Anytime Multi-Agent Path Finding via Machine Learning-Guided Large Neighborhood Search

Taoan Huang, Jiaoyang Li, Sven Koenig, Bistra Dilkina

University of Southern California
{taoanhua, jiaoyanl, skoenig, dilkina}@usc.edu

Abstract

Multi-Agent Path Finding (MAPF) is the problem of finding a set of collision-free paths for a team of agents in a common environment. MAPF is NP-hard to solve optimally and, in some cases, also bounded-suboptimally. It is thus time-consuming for (bounded-sub)optimal solvers to solve large MAPF instances. Anytime algorithms find solutions quickly for large instances and then improve them to close-to-optimal ones over time. In this paper, we improve the current state-of-the-art anytime solver MAPF-LNS, that first finds an initial solution fast and then repeatedly replans the paths of subsets of agents via Large Neighborhood Search (LNS). It generates the subsets of agents for replanning by randomized destroy heuristics, but not all of them increase the solution quality substantially. We propose to use machine learning to learn how to select a subset of agents from a collection of subsets, such that replanning increases the solution quality more. We show experimentally that our solver, MAPF-ML-LNS, significantly outperforms MAPF-LNS on the standard MAPF benchmark set in terms of both the speed of improving the solution and the final solution quality.

1 Introduction

Multi-Agent Path Finding (MAPF) is an NP-hard problem (Yu and LaValle 2013; Banfi, Basilico, and Amigoni 2017) that consists of computing a set of collision-free paths for a team of agents on a given graph that minimizes the sum of path costs or the makespan. Significant research effort has been devoted to MAPF due to its applications in distribution centers (Ma et al. 2017a; Hönig et al. 2019), computer games (Ma et al. 2017b) and traffic management (Dresner and Stone 2008).

One category of leading MAPF solvers is (bounded-sub)optimal solvers (that provide optimality guarantees). Examples include Conflict-Based Search (CBS) (Sharon et al. 2015) and its variants, such as Bounded CBS (BCBS), Enhanced CBS (ECBS) (Barer et al. 2014) and Explicit Estimation CBS (EECBS) (Li, Ruml, and Koenig 2021), and Branch-and-Cut-and-Price (BCP) solvers (Lam et al. 2019). Another category of leading MAPF solvers is unbounded-suboptimal solvers (that can find solutions fast for large instances but provide no optimality guarantees). Examples include Push-and-Swap (Luna and Bekris 2011), Parallel-Push-and-Swap (PPS) (Sajid, Luna, and Bekris 2012) and Prioritized Planning (PP) (Silver 2005). These existing solvers often either run too slowly or provide solutions of bad quality, especially when solving large MAPF instances with high agent or obstacle densities. Motivated by these issues, researchers have studied anytime MAPF solvers. The appeal of an anytime MAPF solver is that it first finds an initial solution quickly using any existing solver and, if more runtime is available, then improves the solution quality to near-optimal over time. MAPF-LNS (Li et al. 2021b) is a state-of-the-art anytime MAPF solver that uses Large Neighborhood Search (LNS) (Ahuja et al. 2002). MAPF-LNS first finds a solution quickly using an existing MAPF solver. It then iteratively destroys the paths of a set of agents generated by *destroy heuristics* and replans them using a *repair operator* while leaving the remaining paths unchanged. The number of agent sets that could be generated by the (randomized) destroy heuristics can be exponential in the cardinality of the agent sets, and MAPF-LNS randomly selects one of them (namely the one that is first randomly generated). However, some agent sets might not improve the solution as much as other agent sets and even result in no improvement at all, even if they are all generated by the same destroy heuristic. Thus, we propose an agent set-selection oracle that first samples a collection of candidate agents sets using the randomized destroy heuristics and applies the repair operator for each sampled agent set to select the best one, which obviously results in larger improvements than MAPF-LNS but is much more computationally expensive.

In this paper, we propose MAPF-ML-LNS, which uses machine learning (ML) to effectively select an agent set from a collection of candidate agent sets generated by the destroy heuristics. During training, we first record decisions made by the oracle on a set of MAPF instances and collect features that characterize these agent sets. The agent sets are labeled with the obtained improvement after replanning. Then, we use supervised learning to learn a ranking function for agent sets that imitates the oracle but is faster to compute. During testing, MAPF-ML-LNS uses the learned ranking function to select agent sets. We test two variants of MAPF-ML-LNS, namely ML-S, that is trained and tested on the same grid map, and ML-O, that is trained on a set of grid maps and tested on a different one. Both variants of MAPF-

Algorithm 1: MAPF-LNS

0: **Input:** A MAPF instance I
0: $P = \{p_i : i \in [k]\} \leftarrow \mathsf{runInitialSolver}(I)$
0: Initialize the weights ω of the destroy heuristics
0: **while** runtime limit not exceeded **do**
0: $\quad \mathcal{H} \leftarrow \mathsf{selectDestroyHeuristic}(w)$
0: $\quad A \leftarrow \mathsf{selectAgentSet}(I, \mathcal{H})$
0: $\quad P^- \leftarrow \{p_i \in P : a_i \in A\}$
0: $\quad P^+ \leftarrow \mathsf{runReplanSolver}(I, A, P \setminus P^-)$
0: $\quad$ Update the weights ω of the destroy heuristics
0: $\quad$ **if** $\sum_{p \in P^+} l(p) < \sum_{p \in P^-} l(p)$ **then**
0: $\quad\quad P \leftarrow (P \setminus P^-) \cup P^+$
0: **return** P =0

ML-LNS significantly improve MAPF-LNS's speed of improving the solution and the quality of the solution found within given a runtime limit. MAPF-ML-LNS performs well on grid maps unseen during training, which suggests that it is useful not only in environments with fixed layouts but also in those with unknown or dynamic layouts.

2 MAPF

The *Multi-Agent Path Finding (MAPF) problem* is to find a set of conflict-free (that is, collision-free) paths for a set of agents $N = \{a_1, \ldots, a_k\}$ on a given 2D four-neighbor grid map with blocked cells, that is represented as an undirected unweighted graph $G = (V, E)$. Each agent a_i has a start vertex $s_i \in V$ and a goal vertex $t_i \in V$. A path $p_i = (p_{i,0}, \ldots, p_{i,l(p_i)})$ for agent a_i is a sequence of vertices, where $p_{i,0} = s_i$, $p_{i,l(p_i)} = t_i$ and $l(p_i)$ is the length of the path. Time is discretized into time steps, and, at each time step t, every agent takes an action: It either moves to an adjacent vertex, i.e., $(p_{i,t}, p_{i,t+1}) \in E$, or waits at its current vertex, i.e., $p_{i,t} = p_{i,t+1} \in V$. Two types of conflicts are considered: i) A vertex conflict $\langle a_i, a_j, v, t\rangle$ occurs when agents a_i and a_j are at the same vertex v at time step t; and ii) an edge conflict $\langle a_i, a_j, u, v, t\rangle$ occurs when agents a_i and a_j traverse the same edge (u, v) in opposite directions from time step t to time step $t+1$. The cost of agent a_i is defined as $l(p_i)$, which is the number of time steps until it reaches its goal vertex t_i and remains there. The delay of agent a_i is defined as the difference between $l(p_i)$ and the distance between its start and goal vertices. A solution is a set of conflict-free paths that move all agents from their start vertices to their goal vertices. The sum of costs (and delays) of a solution is the sum of all agent costs $\sum_{i=1}^{k} l(p_i)$ (and delays, respectively). Our goal is to find a solution with the minimum sum of costs, or, equivalently, the minimum sum of delays.

3 Background and Related Work

In this section, we introduce MAPF-LNS in detail and summarize other related work on MAPF, ML and LNS.

3.1 MAPF-LNS

MAPF-LNS (Li et al. 2021b) is the state-of-the-art anytime MAPF solver. Anytime solvers find a solution fast and improve it to near-optimal if more runtime is available. They are able to solve large MAPF instances that most existing MAPF solvers fail to either solve or provide high-quality solutions to. However, developing anytime MAPF solvers has not been a focus of MAPF research. To the best of our knowledge, anytime BCBS (Cohen et al. 2018), which changes the focal search of BCBS to an anytime focal search, was the state-of-the-art anytime MAPF solver before MAPF-LNS.

MAPF-LNS, shown in Algorithm 1, takes a MAPF instance as input and first calls an efficient initial solver to compute a solution P (Line 2). In each iteration, it selects an agent set A using a destroy heuristic $\mathcal{H}$ (Line 6), deletes the current paths P^- of the agents in A from P (Line 7) and calls a replan solver to replan new paths P^+ for them that conflict with neither each other nor the paths in $P \setminus P^-$ (Line 8). If P^+ improves the solution (Line 10), then MAPF-LNS replaces P^- with P^+ (Line 11). The initial solver could be any off-the-shelf MAPF solver, and the replan solver could be any off-the-shelf MAPF solver that can handle moving obstacles.

MAPF-LNS uses two randomized destroy heuristics, namely an agent-based heuristic and a map-based heuristic, to generate agent sets[1]. The agent-based heuristic generates the agent set A by including the agent a_i with the largest delay and other agents (found via a random walk procedure) whose paths prevent it from getting a lower agent cost. The map-based heuristic randomly chooses a vertex with a degree greater than 2 in graph G and generates the agent set A by including some agents whose paths visit the chosen vertex. Both the agent-based and the map-based heuristics impose a limit on the cardinality of the agent set. MAPF-LNS uses Adaptive LNS (Ropke and Pisinger 2006), essentially an online learning algorithm, to select one of the two destroy heuristics by maintaining a weight for each of them.

3.2 Other Related Work

Our work is one of the first papers that use ML for MAPF. Sartoretti et al. (2019) have proposed a reinforcement-learning framework to learn decentralized policies for agents to avoid expensive centralized planning. Huang, Dilkina, and Koenig (2021b) have proposed a data-driven framework to learn conflict-selection strategies to speed up CBS. Huang, Dilkina, and Koenig (2021a) have proposed imitation learning and curriculum learning to learn node-selection strategies for ECBS. ML has also been used to select the best MAPF solvers for optimal MAPF (Kaduri, Boyarski, and Stern 2020; Ren et al. 2021).

LNS has been studied extensively for several combinatorial optimization problems, such as vehicle routing (Ropke and Pisinger 2006; Demir, Bektaş, and Laporte 2012) and solving mixed-integer linear programs (MILPs) (Munguía

[1] A more recent version of MAPF-LNS (Li et al. 2021a) uses a third heuristic that randomly generates agent sets, which improves its performance on small maps, such as the 32×32 empty map. In our study, we focus on maps that match real-world settings, such as warehouses, cities and games, which are typically of medium to large sizes, and we thus do not include the random heuristic.

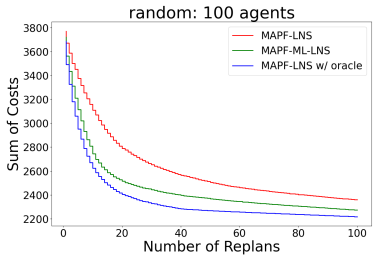

Figure 1: Evolution of the solution quality as a function of the number of replans for MAPF-LNS, MAPF-ML-LNS and MAPF-LNS with the oracle.

et al. 2018). Recently, deep-learning techniques have been applied to learn how to repair solutions for capacitated vehicle routing (Hottung and Tierney 2020) and good destroy heuristics for LNS for solving MILPs (Addanki, Nair, and Alizadeh 2020; Song et al. 2020; Sonnerat et al. 2021) and other combinatorial optimization problems (Chen and Tian 2019).

4 MAPF-ML-LNS

In this section, we introduce MAPF-ML-LNS. We first propose a sampling-based oracle for agent-set selection. The oracle samples a collection of agent sets using one of the two destroy heuristics in MAPF-LNS, replans the paths for all agents in the agent sets and selects the agent set that reduces the sum of costs most. However, the oracle is time-consuming to compute. We therefore use data-driven methods to learn a model for agent-set selection. The key idea is that, by recording the decisions made by the oracle, we learn a ranking function that ranks the agent sets as similarly as possible to the oracle but is much faster to compute than the oracle. Finally, we use the learned ranking function to guide agent-set selection during the search.

4.1 Oracle for Agent-Set Selection

Given a MAPF instance and its current solution, the oracle for agent-set selection first calls the destroy heuristics to sample a collection of S agent sets $\mathcal{A}$, where S is a constant that is set to 20 throughout the experiments. Each agent set sample is generated by a randomized destroy heuristic chosen from the agent-based and map-based heuristics with uniform probability, and its size is chosen uniform at random from 5 to 16. For each of the S agent sets, the oracle replans the paths of the agents in it and records the cost improvement, i.e., the resulting improvement in the sum of costs. Finally, the oracle outputs the agent set with the highest rank, i.e., the one with the largest cost improvement.

We replace the agent-set selection in MAPF-LNS (Lines 5-6 in Algorithm 1) with the oracle and compare the resulting version of MAPF-LNS with the oracle to MAPF-LNS for 100 agents on the random map "random-32-32-10", which is a 32×32 grid map from the MAPF benchmark set (Stern et al. 2019) with 10% randomly blocked cells. The grid map is shown in Table 3. We follow the experimental setup introduced in Section 5. We allocate a budget of 100 replans to each algorithm (instead of a runtime limit). Figure 1 shows how the average sum of costs changes after each replan. The average runtime of MAPF-LNS is 0.8 seconds, while the one of MAPF-LNS with the oracle is more than 16 seconds, which is too slow to be useful during MAPF solving. However, the significant difference between the curves of MAPF-LNS (red) and MAPF-LNS with the oracle (blue) in Figure 1 suggests that, if we could learn an ML model that approximates the oracle accurately with a small computational overhead during MAPF solving, then a version of MAPF-LNS with ML-guided LNS might be able to improve the solution quality faster early in the search than MAPF-LNS. The curves of MAPF-ML-LNS (green) in Figures 1 and 2 show that this is indeed possible.

4.2 Model Learning

We use imitation learning to learn a strategy for agent-set selection. Imitation learning relies on the demonstations of an "expert", which is the oracle in our case. Since we propose a one-time offline learning algorithm, the compute budget for training can be much higher than that of solving MAPF, which allows us to use a slow oracle. We adapt the data aggregation algorithm (Ross, Gordon, and Bagnell 2011) and the forward training algorithm (Ross and Bagnell 2010) to our use case.

The training algorithm, shown in Algorithm 2, takes as input $\mathcal{I}$ a set of training instances and runs for T iterations. We fix the grid map and the number of agents for the training instances, where the start and goal vertices of the agents are drawn i.i.d. from a given distribution. The training algorithm first computes an initial solution P_I for each $I \in \mathcal{I}$ (Lines 2-3). In each iteration t ($1 \leq t \leq T$), it collects training data $D_{I,t}$ for each $I \in \mathcal{I}$ by probing the oracle and recording the oracle's decision with respect to the incumbent solution P_I as well as the features of the agent sets sampled by the oracle (Lines 6-7). Then, it trains a ranking function π_t that minimizes a loss function over the aggregated training data D (Lines 8-9). To improve P_I, it evaluates $D_{I,t}$ to select an agent set A using π_t (Line 11), replans the paths of all agents in A (Line 13) and updates P_I if the solution improves (Lines 14-15). After T iterations, it returns the ranking function that performs best during validation (Lines 16-17). Algorithm 2 repeatedly determines a ranking function that makes good decisions in those situations encountered in previous iterations when using the previously-learned ranking functions to guide agent-set selection.

In the following two subsections, we explain in detail how we collect the training data and learn the ranking function.

Collecting Training Data Given an instance I and the incumbent solution P_I, the subroutine $\mathsf{collectData}(I, P_I)$ for data collection called in Algorithm 2 returns $D_{I,t}$, which consists of: (1) a collection of S agent sets $\mathcal{A}$ sampled by the oracle; (2) a ground-truth label vector $y_{\mathcal{A}} \in \{0, 1, 2\}^S$ for learning that consists of one label for each agent set, transformed from the oracle's ranking of the agent sets; and (3) a feature map $\phi : \mathcal{A} \rightarrow [0, 1]^p$ that describes agent set $A \in \mathcal{A}$ with a p-dimensional feature vector $\phi(A)$.

Algorithm 2: Training Algorithm

Input: Training instance set $\mathcal{I}$, number of iterations T and number S of agent set samples
for $I \in \mathcal{I}$ **do**
  $P_I \leftarrow \mathsf{runInitialSolver}(I)$
$D = \emptyset$
for $t = 1$ to T **do**
  for $I \in \mathcal{I}$ **do**
    $D_{I,t} \leftarrow \mathsf{collectData}(I, P_I)$ {Sample S agent sets for instance I and collect their features and labels using the oracle}
  $D \leftarrow D \cup \{D_{I,t} : I \in \mathcal{I}\}$ {Aggregate data}
  Train π_t with D
  for $I \in \mathcal{I}$ **do**
    $A \leftarrow \pi_t(D_{I,t})$
    $P^- \leftarrow \{p_i \in P_I : a_i \in A\}$
    $P^+ \leftarrow \mathsf{runReplanSolver}(I, A, P_I \setminus P^-)$
    if $\sum_{p \in P^+} l(p) < \sum_{p \in P^-} l(p)$ **then**
      $P_I \leftarrow (P_I \setminus P^-) \cup P^+$
$\pi \leftarrow \mathsf{validate}(\{\pi_1, \ldots, \pi_T\})$
return π =0

Feature Descriptions	Count
Static Features	6
Distance between a_i's start and goal vertices.	1
Row and column numbers of a_i's start and goal vertices.	4
Degree of a_i's goal vertex.	1
Dynamic Features	11
Delay of a_i.	1
Ratio between the delay of a_i and the distance between a_i's start and goal vertices.	1
The minimum, maximum, sum and average of the heat values of the vertices on a_i's path p_i: The heat value of vertex $v \in V$ is the number of time steps that v is occupied by an agent. The heat value of a vertex counts multiple times in the sum and average if the vertex is visited by the agent multiple times before reaching its goal vertex.	4
The number of time steps that a_i is on a vertex with degree j ($1 \leq j \leq 4$) before reaching its goal vertex.	4

Table 1: Agent a_i's features w.r.t. instance I and incumbent solution $P_I = \{p_i : i \in [k]\}$. The counts are the numbers of features contributed by the corresponding entries.

Features To compute the feature vector $\phi(A)$ of a given agent set $A \in \mathcal{A}$, we first compute a set of 17 agent features for each agent $a_i \in N = \{a_1, \ldots, a_k\}$, which are summarized in Table 1. We then divide the set of agents into two subsets, A and $N \setminus A$. For each subset, we compute the minimum, maximum, sum and average of the value of each of the 17 agent features over all agents in the subset, resulting in $4 \times 17 = 68$ features for the subset and $p = 2 \times 68 = 136$ features for both subsets. We normalize the value of each feature to the range of $[0, 1]$ across all agent sets in $\mathcal{A}$ and concatenate them to obtain the feature vector $\phi(A)$.

Labels A ground-truth label y_A is a value assigned to each agent set $A \in \mathcal{A}$, such that agent sets that result in higher cost improvements have smaller values. We use a simple and intuitive soft labeling scheme following previous work (Khalil et al. 2016): Let α and β ($\alpha \geq \beta$) be the cost improvements of the agent sets ranked at the 75 and 50 percentiles by the oracle, respectively, and set $y_A = \mathbf{1}_{[\Delta_A \geq \alpha]} + \mathbf{1}_{[\Delta_A \geq \beta]}$, where Δ_A is the cost improvement of A (in our study, we achieved similar results when labeling with 75, 50 and 25 percentiles as well as 80 and 50 percentiles). This labeling scheme assigns label 2 to the agent sets ranked in the top 25%, label 1 to the ones ranked in the top 50% but not the top 25% (i.e., the ones better than a choice at random) and label 0 to the rest. The ground-truth label vector $y_{\mathcal{A}}$ is obtained by concatenating all labels y_A. Our labeling scheme relaxes the definition of the best agent sets and allows us to learn a ranking function that focuses on selecting only high-ranking agent sets w.r.t. to their cost improvements and avoids having to correctly rank agent sets with small or no cost improvements.

Learning a Ranking Function There is a rich literature on learning to rank. Most of the existing models, such as RankNet (Burges et al. 2005) and LambdaRank (Quoc and Le 2007), rely on deep neural networks, which would introduce an undesirably large computational overhead if used in our MAPF solver. We thus learn a linear ranking function using SVM$^{\mathsf{rank}}$ (Joachims 2002) instead, which was efficient and effective in previous work on learning to rank (Khalil et al. 2016; Huang, Dilkina, and Koenig 2021b,a) in the context of search algorithms for different kinds of applications.

Given the dataset D collected during training, SVM$^{\mathsf{rank}}$ learns a linear ranking function $\pi : \mathbb{R}^p \rightarrow \mathbb{R} : \pi(\phi(A)) = \boldsymbol{w}^{\mathsf{T}}\phi(A)$ with parameter $\boldsymbol{w} \in \mathbb{R}^p$, that minimizes the loss function $L(\boldsymbol{w}) = \sum_{\mathcal{A} \in D} l(y_{\mathcal{A}}, \hat{y}_{\mathcal{A}}) + \frac{C}{2}||\boldsymbol{w}||_2^2$, where $y_{\mathcal{A}}$ is the ground-truth label vector, $\hat{y}_{\mathcal{A}}$ is the predicted score vector resulting from applying π to each feature vector $\phi(A)$ for $A \in \mathcal{A}$, $l : \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}$ is the loss resulting from ranking the agent sets in $\mathcal{A}$ according to $\hat{y}_{\mathcal{A}}$ instead of the ground-truth labels $y_{\mathcal{A}}$ and $C > 0$ is a regularization parameter. Specifically, $l(y_{\mathcal{A}}, \hat{y}_{\mathcal{A}})$ calculates the fraction of swapped pairs

$$l(y_{\mathcal{A}}, \hat{y}_{\mathcal{A}}) = \frac{|\{(A_i, A_j) \in \mathcal{P}_{\mathcal{A}} : \hat{y}_{A_i} \leq \hat{y}_{A_j}\}|}{|\mathcal{P}_{\mathcal{A}}|}$$

based on the set of ordered pairs of agent sets $\mathcal{P}_{\mathcal{A}} = \{(A_i, A_j) : A_i, A_j \in \mathcal{A} \wedge y_{A_i} > y_{A_j}\}$. We use an open-source solver (Joachims 2006) to learn the ranking function that minimizes an upper bound on the loss, since the loss itself is NP-hard to minimize.

4.3 ML-Guided LNS

After learning the ranking function π, we test it in MAPF-ML-LNS. In each iteration during testing, MAPF-ML-LNS samples a collection $\mathcal{A}$ of S agent sets using the two destroy heuristics and computes the predicted score $\pi(\phi(A))$ for each agent set $A \in \mathcal{A}$. The destroy heuristics are chosen from the agent-based and map-based heuristics with probabilities according to the weights ω maintained by adaptive LNS. MAPF-ML-LNS replans the paths for the agents in agent sets in descending order of the predicted scores of the agent sets. If a new incumbent solution (a solution with a

Algorithm 3: MAPF-ML-LNS

Input: MAPF instance I, ranking function π and number of agent set samples S
$P = \{p_i : i \in [k]\} \leftarrow \mathsf{runInitialSolver}(I)$
Initialize the weights ω of the destroy heuristics
while runtime limit not exceeded **do**
  $\mathcal{A} \leftarrow \emptyset$
  for $i = 1$ to S **do**
    $\mathcal{H} \leftarrow \mathsf{selectDestroyHeuristic}(w)$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \mathsf{selectAgentSet}(I, \mathcal{H})$
  Compute $\phi(A)$ for all $A \in \mathcal{A}$
  Compute $\pi(\phi(A))$ for all $A \in \mathcal{A}$
  for $A \in \mathcal{A}$ in descending order of $\pi(\phi(A))$ **do**
    $P^- \leftarrow \{p_i : a_i \in A\}$
    $P^+ \leftarrow \mathsf{runReplanSolver}(I, A, P \setminus P^-)$
    Update the weights ω of the destroy heuristics
    if $\sum_{p \in P^+} l(p) < \sum_{p \in P^-} l(p)$ **then**
      $P \leftarrow (P \setminus P^-) \cup P^+$
      break
return P =0

	Training k	Average Ranking	Improving Choice	Regret
random	100	6.5/20	90%	25%
den520d	200	7.0/20	96%	33%
city	250	6.7/20	99%	19%
ost003d	100	5.4/20	91%	26%
warehouse	100	6.0/20	90%	28%

Table 2: Validation results for the learned ranking function π. "Training k" is the number of agents of the training instances. "Average ranking" is the average rank of the first agent set selected by π among the $S = 20$ agent sets. "Improving choice" is the fraction of times π selects an agent set that results in a positive cost improvement. "Regret" is calculated as the average of 100% minus the cost improvement achieved by π as a percentage of the cost improvement achieved by the oracle.

smaller sum of costs) is found, it discards the remaining agent sets, recomputes the agent features and continues to the next iteration. MAPF-ML-LNS is summarized in Algorithm 3.

4.4 Discussion

To improve MAPF-LNS, we frame our ML problem as learning an agent set-selection strategy to guide destroying part of the solution in LNS. This allows us to use a lightweight linear ML model, such as SVM$^{\mathsf{rank}}$, that is easy to train and fast to evaluate during MAPF solving. We do not learn how to construct agent sets or predict the cost improvement of given agent sets since these are much more complicated ML problems that require using larger ML models, such as deep neural networks. We experimented with graph convolutional networks for these tasks on an agent dependency graph (Li et al. 2019) and ended up with good ML performance but an undesirably large computational overhead due to their high model complexity, rendering them useless.

5 Empirical Evaluation

In this section, we demonstrate the efficiency and effectiveness of MAPF-ML-LNS through extensive experiments. We implement MAPF-ML-LNS in C++ and conduct our experiments on a 2.4 GHz Intel Core i7 CPU with 16 GB RAM. We compare against MAPF-LNS on five grid maps of different sizes and structures from the MAPF benchmark set (Stern et al. 2019): (1) the random map "random-32-32-10"; (2) the game map "den520d", which is a 257×256 grid map from the video game *Dragon Age: Origins*; (3) the city map "Paris_1_256", which is a 256×256 grid map of Paris; (4) the game map "ost003d", which is a 194×194 grid map from the video game *Dragon Age: Origins*; and (5) the warehouse map "warehouse-10-20-10-2-1", which is a 163×63 grid map with 200 10×2 rectangular obstacles. The five grid maps are shown in Table 3. We do not compare to other anytime solvers, such as anytime BCBS and anytime EECBS, since they have been shown to be inferior to MAPF-LNS (Li et al. 2021b).

MAPF-LNS and MAPF-ML-LNS use the same setup. For the initial solvers and each grid map, we follow Li et al. (2021a) and select the MAPF solver from PP, PPS and EECBS that has the highest success rate on the instances with the largest number of agents within a runtime limit of 10 seconds as reported by them. We use that MAPF solver consistently for training and MAPF solving. That is, we use PP as the initial solver for den520d, ost003d and the city map, and PPS for the random and warehouse maps. We use PP as the replan solver for all grid maps, since PP dominates the other MAPF solvers used by them, namely CBS and EECBS (Li et al. 2021b).

During training, we run Algorithm 2 for $T = 100$ iterations. For each grid map, we use $|\mathcal{I}| = 16$ instances with a fixed number of agents from 16 "random" scenarios in the MAPF benchmark set to collect training data. The number of agents k of the training instances is reported in Table 2. Since we are using a randomized version of PP that uses random agent priorities to plan agents' paths, the cost improvement of each agent set used for creating its label is the average taken over 6 runs. We use regularization parameter $C = 0.1$ and the default values for the other parameters in the SVM$^{\mathsf{rank}}$ solver. We also tried $C \in \{0.01, 0.001\}$ and achieved similar results. It takes 2 to 8 hours, depending on the grid map, to run Algorithm 2 on a single CPU. If collecting training data for the 16 instances weres done in parallel on 16 CPUs in each iteration (Lines 6-7 in Algorithm 2), the training time could be reduced to less than 1 hour.

During validation, we evaluate $\pi_1, \ldots, \pi_T$ on the validation data and return the ranking function π that selects agent sets with the highest average ranking. We run MAPF-LNS with the oracle for 100 iterations on 4 MAPF instances from another 4 "random" scenarios to collect the validation data. The validation results for π are summarized in Table 2. During testing, we generate another 25 MAPF instances for each grid map from the same distribution as the "random" scenarios and set a runtime limit of 60 seconds per instance. For both MAPF-LNS and MAPF-ML-LNS, the runtime limit for finding the initial solution is set to 10 seconds. Those instances for which they fail to find an initial solution within

	k	AUC Ratio		Win/Loss		Sum of Agents' Delay (Suboptimality)		
		ML-S	ML-O	ML-S	ML-O	MAPF-LNS	ML-S	ML-O
random 	**100**	**1.15±0.23**	**1.12±0.20**	**20/5**	**20/5**	30 (1.01)	**28 (1.01)**	**28 (1.01)**
	150	**1.14±0.12**	**1.07±0.12**	**22/3**	**21/4**	105 (1.03)	**96 (1.03)**	**96 (1.03)**
	200	**1.03±0.10**	**1.07±0.19**	**15/9**	**15/9**	309 (1.07)	**275 (1.06)**	**270 (1.06)**
	250	0.98±0.17	0.95±0.12	10/15	8/17	806 (1.15)	843 (1.15)	845 (1.15)
	300	**1.13±0.14**	**1.06±0.15**	**18/6**	**13/11**	4,460 (1.67)	**3,754 (1.56)**	**4,301 (1.61)**
	350	0.99±0.08	0.94±0.08	11/12	6/17	21,310 (3.78)	22,234 (3.90)	23,674 (4.08)
den520d 	**200**	**1.97±0.56**	**1.75±0.53**	**23/2**	**24/1**	64 (1.00)	65 (1.00)	66 (1.00)
	300	**1.62±0.55**	**1.45±0.43**	**21/4**	**20/5**	400 (1.01)	**298 (1.00)**	**328 (1.01)**
	400	**1.65±0.54**	**1.31±0.30**	**25/0**	**22/3**	1,327 (1.02)	**778 (1.01)**	**1,121 (1.01)**
	500	**1.25±0.35**	**1.13±0.22**	**19/6**	**18/7**	3,616 (1.04)	**2,676 (1.03)**	**3,281 (1.03)**
	600	**1.10±0.15**	**1.10±0.08**	**18/7**	**24/1**	8,134 (1.08)	**6,654 (1.06)**	**6,967 (1.07)**
	700	**1.07±0.06**	**1.05±0.06**	**22/3**	**20/5**	12,558 (1.10)	**11,785 (1.10)**	**11,535 (1.09)**
city 	**250**	**1.75±0.41**	**1.14±0.32**	**22/3**	**14/11**	229 (1.00)	**110 (1.00)**	**128 (1.00)**
	350	**1.12±0.34**	**1.02±0.24**	**19/6**	**14/11**	469 (1.01)	**372 (1.01)**	**368 (1.01)**
	450	**1.30±0.35**	**1.01±0.22**	**19/6**	**13/12**	763 (1.01)	**629 (1.01)**	**753 (1.01)**
	550	**1.05±0.18**	**1.06±0.24**	**16/9**	**14/11**	1,932 (1.02)	2,056 (1.02)	**1,536 (1.01)**
	650	**1.08±0.13**	**1.10±0.25**	**17/8**	**17/8**	3,274 (1.03)	**3,041 (1.02)**	**3,033 (1.02)**
	750	**1.07±0.14**	**1.09±0.08**	**17/6**	**19/4**	8,371 (1.06)	**8,363 (1.06)**	**7,413 (1.05)**
ost003d 	**100**	**1.28±0.33**	**1.17±0.28**	**21/4**	**15/10**	42 (1.00)	42 (1.00)	42 (1.00)
	200	**1.43±0.36**	**1.20±0.27**	**19/4**	**17/6**	458 (1.01)	**332 (1.01)**	**372 (1.01)**
	300	**1.14±0.19**	**1.16±0.16**	**16/8**	**20/4**	2,509 (1.05)	**2,379 (1.05)**	**2,152 (1.04)**
	400	**1.05±0.08**	**1.06±0.08**	**17/6**	**17/6**	6,907 (1.11)	**6,584 (1.10)**	**6,417 (1.10)**
	500	**1.02±0.03**	**1.04±0.05**	**15/7**	**16/6**	14,750 (1.19)	**14,431 (1.19)**	**14,251 (1.18)**
	600	**1.02±0.03**	**1.03±0.04**	**14/6**	**16/4**	24,684 (1.27)	**24,468 (1.27)**	**24,401 (1.27)**
warehouse 	**100**	**1.35±0.33**	**1.25±0.30**	**20/5**	**20/5**	57 (1.01)	**37 (1.00)**	**37 (1.00)**
	150	**1.21±0.24**	**1.14±0.22**	**18/7**	**16/9**	295 (1.02)	**195 (1.01)**	**217 (1.02)**
	200	**1.19±0.22**	**1.05±0.13**	**21/4**	**15/10**	925 (1.06)	**736 (1.05)**	**842 (1.05)**
	250	**1.17±0.20**	**1.11±0.18**	**17/8**	**16/9**	1,817 (1.09)	**1,595 (1.08)**	**1,805 (1.09)**
	300	**1.18±0.21**	**1.13±0.19**	**17/8**	**18/7**	4,719 (1.20)	**3,852 (1.16)**	**3,547 (1.15)**
	350	**1.07±0.10**	**1.02±0.07**	**15/9**	**13/11**	12,004 (1.43)	**10,191 (1.36)**	12,143 (1.43)

Table 3: The average ratios of the AUCs of MAPF-LNS and our MAPF solvers with their standard deviations, the win/loss counts w.r.t. the AUCs and the average sums of delays with the average suboptimalities for a runtime limit of 60 seconds. All entries take only the solved MAPF instances into account. The win/loss counts are the numbers of instances where the AUCs of our MAPF solvers are smaller/larger than those of MAPF-LNS. The suboptimalities are overestimated values calculated as the ratio between the final sum of costs and the sum of distances between the agents' start and goal vertices. We bold the number of agents k on which ML-S is trained and the entries where one of our MAPF solvers outperforms MAPF-LNS.

10 seconds are considered unsolvable and not included in our results. We use the same random seed to ensure that both MAPF solvers compute the same instances and initial solutions. The runtime limit of PP per replan is set to 2 seconds for the warehouse map and 0.6 seconds for the other grid maps initially and then adaptively set to twice the average runtime of all successful replans so far after the first 30 successful replans. During both training and MAPF solving, when generating an agent set using the destroy heuristics in a MAPF solver, we draw its cardinality uniformly from 5 to 16. We sample $S = 20$ agent sets in each iteration of MAPF-ML-LNS.

Our results provide answers to the following questions: (1) If the grid map is known in advance, can we learn a ranking function that performs well on the same grid map with the same and different numbers of agents? (2) If the grid map is unknown in advance, can we learn a ranking function from other grid maps that performs well on the unknown one? We therefore learn two ranking functions with SVM$^{\mathsf{rank}}$ for each grid map, namely a ranking function trained on MAPF instances on that grid map (resulting in MAPF solver ML-S) and a ranking function trained on MAPF instances from the other four grid maps (resulting in MAPF solver ML-O).

An important metric for evaluating the performance of an anytime MAPF solver is its speed of improving the solution. Let $\mathcal{I}_{\mathsf{test}}$ be the set of test instances and, for each $I \in \mathcal{I}_{\mathsf{test}}$, let $t^{\mathcal{S}}_{I,\mathsf{init}}, SOC^{\mathcal{S}}_I(t)$ and $SOD^{\mathcal{S}}_I(t)$ be the runtime needed to find the initial solution, the sum of costs and the sum of delays of the solution at runtime t, respectively, when solving instance I using solver $\mathcal{S}$. Following Li et al. (2021b), we compute the area under the curve (AUC) of the sum of delays as a function of the runtime for each instance I with solver $\mathcal{S}$, which is formally defined as $AUC^{\mathcal{S}}_I(t_{\mathsf{limit}}) = \int_{t^{\mathcal{S}}_{I,\mathsf{init}}}^{t_{\mathsf{limit}}} SOD^{\mathcal{S}}_I(t)\mathrm{d}t$, where t_{limit} is the runtime limit (60 seconds). The smaller the AUC, the faster the speed of improving the solution is. In Table 3, we report the average ratios of the AUCs of MAPF-LNS and our MAPF solvers, the win/loss counts w.r.t. the AUC and the average sums of delays with the the average suboptimalities over

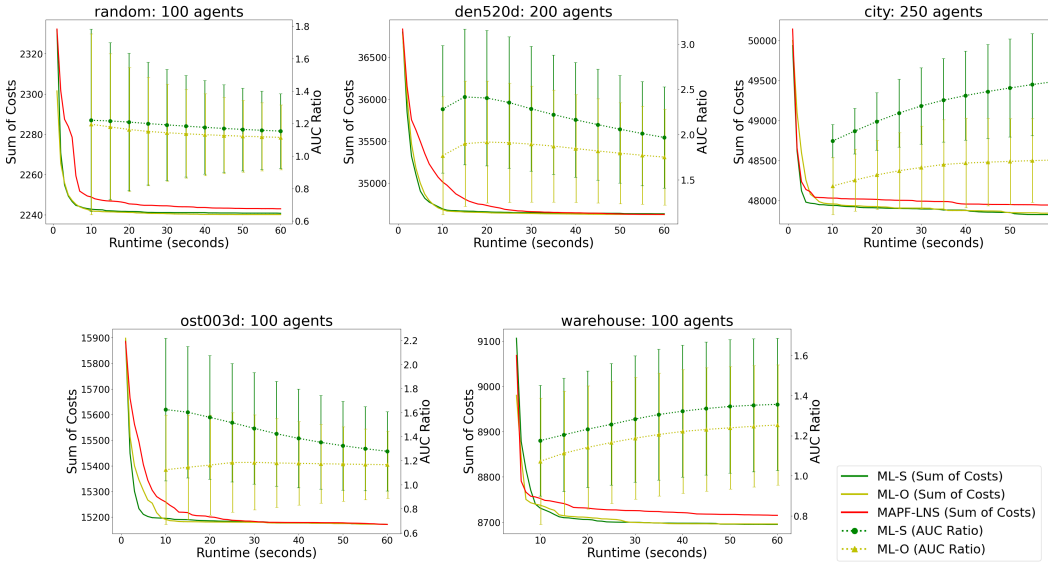

Figure 2: Evolutions of the sum of costs (solid curves with the y-axis on the left side, smaller is better) from 1 second to 60 seconds for MAPF-LNS, ML-S and ML-O, averaged over all solved instances, and the average ratio of the AUCs of MAPF-LNS and our MAPF solvers (dotted curves with the y-axis on the right side, greater than 1 is better), also averaged over all solved instances, as a function of the runtime limit. The error bars represent the standard deviation.

all solved test instances[2]. On the city and two game maps (den520d and ost003d), the AUCs of MAPF-LNS are 43% to 97% worse than the ones of ML-S. On these three maps, ML-S significantly outperforms MAPF-LNS also w.r.t. the win/loss counts and, for almost all tested numbers of agents, w.r.t. the final solution qualities. On the random and warehouse maps, ML-S also outperforms MAPF-LNS w.r.t. all metrics, except for a few cases with huge numbers of agents (250 and 350 agents on the random map). Even though ML-S learns the ranking functions on MAPF instances with a fixed number of agents, they generalize well to MAPF instances with larger numbers of agents on the same grid map and outperform MAPF-LNS in almost all cases. ML-O also significantly outperforms MAPF-LNS. ML-O, without seeing the test grid map during training, is competitive with ML-S and even outperforms it sometimes on the two game maps and the city map. For the random map, the improvement of MAPF-ML-LNS over MAPF-LNS is not as substantial as for the other grid maps, especially on MAPF instances with huge numbers of agents. We tried retraining the ranking functions on MAPF instances with larger numbers of agents (e.g., 250 agents for the random map) but achieved similar results. It is future work to improve the effectiveness of MAPF-ML-LNS on this grid map.

To demonstrate the effectiveness of our MAPF solvers further, we show the average sum of costs for MAPF-LNS, ML-S and ML-O in Figure 2 together with the average ratios between the AUCs of MAPF-LNS and our MAPF solvers as functions of the runtime limit t_{limit}, i.e., $\frac{1}{|\mathcal{I}_{\text{test}}|}\sum_{I\in\mathcal{I}_{\text{test}}} SOC_I^{\mathcal{S}}(t_{\text{limit}})$ and $\frac{1}{|\mathcal{I}_{\text{test}}|}\sum_{I\in\mathcal{I}_{\text{test}}} \frac{AUC_I^{\text{MAPF-LNS}}(t_{\text{limit}})}{AUC_I^{\mathcal{S}}(t_{\text{limit}})}$ for each solver $\mathcal{S}$. In these cases, ML-S and ML-O establish advantages early in the search and substantially outperform MAPF-LNS for several shorter runtime limits, e.g., 20 or 30 seconds.

The runtime overhead of MAPF-ML-LNS induced by computing the features and evaluating the ranking function is small. MAPF-ML-LNS performs fewer replans than MAPF-LNS on average. These results suggest that our learned ranking functions select agent sets more effectively since they improve the solutions faster and achieve better solution qualities than MAPF-LNS with fewer replans.

Finally, we study the feature importance of the learned ranking function for ML-S for each grid map, measured by the absolute values of the learned feature weights. It makes sense to do so since the features $\phi(A)$ are normalized. Features related to the delays are the most important ones for all five grid maps. The other important features are related to the costs of the paths, the ratios between the delays and costs, the sums of the heat values on the paths and the numbers of time steps that the agents are on a vertex with degree 2 or 3 (see Table 1 for definitions).

6 Conclusion

In this paper, we proposed MAPF-ML-LNS, an anytime MAPF solver that uses ML-guided LNS. We compared MAPF-ML-LNS to MAPF-LNS, the state-of-the-art anytime MAPF solver. Our experimental results showed that our learned ranking function can generalize to different numbers of agents on both fixed and unseen grid maps. In fact, MAPF-ML-LNS performed substantially better than MAPF-LNS on the large grid maps tested, such as the two game maps, the warehouse map and the city map.

[2] All solvers have the same set of solved instances since they use the same initial solver with the same random seeds.

Acknowledgments

The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, 1837779, 1935712 and 2112533, the U.S. Department of Homeland Security under grant number 2015-ST-061-CIRC01 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 necessarily representing the official policies, either expressed or implied, of NSF or the U.S Department of Homeland Security.

References

Addanki, R.; Nair, V.; and Alizadeh, M. 2020. Neural large neighborhood search. In *LMCA NeurIPS Workshop*.

Ahuja, R. K.; Ergun, Ö.; Orlin, J. B.; and Punnen, A. P. 2002. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3): 75–102.

Banfi, J.; Basilico, N.; and Amigoni, F. 2017. Intractability of time-optimal multirobot path planning on 2D grid graphs with holes. *IEEE Robotics and Automation Letters*, 2(4): 1941–1947.

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *SoCS*.

Burges, C.; Shaked, T.; Renshaw, E.; Lazier, A.; Deeds, M.; Hamilton, N.; and Hullender, G. 2005. Learning to rank using gradient descent. In *ICML*, 89–96.

Chen, X.; and Tian, Y. 2019. Learning to perform local rewriting for combinatorial optimization. In *NeurIPS*, 6281–6292.

Cohen, L.; Greco, M.; Ma, H.; Hernández, C.; Felner, A.; Kumar, T. S.; and Koenig, S. 2018. Anytime focal search with applications. In *IJCAI*, 1434–1441.

Demir, E.; Bektaş, T.; and Laporte, G. 2012. An adaptive large neighborhood search heuristic for the pollution-routing problem. *European Journal of Operational Research*, 223(2): 346–359.

Dresner, K.; and Stone, P. 2008. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31: 591–656.

Hönig, W.; Kiesel, S.; Tinka, A.; Durham, J. W.; and Ayanian, N. 2019. Persistent and robust execution of MAPF schedules in warehouses. *IEEE Robotics and Automation Letters*, 4(2): 1125–1131.

Hottung, A.; and Tierney, K. 2020. Neural large neighborhood search for the capacitated vehicle routing problem. In *ECAI*, 443–450.

Huang, T.; Dilkina, B.; and Koenig, S. 2021a. Learning node-selection strategies in bounded-suboptimal conflict-based search for multi-agent path finding. In *AAMAS*, 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*, 11246–11253.

Joachims, T. 2002. Optimizing search engines using clickthrough data. In *SIGKDD*, 133–142.

Joachims, T. 2006. Training linear SVMs in linear time. In *SIGKDD*, 217–226.

Kaduri, O.; Boyarski, E.; and Stern, R. 2020. Algorithm selection for optimal multi-agent pathfinding. In *ICAPS*, 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*, 724–731.

Lam, E.; Le Bodic, P.; Harabor, D.; and Stuckey, P. J. 2019. Branch-and-cut-and-price for multi-agent pathfinding. In *IJCAI*, 1289–1296.

Li, J.; Boyarski, E.; Felner, A.; Ma, H.; and Koenig, S. 2019. Improved heuristics for multi-agent path finding with conflict-based search: Preliminary results. In *SoCS*.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021a. Anytime multi-agent path finding via large neighborhood search. In *IJCAI*, 4127–4135.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021b. Anytime multi-agent path finding via large neighborhood search: Extended abstract. In *AAMAS*, 1581–1583.

Li, J.; Ruml, W.; and Koenig, S. 2021. EECBS: Bounded-suboptimal search for multi-agent path finding. In *AAAI*, 12353–12362.

Luna, R.; and Bekris, K. E. 2011. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, 294–300.

Ma, H.; Li, J.; Kumar, T. S.; and Koenig, S. 2017a. Lifelong multi-agent path finding for online pickup and delivery tasks. In *AAMAS*, 837–845.

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

Munguía, L.-M.; Ahmed, S.; Bader, D. A.; Nemhauser, G. L.; and Shao, Y. 2018. Alternating criteria search: A parallel large neighborhood search algorithm for mixed integer programs. *Computational Optimization and Applications*, 69(1): 1–24.

Quoc, C.; and Le, V. 2007. Learning to rank with nonsmooth cost functions. In *NeurIPS*, 193–200.

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 *AAMAS*, 1055–1063.

Ropke, S.; and Pisinger, D. 2006. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4): 455–472.

Ross, S.; and Bagnell, D. 2010. Efficient reductions for imitation learning. In *AISTATS*, 661–668.

Ross, S.; Gordon, G.; and Bagnell, D. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, 627–635.

Sajid, Q.; Luna, R.; and Bekris, K. E. 2012. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *SoCS*.

Sartoretti, G.; Kerr, J.; Shi, Y.; Wagner, G.; Kumar, T. S.; Koenig, S.; and Choset, H. 2019. PRIMAL: Pathfinding via reinforcement and imitation multi-agent learning. *IEEE Robotics and Automation Letters*, 2378–2385.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219: 40–66.

Silver, D. 2005. Cooperative pathfinding. In *AIIDE*, 117–122.

Song, J.; Lanka, R.; Yue, Y.; and Dilkina, B. 2020. A general large neighborhood search framework for solving integer linear programs. In *NeurIPS*, 20012–20023.

Sonnerat, N.; Wang, P.; Ktena, I.; Bartunov, S.; and Nair, V. 2021. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201*.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. S.; Boyarski, E.; and Bartak, R. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *SoCS*.

Yu, J.; and LaValle, S. M. 2013. Planning optimal paths for multiple robots on graphs. In *ICRA*, 3612–3617.