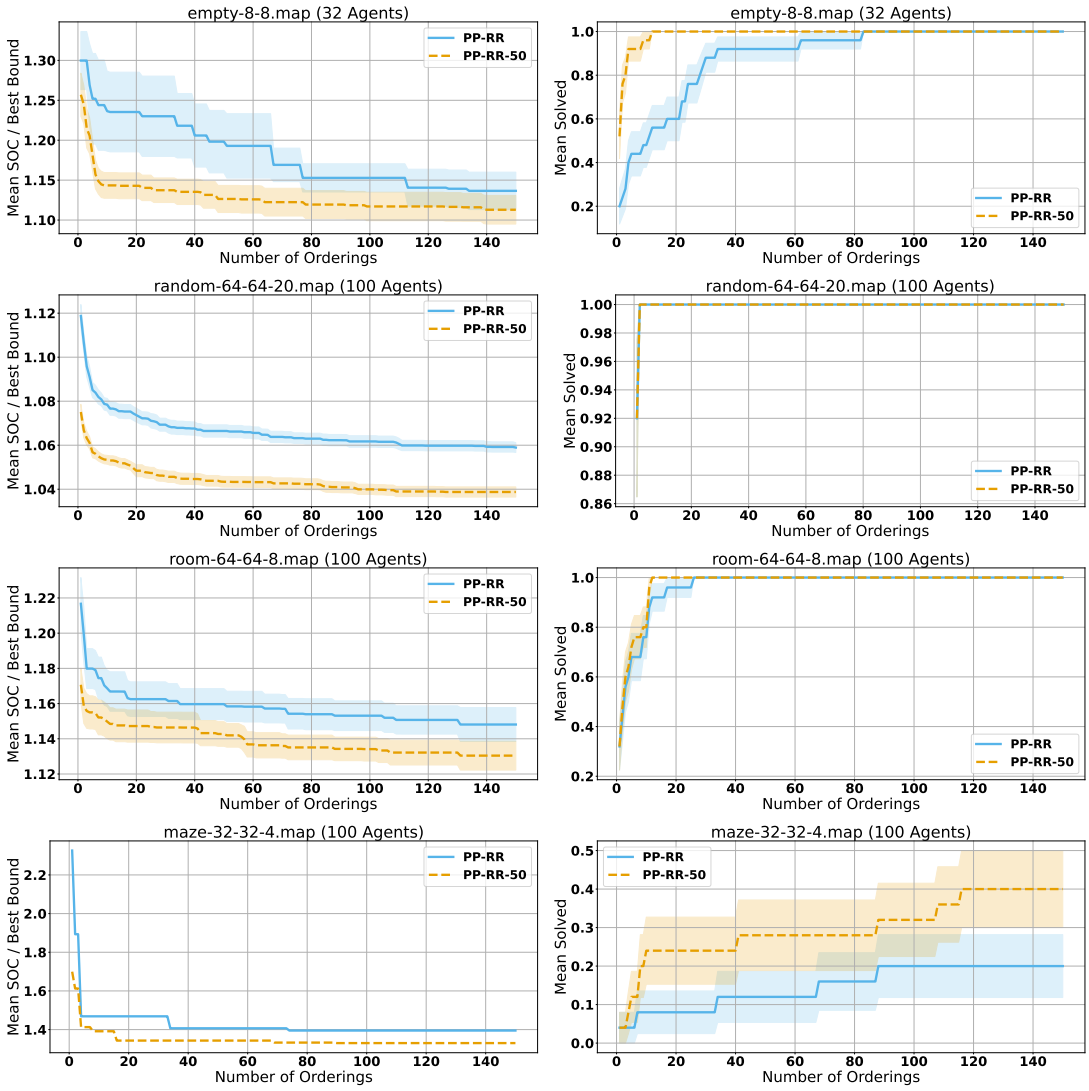

Fig. 9. Suboptimality of solutions and Success Rate of PP-RR with 1 or 50 path-functions, as a factor of the number of orderings attempted.

solved by both algorithms since the first ordering. The results show that the cost graph approaches a plateau as we try more and more orderings, as each additional ordering attempted is less likely to find a solution better than the current best. This is reasonable – if we assume attempting a random ordering samples uniformly from the set of all PFC-MAPF solutions to the problem, then the more we sample, the less likely we are to find a solution that is better than all previous samplings. Of course, this process can potentially be shortened by employing heuristic methods of guessing which orderings would produce good solutions. However, this is not our focus. Instead, we see here that this process can also be shortened through finding better path-functions for the same orderings.

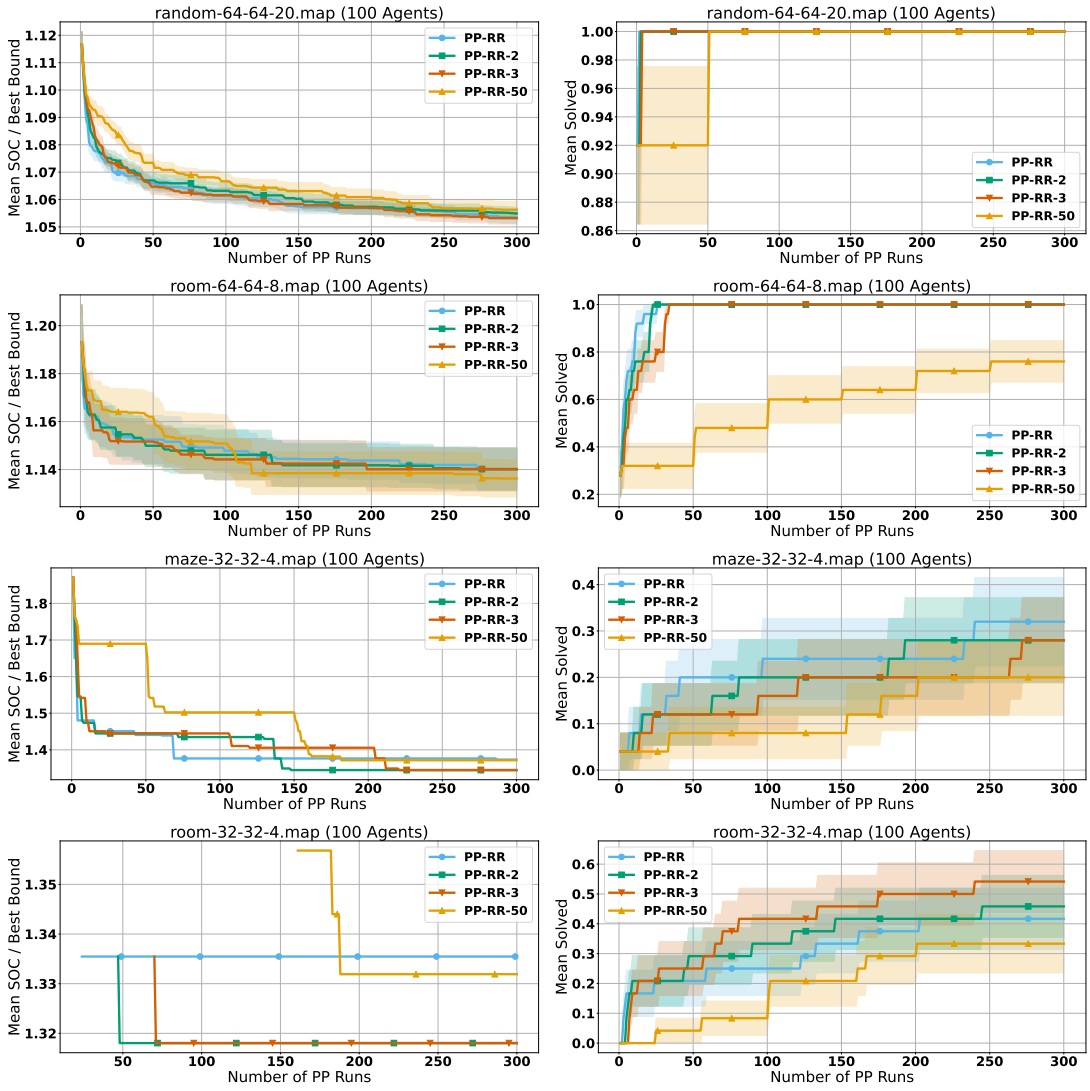

Fig. 10. Suboptimality of solutions and Success Rate of PP-RR with 1 or 50 path-functions, as a factor of the number of PP runs (pairs of ordering and path-function).

Indeed, we see that PP-RR-50 consistently found better solutions. This held true not only for the first orderings attempted, where improvements are easy, but even after more than 100 orderings.

The right column of Figure 9 shows the success rate (the number of solved instances divided by the total number of instances) as a factor of the number of orderings attempted. The shaded area is the standard error. Here we see similar trends to those we saw with solution costs, though more subtle. By using 50 path-functions per ordering, we were able to find solutions earlier in terms of the number of orderings attempted.

These results should not be interpreted to mean that trying 50 path-functions (using PP-RR-50) is a better use of runtime than trying new orderings. In fact, it was more often, though not always, more beneficial to attempt a new random ordering rather than re-try the same ordering with a new random path-function. Figure 10 shows an effort-based comparison of the benefits of trying random orderings versus random path-functions. It is organised in the same manner as Figure 9, except that the x-axis is now the number of PP runs (pairs of ordering and path-function) attempted. We gave each solver a budget of 300 runs, which translates into different numbers of orderings and path-functions, depending on the solver. PP-RR-50 is an extreme example of preferring to explore path-functions over orderings, so we add two solvers that represent more reasonable middle grounds – PP-RR-2 and PP-RR-3, which try 2 or 3 path-functions respectively, for each ordering. For both solution cost and coverage, PP-RR-50 performed worse in most cases, though in some examples, like room-64-64-8 with more than 116 runs, it had the best cost. The other three solvers performed similarly to each other in terms of solution cost, with no clear winner. In terms of coverage, trying fewer path-functions (and more orderings instead) was sometimes advantageous, though differences were quite small. For example, on maze-32-32-4 with 110 runs, PP-RR-3 had 16% coverage, PP-RR-2 had 20% coverage, and PP-RR had 24% coverage. One notable counter-example is maze-32-32-4, where PP-RR-2 had better coverage than PP-RR, and PP-RR-3 had the best coverage, indicating there are some conditions under which finding a good path-function can be more important for coverage than the choice of ordering. None of the instances on this map was solved by any algorithm on the first run, so we only display the solution cost results of instance number 1.

These results motivate the need to find a good path-function given an ordering, or find both in conjunction. This is a wide topic that is left for future work. Possible approaches include heuristic methods and machine learning.

7 Discussion and Future Work

Prioritised Planning (PP) is a widely popular method to simplify and tackle *Multi-Agent Path Finding* (MAPF). In this work, we studied PP. To this end, we defined the concept of a *PP algorithm*, as a pair $\langle \mathcal{P}, \mathcal{F} \rangle$. $\mathcal{P}$, referred to as the *priority order*, is a total order over the set of agents. $\mathcal{F}$, referred to as the *path-function*, is a function that accepts as input a pair of vertices $v_s, v_t \in V$ and a set of constraints, and outputs a shortest path from v_s to v_t that avoids the given constraints. The path-function is a concept new to this work. We described a hierarchy of the problems solved by PP. This hierarchy starts with Prioritised MAPF (P-MAPF), which includes all solutions to a MAPF problem that could be found by PP. It is then further divided into two problems. The first problem is Path-Function Constrained MAPF (PFC-MAPF), which includes all P-MAPF solutions that choose individually optimal paths using a given path-function. PFC-MAPF is the problem that most textbook implementations of PP solve, as most prior works only dealt with modifying $\mathcal{P}$. The second problem is Priority Constrained MAPF (PC-MAPF), which includes all P-MAPF solutions that are constrained to a specific priority ordering. Solutions to PC-MAPF are required by important industrial applications.

We gave a first analysis characterising the complexity of the problems we formalised. Table 3 summarises these results. We showed that P-MAPF, PC-MAPF, and PFC-MAPF are all generally in NP-hard. This result is somewhat unintuitive, as PP algorithms are usually used to make the difficult MAPF problem easier to solve, sacrificing completeness and solution quality for low runtime. One may assume that this speed-up is achieved precisely because PP algorithms limit the space of solutions they can find to the space of solutions to P-MAPF problems, potentially missing some MAPF solutions. Yet, this is evidently not the case, as we have shown P-MAPF is at least as hard as MAPF. At this time, the complexity of the problems of finding feasible P-MAPF and PFC-MAPF solutions on undirected graphs remains an open question. However, we proved that the more general problem of finding feasible (P-MAPF or PFC-MAPF) solutions on directed graphs is in NP-hard. We found that surprisingly,

Table 3. Summary of complexity results for the problems associated with PP. Each problem formulation is divided into the problems of finding feasible or optimal solutions on directed or undirected graphs. Known MAPF results are included for reference.

	P-MAPF		MAPF	
	Directed	Undirected	Directed	Undirected
Feasible	NP-hard (Theorem 2)	?	NP-hard (Nebel 2020)	Poynomial (Daniel Kornhauser 1984)
Optimal	NP-hard (Theorem 3/2)	NP-hard (Theorem 3)	NP-hard (Yu and LaValle 2013)	NP-hard (Yu and LaValle 2013)

	PC-MAPF		PFC-MAPF	
	Directed	Undirected	Directed	Undirected
Feasible	NP-hard (Theorem 8)	NP-hard (Theorem 8)	NP-hard (Theorem 6)	?
Optimal	NP-hard (Theorem 8)	NP-hard (Theorem 8)	NP-hard (Theorem 7/6)	NP-hard (Theorem 7)

PC-MAPF is potentially a harder problem than MAPF, as finding feasible solutions on undirected graphs is in NP-hard for PC-MAPF, whereas it can be done in polynomial time for MAPF.

These three problems can be solved using a broad family of PP algorithms. Unfortunately, PP algorithms suffer two notable drawbacks: 1. The quality of the solutions they return is unbounded sub-optimal, and it is not clear if solutions can be further improved. 2. PP can produce deadlock failures, leaving practitioners without explanation or recourse.

We developed algorithms to solve these problems:

- PaPS, which is complete and optimal for P-MAPF.
- PFCS, which is complete and optimal for PFC-MAPF.
- PCS, which is complete and optimal for PC-MAPF.
- PPR*, which solves PC-MAPF sub-optimally, but quickly.

We evaluated our algorithms against existing baselines on a range of common MAPF problems, and showed that PaPS, PCS, and PFCS can succeed where PP fails. We gave tighter bounds for the quality of P-MAPF, PC-MAPF, and PFC-MAPF solutions, and optimally solved many associated problem instances for the first time. Together, these results give guidance to practitioners using PP in real applications. When PP fails, PaPS can help explain the reasons for that failure, and when PP succeeds, PaPS can be used to improve solution quality.

In this work, we focused on optimal algorithms, creating PaPS, PFCS, and PCS. We also investigated some options for suboptimal algorithms, creating PPR*, and comparing it with PP-RR. Most existing related algorithms, and indeed many MAPF algorithms, fall into the category of fixed $\mathcal{F}$ with sub-optimal $\mathcal{P}$. This leaves open the question of how algorithms that are sub-optimal in regard to both $\mathcal{F}$ and $\mathcal{P}$ would behave. Such an algorithm would map to the empty middle bracket in Table 1. We created a naive implementation of such an algorithm by taking PP-RR and randomising the $\mathcal{F}$ every time one is needed to generate a path, as in PPR*. In practice, this algorithm behaved similarly to PP-RR, so we did not include it in our results. In another experiment, we approximated how this algorithm would have behaved if, instead of choosing $\mathcal{F}$ randomly, it had chosen $\mathcal{F}$ to complement the current $\mathcal{P}$. We made this approximation using a computationally expensive, brute-force approach. Our experiment showed that this can significantly improve the performance of PP-RR in both success rate and

solution quality, for the same number of attempted $\mathcal{P}$s. We believe finding a computationally efficient function to approximate a near-optimal $\mathcal{F}$ choice could be a promising avenue for future work. Naturally, other approaches besides PP-RR could also be applied to create algorithms in this bracket of sub-optimal $\mathcal{F}$ with sub-optimal $\mathcal{P}$. The remaining brackets in Table 1, corresponding to the combinations of sub-optimal $\mathcal{F}$ with optimal $\mathcal{P}$ or of optimal $\mathcal{F}$ with sub-optimal $\mathcal{P}$, are hard to define, and their usefulness is questionable.

Other directions for future work could be to consider applying speed-up techniques from the CBS family of algorithms to PaPS, PCS, and PFCS. Additionally, we believe that relaxing the termination criteria of these algorithms can lead to new types of PP algorithms with different tradeoffs and guarantees. Another interesting use for PaPS would be to use it to generate training data for machine learning algorithms that generate priority orderings (Zhang et al. 2022), as those rely on learning from many examples of priority orderings that produce high-quality solutions.

8 Acknowledgements

This work was supported by the Israel Science Foundation (ISF) grant #909/23 awarded to Ariel Felner, the United States-Israel Binational Science Foundation (BSF) grant #2021643 awarded to Ariel Felner, and by BSF grant #2018684 and ISF grant #1238/23 to Roni Stern. Part of this work was performed while Jonathan Morag was on a research visit to Monash University. Research at Monash is partially funded by The Australian Research Council under grant DP200100025 and by a gift from Amazon.

References

A. Andreychuk, K. Yakovlev, P. Surynek, D. Atzmon, and R. Stern. 2022. "Multi-agent pathfinding with continuous time." *Artificial Intelligence*, 305, 103662.

D. Atzmon, R. Stern, A. Felner, N. R. Sturtevant, and S. Koenig. 2020. "Probabilistic robust multi-agent path finding." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30, 29–37.

D. Atzmon, R. Stern, A. Felner, G. Wagner, R. Barták, and N.-F. Zhou. 2020. "Robust multi-agent path finding and executing." *Journal of Artificial Intelligence Research*, 67, 549–579.

D. Atzmon, Y. Zax, E. Kivity, L. Avitan, J. Morag, and A. Felner. 2020. "Generalizing multi-agent path finding for heterogeneous agents." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 11, 101–105.

R. Barták, J. í. Švancara, and M. Vlk. 2018. "A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs." In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, 748–756.

M. Bennewitz, W. Burgard, and S. Thrun. 2002. "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.

M. Čáp, P. Novák, A. Kleiner, and M. Seleckỳ. 2015. "Prioritized planning algorithms for trajectory coordination of multiple mobile robots." *IEEE transactions on automation science and engineering*, 12, 3, 835–849.

M. Čáp, J. Vokřínek, and A. Kleiner. Apr. 2015. *Complete Decentralized Method for On-Line Multi-Robot Trajectory Planning in Well-formed Infrastructures*. (Apr. 2015).

P. S. Daniel Kornhauser Gary Miller. 1984. "Coordinating pebble motion on graphs, the diameter of permutation groups, and applications." In: *FOCS*.

R. Dechter and J. Pearl. 1988. *The optimality of A**. Springer.

M. A. Erdmann and T. Lozano-Pérez. 1986. "On multiple moving objects." In: *Proceedings of the 1986 IEEE International Conference on Robotics and Automation, San Francisco, California, USA, April 7-10, 1986*. IEEE, 1419–1424. doi:10.1109/ROBOT.1986.1087401.

P. Godefroid. 1996. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer.

P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. "A formal basis for the heuristic determination of minimum cost paths." *IEEE transactions on Systems Science and Cybernetics*, 4, 2, 100–107.

W. Hönig, T. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, and S. Koenig. 2016. "Multi-agent path finding with kinematic constraints." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 26, 477–485.

S. Hu, D. D. Harabor, G. Gange, P. J. Stuckey, and N. R. Sturtevant. 2022. "Multi-agent path finding with temporal jump point search." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 32, 169–173.

J. Li, Z. Chen, D. Harabor, P. Stuckey, and S. Koenig. 2021. "Anytime multi-agent path finding via large neighborhood search." In: *IJCAI*.

J. Li, Z. Chen, D. Harabor, P. J. Stuckey, and S. Koenig. 2022. "MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search." In: *AAAI*.

J. Li, D. Harabor, P. J. Stuckey, A. Felner, H. Ma, and S. Koenig. 2019. "Disjoint splitting for multi-agent path finding with conflict-based search." In: *Proceedings of the international conference on automated planning and scheduling*. Vol. 29, 279–283.
J. Li, D. Harabor, P. J. Stuckey, H. Ma, and S. Koenig. 2019. "Symmetry-breaking constraints for grid-based multi-agent path finding." In: *Proceedings of the AAAI conference on artificial intelligence* 01. Vol. 33, 6087–6095.
J. Li, P. Surynek, A. Felner, H. Ma, T. S. Kumar, and S. Koenig. 2019. "Multi-agent path finding for large agents." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33, 7627–7634.
J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. K. S. Kumar, and S. Koenig. May 2021. *Lifelong Multi-Agent Path Finding in Large-Scale Warehouses*. (May 2021).
H. Ma, D. Harabor, P. J. Stuckey, J. Li, and S. Koenig. 2019. "Searching with consistent prioritization for multi-agent path finding." In: *AAAI*. Vol. 33, 7643–7650.
H. Ma, T. S. Kumar, and S. Koenig. 2017. "Multi-agent path finding with delay probabilities." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31.
H. Ma, J. Li, T. Kumar, and S. Koenig. Jan. 2017. *Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks*. (Jan. 2017).
J. Morag, A. Felner, R. Stern, D. Atzmon, and E. Boyarski. 2022. "Online multi-agent path finding: New results." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 15, 229–233.
J. Morag, R. Stern, and A. Felner. 2023. "Adapting to planning failures in lifelong multi-agent path finding." In: *Proceedings of the International Symposium on Combinatorial Search* 1. Vol. 16, 47–55.
B. Nebel. 2020. "On the computational complexity of multi-agent pathfinding on directed graphs." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30, 212–216.
M. Phillips and M. Likhachev. 2011. "Sipp: Safe interval path planning for dynamic environments." In: *2011 IEEE international conference on robotics and automation*. IEEE, 5628–5635.
T. Shahar, S. Shekhar, D. Atzmon, A. Saffidine, B. Juba, and R. Stern. 2021. "Safe multi-agent pathfinding with time uncertainty." *Journal of Artificial Intelligence Research*, 70, 923–954.
G. Sharon, R. Stern, A. Felner, and N. Sturtevant. 2012. "Meta-agent conflict-based search for optimal multi-agent path finding." In: *Proceedings of the International Symposium on Combinatorial Search* 1. Vol. 3, 97–104.
G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. 2015. "Conflict-based search for optimal multi-agent pathfinding." *Artificial intelligence*, 219, 40–66.
G. Sharon, R. Stern, M. Goldenberg, and A. Felner. 2013. "The increasing cost tree search for optimal multi-agent pathfinding." *Artificial intelligence*, 195, 470–495.
B. Shen, Z. Chen, M. A. Cheema, D. D. Harabor, and P. J. Stuckey. 2023. *Tracking Progress in Multi-Agent Path Finding*. (2023). doi:10.48550/arXiv.2305.08446.
D. Silver. 2005. "Cooperative Pathfinding." In: *AIIDE*.
T. Standley. 2010. "Finding optimal solutions to cooperative pathfinding problems." In: *Proceedings of the AAAI conference on artificial intelligence* 1. Vol. 24, 173–178.
R. Stern et al.. 2019. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks." In: *SoCS*, 151–158.
J. Švancara, M. Vlk, R. Stern, D. Atzmon, and R. Barták. 2019. "Online multi-agent pathfinding." In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33, 7732–7739.
J. P. Van Den Berg and M. H. Overmars. 2005. "Prioritized motion planning for multiple robots." In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 430–435.
S. Varambally, J. Li, and S. Koenig. 2022. "Which MAPF Model Works Best for Automated Warehousing?" In: *SoCS*. https://www.autostoresystem.com/.
T. T. Walker. 2022. "Multi-Agent Pathfinding in Mixed Discrete-Continuous Time and Space." Ph.D. Dissertation. University of Denver.
J. Yu and S. LaValle. 2013. "Structure and intractability of optimal multi-robot path planning on graphs." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 27, 1443–1449.
S. Zhang, J. Li, T. Huang, S. Koenig, and B. Dilkina. 2022. "Learning a priority ordering for prioritized planning in multi-agent path finding." In: *Proceedings of the International Symposium on Combinatorial Search* 1. Vol. 15, 208–216.

A Complexity Proofs

In this section, we provide complexity proofs that were too long to include in the main body of the paper. We adapt ideas from the proofs made by Yu and LaValle (2013) and use the same terminology. Accordingly, x is used to refer to variable agents rather than time steps, while k refers to the cost (SOC) of a solution. The following proofs are based on a reduction from 3SAT. 3SAT is an NP-complete problem that receives a formula F and outputs whether or not the formula is satisfiable. The formula F is a conjunctive normal form (CNF) consisting of

n variables x_i and m clauses c_j such that each clause has exactly three literals. Each literal in the clauses can be a negated or unnegated form of a variable.

A.1 Proof of Theorem 2

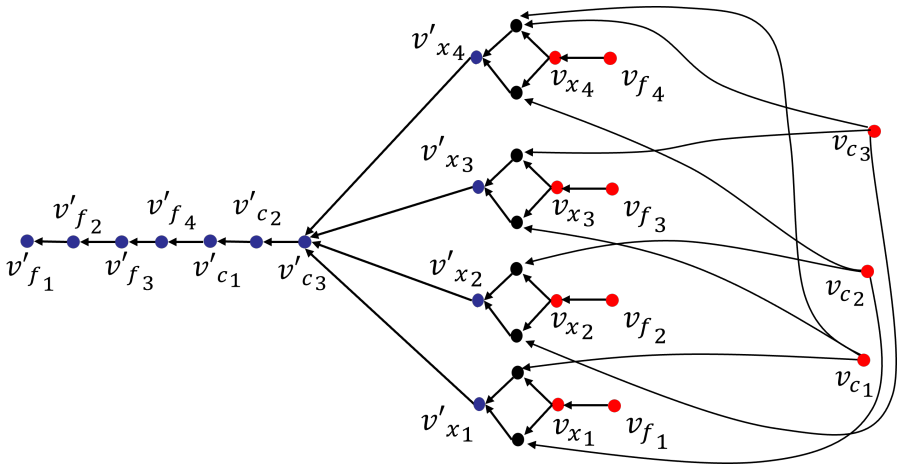

Fig. 11. Reduction from 3SAT to P-MAPF for the instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. The red vertices are the source vertices and the blue vertices are the targets.

Theorem 2. *The decision problem of whether a prioritised solution exists is* NP*-hard on directed graphs.*

Proof. We prove Theorem 2 by a reduction from the 3SAT problem. We create a corresponding P-MAPF problem with $2n + m$ agents on a directed graph G. The set of agents is:

$$A = \{x_1, \ldots, x_n, c_1, \ldots, c_m, f_1, \ldots, f_n\}$$

The x_i agents are called *variable agents*, the c_j agents are called *clause agents*, and the f_i's are called *filler agents*. We construct the graph similarly to Yu and LaValle (2013). The most significant difference is that we use a *directed* graph. First, we construct a simple gadget for each variable x_i called a *mouse gadget*. For x_i, we create a vertex v_{x_i} that represents its source vertex. We connect two vertices to the source vertex through directed edges, and then join these vertices with more directed edges to the target vertex v'_{x_i}. The agent can traverse through either the top or the bottom paths (vertices) to reach its target. Let these two paths be the i-th upper and lower paths. This section of the gadget is the body of the mouse.

Next, for each clause, we add a clause agent c_j whose source is vertex v_{c_j}. This vertex is connected through directed edges to three paths that represent the literals in the clause of c_j. If a literal is unnegated (resp., negated), then v_{c_j} is connected to the i-th upper (resp., lower) path in the gadget that belongs to v_{x_i}. The target vertices for the clause agents are created as follows: A path of length m is added by connecting a directed edge from vertex v'_{x_i} to the start of the new path. The path itself is comprised only of directed edges. The first vertex in the path is the target vertex of agent c_m (v'_{c_m}) followed by the target vertex of agent c_{m-1} ($v'_{c_{m-1}}$) and so on, until the end of the path where the target vertex of agent c_1 (v'_{c_1}) is located.

We add each filler agent f_i as follows. We create an additional edge (the tail of the mouse) from v_{f_i}, which is the source vertex of f_i, to v_{x_i}. The target vertex of f_i is v'_{f_i}. We add these target vertices by extending the path of

clause target vertices, connecting v'_{c_1} to v'_{f_m} with a directed edge, then v'_{f_m} to $v'_{f_{m-1}}$ and so on, until the end of the path where the target vertex of agent f_1 (v'_{f_1}) is located.

Figure 11 shows the complete graph for the P-MAPF problem constructed from 3SAT instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. A red vertex represents the source vertex of an agent, and a blue vertex is the target vertex.

The construction we defined limits the set of priority orderings over the agents for which a solution exists.

- The construction of the filler (f) and clause (c) targets forces the following in order for filler and clause agents not to block each other while sitting at their targets. The filler agents must be ordered according to their index. The clause agents must similarly be ordered according to their index. Additionally, all filler agents must precede all clause agents. Thus, $f_1 \prec f_2 \prec \ldots \prec f_n \prec c_1 \prec c_2 \prec \cdots \prec c_m$.
- The positioning of the variable (x) agents' targets forces each variable agent x_i to have lower priority than its corresponding filler agent f_i. Otherwise, the variable agent would enter its target too early and block the filler agent from advancing towards its target. Thus, $\forall_i f_i \prec x_i$.
- Each clause agent c_j must have higher priority than at least one variable agent x_i that appears in clause c_j. The specific prioritisation is part of the solution to the problem.

If the 3SAT instance is satisfiable, each variable x_i receives an assignment of truth value $\tilde{x_i}$. If $\tilde{x_i}$ is true (resp., false), then let agent x_i take the lower (resp., upper) path of the body of the mouse gadget. The path that was not chosen is free to transit for the clause and filler agents corresponding to the x_i variable. The filler agents will move through in order of their priority. The path-function will choose for the filler agents to take the upper or lower path according to the $\tilde{x_i}$ assignment (opposite to the path taken by the corresponding variable agent). The clause agents will similarly move in order of their own prioritisation, following the filler agents as soon as they clear a path. Since the assignment satisfies the 3SAT instance, we are guaranteed that each clause agent has at least one path free to transit. The variable agents will move a single step into their paths and wait there, as they are forced to avoid the paths of higher-priority agents. Finally, each variable agent will move into its own target as soon as the relevant filler and clause agents have moved past it. This solution respects a priority that obeys the restrictions listed above. Specifically, the ordering $f_1 \prec f_2 \prec \ldots \prec f_n \prec c_1 \prec c_2 \prec \cdots \prec c_m \prec x_1 \prec x_2 \prec \cdots \prec x_n$ will always be correct.

If P-MAPF has a solution, then each clause agent was assigned a path to its target. For each clause agent, there are up to three options for this path, each passing through some gadget corresponding to a variable that appears in the clause. Each variable agent x_i has two possible paths, and is forced to take the path opposite to that taken by its relevant clause and filler agents. If no clause agent passes through the variable agent's gadget, then this choice is determined solely by the filler agent and does not affect whether the assignment satisfies the 3SAT instance. Each filler agent similarly has two possible paths. By choosing a path, it forces the variable agent to advance rather than wait at its source vertex. If agent x_i uses the lower (resp. upper) path, let $\tilde{x_i} = true$ (resp. false). Since clause agents can pass through the upper path if they contain a variable x_i or through the lower path if they contain its negation $\neg x_i$, and each clause agent passed through exactly one such path, then for each clause, at least one variable resolves to *true*. Thus, the resulting assignment satisfies the 3SAT instance. □

A.2 Proof of Theorem 3

Theorem 3. *The optimisation problem of P-MAPF is* NP*-hard on undirected graphs.*

Proof. We prove Theorem 3 by closely following a reduction from the 3SAT problem proposed by Yu and LaValle (2013), and showing it can similarly be used for P-MAPF. To start the reduction, we create a P-MAPF problem corresponding to the 3SAT instance with $n + m$ agents on an undirected graph G. The set of agents is:

$$A = \{x_1, \ldots, x_n, c_1, \ldots, c_m\}$$

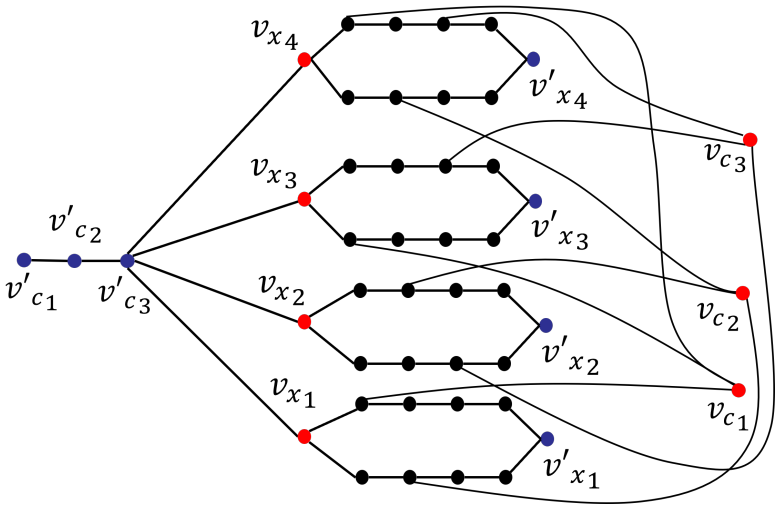

Fig. 12. Reduction from 3SAT to optimal MAPF, for the instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$ (Yu and LaValle 2013). The red vertices are the source vertices and the blue vertices are the targets. We apply this reduction to P-MAPF and PFC-MAPF.

The x_i agents are called *variable agents*, the c_j agents are called *clause agents*. For each variable x_i, we create a vertex v_{x_i} that represents its source vertex. We connect two paths of length $m+2$ to the source vertex, and join their ends such that the target is at the end vertex v'_{x_i}. The agent can traverse along either of the two paths to reach its target in $m+2$ steps. Let these two paths be the i-th upper and lower paths.

Next, for each clause, we add a clause agent c_j whose source is vertex v_{c_j}. This vertex is connected to three paths that represent the literals in the clause of c_j. If a literal is unnegated (resp., negated), then v_{c_j} is connected to the i-th upper (resp., lower) path at a vertex of distance j from v_{x_i}. The target vertices for the clause agents are created as follows: A path of length m is added, where one end of the path is connected to the source vertex v_{x_i}. The connected vertex is the target vertex of agent c_m (v'_{c_m}) followed by the target vertex of agent c_{m-1} ($v'_{c_{m-1}}$) and so on, until the end of the path where the target vertex of agent c_1 (v'_{c_1}) is located.

Figure 12 shows the complete graph for the P-MAPF problem constructed from 3SAT instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. A red vertex represents the source vertex of an agent, and a blue vertex is the target vertex.

To prove the difficulty of solving P-MAPF optimally, we must ask the question "does a solution with cost k exist for a given P-MAPF problem instance?". We set $k := (n+m)(m+2)$, meaning all agents reach their targets without performing any wait actions.

If the 3SAT instance is satisfiable, each variable x_i receives an assignment of truth value $\tilde{x}_i$. There exists a path-function $\mathcal{F}$ as follows. If $\tilde{x}_i$ is true (resp., false), then let agent x_i take the lower (resp., upper) path to its target. The path that was not chosen is left unused, or transited in the opposite direction by one or more of the clause agents corresponding to the x_i variable. This means that all variable agents and clause agents can start moving at time step zero and arrive at their targets at time step $m+2$. Thus, any ordering $\mathcal{P}$ will produce a solution given $\mathcal{F}$, as no agent takes a path that interferes with the preferred path of another agent. Thus, if the 3SAT instance is satisfiable, the P-MAPF instance has a solution with the optimal cost $(n+m)(m+2)$.

If the P-MAPF instance has a solution with cost $(n+m)(m+2)$, then each agent has a path that arrives at its target without waiting. This means each variable agent x_i must move exclusively through either its lower or

upper path, preventing any clause agent c_j from taking the same path in the opposite direction, but leaving the other path free for c_j to traverse. If agent x_i uses the lower (resp. upper) path, let $\tilde{x_i} = true$ (resp. false). Thus, the resulting assignment satisfies the 3SAT instance. □

A.3 Proof of Theorem 6

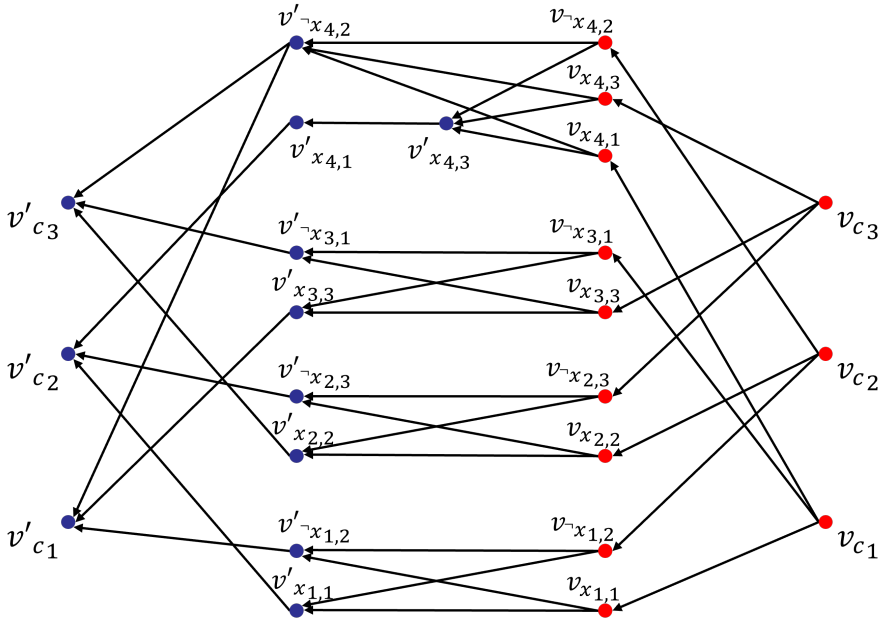

Fig. 13. Reduction from 3SAT to PFC-MAPF for the instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. The red vertices are the source vertices and the blue vertices are the targets.

Theorem 6. *The decision problem of whether a PFC-MAPF solution exists is* NP*-hard on directed graphs.*

Proof. We prove Theorem 6 by reduction from the 3SAT. To start the reduction, we create a PFC-MAPF problem corresponding to the 3SAT instance with $4m$ agents on a directed graph G. The set of agents is:

$$A = \{x_{i,j} \mid x_i \in c_j\} \cup \{\neg x_{i,j} \mid \neg x_i \in c_j\} \cup \{c_1, \ldots, c_m\}$$

The c_j agents are called *clause agents*. The $x_{i,j}$ agents are called *unnegated literal agents*. The i index corresponds to the identity of the variable in the 3SAT instance, and the j index corresponds to the clause where a literal of the x_i variable appears. The $\neg x_{i,j}$ agents are called *negated literal agents*, and are the same as the literal agents, except that they correspond to negated literals that appear in clauses. For variables that only appear in their unnegated (resp. negated) form, we only need to create the corresponding unnegated (resp. negated) literal agent.

For each clause, we add a clause agent c_j whose source is vertex v_{c_j}. For each unnegated literal agent $x_{i,j}$ (resp. negated literal agent $\neg x_{i,j}$), we create a vertex $v_{x_{i,j}}$ (resp. $v_{\neg x_{i,j}}$) that represents its source vertex. We connect each v_{c_j} to each of the $v_{x_{i,j}}$ (resp. $v_{\neg x_{i,j}}$) vertices corresponding to unnegated (resp. negated) literals in clause c_j. For each variable x_i, we create two sequences of vertices called the x_i sequence and the $\neg x_i$ sequence. The

number of vertices in each sequence corresponds to the number of unnegated (for the x_i sequence) or negated (for the $\neg x_i$ sequence) literals that exist for variable x_i. Each vertex in sequence x_i (resp. $\neg x_i$) is the target vertex $v'_{x_{i,j}}$ (resp. $v'_{\neg x_{i,j}}$) of an unnegated literal agent $x_{i,j}$ (resp. negated literal agent $\neg x_{i,j}$). These target vertices are sequenced such that the first corresponds to the literal that belongs to the clause with the maximal index j, and each subsequent target vertex corresponds to a literal that belongs to a clause with a smaller index j. The edges in the sequence are directed edges. We connect, using a directed edge, each unnegated (resp. negated) literal source vertex $v_{x_{i,j}}$ (resp. $v_{\neg x_{i,j}}$) to the first vertex in both sequence x_i and $\neg x_i$. For example, if there are 4 literals for variable x_i, $x_{i,1}$, $x_{i,3}$, $x_{i,4}$, and $\neg x_{i,1}$, then we will create two sequences of vertices. The sequence of target vertices of unnegated literals will be $v'_{x_{i,4}} \rightarrow v'_{x_{i,3}} \rightarrow v'_{x_{i,1}}$. The sequence of target vertices of negated literals will consist only of $v'_{\neg x_{i,1}}$. Lastly, we connect the end of each sequence x_i (resp. $\neg x_i$), again via a directed edge, to a target vertex v'_{c_j} for each clause agent c_j whose clause contains a literal whose target vertex is in $\neg x_i$ (resp. x_i). Note that sequences can be empty, in the event that some variable x_i only has unnegated literals or only has negated literals. In this case, we will connect the vertex $v_{x_{i,j}}$ (resp. $v_{\neg x_{i,j}}$) of the existing literal $x_{i,j}$ (resp. $\neg x_{i,j}$) directly to the clause target vertex v'_{c_j} that would have otherwise been connected to the end of sequence $\neg x_i$ (resp. x_i).

Next, we define the path-function $\mathcal{F}$: The construction guarantees that each literal agent will only ever have one shortest path that avoids constraints, so we do not need to define it further for literal agents. Clause agents prefer to traverse a sequence x_i if the complementing $\neg x_i$ sequence has constraints associated with it, and vice versa; i.e., if literal agent $x_{i,j}$ has already planned when clause agent c_j plans, then the $\neg x_i$ sequence is considered a preferred sequence. Naturally, if a clause agent c_j can reach a sequence that has constraints, then that sequence will not appear as part of a shortest path for c_j. If c_j can move through multiple sequences of the same length whose complementing sequences have constraints, then the sequence with the minimal index (i) among them is preferred. If c_j is not connected to any sequence whose complementing sequence has constraints, then it prefers to take an x_i (unnegated) sequence. If c_j can go through multiple such sequences, it will choose the one with the minimal index (i). The preference towards paths with lower i is made for completeness, and does not affect the construction.

Figure 13 shows the complete graph for the PFC-MAPF problem constructed from 3SAT instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. A red vertex represents the source vertex of an agent, and a blue vertex is the target vertex.

If the 3SAT instance is satisfiable, each variable x_i receives an assignment of truth value $\tilde{x_i}$. There exists a total order $\mathcal{P}$ over the agents as follows: All unnegated literal agents associated with a variable receiving a *true* assignment have the highest priority. All negated literal agents associated with a variable receiving a *false* assignment similarly have the highest priority. The prioritisation between these two groups is unimportant, but we shall give priority to the unnegated literal agents for convenience. Within each group, higher prioritisation is given to a literal agent with a smaller j (clause) index. Next in the priority order are all clause agents, and then all remaining literal agents. Within these remaining literal agents, higher prioritisation is given to a literal agent with a smaller j (clause) index. This describes a partial ordering, but any totalization of this partial ordering will serve our purpose. Formally:

$$\begin{aligned}\mathcal{P} = \{\{x_{i,1} \prec \ldots \prec x_{i,m} \mid \tilde{x_i} = \mathit{true}\} \prec \{\neg x_{i,1} \prec \ldots \prec \neg x_{i,m} \mid \tilde{x_i} = \mathit{false}\} \prec \{c_1 \ldots c_m\} \\ \prec \{x_{i,1} \prec \ldots \prec x_{i,m} \mid \tilde{x_i} = \mathit{false}\} \prec \{\neg x_{i,1} \prec \ldots \prec \neg x_{i,m} \mid \tilde{x_i} = \mathit{true}\}\}\end{aligned}$$

The basic idea of this proof is that literal agents can only wait at their source vertices, and that once any unnegated (resp. negated) literal agent $x_{i,j}$ (resp. $\neg x_{i,j}$) starts moving, it permanently blocks sequence x_i (resp. $\neg x_i$). This means any clause agent connected to $x_{i,j}$ (resp. $\neg x_{i,j}$) can only reach its own target through sequence $\neg x_i$ (resp. x_i), or through a literal related to a different variable (if the clause contains such literals).

All unnegated (resp. negated) literal agents associated with a variable receiving a *true* (resp. *false*) assignment immediately start moving into their respective sequences of targets. In sequences containing multiple targets, the agents will wait at their source vertex and move into the sequence in increasing order of their j index. Each clause agent will move through the shortest sequence that it can reach and is not blocked by constraints. Without loss of generality, let us assume one shortest sequence. If some $\tilde{x_i} = true$ (resp. $\tilde{x_i} = false$) appears in the clause associated with agent c_j, then agent c_j can wait at its source vertex until agent $x_{i,j}$ (resp. $\neg x_{i,j}$) clears vertex $v_{x_{i,j}}$ (resp. $v_{\neg x_{i,j}}$). Then, c_j can move through sequence $\neg x_i$ (resp. x_i), as the literal agents whose targets appear in sequence $\neg x_i$ (resp. x_i) are still waiting at their source vertices. c_j may have to wait until other clause agents that move through the same sequence clear the first vertex in the sequence, but this will not prevent c_j from reaching its target, only delay it. As soon as all clause agents that move through sequence $\neg x_i$ (resp. x_i) have cleared it, the literal agents $\neg x_{i,j}$ (resp. $x_{i,j}$) that connect to it will start moving into the sequence in order of their j index. With that, all agents have been assigned paths to their targets. Thus, if the 3SAT instance is satisfiable, the PFC-MAPF instance has a solution.

If PFC-MAPF has a solution, then each clause agent was assigned a path to its target. For each clause agent, this path passes through one of (up to) three sequences corresponding to the negated (resp. unnegated) version of an unnegated (resp. negated) literal that appears in the clause. Each literal agent $x_{i,j}$ (resp. $\neg x_{i,j}$) can only reach its target through one sequence of vertices, with the number of wait actions that it will have to perform depending how many clause agents traverse this sequence, and how many other literals have their targets in this sequence. All unnegated (resp. negated) literal agents $x_{i,j}$ (resp. $\neg x_{i,j}$) must have moved into and blocked sequence x_i (resp. $\neg x_i$) before any clause agent c_j may traverse the complementing sequence $\neg x_i$ (resp. x_i). Therefore, we can see that for every variable x_i, either sequence x_i or $\neg x_i$ (and not both) was traversed by clause agents, giving us an assignment of $\tilde{x_i} = false$ or $\tilde{x_i} = true$, respectively. For every variable x_i where neither sequence x_i nor $\neg x_i$ was traversed by any clause agent, any assignment would suffice, so we arbitrarily assign $\tilde{x_i} = true$. Thus, the resulting assignment satisfies the 3SAT instance. □

A.4 Proof of Theorem 7

Theorem 7. *The optimisation problem of PFC-MAPF is* NP*-hard on undirected graphs.*

Proof. We prove Theorem 7 by following a reduction from the 3SAT problem proposed by Yu and LaValle (2013), and showing it can similarly be used for PFC-MAPF. This proof is similar to the proof presented in appendix A.2, except that the path-function $\mathcal{F}$ is now handled as part of the reduction. To start the reduction, we create a PFC-MAPF problem corresponding to the 3SAT instance with $n + m$ agents on an undirected graph G. The set of agents is:

$$A = \{x_1, \ldots, x_n, c_1, \ldots, c_m\}$$

The x_i agents are called *variable agents*, the c_j agents are called *clause agents*. For each variable x_i, we create a vertex v_{x_i} that represents its source vertex. We connect two paths of length $m + 2$ to the source vertex, and join their ends such that the target is at the end vertex v'_{x_i}. The agent can traverse along either of the two paths to reach its target in $m + 2$ steps. Let these two paths be the i-th upper and lower paths. We will refer to the circuit that each such pair of paths creates as the i-th gadget.

Next, for each clause, we add a clause agent c_j whose source is vertex v_{c_j}. This vertex is connected to three paths that represent the literals in the clause of c_j. If a literal is unnegated (resp., negated), then v_{c_j} is connected to the i-th upper (resp., lower) path at a vertex of distance j from v_{x_i}. The target vertices for the clause agents are created as follows: A path of length m is added, where one end of the path is connected to the source vertex v_{x_i}. The connected vertex is the target vertex of agent c_m (v'_{c_m}) followed by the target vertex of agent c_{m-1} ($v'_{c_{m-1}}$) and so on, until the end of the path where the target vertex of agent c_1 (v'_{c_1}) is located.

Lastly, we define the path-function $\mathcal{F}$: We will only regard paths that do not include wait actions, as we only discuss optimal solutions in this proof. We can thus leave $\mathcal{F}$ undefined for paths that include wait actions. All variable agents prefer to take the lower path. All clause agents prefer to traverse paths belonging to gadgets that have constraints imposed by the variable agent associated with them; i.e., if variable agent x_i has already planned when the clause agent plans, then the i-th gadget is considered a preferred gadget. Naturally, if a clause agent c_j is connected to a path that has constraints, then that path will not appear as a shortest path for c_j. Similarly, if c_j is connected to both the lower and upper paths of the i-th gadget, and variable agent x_i has taken the upper (resp. lower) path, then the upper (resp. lower) c_j path will not appear as a shortest path for c_j. c_j will thus traverse the lower (resp. upper) path. If c_j is connected to multiple gadgets with constraints, then the gadget with the minimal index (i) among them is preferred. If c_j is not connected to any gadget with constraints, then it prefers to take a lower path if it is connected to one. Otherwise, it will have to take an upper path. If c_j is connected to multiple lower (resp. upper) paths, it will choose the one with minimal index (i). The preference towards paths with lower i is made for completeness, and does not affect the construction.

Figure 12 shows the complete graph for the PFC-MAPF problem constructed from 3SAT instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. A red vertex represents the source vertex of an agent, and a blue vertex is the target vertex.

To prove the difficulty of solving PFC-MAPF optimally, we must ask the question "does a solution with cost k exist for a given PFC-MAPF problem instance?". We set $k := (n + m)(m + 2)$, meaning all agents reach their targets without performing any wait actions.

If the 3SAT instance is satisfiable, each variable x_i receives an assignment of truth value $\tilde{x_i}$. There exists a total order $\mathcal{P}$ over the agents where all variable agents associated with a *true* assignment have the highest priority, followed by all clause agents, and then all variable agents associated with a *false* assignment. This describes a partial ordering, but any totalization of this partial ordering will serve our purpose. Formally:

$$\mathcal{P} = \{\{x_i \mid \tilde{x_i} = true\} \prec \{c_1 \dots c_m\} \prec \{x_i \mid \tilde{x_i} = false\}\}$$

All variable agents corresponding to *true* variables take their lower paths, leaving their upper paths empty, and so those will be traversed by corresponding clause agents. Clause agents that do not connect to the upper path of any *true* agent must therefore connect to the lower path of at least one variable agent assigned *false*, since the clause must contain a negation of some $\tilde{x_i} = false$ variable for the 3SAT instance to still be satisfiable. The variable agents assigned *false* then take upper paths if some clause agent traversed their lower path. Otherwise, they will take the empty lower path. This means that all variable agents and clause agents can start moving at time step zero and arrive at their targets at time step $m + 2$. Thus, if the 3SAT instance is satisfiable, the PFC-MAPF instance has a solution with the optimal cost $(n + m)(m + 2)$.

If the PFC-MAPF instance has a solution with cost $(n + m)(m + 2)$, then each agent has a path that arrives at its target without waiting. This means each variable agent x_i must move exclusively through either its lower or upper path, preventing any clause agent c_j from taking the same path in the opposite direction, but leaving the other path free for c_j to traverse. If agent x_i uses the lower (resp. upper) path, let $\tilde{x_i} = true$ (resp. false). Thus, the resulting assignment satisfies the 3SAT instance. □

A.5 Proof of Theorem 8

Theorem 8. *The decision problem of whether a priority-constrained solution exists is* NP*-hard on undirected graphs.*

Proof. We prove Theorem 8 by a reduction from the 3SAT problem. We create a corresponding PC-MAPF problem with $2n + m$ agents on an undirected graph G. The set of agents is:

$$A = \{x_1, \dots, x_n, c_1, \dots, c_m, f_1, \dots, f_n\}$$

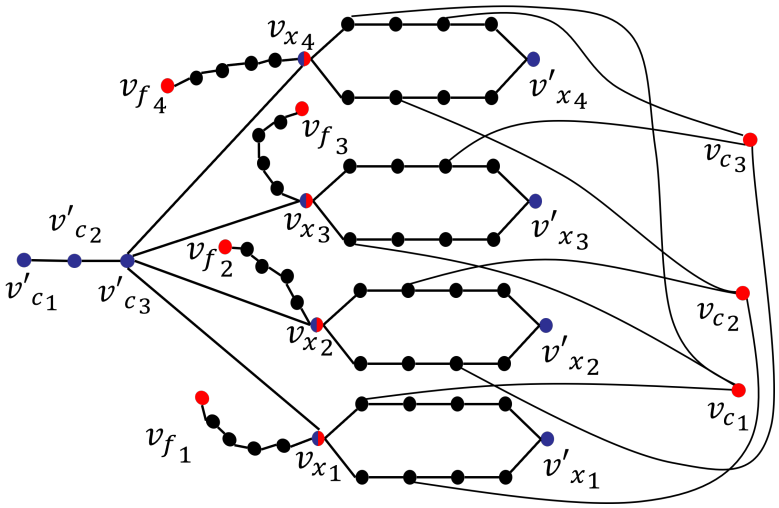

Fig. 14. Reduction from 3SAT to PC-MAPF for the instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. The red vertices are the source vertices, and the blue vertices are the targets.

The x_i agents are called *variable agents*, the c_j agents are called *clause agents*, and the f_i's are called *filler agents*. We construct the graph exactly like Yu and LaValle (2013) and expand upon it to prove our theorem. First, we construct a simple gadget for each variable x_i called a *mouse gadget*. For x_i, we create a vertex v_{x_i} that represents its source vertex. We connect two paths of length $m+2$ to the source vertex, and join their ends such that the target is at the end vertex v'_{x_i}. The agent can traverse along either of the two paths to reach its target in $m+2$ steps. Let these two paths be the i-th upper and lower paths. This section of the gadget is the body of the mouse. We create an additional path of length $m+2$ (the tail of the mouse) such that the last node of the path is v_{x_i}. We add a filler agent f_i to the first vertex of that path and call it v_{f_i}. The target vertex of f_i is v_{x_i}.

Next, for each clause, we add a clause agent c_j whose source is vertex v_{c_j}. This vertex is connected to three paths that represent the literals in the clause of c_j. If a literal is unnegated (resp., negated), then v_{c_j} is connected to the i-th upper (resp., lower) path at a vertex of distance j from v_{x_i}. The target vertices for the clause agents are created as follows: A path of length m is added, where one end of the path is connected to the source vertex v_{x_i}. The connected vertex is the target vertex of agent c_m (v'_{c_m}) followed by the target vertex of agent c_{m-1} ($v'_{c_{m-1}}$) and so on, until the end of the path where the target vertex of agent c_1 (v'_{c_1}) is located. Lastly, we define the total order over the set of agents A:

$$\mathcal{P} = \{f_1 \prec \ldots \prec f_n \prec x_1 \prec \ldots \prec x_n \prec c_1 \prec \ldots \prec c_m\}$$

Note that agent f_1 has the highest priority and agent c_m has the lowest priority. All filler agents must arrive at their target on time step $m+2$, as they have higher priority over clause and variable agents.

Figure 14 shows the complete graph for the PC-MAPF problem constructed from 3SAT instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. A red vertex represents the source vertex of an agent, and a blue vertex is the target vertex. Note that v_{x_i} is both red and blue as it is the source vertex of agent x_i and the target vertex of agent f_i.

If the 3SAT instance is satisfiable, each variable x_i receives an assignment of truth value $\tilde{x_i}$. If $\tilde{x_i}$ is true (resp., false), then let agent x_i take the lower (resp., upper) path of the body of the mouse gadget. The path that was not chosen is free to transit for the clause agents corresponding to the x_i variable. This means that all variable agents

and clause agents can start moving at time step zero and arrive at their targets at time step $m + 2$. Thus, vertices v_{x_i} are free at time step $m + 2$ for all agents f_i to arrive at their target. Since all agents can reach their targets within their individually optimal time, they do not violate the total order $\mathcal{P}$.

If PC-MAPF has a solution, then all clause agents have an optimal path to their target. Since the filler agents must reach their targets at time step $m + 2$ then agent c_m has to be at its vertex target v'_{c_m} at time step $m + 2$ and at time step $m + 1$ at one of the vertices v_{x_i}. Any delay in the arrival of a clause agent c_m would prevent it from arriving at vertex v'_{c_m} at time step $m + 2$, and thus agent c_m will be blocked by the filler agents. Variable agents x_i also cannot delay the path of the clause agents, and so they need to choose the opposite path to the clause agents' path in the body of the mouse gadget. If agent x_i uses the lower (resp. upper) path, let $\tilde{x_i} = true$ (resp. false). Thus, the resulting assignment satisfies the 3SAT instance.

□

B Extended Results

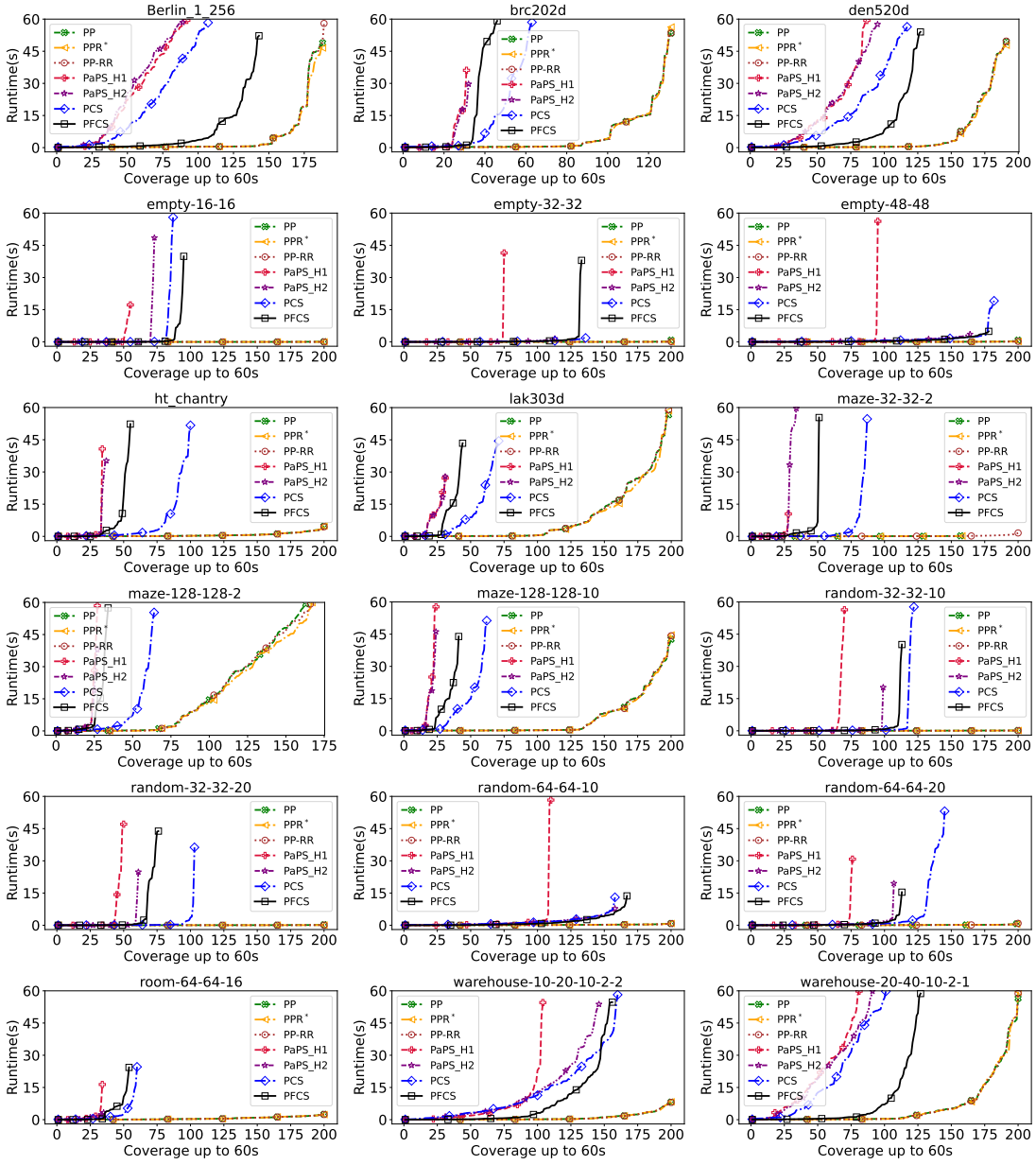

Fig. 15. Additional results for max runtime per instance, as a factor of coverage.

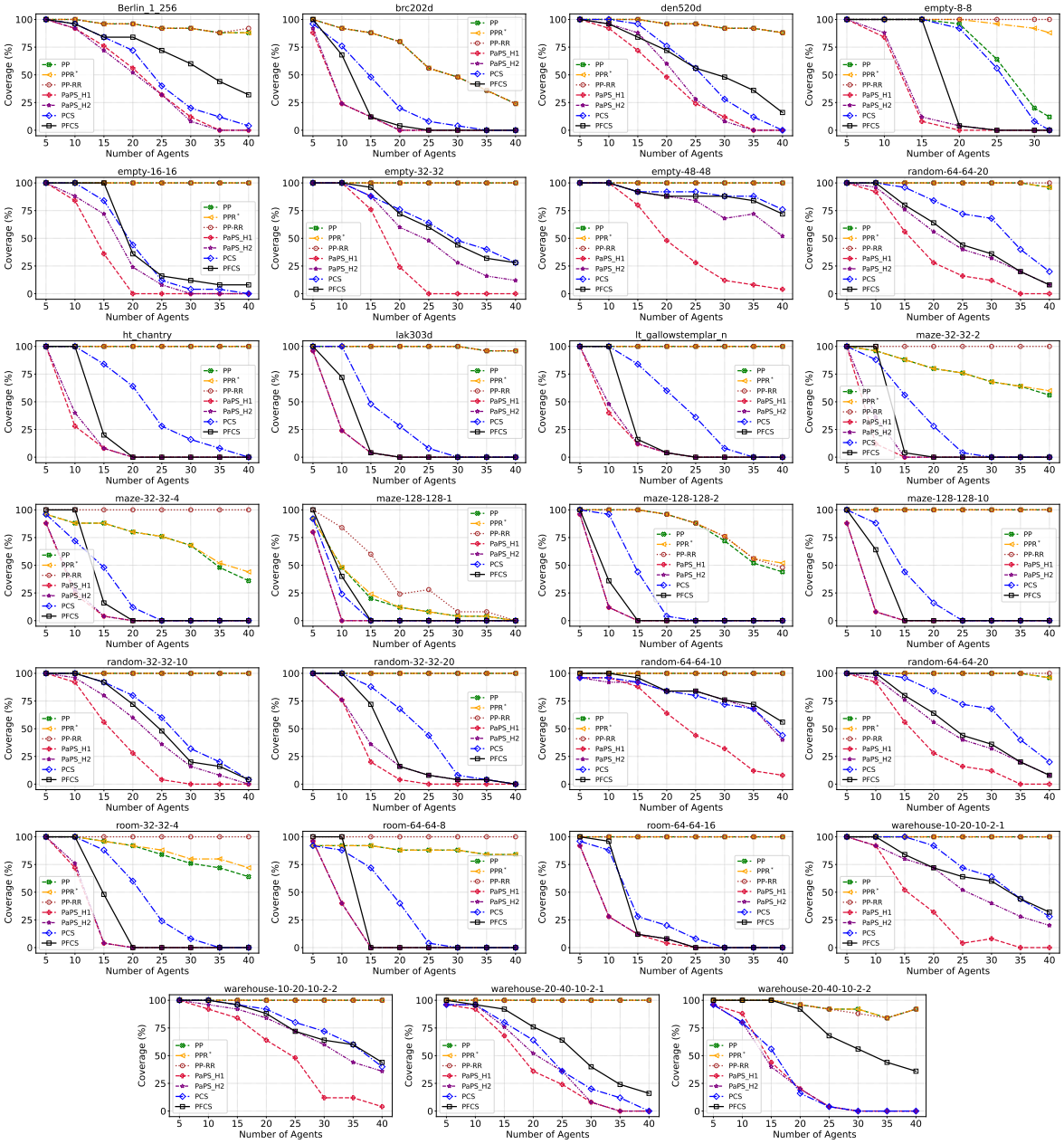

Fig. 16. Results for coverage as a factor of the number of agents.

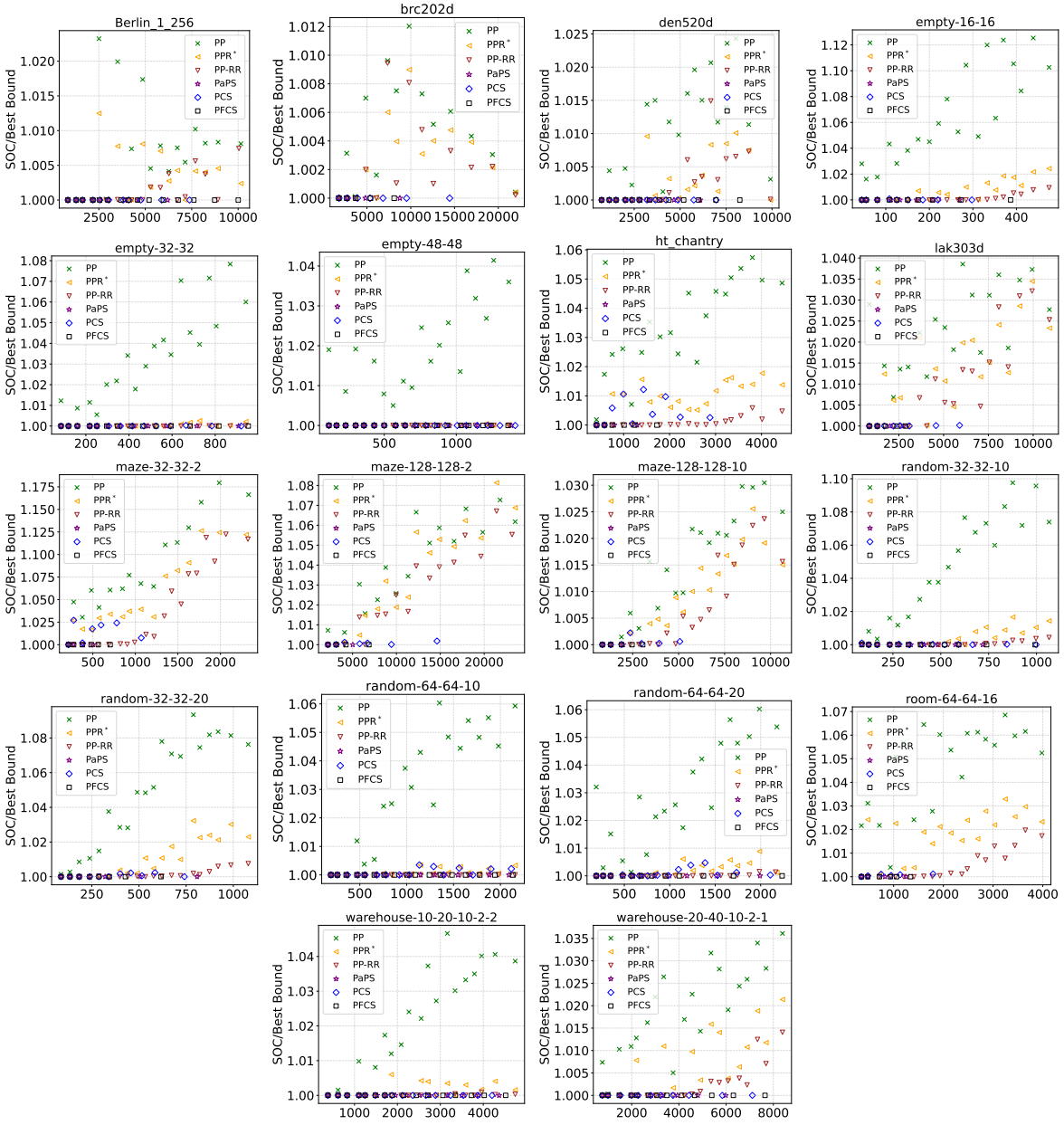

Fig. 17. Additional results for the cost of solutions, relative to the best known lower bound on the cost of optimal solutions to the equivalent MAPF problems.

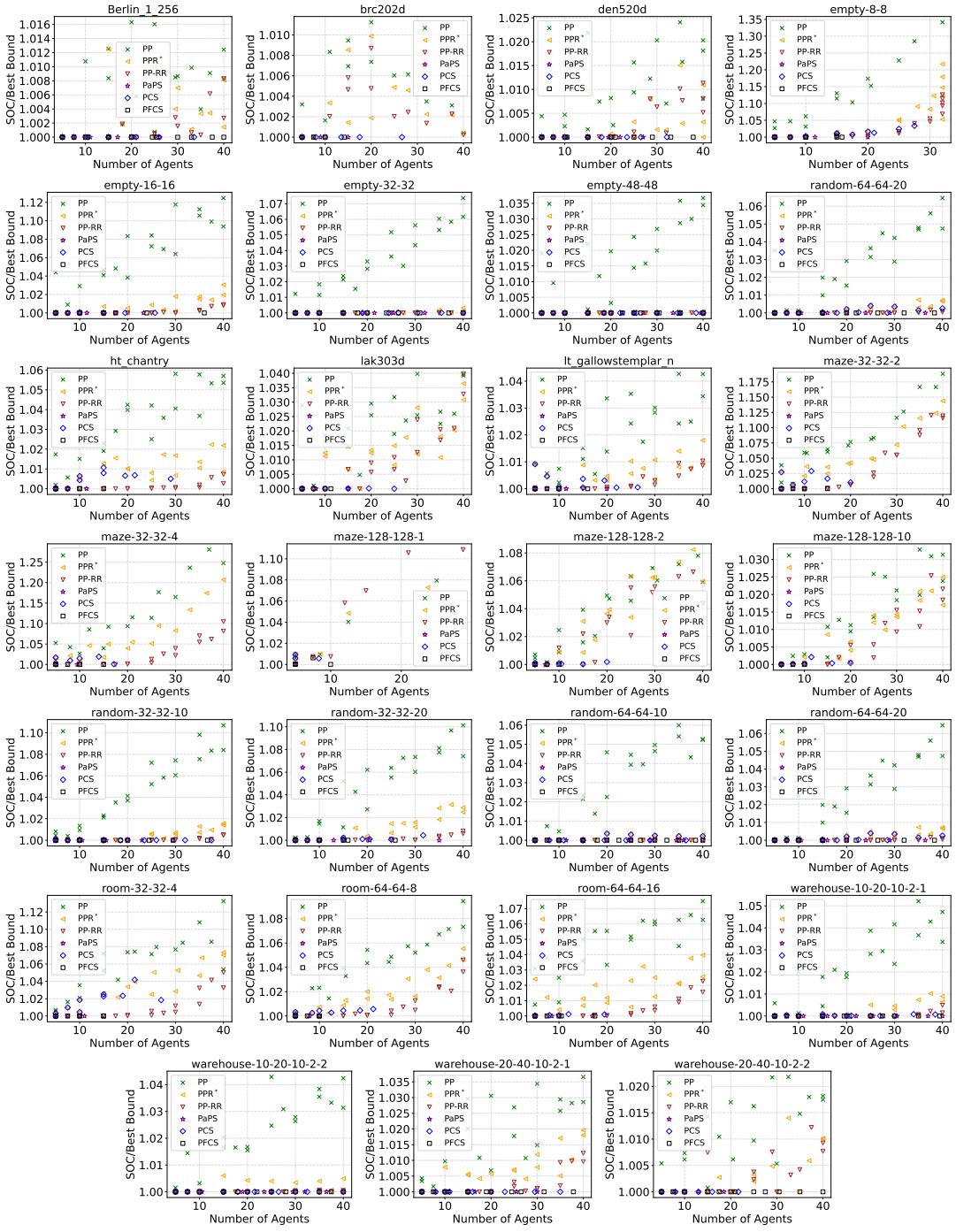

Fig. 18. Results for the cost of solutions, relative to the best known lower bound on the cost of optimal solutions to the equivalent MAPF problems, as a factor of the number of agents.

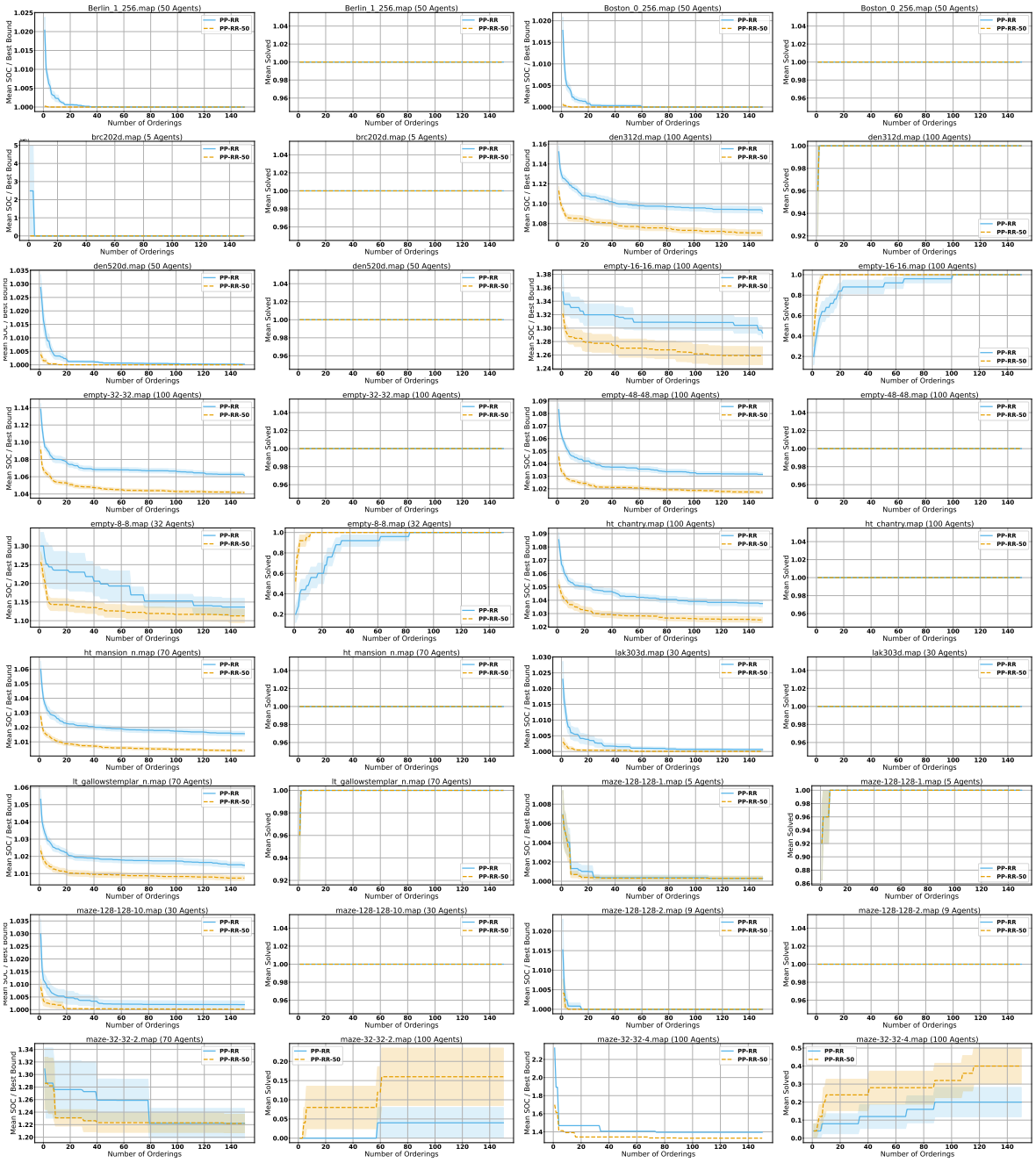

Fig. 19. Additional results for suboptimality of solutions and Success Rate of PP-RR with 1 or 50 path-functions, as a factor of the number of orderings attempted. Part 1.

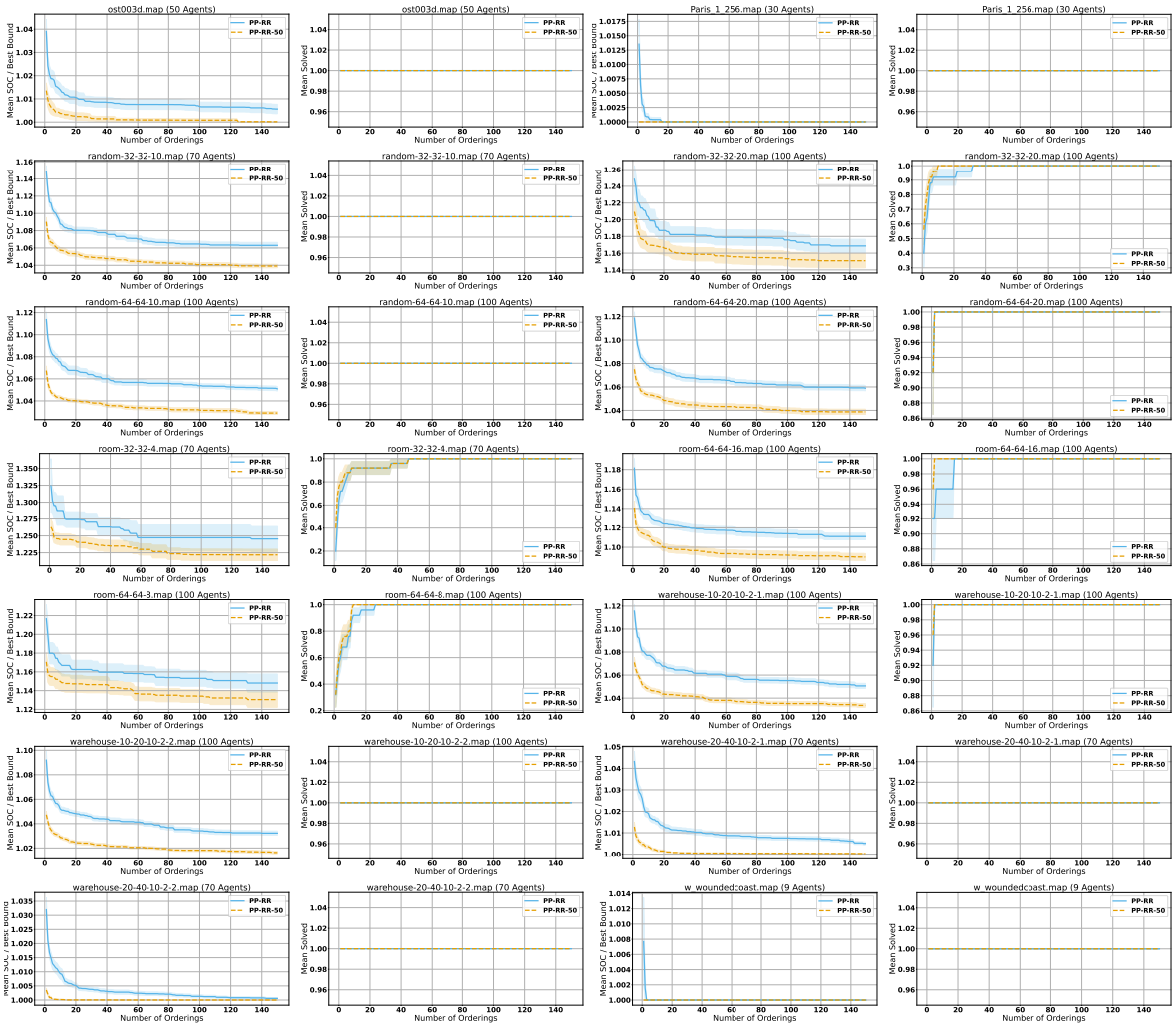

Fig. 20. Additional results for suboptimality of solutions and Success Rate of PP-RR with 1 or 50 path-functions, as a factor of the number of orderings attempted. Part 2.

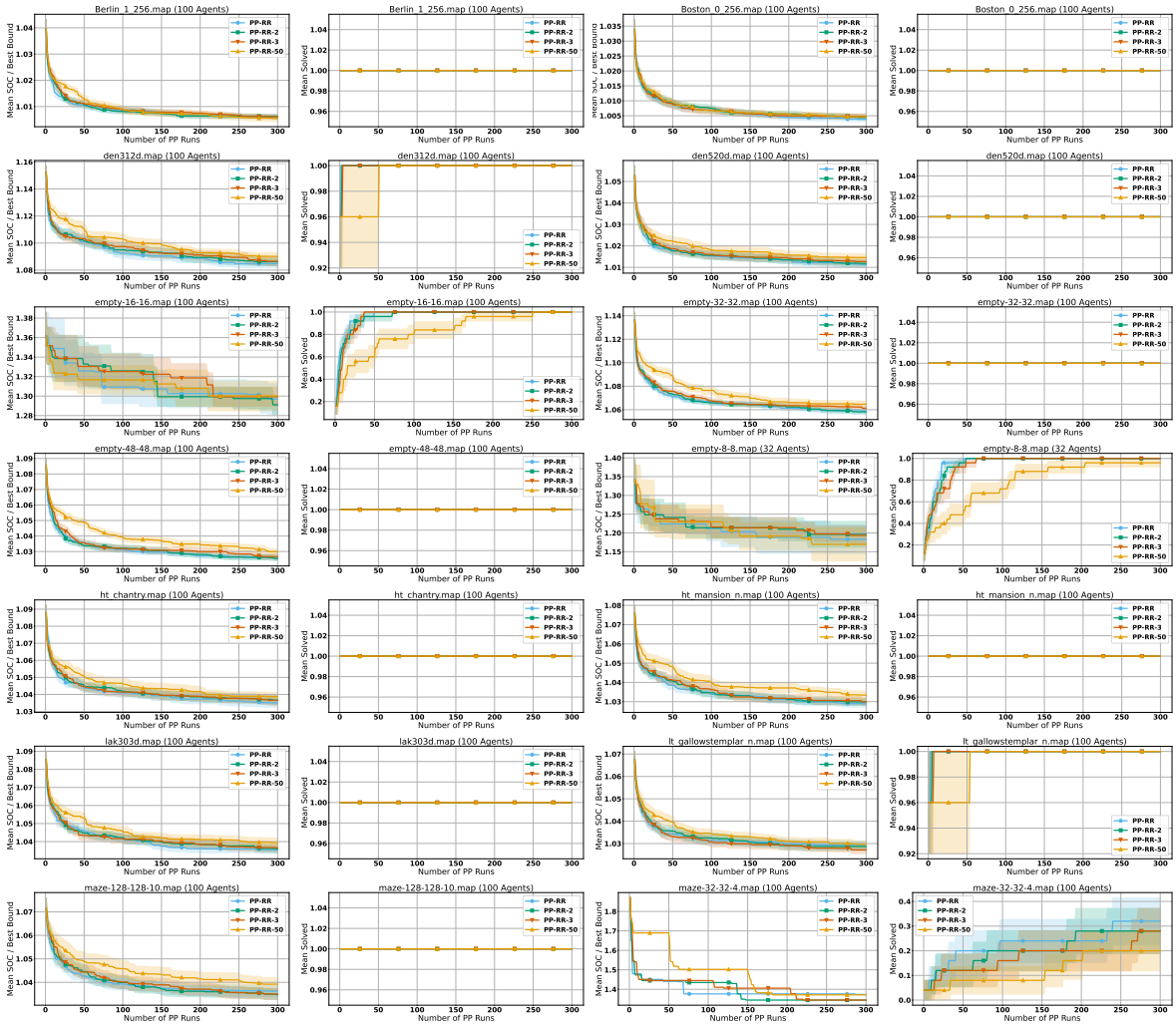

Fig. 21. Additional results for suboptimality of solutions and Success Rate of PP-RR with 1 or 50 path-functions, as a factor of the number of PP runs attempted. Part 1.

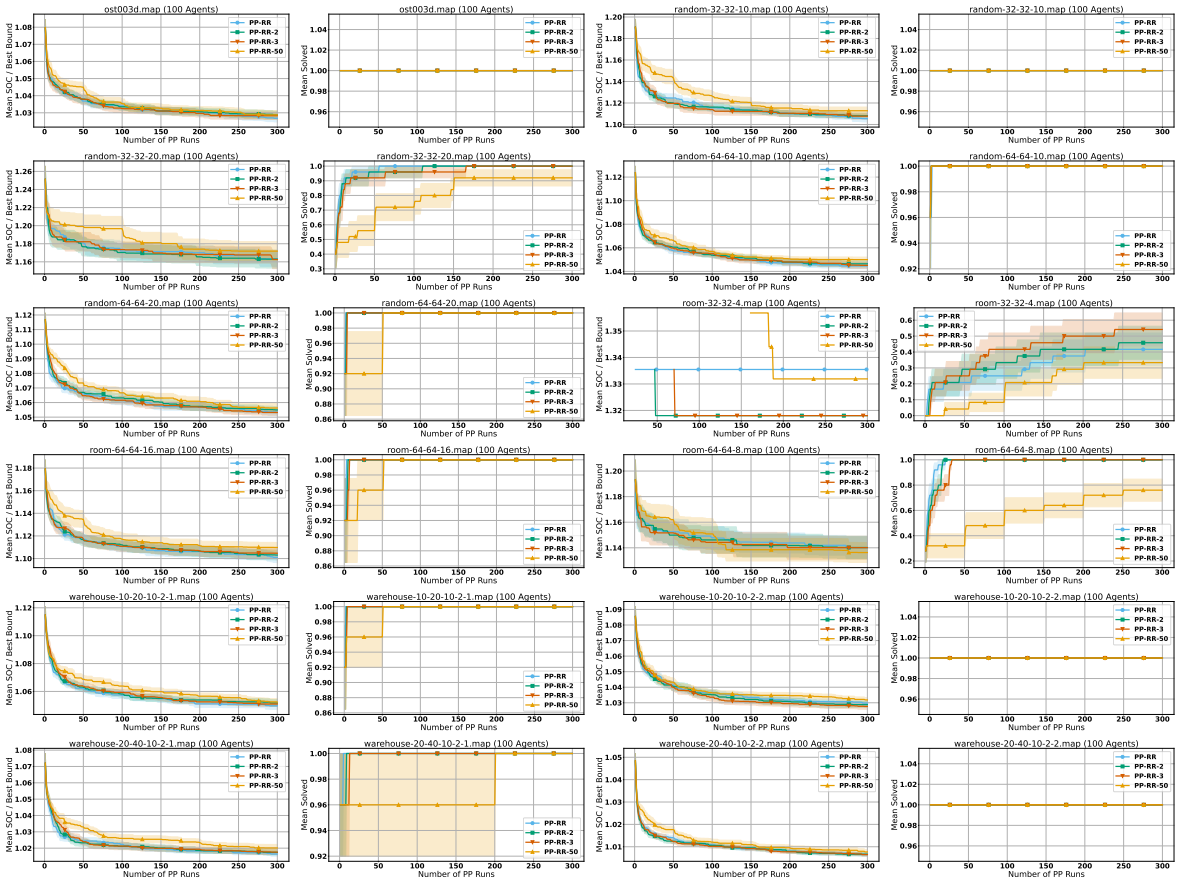

Fig. 22. Additional results for suboptimality of solutions and Success Rate of PP-RR with 1 or 50 path-functions, as a factor of the number of PP runs attempted. Part 2.

Received 01 June 2025; accepted 06 October 2025