Prioritised Planning with Guarantees

Jonathan Morag[1,2], Yue Zhang[2], Daniel Koyfman[1], Zhe Chen[2], Ariel Felner[1], Daniel Harabor[2], Roni Stern[1]

[1]Ben-Gurion University of the Negev
[2]Monash University
moragj@post.bgu.ac.il, Yue.Zhang@monash.edu, koyfdan@post.bgu.ac.il, zhe.chen@monash.edu, felner@bgu.ac.il, Daniel.Harabor@monash.edu, roni.stern@gmail.com

Abstract

Prioritised Planning (PP) is a family of incomplete and sub-optimal algorithms for multi-agent and multi-robot navigation. In PP, agents compute collision-free paths in a fixed order, one at a time. Although fast and usually effective, PP can still fail, leaving users without explanation or recourse. In this work, we give a theoretical and empirical basis for better understanding the underlying problem solved by PP, which we call Priority Constrained MAPF (PC-MAPF). We first investigate the complexity of PC-MAPF and show that the decision problem is NP-hard. We then develop Priority Constrained Search (PCS), a new algorithm that is both complete and optimal with respect to a fixed priority ordering. We experiment with PCS in a range of settings, including comparisons with existing PP baselines, and we give first-known results for optimal PC-MAPF on a popular benchmark set.

Introduction

Prioritised Planning (PP) (Erdmann and Lozano-Pérez 1986) is a family of related algorithms that are popularly applied in Robotics and AI; e.g., for coordinating the motion and for navigating multiple moving agents (Van Den Berg and Overmars 2005; Silver 2005). In PP algorithms each agent is planned in turn, following a fixed priority order $\mathcal{P} = \{a_1 \prec a_2 \prec \ldots \prec a_k\}$. Each agent a_i attempts to reach its target as efficiently as possible, but lower-priority agents a_j must avoid the previously planned paths of higher-priority agents; i.e., $a_i \prec a_j$. PP algorithms are usually fast and scalable (Li et al. 2021), since each agent is only planned once. Moreover, the quality of solutions found by PP can be close to optimal (Ma et al. 2019) as each agent always takes a shortest available path to reach its target.

There are two main drawbacks to PP: (i) computed solutions have no quality guarantees and; (ii) planning agents in priority order may cause deadlock failures. Figure 1(a) shows an example using the agent ordering $\mathcal{P} = \{a_1 \prec a_2 \prec a_3\}$. Notice that agent a_2 can select among several equivalent-cost paths to reach its target while avoiding the path of a_1. The option of waiting at vertex s_2 and then moving to v_3 or s_3 minimises the sum-of-(path)-costs for all agents (which is 13). Alternatively, a_2 could move directly to v_3, which causes additional waiting (+1) for agent a_3. Finally, a_2 could move directly to s_3, which produces a deadlock failure (since a_3 is left without recourse). Thus, it is not clear how to interpret the results from PP. If a solution is found, could its cost be improved? If the result is a deadlock, is the given instance not solvable using priority order $\mathcal{P}$?

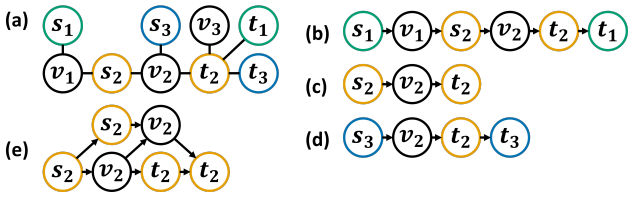

Figure 1: **(a)** A PC-MAPF problem with three agents and priority ordering $\mathcal{P} = \{a_1 \prec a_2 \prec a_3\}$. Vertices s_i and t_i indicate the start and target positions of agent i. **(b-d)** The minimal MDD of each agent. **(e)** MDD of a_2 with +1 depth.

One idea to mitigate the drawbacks of PP involves generating a different priority ordering $\mathcal{P}'$, e.g., at random (Bennewitz, Burgard, and Thrun 2002; Li et al. 2021) or following rule-of-thumb heuristics (Van Den Berg and Overmars 2005; Silver 2005), and then planning new paths for all agents. However, each $\mathcal{P}'$ may suffer the same weaknesses as $\mathcal{P}$. Even enumerating all $k!$ possible orderings (prohibitive, except for small k) is insufficient to settle the optimality or solvability questions. Moreover, important real-world applications can demand coordinated plans which respect a given priority ordering, such as construction-crane planning (Zhang 2010; Zhang and Hammad 2012) or autonomous intersection management (Li et al. 2023), where safety, cost, and task precedence determine movement orderings. Another strategy involves modifying the problem settings to allow agents to safely wait at their start or target locations (Čáp et al. 2015). This eliminates the possibility of deadlocks and ensures that PP always succeeds. However, such modifications are not suitable for all applications.

In this work, we investigate the underlying problem solved by PP, which we call PC-MAPF. We give a first theoretical analysis of the problem and show that computing feasible plans with respect to a given priority ordering $\mathcal{P}$ is NP-hard. We then propose a new algorithm, Priority Constrained Search (PCS), which computes optimal solutions to

PC-MAPF problems, and returns failure if no such solution exists. PCS is the first prioritised algorithm to have these guarantees. We also propose a simple and sub-optimal algorithm, PPR*, which uses randomisation to sample the space of feasible PC-MAPF solutions. In an empirical analysis, we compare PCS, PPR*, and PP in terms of performance and success rate. We show that our proposed methods can succeed where PP fails, and we report first optimal solutions for a wide range of PC-MAPF problems from recent literature.

Background and Definitions

In this work, we consider a popular simplified Multi-Agent Path Finding (**MAPF**) model from Stern et al. (2019). The input of a MAPF problem is a graph $G = (V, E)$ and a set of k agents. Each agent a_i has an associated start location $s_i \in V$ and target location $t_i \in V$. Time is discretised into time steps. At each time step an agent can move to an adjacent location in the graph (provided there exists an appropriate edge) 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 t, called *vertex conflict* (denoted as $\langle a_i, a_j, v, t\rangle$) or traverse the same edge $(u, v) \in E$ at the same time t, called *edge conflict* (denoted as $\langle a_i, a_j, u, v, t\rangle$). A *solution* π to the MAPF problem is a set of k paths, one for each agent. We say that a solution is *valid* if the paths are conflict-free. The cost of a solution is defined as the *Sum-of-Costs* (SOC), which is the sum of path cost for each agent; i.e., $\sum_{i=1}^{k} cost(\pi_i)$. A solution is called *optimal* if it has the lowest SOC of all valid solutions.

Some MAPF algorithms work by imposing *constraints* on the agents, limiting their choice of paths. A *positive constraint* $\langle a_i, v, t\rangle$ says that agent a_i must use vertex v at time step t. Similarly, the constraint $\langle a_i, u, v, t\rangle$ says that agent a_i must use edge (u, v) at time step t. Their negated forms, $\neg\langle a_i, v, t\rangle$ and $\neg\langle a_i, u, v, t\rangle$, indicate a prohibition (a_i must not be at v or use (u, v) at time t), and are known as *negative constraints*. Naturally, a positive constraint on agent a_i directly adds a negative constraint on all other agents $a_j \neq a_i$.

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. Each agent, in its turn, is given an individually optimal path that avoids the paths of preceding (equiv., 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 et al. 2019).

When planning 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.

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 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 start 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 1, (b-d) shows the minimal (assuming no constraints) MDDs for the three agents in the problem shown in (a). (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 t on any optimal path is added as an MDD node v at depth t.

Priority Constrained MAPF

The Priority Constrained MAPF (PC-MAPF) problem 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 functions, mapping each agent to its initial and final location in G; $\mathcal{P}$ is a total order (priority) over the set of agents, and thus for any pair of agents i and j, either $i \prec j$ or $j \prec i$. We say that agent i has a *higher priority* than agent j if $i \prec j$.

A solution π to the PC-MAPF problem is a set of paths, one for each agent. The path of agent a_i induces a set of edge and vertex constraints on all other agents; i.e., no two agents can use a resource at the same time. We call a solution *priority-constrained* (equiv. *consistent w.r.t.* $\mathcal{P}$ (Ma et al. 2019)) if for every agent a_i, removing the paths of all lower-priority agents $\{a_j | a_i \prec a_j\}$ does not enable to assign a_i a lower-cost path (while still avoiding the paths of higher-priority agents). A solution to a PC-MAPF problem is called *priority-optimal* if it has the smallest cost among all priority-constrained solutions. If no priority-constrained solution exists we consider the problem as *unsolvable*. An algorithm is called *priority-complete* 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.

It is easy to see that for every PC-MAPF problem there exists an analogous MAPF problem where the input is identical, but the returned paths may not be priority-constrained. Hence, for any priority-optimal solution, there exists a corresponding optimal MAPF solution with an equal or smaller cost. Similarly, if a MAPF problem is unsolvable, then the corresponding PC-MAPF problem is unsolvable. However, the reverse is not true; i.e., for an unsolvable PC-MAPF problem there may exist a valid MAPF solution. The following theorem proves the equivalence of solutions to PC-MAPF problems and solutions that may be computed by PP to the equivalent MAPF problems.

Theorem 1. *A MAPF problem can be solved by PP iff the corresponding PC-MAPF problem is solvable.*

Proof. ($\rightarrow$) PP plans agents one by one, from highest priority to lowest priority, as specified by $\mathcal{P}$. Each agent a_j takes an individually optimal path but must avoid the previously planned paths of higher-priority agents a_i. Removing the path of a_j cannot improve the arrival time of any $a_i \prec_{\mathcal{P}} a_j$, 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 priority-constrained. ($\leftarrow$) A PC-MAPF solution is a collection of shortest priority-constrained paths. By definition of priority-constrained, the path of agent a_i is independent of any constraints introduced by a_j. In other words, the path appears as an individually optimal path for a_i when that agent is planned by PP. □

Theorem 1 shows that PP has the potential to find any one of the priority-constrained solutions. However, when planning an agent with multiple optimal paths to its target, PP algorithms make arbitrary choices, and these decisions can produce a deadlock. Reasoning about the causes of PP failure is challenging as: (i) the root cause is unclear and; (ii) changing the paths of higher-priority agents, to restore feasibility in case of deadlock, can affect the paths of all subsequent agents, leading to yet more failures. For example, in Figure 1(a), PP produces deadlock failure if a_2 is assigned a path that proceeds directly to s_3. Yet a_2 may not be responsible for the deadlock; its choice of paths is in turn constrained by the path choices of a_1. Thus, replanning the last successful agent (here a_2) may not be sufficient to avoid failure in general (although in this example, it is).

We summarise the situation as follows: when PP fails we do not know if the failure is due to the choice of paths or if the problem is unsolvable. Similarly, when PP succeeds, the solution is guaranteed to be priority-constrained, but it may not be priority-optimal. Despite several decades of research on the topic, no previous study has been able to improve the theoretical properties of PP in general. The following theorem may help to explain why. We show that deciding whether a priority-constrained solution exists is at least as hard as MAPF in general. Moreover, it is potentially harder on undirected graphs, as the decision problem for those is polynomial for MAPF (Daniel Kornhauser 1984).

Theorem 2. *The decision problem of whether a priority-constrained solution exists is* NP*-hard on undirected graphs.*

Proof. We prove this theorem by a reduction from the 3SAT problem. 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.

We create a corresponding PC-MAPF problem with $2n + m$ agents on an undirected 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 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 start vertex. We connect two paths of length $m + 2$ to the start 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 that starts at 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 start 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 time step $m+2$ to their target as they have higher priority over clause and variable agents.

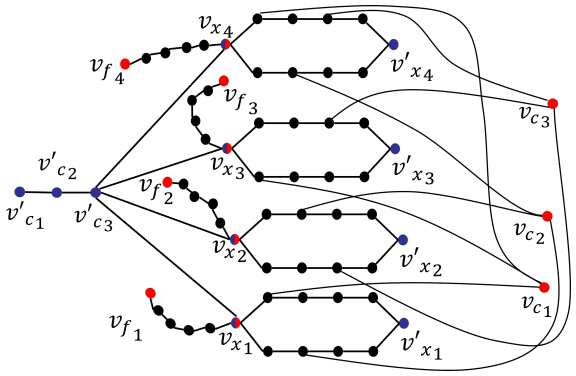

Figure 2: Reduction from 3SAT to PC-MAPF for the instance $\langle\{x_1, x_2, x_3, x_4\}, \{(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)\}\rangle$. The red vertices are the start vertices and the blue are the targets.

Figure 2 shows the complete graph for the PC-MAPF problem constructed from the 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 start vertex of an agent and a blue vertex is the target vertex. Note that v_{x_i} is both red and blue as it the start 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.

□

Observation 1. *The decision problem of PC-MAPF in directed graphs and the optimization problems both in directed and undirected graphs are* NP-hard.

Prioritised Planning with Randomised A*

A popular approach to extend PP for MAPF is by repeatedly randomising the priority order of the agents (Bennewitz, Burgard, and Thrun 2002). This randomisation is performed whenever PP fails, but can also be applied to find better-quality solutions when excess planning time remains. This approach can not be used for PC-MAPF, since the priority order is fixed. As an alternative, we suggest to randomise the tie-breaking used by the single-agent planner in order to choose between different paths with equal costs. We call this simple and scalable algorithm Prioritised Planning with Randomised A* (PPR*). The main advantage is that 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.

Priority Constrained Search (PCS)

A straightforward approach for optimal PC-MAPF is to find all optimal constrained paths for the current priority agent, create a child node for each path, and then add the next agent in each of these child nodes. If the nodes are expanded in best-first order (smallest SOC) this approach guarantees to eventually find a priority-optimal solution and eventually terminate if no such solution exists. The main disadvantage is a potentially large branching factor, as many individually-optimal paths can exist for a given agent. In this section, we introduce a new and more effective optimal algorithm for PC-MAPF called *Priority Constrained Search* (PCS).

PCS is a two-level search algorithm, similar in broad strokes to CBS (Sharon et al. 2015), a popular approach for optimal MAPF. The *high-level* of PCS explores a binary tree, searching over the space of possible conflicts, similar to CBS. The *low-level* of PCS resolves conflicts by replanning agents one at a time, each time subject to new constraints, again akin to CBS. Yet PCS differs in a few important ways:

1. CBS stores a single path for each agent, that satisfies its relevant constraints. PCS uses MDDs instead of paths.
2. CBS generates paths for all agents in the root node. PCS starts with the highest-priority agent and gradually adds agents (MDDs) as the search progresses.
3. To resolve a conflict, CBS constrains and replans affected agents, potentially increasing their individual costs. By contrast, PCS favours high-priority agents, and it never generates nodes where the cost of these agents increases.
4. CBS creates constraints only when splitting, and stores them on the search tree branches. PCS has two distinct ways that constraints are created and stored. Constraints on high-priority agents are directly created through splits. Constraints on low-priority agents are gradually created on the fly while generating a search node.

We next provide full details on PCS.

High-level Search

At the high-level, PCS conducts a best-first search based on the cost function $f(n) = g(n) + h(n)$ (detailed below), over a *Priority-Constrained Tree* (PCT); a binary tree where each node n has the following fields:

- $n.c$ - The index of the *current agent*; i.e., agent a_c is the agent currently being planned. We note that a_c might gradually change inside node n to be the next agent in the order of $\mathcal{P}$ (if no conflicts are found, see below).
- $n.M$ - a list of MDDs stored by this node. We store one MDD for each agent a_i with priority $a_i \preceq_{\mathcal{P}} a_c$. The MDDs of higher-priority agents, $M[i]$ are disjoint (no shared vertices/edges at the same time; i.e., no conflicts). These MDDs might be modified in PCT nodes below n but their depth must not increase. The MDD for the current agent, $M[c]$, may be replaced by an MDD with larger depth in response to newly added constraints at node n.
- $n.constraints$ - a set of constraints relevant for node n. Each node inherits the constraints of its parent. New constraints are added to node n to resolve or prevent conflicts, based on the MDDs of preceding agents.
- $n.conflict$ - a tuple comprising two agents and a contested resource; e.g., a vertex conflict $\langle a_{hi}, a_c, v, t\rangle$. One agent is always the current agent, a_c; the other agent is always a higher-priority agent $a_{hi} \prec_{\mathcal{P}} a_c$. The conflicting resource (vertex or edge) appears in the MDDs of both.
- $n.g$ - cost of the plan thus far, n, derived by summing the depths of all MDDs in $n.M$.
- $n.h$ - an estimate of the remaining plan cost.

Algorithm 1 shows the pseudo code for the high-level. We use a priority queue, $OPEN$ to determine the order of node expansions. We pop nodes from $OPEN$ with minimum f-value, where $f(n) = n.g + n.h$ (line 5). Let n denote the

Algorithm 1: PCS

Input: a problem instance $\langle G, k, s, t, O \rangle$, a heuristic function H

1: $OPEN \leftarrow$ min-heap
2: $root \leftarrow Generate(null, null, null)$
3: $OPEN.insert(root)$
4: **while** $OPEN$ not empty **do**
5: $n \leftarrow OPEN.pop()$
6: **if** $isGoal(n)$ **then**
7: **return** solution from $n.M$
8: // Expand
9: $a_{hi} \leftarrow n.conflict.a_{hi}$
10: $r \leftarrow \langle a_{hi}, n.conflict.v, n.conflict.t \rangle$
11: $OPEN.insert(Generate(n, r, a_{hi}))$
12: $OPEN.insert(Generate(n, \neg r, a_{hi}))$
13: return no-solution

Algorithm 2: Generate

Input: parent node p, constraint r, constrained agent *hi*

1: $n \leftarrow$ an empty PCT node
2: **if** p is not $null$ **then**
3: $n \leftarrow$ a copy of node p
4: $n.M[hi].Restrict(r)$
5: Add all new critical resources of $n.M[hi]$ to $n.constraints$ as constraints
6: $n.M[c] \leftarrow lowLevel(a_c, constraints)$
7: **if** $n.M[c] = null$ **then**
8: Return *null*
9: $n.conflict \leftarrow$ conflict with another MDD if one exists
10: **while** $n.conflict = null \wedge |n.M| < k$ **do**
11: $n.c \leftarrow$ next agent according to $\mathcal{P}$
12: $n.M[c] \leftarrow lowLevel(a_c, constraints)$
13: **if** $n.M[c] = null$ **then**
14: Return *null*
15: **else if** exists conflict between $n.M[c]$ and another MDD **then**
16: $n.conflict \leftarrow$ conflict with another MDD
17: **else**
18: Add all critical resources of $n.M[c]$ to $n.constraints$
19: Return n

most recently popped node. If n is conflict-free then it is a goal, and the search is done (line 7). That is, node n contains disjoint MDDs, one for each agent, i.e., each agent can reach its target by following any individually optimal path without any collisions. Otherwise, as a best-first search, PCS uses the regular *expand* and *generate* functions described next.

Expand If n contains a conflict (assume vertex conflict for simplicity) between the current agent a_c and some other higher priority agent a_{hi}, then we *expand* the node (lines 8-12). This procedure branches at the current node n and produces two child nodes to resolve the current conflict. The left child will have a positive constraint $\langle a_{hi}, v, t \rangle$, which says a_{hi} must occupy vertex v at time t; i.e., $MDD_{hi}[t] = \{v\}$ (line 11). The right child will have a negative constraint $\neg \langle a_{hi}, v, t \rangle$, which says that a_{hi} must not occupy vertex v at time t; i.e., $v \notin MDD_{hi}[t]$ (line 12). This operation is known as *disjoint splitting* (Li et al. 2019) for CBS.

Generate *Generate* is invoked at the start of Algorithm 1 to create the root node (when it is called with null-valued parameters, line 3), or as a consequence of expanding (splitting) a parent node p (lines 11–12). *Generate* is the primary step of PCS and its pseudo code is given in Algorithm 2. *Generate* receives as input a *constraint* r and enforces r on the MDD of the high-priority agent, $M[hi]$, using the *Restrict* function (line 4, Alg. 2) as follows. If r is a negative constraint $\neg(a_{hi}, v, t)$ (similarly for an edge conflict) then v is removed 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. If r is a positive constraint $\langle a, v, t \rangle$, we want a sub-graph that only contains paths that satisfy r. This is done by taking the sub-graph that is rooted at (v, t) and connecting it to all paths from the root that lead to (v, t) (by searching back from (v, t) to the MDD's root). Other paths are deleted from the MDD.

The main difference between CBS and PCS is that in PCS 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. 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. During the *restrict* operation (line 4), if critical resources are identified, then we preemptively constrain all lower-priority agents from conflicting with these critical resources (line 5). We also do this when we are done with the current agent (line 18). This ensures that a conflict with a critical resource of a high-priority agent will never occur.

Next, the *Generate* function calls the low-level search to update/find $M[c]$ (the MDD of the current agent), satisfying the new constraints (line 6). Finally, the *Generate* function checks for further conflicts between $M[c]$ and the MDDs of preceding agents. When there are multiple conflicts, we pick the earliest one. If a conflict is found, it will be handled when this node is expanded, so the *Generate* function returns. Otherwise, if no conflicts are found, (lines 10-18),[1] *Generate* adds the next agent according to the priority order. This agent becomes the current agent and we compute its MDD by calling the low-level search. This is repeated until all agents are added or until a conflict is detected. The root node is generated similarly (adding agents until conflicts arise).

Low-Level Search

The low-level search for PCS 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 node, to

[1] We can pause generating a node n when its cost lower bound ($f(n)$) exceeds the current lower bound on the cost of the solution, insert it to $OPEN$, and continue once it is popped.

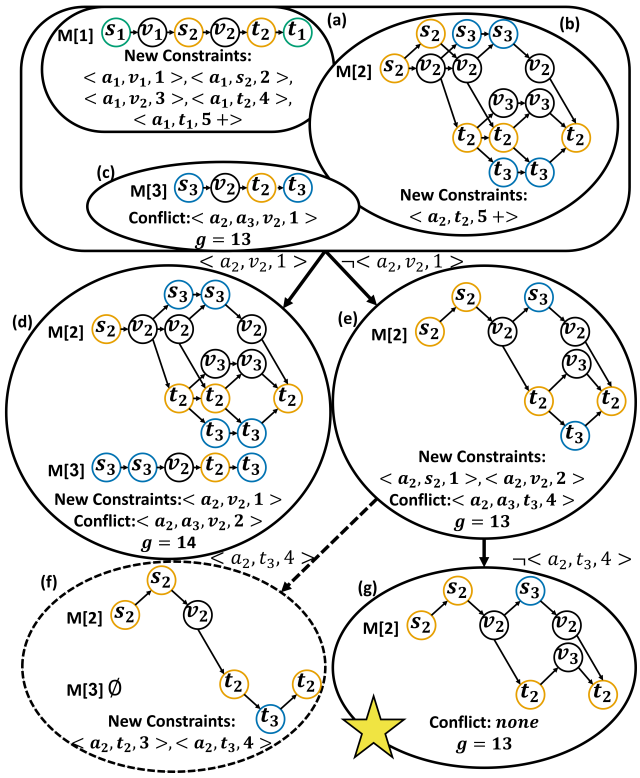

Figure 3: Part of the PCT for the problem in Figure 1(a).

maintain all equivalent length paths to each node.

PCS Example

Figure 3 shows part of the PCT for the problem in Figure 1(a). The full tree is available in the supplementary materials.[2] The figure only shows new or changed MDDs and new constraints at each node, and we limit our focus to vertex constraints. Generated nodes have a solid border, while invalid nodes that were not generated have a dashed border. The goal node is marked with a star. Generating the root node is illustrated in three stages (a-c). In (a), $M[1]$, 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 $M[1]$, so 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 $M[2]$, and so we add constraints for its only critical resource, $\langle a_2, t_2, 5+\rangle$. Next, we build $M[3]$ while obeying the current 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 finish generating the root, both because we have detected a conflict, and because all agents have been added. We expand the root, generating two child nodes. The left child (d) 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 (d) into $OPEN$. The right child (e) 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. (e) is inserted to $OPEN$, and immediately popped, as its cost is 13, making it the current minimum. We expand (e), generating (f) and (g), but (f) is pruned since the constraints it adds result in a deadlock where it is impossible for a_3 to reach its target. Next, (g) 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, (g) would not have been a goal node yet. Instead, we would have continued generating (g) by closing $M[3]$ and building an MDD $M[4]$, checking for conflicts, etc.

PCS Properties

Theorem 3. *PCS is priority-complete*

Proof by induction. A solution is *permitted* by a PCT node n if its paths obey all constraints in $n.constraints$. The following proof will show that two properties are maintained throughout the search - no priority-constrained solutions are lost (all are permitted by at least one leaf node), and all represented solutions (or parts thereof) are priority-constrained. Step 1 is the base case of the induction; step 2 is the induction hypothesis; steps 3 and 4 are two separate induction steps, for the two primary functions of PCS, adding agents (*Generate*) and splitting on conflicts (*Expand*), respectively.

(1) At the start of the root node's generation, before any MDD is added, the constraints set is empty. Therefore, all priority-constrained solutions are permitted at this point. Additionally, since no solution is explicitly represented yet, all represented solutions are priority-constrained.

(2) Assume a PCT node that contains m disjoint MDDs. Any set of paths (one for each agent) derived from the MDDs is part of a priority-constrained solution.

(3) When the m+1th MDD is added, this property is maintained: The new MDD is built to have the minimal possible depth while avoiding existing constraints, all of which are critical resources of some existing MDD. No priority-constrained solutions are lost, because the MDD represents all paths of that depth that obey the constraints.

(4) When the MDD of a higher-priority agent (not $M[c]$) is constrained, no priority-constrained 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 negatively and positively 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. Additionally, when the current MDD ($M[c]$) is updated to obey new constraints

(avoid new critical resources), the priority-constrained property is maintained just as it is maintained when it is first generated (see (3)). Thus, both the addition of new MDDs and the addition of new constraints maintain the priority-constrained property for each node and never eliminates any priority-constrained solutions from the PCT. In particular, goal nodes also represent priority-constrained solutions. □

Theorem 4. *PCS is priority-optimal*

Proof. Assume a heuristic function that always returns 0.

The cost function ($n.g$) is monotonically non-decreasing: When generating a node, 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. We may also add more MDDs for more agents that are not covered by the parent node. This action only increases the cost of the node. Thus, we see that by always expanding the node with the minimal g from $OPEN$ in a best-first-search manner, the first goal node we expand will have minimal cost. □

Heuristic Functions

We propose two heuristic functions for PCS.

H1 is a simple heuristic that sums the lengths of the shortest path (while ignoring constraints) of each agent not yet in the node ($\sum_{i>|M|} cost(minPath(i))$). This information can be cached, and then quickly retrieved.

Lemma 1. *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 ($\sum_{i>|M|} cost(minPath(i, constraints))$). 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.

Lemma 2. *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.*

Experimental Results

We implemented PCS with H1 and H2, PPR*, and PP.[2] The first iteration for PPR* was made to run identically to PP. We experimented using a grid-based MAPF benchmark from Stern et al. (2019). For each map, we used 25 different scenarios, each with 7 PC-MAPF problem instances. In each instance, we varied the number of agents from 10 to 40, increasing by 5. We present results on 8 diverse maps from the benchmark, and provide full results in the supplementary materials.[2] We ran our experiments on a cluster of AMD EPYC 7702P CPUs, with 16GB RAM each.

[2] github.com/J-morag/MAPF/releases/tag/24.SoCS.PPwG

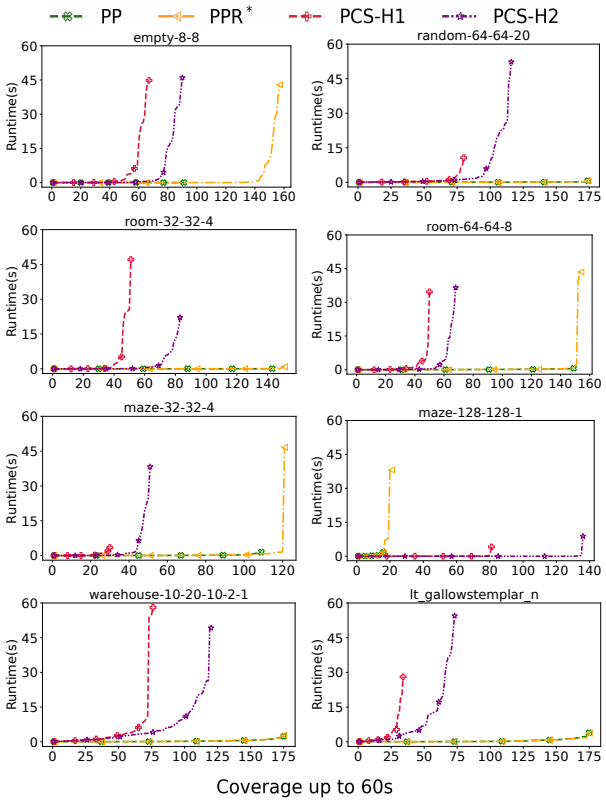

Figure 4: Max runtime per instance, as a factor of coverage.

Coverage up to runtime limit: Figure 4 shows the maximal runtime of each algorithm (y-axis), as a factor of the number of successes (x-axis). We count a success if either the algorithm found a solution, or it proved that the problem is unsolvable. Naturally, as the only priority-complete algorithm, this is only possible for PCS. For PPR*, the runtime until the first solution was found is used, since it always uses all the allowed runtime to attempt to improve the solution.

Comparing PCS with the two proposed heuristics, PCS-H2 solves more problems in less time. For example, in map warehouse-10-20-10-2-1, PCS-H2 achieves 121 successes, whereas PCS-H1 only succeeded on 77 problems. As could be expected, PP and PPR* take the same amount of runtime to find the first solution, but in cases where PP fails to find a solution, PPR* continues to run and often eventually finds a solution. For example, PPR* solved around 10% more instances than PP in maze-32-32-4. Clearly, PCS-H2 had less successes than PPR* on most maps, and often required more time. However, where many of the instances were unsolvable, like maze-128-128-1, a maze map comprised of many corridors that are one vertex wide, PCS-H2 was able to succeed much more often. This is because it was able to prove that many of the instances were unsolvable.

Solution Costs of Different Algorithms: We compare the solution costs of PP, PPR* and PCS in Figure 5. For PCS

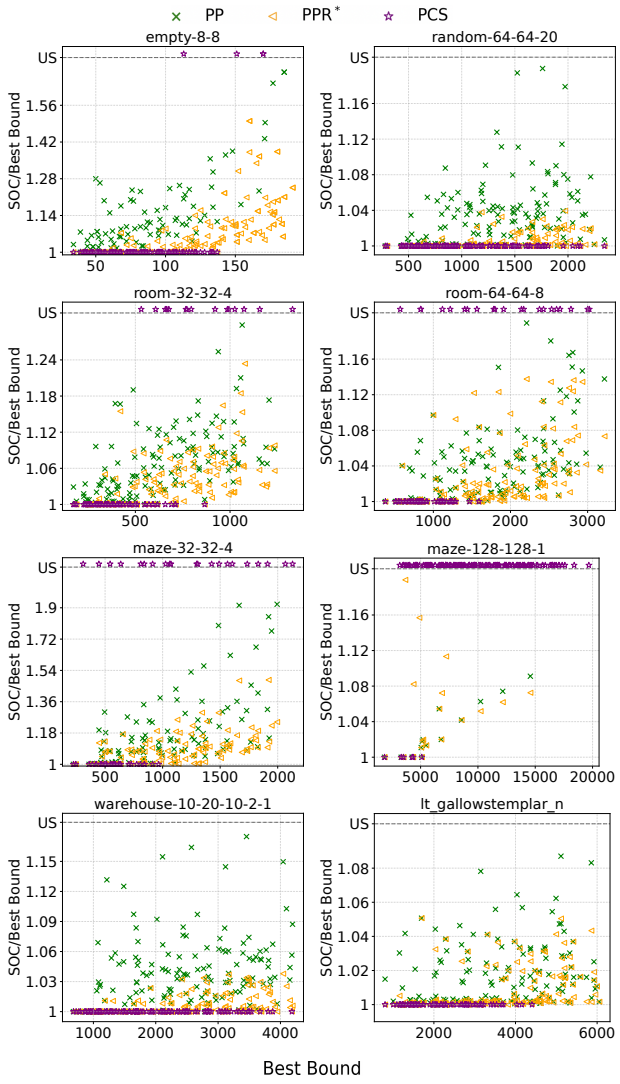

Figure 5: The cost of solutions, relative to the best lower bound. 'US' refers instances proven unsolvable.

heuristics, we use H2, as the previous experiment suggests it is always better to use it. The x-axis is the *best bound* on the cost of each PC-MAPF problem - when the optimal cost is known (PCS solved the problem), we use that as the bound, and when it is not known, we use the best known lower-bound (or optimal cost) for the corresponding MAPF problem, extracted from an online tracker (Shen et al. 2023). The MAPF bound is also a lower bound on the priority-optimal cost. 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. If an algorithm failed to solve an instance, it simply has no marker that corresponds to that instance. Instances that PCS found to be unsolvable are shown at the top of the plot, labelled 'US'.

In comparing PP and PCS, it is clear to see that PP often finds solutions with a significantly higher cost. For example, in empty-8-8, on instances solved by both PCS and PP, the 95th percentile of the ratio of their costs was 1.24. PCS was also able to prove some problem instances were unsolvable. This was especially prominent on maze-128-128-1, where most algorithms failed to find solutions, but PCS was able to prove that 75 of the 175 problems were in fact unsolvable. PPR* provided a balance of solution quality and scalability. It was able to find solutions with significantly lower SOC than those found by PP, though often not as low as those found by PCS. For example, on maze-32-32-4, the 95th percentile for the ratio of SOC/bound for PP solutions was 1.19, whereas the 95th percentile for PPR* solutions was only 1.06. PPR* was also able to solve many problems that both PP and PCS failed to.

We also compared the difference between the priority-optimal cost (PCS) and the optimal cost of the analogous MAPF problems. Results (omitted for brevity) show the gap is usually quite small - less than 10% in our experiments. As could be expected, these differences were smaller in large and open maps, where agents have many possible optimal paths. In dense environments, there were larger differences. Note that in such cases, the MAPF solution is not a feasible PC-MAPF solution, and the lowered cost is due to agents not being required to respect the priority constraints.

Conclusion and Future Work

In this work, we studied Priority-Constrained Multi-Agent Path Finding (PC-MAPF), a problem where agents must be planned according to a specific priority ordering. Solutions to PC-MAPF are required by important industrial applications. Meanwhile, planning agents subject to any priority order is a widely popular method to simplify and tackle MAPF. Both problems can be solved using a broad family of Prioritised Planning (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 gave a first analysis characterising the complexity of PC-MAPF, and showed that it is in NP-hard. We then presented Priority Constrained Search (PCS), the first PC-MAPF algorithm with theoretical guarantees on optimality and completeness. We evaluated PCS and PP on a range of common MAPF problems, and showed that PCS can succeed where PP fails. We gave tighter bounds for the quality of PC-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, PCS can help explain the reasons for that failure, and when PP succeeds, PCS can be used to improve solution quality.

Future work could consider applying speed-up techniques from the CBS family of algorithms to PCS. Additionally, we believe that relaxing the termination criteria of PCS can lead to new types of PP algorithms with different tradeoffs and guarantees. Another interesting future direction is extending our results to the more general problem of priority-optimal planning under a partial order or under any total order.

Acknowledgements

This work was performed while Jonathan Morag was on a research visit to Monash University.

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

References

Bennewitz, M.; Burgard, W.; and Thrun, S. 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.

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

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

Erdmann, M. A.; and Lozano-Pérez, T. 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*, 1419–1424. IEEE.

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

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

Li, J.; Harabor, D.; Stuckey, P. J.; Felner, A.; Ma, H.; and Koenig, S. 2019. Disjoint splitting for multi-agent path finding with conflict-based search. In *Proceedings of the international conference on automated planning and scheduling*, volume 29, 279–283.

Li, J.; Hoang, T. A.; Lin, E.; Vu, H. L.; and Koenig, S. 2023. Intersection Coordination with Priority-Based Search for Autonomous Vehicles. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 11578–11585.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with consistent prioritization for multi-agent path finding. In *AAAI*, volume 33, 7643–7650.

Phillips, M.; and Likhachev, M. 2011. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE international conference on robotics and automation*, 5628–5635. IEEE.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial intelligence*, 195: 470–495.

Shen, B.; Chen, Z.; Cheema, M. A.; Harabor, D. D.; and Stuckey, P. J. 2023. Tracking Progress in Multi-Agent Path Finding.

Silver, D. 2005. Cooperative Pathfinding. In *AIIDE*.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *SoCS*, 151–158.

Van Den Berg, J. P.; and Overmars, M. H. 2005. Prioritized motion planning for multiple robots. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 430–435. IEEE.

Yu, J.; and LaValle, S. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, 1443–1449.

Zhang, C. 2010. *Improving crane safety by agent-based dynamic motion planning using UWB real-time location system*. Ph.D. thesis, Concordia University.

Zhang, C.; and Hammad, A. 2012. Improving lifting motion planning and re-planning of cranes with consideration for safety and efficiency. *Advanced Engineering Informatics*, 26(2): 396–410.