Learning a Priority Ordering for Prioritized Planning in Multi-Agent Path Finding

Shuyang Zhang, Jiaoyang Li, Taoan Huang, Sven Koenig, and Bistra Dilkina
University of Southern California, Los Angeles, USA
{zhan270, jiaoyanl, taoanhua, skoenig, dilkina}@usc.edu

Abstract

Prioritized Planning (PP) is a fast and popular framework for solving Multi-Agent Path Finding, but its solution quality depends heavily on the predetermined priority ordering of the agents. Current PP algorithms use either greedy policies or random assignments to determine a total priority ordering, but none of them dominates the others in terms of the success rate and solution quality (measured by the sum-of-costs). We propose a machine-learning (ML) framework to learn a good priority ordering for PP. We develop two models, namely ML-T, which is trained on a total priority ordering, and ML-P, which is trained on a partial priority ordering. We propose to boost the effectiveness of PP by further applying stochastic ranking and random restarts. The results show that our ML-guided PP algorithms outperform the existing PP algorithms in success rate, runtime, and solution quality on small maps in most cases and are competitive with them on large maps despite the difficulty of collecting training data on these maps.

1 Introduction

Multi-Agent Path Finding (MAPF) is the problem of finding a set of collision-free paths for a group of agents in a shared environment. The objective of MAPF is to minimize the sum-of-costs (the sum of the arrival times of all agents at their goal locations, also known as flowtime) or the makespan (the largest arrival time of any agent at its goal location). Finding an optimal solution for MAPF is NP-hard (Yu and LaValle 2013; Surynek 2010). MAPF has many real-world applications in warehouse management (Wurman, D'Andrea, and Mountz 2008), airport surface operations (Morris et al. 2016), autonomous vehicles (Veloso et al. 2015), video games (Ma et al. 2017), and other multi-agent systems.

Prioritized planning (PP) (Silver 2005) is one of the fastest algorithms for solving MAPF suboptimally. It is based on a simple planning scheme (Erdmann and Lozano-Perez 1987): We assign each agent a unique priority and compute, in descending priority ordering, each agent's shortest path that avoids collisions with both static obstacles and the already-planned agents (moving obstacles). Because of its computational efficiency, scalability, and simplicity, PP remains the most commonly-adopted MAPF algorithm in practice. However, its solution quality is sensitive to the predetermined total priority ordering. Good priority orderings can yield (near-)optimal solutions, whereas bad priority orderings can lead to solutions with large sums-of-costs or even failures to find any solution for solvable MAPF instances, as shown in Figure 1.

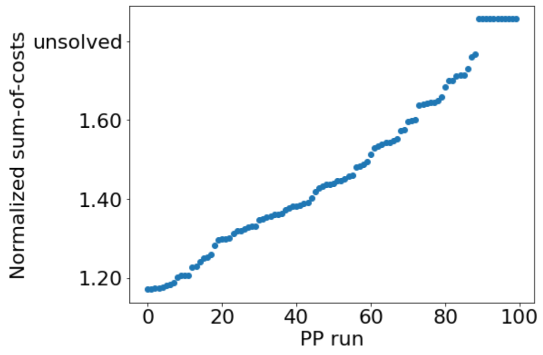


Figure 1: Normalized sum-of-costs (= the ratio of the sum-of-costs of the solution over the sum of the lengths of the shortest paths of all agents) of 100 PP runs with different random priority orderings on MAPF instance "room-32-32-4-random-1.scen" from (Stern et al. 2019) with 20 agents, sorted by their normalized sums-of-costs. PP runs that fail to find any solutions are shown on top of the plot.

Existing PP algorithms use either randomized assignments or greedy heuristics to determine the priority ordering, such as the query-distance heuristic (van den Berg and Overmars 2005), least-option heuristic (Wang et al. 2019; Wu, Bhattacharya, and Prorok 2020), and start-and-goal-conflict heuristic (Buckley 1989; Li et al. 2019). However, these hand-crafted heuristics have been developed in the context of specific usage scenarios, and none of them dominates the others in all cases in terms of the success rate and solution quality (measured by the sum-of-costs).

To fill the gap, we propose a Machine-Learning (ML) framework to learn good priority orderings in this paper. In the first phase, we extract features from a set of MAPF instances. We then run PP for a large number of times with different priority orderings and generate labels derived from the priority orderings that yield the best solutions. In the second phase, we use supervised learning to obtain a ranking function that produces a priority score for each agent. We propose two models, namely ML-T, which is trained on a total priority ordering, and ML-P, which is trained on a partial priority ordering. In the last phase, we use the learned ranking functions to find total priority orderings for unseen MAPF instances and compare the results with those of the benchmark PP algorithms. Besides exploring deterministic ranking, we also apply stochastic ranking and random restarts to all PP algorithms in our portfolio. Our results show that our ML-guided PP algorithms significantly outperform the existing PP algorithms in success rate, runtime, and sum-of-costs in almost all cases on small maps, especially in "hard" scenarios with large numbers of agents where a solution is extremely difficult to find with existing PP algorithms. Our ML-guided PP algorithms are also competitive with the existing PP algorithms on large maps despite the difficulty in collecting training data on these maps.

2 Problem Definition

The *Multi-Agent Path Finding (MAPF) problem* is to find a set of conflict-free paths for a set of agents $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ on an unweighted undirected graph $G = (V, E)$. Each agent has a start location $s_i \in V$ and a goal location $g_i \in V$. Time is discretized into time steps. Agents can either move to adjacent locations or wait at their current locations at each time step. We consider two types of conflicts, namely vertex conflicts, that occur when two agents are at the same location at the same time step, and edge conflicts, that occur when two agents cross the same edge in opposite directions at the same time step. The cost of agent a_i (or, synonymously, the path length of a_i) is the number of time steps needed for a_i to complete its path from s_i to g_i and remain motionless at g_i. The sum-of-costs is the sum of the costs of all agents. A solution is a set of conflict-free paths that move all agents from their start locations to their goal locations. Our objective is to find a solution that minimizes the sum-of-costs. In this paper, we assume that G is always a four-neighbor grid, which we also refer to as a map.

Definition 2.1 (Graph distance)**.** The *graph distance* $dist(u, v)$ between two locations $v \in V$ and $u \in V$ is the length of a shortest path that connects both locations.

Definition 2.2 (MDD)**.** An MDD (Sharon et al. 2013) MDD_i for agent a_i is a directed acyclic multi-level graph consisting of all shortest paths from location s_i to location g_i. Level t of MDD_i contains all possible locations that a_i can occupy at time step t along one of its shortest paths. In particular, levels 0 and $dist(s_i, g_i)$ (i.e., the first and last levels) contain only s_i and g_i, respectively. The width of a level is the number of locations in that level.

Definition 2.3 (Cardinal conflicts)**.** A vertex (edge) conflict between agents a_i and a_j is cardinal iff the contested vertex (edge) is the only vertex (edge) in the MDDs of both agents at some level (between two consecutive levels).

3 Background and Related Work

In this section, we first introduce prioritized planning, its challenges, and the commonly-used prioritization rules. We then briefly introduce some recent work on ML for MAPF.

3.1 Prioritized Planning

Definition 3.1 (Priority ordering)**.** A *priority ordering* $\prec$ is a strict partial order on $\mathcal{A}$: $a_i \prec a_j$ iff agent a_i has higher priority than agent a_j (Ma et al. 2019). $\prec$ is a *total* priority ordering iff any two agents in $\mathcal{A}$ are comparable (i.e., either $a_i \prec a_j$ or $a_j \prec a_i$ for all $a_i \neq a_j$) and a *partial* priority ordering otherwise.

Prioritized Planning (PP) (Erdmann and Lozano-Perez 1987) is a decoupled approach to solving MAPF. In PP, we arrange all agents into a predefined total priority ordering. Then, we plan paths for all agent one by one in descending order according to the priority ordering. The path of each agent is a shortest path that has no conflicts with the paths of all higher-priority agents. So, instead of planning paths for all agents at once, PP decouples the planning process and plans for the agents sequentially.

PP does not guarantee completeness or optimality, but it is popular because of its efficiency and simplicity. A key consideration in PP is how to determine the predefined total priority ordering. It is typically determined either randomly or via manually designed heuristics.

Query-distance heuristic. Van den Berg and Overmars (2005) proposed the query-distance heuristic, which measures the start-goal graph distance $dist(s_i, g_i)$ of each agent a_i and assigns higher priority to agents with longer distances. The motivation behind this heuristic was to prioritize agents that need to travel longer distances and thus minimize the makespan (i.e., the largest cost of all agents). An opposite version of the query-distance heuristic, which assigns higher priority to agents with shorter start-goal graph distances, has been used in (Ma et al. 2019).

Least-option heuristic. Building on the idea behind the most-constrained-variable heuristic for solving constraint satisfaction problems, Wang et al. (2019) and Wu, Bhattacharya, and Prorok (2020) proposed the least-option heuristic, which assigns higher priority to agents with fewer paths options, where the number of path options for an agent is defined as the number of conflict-free paths within given time steps in (Wang et al. 2019) and the number of homology classes of paths in (Wu, Bhattacharya, and Prorok 2020).

Start-and-goal-conflict heuristic. Buckley (1989) proposed prioritization rules that consider the potential conflicts at the start and goal locations of the agents. Intuitively, if the shortest path of agent a_i visits the start location of another agent a_j, then a_j needs to be planned prior to a_i; if the shortest path of agent a_i visits the goal location of another agent a_j, then a_i needs to be planned prior to a_j. This heuristic tends to reduce the runtime of PP (Buckley 1989) and increase its success rate (van den Berg et al. 2009).

Random restarts. When the priority ordering is assigned randomly, researchers often apply random restarts to improve the performance of PP (Bennewitz, Burgard, and Thrun 2002). When PP with a particular priority ordering fails to find a solution for a MAPF instance, we can "restart" it with a new randomized priority ordering.

3.2 Related Work on ML for MAPF

Recently, ML techniques have been applied to MAPF. Kaduri, Boyarski, and Stern (2020), Ren et al. (2021), and Ewing et al. (2022) proposed algorithm selection models that learn to select the fastest algorithm to solve MAPF problems optimally. Huang, Dilkina, and Koenig (2021a,b) and Huang et al. (2022) applied ML techniques to speed up MAPF algorithms such as Conflict-Based Search and its variants. We leverage their insights into feature crafting and ML methodologies and propose a data-driven ML framework for learning priority orderings for PP. Compared to ML frameworks that use deep learning, our approach is simple in both structure and implementation and requires only a small to medium training dataset to obtain good test results.

4 Machine-Learning Methodology

In this section, we introduce our supervised learning framework to learn a priority ordering for PP to solve MAPF. The idea is to train the ML model on hand-crafted features and labels such that it produces a score for each agent. We then use this score to rank the agents to produce a total priority ordering. To begin with, we define some terminology used in our subsequent explanation.

Definition 4.1 (Example)**.** An *example* D_I is a single data point derived from a MAPF instance I that consists of a feature vector for each agent and a label.

Definition 4.2 (Dataset)**.** A *dataset* is a set of examples.

Our ML pipeline consists of three phases:

1. Data collection: We obtain two sets of MAPF instances, namely training MAPF instances $\mathcal{I}_{\text{Train}}$ and test MAPF instances $\mathcal{I}_{\text{Test}}$, and a training dataset $\mathcal{D}_{\text{Train}} = \{D_I \mid I \in \mathcal{I}_{\text{Train}}\}$. See Section 5.
2. Model learning: We use $\mathcal{D}_{\text{Train}}$ to train a ranking function that produces a score for each agent such that the priority orderings derived from the scores are as "similar" to the priority orderings derived from the labels in $\mathcal{D}_{\text{Train}}$ as possible. See Section 6.
3. ML-guided search: We use the learned ranking function to generate total priority orderings for PP to solve MAPF instances in $\mathcal{I}_{\text{Test}}$. See Section 7.

5 Data Collection

The first task in our ML pipeline is to generate the training and test datasets. We collect two sets of MAPF instances, namely training MAPF instances $\mathcal{I}_{\text{Train}}$ and test MAPF instances $\mathcal{I}_{\text{Test}}$, on the same map with the same number of agents. For each MAPF instance $I \in \mathcal{I}_{\text{Train}}$, we generate an example D_I that consists of (i) a p-dimensional feature vector Φ_I^i for each agent $a_i \in \mathcal{A}$ that describes agent a_i with p features and (ii) a total/partial priority ordering $\prec_I$, which is used to compute the label for the ML models.

5.1 Feature Vector

We collect a p-dimensional feature vector Φ_I^i for each agent a_i and each example $I \in \mathcal{I}_{\text{Train}}$. The $p = 26$ features in our implementation are summarized in Table 1 and can be classified into four categories.

Start-goal distances. Motivated by the query-distance heuristic (van den Berg and Overmars 2005), we design 4 features about the graph and Manhattan distances between the start and goal locations of a_i (Feature 1). We also generalize this idea to looking at the graph distances between the start/goal locations of a_i and those of the other agents (Features 2 and 3).

MDD. Motivated by the least-option heuristic, we design 5 features about MDD_i (Features 4-6) because MDD_i captures information about the path options of a_i.

Start and goal locations. Motivated by the start-and-goal-conflict heuristic, we design 4 features about the potential conflicts at the start or goal locations of the other agents that a_i might be involved in, namely potential conflicts between a_i and another agent a_j when a_i is at its start or goal location and a_j follows (one of) its shortest paths (Features 7 and 8) or when a_j is at its start or goal location and a_i follows (one of) its shortest paths (Features 9 and 10).

Conflicts. We finally design 7 features about conflicts of different types (Feature 11) and potential conflicts (Feature 12) a_i might be involved in. In particular, Feature 11 counts the number of each type of conflicts that a_i can be involved in if all agents follow their shortest paths. Feature 12 counts the number of potential vertex conflicts that a_i can be involved in if all agents follow their shortest paths but wait for some time steps along their paths. We reason about the conflicts using these two methods because they can be easily computed by reasoning about the MDDs of the agents.

Normalization. We normalize the value of each feature between 0 and 1 across all agents. Since all our features have non-negative values by construction, we normalize them using a simple Min-Max normalization method.

5.2 Priority Ordering

To obtain the priority ordering $\prec_I$ for a given MAPF instance I, we first run PP repeatedly for x times with different total priority orderings (e.g., randomly generated total priority orderings) on MAPF instance I. We then use two different methods to construct $\prec_I$: One method is based on a total priority ordering, and the other one is based on a partial priority ordering.

In the first method, we set $\prec_I$ to the total priority ordering that generates the solution with the smallest sum-of-costs among the x runs. This method is simple and straightforward but has two drawbacks. First, the total priority ordering may be arbitrary in places. For example, if agents a_i and a_j are located far away from each other and do not collide with each other, then it does not matter which agent has the higher priority. Second, the total priority ordering is based on a single example, which may not be sufficiently robust.

In the second method, we therefore collect the $k \geq 1$ samples that result in the smallest sums-of-costs and generate a partial priority ordering by imposing an ordering on two agents only if the reverse ordering may change the sums-of-costs substantially. The method works as follows:

For each PP run $p = 1, ..., x$, we start with an empty partial priority ordering $\prec_I^p$. Each iteration of PP calls (space-

Index	Feature description	Count
1	Graph and Manhattan distances between s_i and g_i: their respective values, absolute difference, and the ratio of the graph distance over the Manhattan distance	4
2	Graph distance between s_i and the start locations of the other agents: their max., min., and mean	3
3	Graph distance between g_i and the goal locations of the other agents: their max., min., and mean	3
4	Sum of the widths of all levels of MDD_i	1
5	Width of each level (excluding the first and the last levels) of MDD_i: their max., min., and mean	3
6	Number of unit-width levels of MDD_i	1
7	Number of the other agents whose MDDs contain s_i	1
8	Number of the other agents whose MDDs contain g_i	1
9	Number of the other agents whose start locations are in MDD_i	1
10	Number of the other agents whose goal locations are in MDD_i	1
11	Number of vertex, edge, and cardinal conflicts between any shortest path of a_i and any shortest path of one of the other agents: counted by agent pair or counted by raw conflict count	6
12	Number of locations in MDD_i that are also in the MDD of at least one other agent	1

Table 1: $p = 26$ features for agent a_i. Column "Count" reports the numbers of features contributed by the corresponding entries.

time) A* (Silver 2005) to plan a shortest path for a single agent a_i that avoids conflicts with the already-planned paths. When this A* search generates an A* node n with an f-value of f_n that moves a_i from one location to another, it checks if this move action leads to a conflict with an already-planned path, say, that of agent a_j, and prunes node n if so. We record such pruned nodes, i.e, we record the pair (a_j, f_n). When the A* search terminates and returns a path of length l_i for a_i, we collect the set of agents $\mathcal{A}^H$ in the recorded pairs whose corresponding f_n values are smaller than l_i and add $a_j \prec a_i$ for all $a_j \in \mathcal{A}^H$ to $\prec_I^p$, for the following reason: If any agent in $\mathcal{A}^H$ had lower priority than a_i, then A* might find a path of length within $[f_n, l_i)$ for a_i, i.e., it might find a shorter path than the current one. In contrast, even if all agents not in $\mathcal{A}^H$ had lower priority than a_i, A* still cannot find a shorter path.

When we select the top k samples, we collect the associated partial priority orderings $\prec_I^p$ for $p = 1, \ldots, k$ and combine them into a joint partial priority ordering $\prec_I$. To do so, we first find all pairs of comparable agents in each $\prec_I^p$.[1] We then sort the agent pairs in descending order of their occurrences in the top k samples ($a_i \prec a_j$ and $a_j \prec a_i$ are treated as two different agent pairs) and add them one by one to $\prec_I$ whenever possible. That is, if the agent pair is $a_i \prec a_j$, we add it to $\prec_I$ iff agents a_i and a_j are not comparable in $\prec_I$. We also record the occurrences and use $\#(a_i \prec a_j)$ to represent how often $a_i \prec a_j$ occurs in the k priority orderings.

6 Model Learning

We learn a linear ranking function with parameters $\boldsymbol{w} \in \mathbb{R}^p$

$$f : \mathbb{R}^p \to \mathbb{R} : f(\Phi_I^i) = \boldsymbol{w}^{\boldsymbol{T}} \Phi_I^i$$

[1]Specifically, we convert $\prec_I^p$ to a directed acyclic graph H_I^p, where node i represents agent a_i and each directed edge $i \to j$ represents $a_i \prec_I^p a_j$. We run the Floyd-Warshall algorithm on H_I^p to find all connected agent pairs.

that minimizes the loss function

$$L(\boldsymbol{w}) = \sum_{I \in \mathcal{I}_{\text{Train}}} l(y_I, \hat{y_I}) + \frac{C}{2}||\boldsymbol{w}||_2^2.$$

Here, y_I is derived from $\prec_I$ that represents the ground-truth priority ordering (details are provided below), and $\hat{y_I}$ is a n-dimensional vector of the predicted scores, i.e., $\hat{y_I}(a_i) = f(\Phi_I^i)$, that represents the predicted priority ordering. $l(\cdot, \cdot)$ is a loss function that measures the difference between the ground-truth priority ordering and the predicted priority ordering (details are provided below), and $C > 0$ is a regularization parameter.

ML-T. The first model, ML-T, learns a form of the total priority ordering $\prec_I$ generated by the first method in Section 5.2. Since $\prec_I$ is noisy, we group the agents into m priority groups (where $m \in \mathbb{N}$ is a hyper-parameter) by setting $y_I(a_i) = \lfloor r_i/m \rfloor$, where r_i represents that agent a_i has the r_i-th lowest priority among all agents. That is, agents with larger y_I values are in higher priority groups, and agents with the same y_I value are in the same priority group. Similar labeling methods are used in (Khalil et al. 2016; Huang, Dilkina, and Koenig 2021b). The loss function $l(y_I, \hat{y_I})$ is defined to be the fraction of discordant pairs over all pairs of agents in different priority groups:

$$l(y_I, \hat{y_I}) = \frac{|\{(a_i, a_j) \in \mathcal{A}^2 \colon y_I(a_i) > y_I(a_j) \wedge \hat{y_I}(a_i) < \hat{y_I}(a_j)\}|}{|\{(a_i, a_j) \in \mathcal{A}^2 \colon y_I(a_i) > y_I(a_j)\}|}.$$

ML-P. The second model, ML-P, learns a pairwise partial priority ordering generated by the second method in Section 5.2. Here, we do not explicitly define y_I. Instead, we define the loss function $l(y_I, \hat{y_I})$ directly to be the fraction of the sum of the occurrence of discordant agent pairs in $\prec_I$ over the sum of the occurrence of agent pairs in $\prec_I$:

$$l(y_I, \hat{y_I}) = \frac{\sum_{(a_i, a_j) \in \mathcal{A}^2 \colon a_i \prec_I a_j \wedge \hat{y_I}(a_i) < \hat{y_I}(a_j)} \#(a_i \prec a_j)}{\sum_{(a_i, a_j) \in \mathcal{A}^2 \colon a_i \prec_I a_j} \#(a_i \prec a_j)}.$$

7 ML-Guided Search

After data collection and model learning, we apply the learned ranking function f to the feature vectors for each test MAPF instance $I \in \mathcal{I}_{\text{Test}}$. Based on the predicted scores $\hat{y_I} : \mathcal{A} \to \mathbb{R}^n$ returned by f, we propose two different methods to produce a total priority ordering.

Deterministic ranking. We rank the agents by their predicted scores, namely $a_i \prec a_j$ iff $\hat{y_I}(a_i) > \hat{y_I}(a_j)$.

Stochastic ranking. We use the predicted scores to produce a probability distribution and generate a total priority ordering sequentially from agents with high priority to agents with low priority. Specifically, we normalize the predicted scores $\hat{y_I}$ using the softmax function

$$\sigma : \mathbb{R}^n \to [0,1]^n : \sigma(\boldsymbol{z}) = \frac{(e^{\beta z_1}, \ldots, e^{\beta z_n})}{\sum_{j=1}^{n} e^{\beta z_j}} \qquad (1)$$

($\beta \in \mathbb{R}^+$ is a hyper-parameter), where we set $z_i = \hat{y_I}(a_i)$. We then repeatedly assign the next highest priority to an agent that is selected with a probability proportional to its normalized predicted score (where, of course, every agent can be selected only once). Agents with higher normalized scores have higher probabilities to be selected earlier and thus assigned higher priorities. This adds randomness to the total priority orderings and allows us to leverage the random restart scheme when experimenting with ML-guided PP.

8 Experimentation

We use two open-source software packages to build our two ML models, namely SVM$^{\text{rank}}$ (Joachims 2006), which implements a Support Vector Machine for ordinal classification and regression, for building **ML-T** and LIBLINEAR (Fan et al. 2008), which implements a Support Vector Machine for large-scale linear classification, for building **ML-P**. We compare our ML-guided PP algorithms against three baseline PP algorithms: (1) **LH**, a query-distance heuristic where agents with longer start-goal graph distances have higher priority (van den Berg and Overmars 2005); (2) **SH**, a query-distance heuristic where agents with shorter start-goal graph distances have higher priority (Ma et al. 2019); and (3) **RND**, a heuristic that generates a random total priority ordering (Bennewitz, Burgard, and Thrun 2002). We implement all algorithms in C++ with the same PP code base and run experiments on Ubuntu 20.04 LTS on an Intel Xeon 8175M processor with a memory limit of 8 GB.

Maps. We use a set of six maps $\mathcal{M}$, illustrated in Table 3, with different sizes and structures from the MAPF benchmark suite (Stern et al. 2019): (1) the random map "random-32-32-20" of size 32×32 with 20% randomly blocked cells and $|V| = 819$, (2) the room map "room-32-32-4" of size 32×32 with 64 square rooms connected by single-cell doors and $|V| = 682$, (3) the maze map "maze-32-32-2" of size 32×32 with two-cell-wide corridors and $|V| = 666$, (4) the warehouse map "warehouse-10-20-10-2-1" of size 161×63 with $|V| = 5{,}699$, (5) the first game map "lak303d" of size 194×194 with $|V| = 14{,}784$, and (6) the second game map "ost003d" of size 194×194 with $|V| = 13{,}214$. We refer to the first four maps as small maps and to the last two as large maps.

Training and test MAPF instances. We use 25 random scenarios from the MAPF benchmark suite for each map. A scenario is a list of $\min\{0.5|V|, 1{,}000\}$ randomly created pairs of start and goal locations. Following the literature, given a map $M \in \mathcal{M}$ and a number of agents n, we generate 25 test MAPF instances $\mathcal{I}_{\text{Test}}^{(M)}$, one from each scenario, by using the first n pairs of start and goal locations. In order to generate training MAPF instances that follow a similar distribution as the test MAPF instances, given a scenario with a map $M \in \mathcal{M}$ and a number of agents n, we generate a training MAPF instance $I \in \mathcal{I}_{\text{Train}}^{(M)}$ by randomly selecting n start locations from all start locations in the scenario, randomly selecting n goal locations from all goal locations in the scenario, and then randomly combining them into n pairs of start and goal locations. For ML-T, we generate 99 training MAPF instances from each of the 25 scenarios, so $|\mathcal{I}_{\text{Train}}^{(M)}| = 2{,}475$. For ML-P, we generate one training MAPF instance from each scenario since the training loss converges already for a small training dataset, so $|\mathcal{I}_{\text{Train}}^{(M)}| = 25$.

Training datasets. We run PP $x = 100$ times, once with LH, once with SH, and 98 times with RND, to solve each MAPF instance $I \in \mathcal{I}_{\text{Train}}^{(M)}$. We pick the PP run with the least sums-of-costs for ML-T and the top $k = 5$ PP runs with the least sums-of-costs for ML-P to generate the training example D_I. However, when we use small maps with large numbers of agents n, most of the 100 PP runs fail to find any solutions. We show in Sections 8.1 and 8.2 that our ML models often have higher success rates than LH, SH, and RND on small maps with small numbers of agents n. Therefore, when the success rate of the 100 PP runs is less than 5% for the given MAPF instances (i.e., on the random, room, and maze maps with $n \geq 200$, $n \geq 125$, and $n \geq 90$, respectively), we replace 10 of the 98 RND runs with ML-guided PP trained on a smaller number of agents on the same map (i.e., the number of agents shown on the previous row of the row in Tables 2 and 3 that corresponds to the map and the number of agents of the given MAPF instance). Specifically, we run ML-guided PP with random restarts with a runtime limit of 3 seconds (i.e., repeatedly run PP using the stochastic ranking method until a solution is found or the runtime limit is reached) in each run and always use the same ML model for training and testing (i.e., train ML-T with datasets partially generated by ML-T and train ML-P with datasets partially generated by ML-P). This is effective in gathering training data for large numbers of agents on all small maps except for the warehouse map, for which we did not use ML-guided PP to generate training datasets (because it did not result in higher success rates).

Training hyper-parameters. We varied the group size $m \in \{1, 5, 10\}$ for ML-T and picked $m = 5$ as it leads to the best results. We varied the regularization parameter $C \in \{0.1, 1, 10, 20, 100\}$ and picked $C = 20$ to train an SVM$^{\text{rank}}$ for model ML-T since there was no significant difference on the test results. We used the built-in cross-

Map	n	Success rate (%)					Solution rank				
		LH	SH	RND	ML-T	ML-P	LH	SH	RND	ML-T	ML-P
random	50	**96**	16	76	84	56	2.00	2.72	**1.24**	1.52	**1.24**
	100	**100**	20	60	32	24	1.20	1.68	**1.16**	1.32	1.68
	150	**68**	4	20	64	8	0.72	1.44	1.28	**0.68**	1.40
	200	24	0	0	**32**	24	0.44	0.80	0.80	**0.28**	0.44
	250	0	0	0	0	0	0.00	0.00	0.00	0.00	0.00
room	50	**88**	12	52	76	16	1.48	2.00	1.28	**0.72**	1.72
	75	**92**	0	8	0	44	**0.28**	1.44	1.36	1.44	0.72
	100	**60**	0	0	56	**60**	0.92	1.76	1.76	0.64	**0.56**
	125	**24**	0	0	16	20	**0.28**	0.60	0.60	0.44	0.32
	150	**4**	0	0	0	0	**0.00**	0.04	0.04	0.04	0.04
maze	50	**84**	0	12	76	68	1.32	2.40	2.12	**0.68**	0.96
	70	76	0	0	84	**88**	**0.84**	2.48	2.48	0.88	0.88
	90	64	0	0	52	**80**	0.96	1.96	1.96	0.84	**0.68**
	110	**44**	0	0	32	**44**	0.56	1.20	1.20	**0.52**	0.56
	130	**16**	0	0	8	**16**	**0.20**	0.40	0.40	0.32	**0.20**
warehouse	100	**92**	32	80	**92**	80	2.72	2.64	1.40	1.96	**0.64**
	200	**80**	28	56	36	60	1.80	1.36	1.56	1.44	**0.88**
	300	**52**	16	12	24	20	0.72	**0.68**	1.04	**0.68**	0.84
	400	**32**	4	8	12	24	**0.44**	0.64	0.64	0.60	**0.44**
	500	**12**	0	4	0	0	**0.04**	0.16	0.08	0.16	0.16
lak303d	300	**100**	28	96	96	76	2.64	2.64	1.96	**1.16**	1.24
	400	**100**	36	88	88	72	2.84	2.24	1.64	1.52	**1.08**
	500	**100**	44	88	84	80	2.76	1.76	1.48	2.24	**0.96**
	600	**88**	8	36	80	12	1.24	1.84	1.36	**0.72**	1.80
	700	16	0	0	**68**	0	0.64	0.84	0.84	**0.16**	0.84
ost003d	300	**100**	28	96	92	80	2.72	2.68	1.20	2.04	**1.04**
	400	88	32	**92**	84	**92**	2.84	2.44	2.04	1.04	**0.96**
	500	**96**	28	84	**96**	64	2.32	2.40	1.52	1.76	**1.20**
	600	**92**	16	60	84	40	1.76	2.16	**1.28**	1.32	1.44
	700	**72**	8	40	68	12	1.08	1.60	**0.88**	**0.88**	1.64

Table 2: Success rate and solution rank for deterministic ranking. The best results achieved among all algorithms are shown in bold. The results are obtained by training and testing on the same map with the same number of agents n, except for maps lak303d and ost003d with $n > 500$, where the results are obtained by training on the same map with $n = 500$.

validation function in LIBLINEAR to obtain the value of $C = 128$ to train an SVM for model ML-P (Fan et al. 2008).

Testing setups. We always train and test on the same map. For small maps, we train and test with the same number of agents n. For large maps, we are only able to gather training datasets for MAPF instances with $n \leq 500$ because the runtime for a single PP run with a larger n is too high. We therefore train and test with the same number of agents when $n \leq 500$ and use the ML models trained on MAPF instances with $n = 500$ to test on MAPF instances with $n > 500$. The runtime limit for testing is set to 1 minute for small maps and 10 minutes for large maps. We pre-compute the graph distances from each goal location to all locations on the map and use them as the admissible heuristics for the (space-time) A* search of PP for all the algorithms.

Metrics. We evaluate the algorithms with four metrics. *Success rate* is the percentage of solved test MAPF instances within the runtime limit. *Runtime to first solution* is the runtime needed to find the first solution, averaged over all 25 test MAPF instances, in which the runtime limit is used for unsolved instances. Here, we consider only the PP runtime and ignore the runtime overhead of generating the total priority orderings for PP because such runtime overhead for SH and LH is negligible as the start-goal graph distances are precomputed, and that for ML-T and ML-P are also considerably small due to the small ML runtime overhead.[2] *Normalized sum-of-costs* is the ratio of the sum-of-costs and the sum of the start-goal graph distances of all agents. *Solution rank* evaluates the relative solution quality as follows: For each test MAPF instance, we rank the algorithms in ascending order of the sums-of-costs of their solutions. The lower the sum-of-costs, the lower the numerical value of the ranking. The lowest numerical value of the ranking is 0. Algorithms that lead to the same sum-of-costs have the same ranking, which is set to the numerical value of the lowest ranking in the tie. For example, if the sums-of-costs of the 5 algorithms are [101, 101, 102, 103, 103], then their rankings are [0, 0, 2, 3, 3]. Algorithms that fail to solve the MAPF instances have the largest numerical value of the ranking. Solution rank is the average numerical value of the ranking over the 25 test MAPF instances.

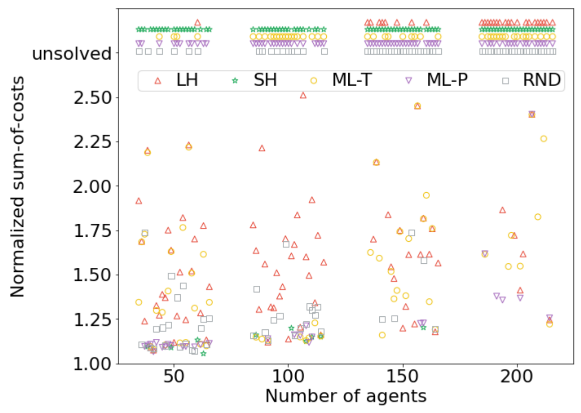


Figure 2: Normalized sum-of-costs for deterministic ranking on the random map. Unsolved MAPF instances are shown on top of the plot.

8.1 Deterministic Ranking

We first experiment with deterministic ranking for the baseline PP algorithms LH, SH, and RND and our ML-guided PP algorithms ML-T and ML-P on test MAPF instances on each of the six maps and vary the number of agents within a reasonable range. Here, each algorithm generates one total priority ordering and runs PP exactly once for each MAPF instance. (RND uses the first total priority ordering that the

[2]The ML runtime overhead mainly comes from the runtime for collecting features, which, for example, is 0.03 seconds, 0.01 seconds, 0.02 seconds, 0.7 seconds, 4.37 seconds, and 3.34 seconds per MAPF instance for the six maps with their respective largest numbers of agents tested in Table 3. Moreover, the features need to be collected only once for each MAPF instance even if we run PP multiple times via random restarts.

randomized algorithm generates). We report, in Table 2, the success rate and the solution rank for all maps and, in Figure 2, the normalized sum-of-costs for the random map.

In terms of success rate, the two ML-guided algorithms ML-T and ML-P achieve comparable results as but do not completely dominate the baseline algorithms. ML-T generally has a higher success rate than ML-P, but both are often more prone to failure than LH. In terms of solution quality, ML-T achives results comparable to RND. Although ML-P often fails to find a solution, once it does find one, it often finds a solution with lower sum-of-costs than the other algorithms, as shown in Figure 2. In other words, ML-P suffers from low success rates but yields promising solution qualities. We therefore consider the random restart technique to boost the success rate of the two ML-guided algorithms while preserving their solution qualities.

8.2 Stochastic Ranking with Random Restarts

We now illustrate stochastic ranking in conjunction with random restarts, which are applied to all five algorithms to ensure a fair comparison. To make random restarts possible, we add randomness to the deterministic algorithms LH and SH. LH relies on the start-goal graph distances of all agents to determine the total priority ordering. Therefore, for LH, we use the stochastic ranking method in Section 7 with $z_i = dist(s_i, g_i)$. For SH, since it is the reversed version of LH, we use the stochastic ranking method in Section 7 with $z_i = dist(s_i, g_i)$ and generate a total priority ordering from low to high (instead of high to low). We varied parameter $\beta \in \{0.1, 0.5, 1.0, 1.5\}$ in the softmax function and picked $\beta = 0.5$ for SH, LH, ML-T, and ML-P as it leads to the best results. RND is directly used with random restarts. We keep restarting each algorithm with a new random seed until the runtime limit is reached. We report, in Table 3, the success rate and the runtime to the first solution and, in Figure 3, the solution rank, where the solution, which we refer to as the final solution, is the one with the least sum-of-costs found within the runtime limit.

ML-T and ML-P outperform the baseline algorithms in terms of the success rate, runtime to the first solution, and sum-of-costs of the final solution on the random, room, and maze maps. ML-P has a slightly higher success rate and a better solution rank than ML-T. The advantage of our ML-guided algorithms is most apparent on these maps when the number of agents is large and a solution is hard to find with the baseline algorithms. On the warehouse map, ML-T and ML-P achieve results comparable to the baseline algorithms. On the large maps, ML-T achieves comparable success rates and solution ranks as the baseline algorithms, while ML-P has a marginally lower success rate but a better solution rank. These results, to some degree, are consistent with the difficulty of obtaining high-quality training datasets: As we described in the "training datasets" and "testing setups" paragraphs in the beginning of this section, it is difficult to get good training datasets on the warehouse map and the large maps due to both the low success rates of existing PP algorithms and their long runtimes. For example, the lak303d and ost003d maps are the only two maps on which we train and test using different numbers of agents. There-

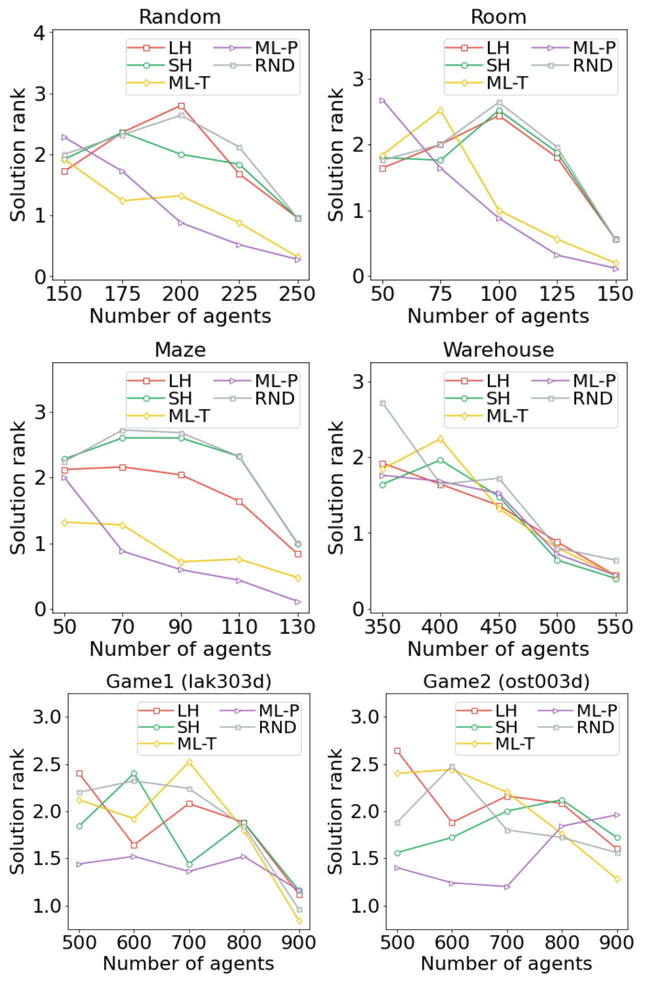


Figure 3: Solution rank for stochastic ranking with random restarts.

fore, the results demonstrate the limited ability of ML-P to generalize to a higher number of agents.

8.3 Feature Importance

We now analyze the feature importance of the learned ranking functions with good success rates and solution ranks, i.e., on the random, room, and maze maps, each with the largest number of agents, because these ranking functions most significantly outperform the baseline algorithms. We sort the feature weights $\boldsymbol{w}$ in decreasing order of their absolute values. Since the features are normalized, we use the absolute values of the feature weights to represent their importance.

The three ranking functions for ML-T, one for each map, have nine features in common among their top ten features with the largest absolute values: the graph and Manhattan distances between s_i and g_i and their absolute difference in Feature 1 (three features); the number of vertex conflicts counted by agent pair in Feature 11 (one feature); and Features 4, 6, 9, 10, and 12 (five features).

The three ranking functions for ML-P, one for each map, have five features in common among their top ten features

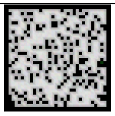
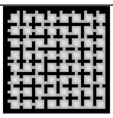
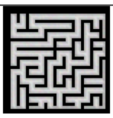
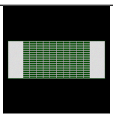
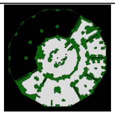
Map	n	Success rate (%)					Runtime to the first solution (seconds)				
		LH	SH	RND	ML-T	ML-P	LH	SH	RND	ML-T	ML-P
	150	100	100	100	100	100	**0.08**	0.65	0.42	0.24	1.37
	175	100	100	100	100	100	3.24	2.54	6.22	**1.06**	1.49
	200	88	80	88	**100**	**100**	15.60	20.56	18.25	**2.13**	2.86
	225	16	20	28	88	**92**	51.09	49.70	46.10	**8.82**	13.17
random	250	0	0	0	44	**52**	60.00	60.00	60.00	40.88	**33.79**
	50	100	100	100	100	100	0.24	0.17	0.15	**0.11**	0.65
	75	100	100	100	100	100	0.66	1.23	1.13	1.47	**0.52**
	100	84	80	76	**100**	**100**	12.56	22.70	23.07	3.35	**0.70**
	125	20	8	4	80	**88**	49.50	59.60	58.21	16.21	**10.18**
room	150	0	0	0	24	**32**	60.00	60.00	60.00	51.53	**44.57**
	50	100	100	100	100	100	**0.61**	3.86	2.19	1.30	1.28
	70	**100**	68	68	96	**100**	3.17	25.48	28.58	3.24	**0.49**
	90	68	16	16	**100**	**100**	22.81	55.27	54.18	2.84	**0.72**
	110	44	0	0	92	**96**	33.67	60.02	60.01	15.22	**9.90**
maze	130	12	0	0	36	**52**	52.85	60.01	60.02	42.78	**33.90**
	350	96	96	96	**100**	92	**9.67**	13.62	13.18	13.45	13.43
	400	80	**84**	72	**84**	76	26.00	25.24	26.80	**24.39**	29.06
	450	**68**	48	52	60	48	**35.80**	39.67	39.45	37.10	40.91
	500	24	28	20	20	**32**	52.86	49.96	52.50	53.06	**49.11**
warehouse	550	12	8	8	**24**	12	58.29	56.73	59.54	**56.11**	56.63
	500	**100**	96	**100**	**100**	**100**	**26.74**	65.87	62.98	43.78	98.97
	600	**100**	96	92	96	88	**58.18**	117.80	135.97	100.26	174.64
	700	**100**	96	88	88	84	**99.51**	140.22	230.52	257.10	270.98
	800	**100**	68	76	68	56	**192.71**	397.20	395.42	381.47	451.95
lak303d	900	**68**	32	32	36	16	**423.82**	565.70	536.69	541.53	584.13
	500	**100**	**100**	96	96	92	**15.68**	49.15	60.60	53.10	87.81
	600	**100**	96	96	96	92	**43.07**	85.94	93.92	94.57	136.83
	700	**96**	92	92	92	88	**77.55**	163.39	205.10	468.27	232.78
	800	**100**	84	84	92	72	**126.35**	287.04	299.36	263.53	369.30
ost003d	900	**88**	64	64	72	40	**273.00**	462.16	456.27	420.71	525.51

Table 3: Success rate and runtime to the first solution for stochastic ranking with random restarts. The best results achieved among all algorithms are shown in bold. The results are obtained by training and testing on the same map with the same number of agents n, except for maps lak303d and ost003d with $n > 500$, where the results are obtained by training on the same map with $n = 500$.

with the largest absolute values: the graph distance between s_i and g_i and the absolute difference between the graph and Manhattan distances between s_i and g_i in Feature 1 (two features); and Features 4, 6, and 10 (three features).

Taking the intersection between the most important features for ML-T and ML-P, we determine the most important features to be Features 1, 4, 6, and 10, which correspond to the query-distance heuristic (Feature 1), the least-option heuristic (Features 4 and 6), and the start-and-goal-conflict heuristic (Features 10). This indicates that our learned ranking functions cleverly combine the strengths of the existing heuristic methods.

9 Conclusions and Future Directions

In this paper, we proposed the first ML framework for learning priority orderings for Prioritized Planing (PP) in the context of MAPF. We developed two models: model ML-T learns from a total priority ordering, and model ML-P learns from a partial priority ordering. Our experimental results showed that both models significantly outperform existing PP algorithms on small maps, especially in difficult scenarios with a large number of agents. ML-T and ML-P still achieve results comparable to the existing PP algorithms on large maps despite the lack of high-quality training data. Going forward, we are interested in training our model on different numbers of agents instead of a fixed number of agents, which may enable our model to generalize better across different numbers of agents. We are also interested in developing a new model that is able to learn features related to the size and structure of the map so that it can generalize across different maps.

Acknowledgments

Shuyang Zhang was partially supported by a University of Southern California's Center for Undergraduate Research in Viterbi Engineering (CURVE) Fellowship. The research was also supported by the National Science Foundation under grant numbers 1409987, 1724392, 1817189, 1837779, 1935712, and 2112533 as well as a gift from Amazon. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

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.

Buckley, S. J. 1989. Fast motion planning for multiple moving robots. In *IEEE International Conference on Robotics and Automation*, 322–326.

Erdmann, M.; and Lozano-Perez, T. 1987. On multiple moving objects. *Algorithmica*, 2(1): 477–521.

Ewing, E.; Ren, J.; Kansara, D.; Sathiyanarayanan, V.; and Ayanian, N. 2022. Betweenness centrality in multi-agent path finding. In *International Conference on Autonomous Agents and Multiagent Systems*, 400–408.

Fan, R.-E.; Chang, K.-W.; Hsieh, C.-J.; Wang, X.-R.; and Lin, C.-J. 2008. LIBLINEAR: A library for large linear classification. *The Journal of Machine Learning Research*, 9: 1871–1874.

Huang, T.; Dilkina, B.; and Koenig, S. 2021a. Learning node-selection strategies in bounded-suboptimal conflict-based search for multi-agent path finding. In *International Conference on Autonomous Agents and Multiagent Systems*, 611–619.

Huang, T.; Dilkina, B.; and Koenig, S. 2021b. Learning to resolve conflicts for multi-agent path finding with conflict-based search. In *AAAI Conference on Artificial Intelligence*, 11246–11253.

Huang, T.; Li, J.; Koenig, S.; and Dilkina, B. 2022. Anytime multi-agent path finding via machine learning-guided large neighborhood search. In *AAAI Conference on Artificial Intelligence*.

Joachims, T. 2006. Training linear SVMs in linear time. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 217–226.

Kaduri, O.; Boyarski, E.; and Stern, R. 2020. Algorithm selection for optimal multi-agent pathfinding. In *International Conference on Automated Planning and Scheduling*, 161–165.

Khalil, E. B.; Le Bodic, P.; Song, L.; Nemhauser, G. L.; and Dilkina, B. 2016. Learning to Branch in Mixed Integer Programming. In *AAAI Conference on Artificial Intelligence*, 724–731.

Li, H.; Long, T.; Xu, G.; and Wang, Y. 2019. Coupling-degree-based heuristic prioritized planning method for UAV swarm path generation. In *Chinese Automation Congress*, 3636–3641.

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

Ma, H.; Yang, J.; Cohen, L.; Kumar, T.; and Koenig, S. 2017. Feasibility study: Moving non-homogeneous teams in congested video game environments. In *Artificial Intelligence and Interactive Digital Entertainment Conference*, 270–272.

Morris, R.; Pasareanu, C. S.; Luckow, K. S.; Malik, W.; Ma, H.; Kumar, T. K. S.; and Koenig, S. 2016. Planning, scheduling and monitoring for airport surface operations. In *AAAI Workshop on Planning for Hybrid Systems*, 608–614.

Ren, J.; Sathiyanarayanan, V.; Ewing, E.; Senbaslar, B.; and Ayanian, N. 2021. MAPFAST: A deep algorithm selector for multi agent path finding using shortest path embeddings. In *International Conference on Autonomous Agents and Multiagent Systems*, 1055–1063.

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.

Silver, D. 2005. Cooperative pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment Conference*, 117–122.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Barták, R.; and Boyarski, E. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Annual Symposium on Combinatorial Search*, 151–158.

Surynek, P. 2010. An optimization variant of multi-robot path planning is intractable. In *AAAI Conference on Artificial Intelligence*, 1261–1263.

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

van den Berg, J. P.; Snoeyink, J.; Lin, M. C.; and Manocha, D. 2009. Centralized path planning for multiple robots: Optimal decoupling into sequential plans. In *Robotics: Science and Systems V*, 2–3.

Veloso, M.; Biswas, J.; Coltin, B.; and Rosenthal, S. 2015. Cobots: Robust symbiotic autonomous mobile service robots. In *International Joint Conference on Artificial Intelligence*, 4423–4429.

Wang, J.; Li, J.; Ma, H.; Koenig, S.; and Kumar, S. 2019. A new constraint satisfaction perspective on multi-agent path finding: Preliminary results. In *International Conference on Autonomous Agents and Multiagent Systems*, 2253–2255.

Wu, W.; Bhattacharya, S.; and Prorok, A. 2020. Multi-robot path deconfliction through prioritization by path prospects. In *IEEE International Conference on Robotics and Automation*, 9809–9815.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1): 9–9.

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