Learning Node-Selection Strategies in Bounded-Suboptimal Conflict-Based Search for Multi-Agent Path Finding

Taoan Huang, Bistra Dilkina, Sven Koenig
University of Southern California
{taoanhua,dilkina,skoenig}@usc.edu

ABSTRACT

Multi-Agent Path Finding (MAPF) is an NP-hard problem that has important applications for distribution centers, traffic management and computer games, and it is still difficult for current solvers to solve large instances optimally. Bounded suboptimal solvers, such as Enhanced Conflict-Based Search (ECBS) and its variants, are more efficient than optimal solvers in finding a solution with suboptimality guarantees. ECBS is a tree search algorithm that expands the search tree by repeatedly selecting search tree nodes from a focal list. In this work, we propose to use machine learning (ML) to learn a node-selection strategy to speed up ECBS. In the first phase of our framework, we use imitation learning and curriculum learning to learn node-selection strategies iteratively for different numbers of agents from training instances. In the second phase, we deploy the learned models in ECBS and test their solving performance on unseen instances drawn from the same distribution as the training instances. We demonstrate that our solver, ECBS+ML, shows substantial improvement in terms of the success rates and runtimes over ECBS on five different types of grid maps from the MAPF benchmark.

KEYWORDS

Multi-Agent Path Finding; Heuristic Search; Machine Learning

ACM Reference Format:
Taoan Huang, Bistra Dilkina, Sven Koenig. 2021. Learning Node-Selection Strategies in Bounded-Suboptimal Conflict-Based Search for Multi-Agent Path Finding. In *Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), Online, May 3–7, 2021*, IFAAMAS, 9 pages.

1 INTRODUCTION

Multi-Agent Path Finding (MAPF) is the problem of finding a set of conflict-free (that is, collision-free) paths for a team of agents that moves on a given underlying graph and minimizes the sum of path costs or the makespan. MAPF has practical applications in distribution centers [14, 21], traffic management [9] and video games [22].

Conflict-Based Search (CBS) [27], a bi-level search algorithm, is one of the leading algorithms for solving MAPF optimally. The high-level search uses a best-first search on a binary search tree, called *constraint tree* (CT), to find a set of constraints to impose on agents to ensure the feasibility and optimality of the solution. The low-level search finds a cost-minimal path for each agent that respects the constraints imposed by the high-level search. MAPF is NP-hard to solve optimally [2, 32] and, therefore, optimal solvers, such as CBS, do not scale to many agents. Enhanced Conflict-Based Search (ECBS) [3] and its variants [6, 7] are guaranteed to find solutions whose sums of costs of the conflict-free paths are at most $w \geq 1$ times the minimum ones, called w-approximate solutions, and run faster than CBS. ECBS uses focal searches [23] instead of best-first searches for both the high-level and the low-level search to guarantee bounded suboptimality. The high-level search of ECBS maintains a focal list that contains the subset of CT nodes on the open list whose costs are at most w times the current lower bound on the optimal solution cost. ECBS can select an arbitrary CT node in the focal list for expansion, but the common practice is to select one with the minimum d-value, where the d-value is a heuristic component of ECBS that is computed for each CT node when it is generated. The d-value of a CT node is an estimate of the effort required to find a solution in the CT subtree rooted at that CT node. Typically, the d-value is a hard-coded function defined to be the number of conflicts in the agents' paths of the CT node or other statistics related to the conflicts. However, such choices for the d-value might result in ECBS being not as fast as it could be since the number of conflicts in the agents' paths is not directly correlated with the effort required to find a solution.

Instead of manually defining the d-values, we borrow tools from the machine learning literature [4, 24, 25, 28] and propose a novel data-driven framework for learning node-selection strategies for the high-level focal search to speed up ECBS. A node-selection strategy assigns a d-value to each CT node and always selects a node with the minimum d-value in the focal list to expand during the search. During training, we do not directly learn the d-values but rather a ranking function that differentiates CT nodes that have shorter distances to a leaf node in the CT with a w-approximate solution from those CT nodes that have longer distances to one. During the search, the ranking function takes a CT node's features as input and calculates a real-valued d-value. Our goal is to learn a ranking function such that its d-values allow ECBS to get closer to a desired solution every time it expands a CT node and, therefore, help it to find a solution more quickly. In the first phase of our framework, we fix the underlying graph and learn node-selection strategies from solving the training instances on that graph where the start and goal locations of the agents are drawn from a given distribution. In the first phase, we start with a small number of agents and use imitation learning [8, 24, 25] to learn a node-selection strategy for that number of agents. We then continue learning node-selection strategies for larger and larger numbers of agents. Instead of learning from scratch, we use curriculum learning [4] to learn more efficiently by using previously-learned node-selection strategies as starting points. In the second phase, we use the learned strategies to solve unseen instances drawn from the same distribution. Our

Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.), May 3–7, 2021, Online.
© 2021 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

solver, ECBS+ML, can scale to large instances beyond those that are solvable by ECBS, the state-of-the-art bounded suboptimal MAPF algorithm. In experiments, we test our solver on five different types of grid maps from the MAPF benchmark and show that we significantly outperform ECBS in terms of both the success rate (the fraction of test instances solved within a given runtime limit) and runtime. We believe that our method is useful for applications in domains where the layouts of the environments, such as warehouses, road networks and maps in computer games, are fixed while only the numbers and locations of agents differ.

2 MAPF

Given an undirected unweighted underlying graph $G = (V, E)$, the *Multi-Agent Path Finding (MAPF) problem* is to find a set of conflict-free paths for a set of agents $\{a_1, \ldots, a_k\}$. Each agent a_i has a start vertex $s_i \in V$ and a goal vertex $t_i \in V$. Time is discretized into time steps, and, at each time step, every agent can either move to an adjacent vertex or wait at its current vertex in the graph. 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 $(u, v) \in E$ in opposite directions between time steps t and $t+1$. The cost of agent a_i is defined as the number of time steps until it reaches its goal vertex t_i and remains there. A conflict-free solution is a set of conflict-free paths that move all agents from their start vertices to their goal vertices. The sum of costs of a solution is the sum of all agents' costs. Given a suboptimality factor $w \geq 1$, our objective is to find a w-approximate conflict-free solution, which is a conflict-free solution with a sum of costs that is at most w times the minimum sum of costs.

3 BACKGROUD AND RELATED WORK

In this section, we first introduce CBS and ECBS. Then, we summarize related work on MAPF solvers and machine learning (ML) in MAPF and branch-and-bound searches.

3.1 CBS

CBS is an optimal bi-level tree search algorithm for MAPF. It records the following information for each CT node N:

(1) The set of constraints imposed so far. There are two types of constraints: i) a vertex constraint $\langle a_i, v, t\rangle$, corresponding to a vertex conflict, prohibits agent a_i from being at vertex v at time step t; and ii) an edge constraint $\langle a_i, u, v, t\rangle$, corresponding to an edge conflict, prohibits agent a_i from moving from vertex u to vertex v between time steps t and $t+1$.
(2) The solution N_{Sol} of N, which consists of a set of individually cost-minimal paths for all agents that respect the set of constraints of N. An individually cost-minimal path for an agent is the cost-minimal path between its start and goal vertices, assuming it is the only agent in the graph.
(3) The cost N_{Cost} of N, defined as the sum of costs of the solution of N.
(4) The set of conflicts N_{Conf} in the solution of N.

On the high level, CBS starts with a CT node with an empty set of constraints and grows the CT by always expanding a CT node with the lowest cost. After choosing a CT node N for expansion, CBS identifies its set of conflicts. If there is no conflict, CBS terminates and returns the conflict-free solution. Otherwise, CBS picks one of the conflicts to resolve and generates two child nodes of N to the CT by adding to the set of constraints of N, depending on the type of conflict, an edge or vertex constraint for one of the two conflicting agents in one of the child nodes and for the other conflicting agent in the other child node. Then, CBS applies the low-level search to each child node to replan the path of each affected agent and records the respective solution and cost. CBS guarantees completeness by exploring both ways of resolving each conflict and optimality by performing best-first searches on both of its high and low levels.

Algorithm 1 ECBS

```
1: Input: A MAPF instance and suboptimality factor w
2: Generate the root CT node R with an initial solution
3: Initialize open list 𝒩 ← {R}
4: LB ← R_LB, and initialize focal list ℱ ← {R}
5: while 𝒩 is not empty do
6:     N ← a CT node with the minimum d-value in ℱ
7:     if N_Conf = ∅ then
8:         return N_Sol
9:     Delete N from the open and focal lists
10:    if min_{N∈𝒩} N_LB > LB then
11:        LB ← min_{N∈𝒩} N_LB
12:        ℱ ← {N ∈ 𝒩 : N_LB ≤ wLB}
13:    Pick a conflict in N_Conf
14:    Generate 2 child CT nodes N^1 and N^2 of N
15:    Call low-level search for N^i to calculate N^i_Sol, N^i_Cost and
       N^i_Conf for i = 1, 2
16:    Add N^i to 𝒩 for i = 1, 2
17:    Add N^i to ℱ if N^i_Cost ≤ wLB for i = 1, 2
18: return No solution
```

3.2 ECBS

ECBS is a bounded-suboptimal version of CBS [3]. Given a suboptimality factor $w \geq 1$, ECBS is guaranteed to find a w-approximate conflict-free solution. Both the high-level and low-level searches of ECBS use focal searches [23] instead of best-first searches. Consider a CT node N. On the low level, ECBS runs a focal search for each agent a_i such that the cost of the path found is at most $wN_{\mathsf{LB},i}$, where $N_{\mathsf{LB},i}$ is the lower bound on the cost of the individually cost-minimal path for a_i that respects the set of constraints of CT node N and is computed by the focal search. Let $N_{\mathsf{LB}} = \sum_{i=1}^{k} N_{\mathsf{LB},i}$. On the high level, ECBS performs a focal search with a focal list that contains all CT nodes N in $\mathcal{N}$ such that $N_{\mathsf{Cost}} \leq w\mathsf{LB}$, where $\mathcal{N}$ is the open list and $\mathsf{LB} = \min_{N \in \mathcal{N}} N_{\mathsf{LB}}$. Since LB is a lower bound on the sum of costs of any conflict-free solution, once a conflict-free solution is found by always expanding a CT node in the focal list, it is guaranteed to be a w-approximate conflict-free solution. ECBS is summarized in Algorithm 1.

3.3 Related Work

A line of research in MAPF focuses on developing optimal solvers. Leading optimal MAPF solvers include integer linear programming-based solvers [18, 19] as well as CBS [27] and its variants, such as Improved CBS [5], CBSH [12] and CBSH2 [20]. Another line of research focuses on developing bounded-suboptimal A*-based MAPF solvers, such as Enhanced Partial-Expansion A* (EPEA*) [11], A* with operator decomposition [29] and M* [31]. ECBS [3] is the current state-of-the-art bounded-suboptimal MAPF solver for MAPF. ECBS with the highway heuristic [6] and Improved ECBS [7] have been proposed to speed up ECBS in environments such as warehouses. In contrast, our method makes no assumptions about the underlying graph and works well for different types of graphs.

Our work is one of the first papers that use machine learning for MAPF. Sartoretti et al. [26] use reinforcement learning to learn decentralized policies for agents to avoid the cost of centralized planning. Huang et al. [15] use ML to learn which conflict to resolve next to speed up CBS. In a non-MAPF context, ML has been use to speed up search in the context of branch-and-bound tree search. He et al. [13] use imitation learning to learn node-selection and node-pruning strategies for solving mixed-integer programs. Song et al. [28] scale up this approach by progressively increasing the instance sizes in the form of curriculum learning. Other related work includes learning to branch [17] and learning to run primal heuristics [16] in tree search for solving mixed-integer programs.

4 LEARNING NODE-SELECTION STRATEGIES

A generic node-selection strategy for ECBS assigns a d-value to each CT node and always selects a CT node with the minimum d-value in the focal list for expansion. The most commonly used node-selection strategy in previous work uses the number of conflicts $|N_{\text{Conf}}|$ as the d-value of a CT node N. We refer to this strategy as h_1. Barer et al. [3] propose other strategies that use the number of pairs of agents that have at least one conflict with each other and the number of agents that have at least one conflict with other agents as the d-values. We refer to these two strategies as h_2 and h_3, respectively. We implement and experiment with h_1, h_2 and h_3 in Section 5.

Next, we introduce our framework for learning node-selection strategies for the high-level focal search of ECBS. Our framework consists of two phases:

(1) Model learning. During learning, we fix the underlying graph $G = (V, E)$ and the suboptimality factor w of the instances and apply the DAgger algorithm [25], an imitation learning algorithm, and curriculum learning [4] to learn node-selection strategies for solving instances with different numbers of agents on that graph.
(2) ML-guided search. During testing, we draw test instances from the same distribution as the training instances and use the learned node-selection strategies in ECBS.

4.1 Model Learning

We solve instances with a fixed underlying graph $G = (V, E)$ and a fixed suboptimality factor w. The first task in our framework is to learn node-selection strategies for instances with different numbers of agents. The idea central to our training algorithm is that we start learning a node-selection strategy by solving easy instances with a small number of agents and iteratively increasing the number of agents to learn another strategy based on the previous one. In particular, we want to learn to solve instances with increasing difficulty, i.e., with m different numbers of agents $k_1, \ldots, k_m$ where $k_1 < \ldots < k_m$. For each k_i, we learn a node-selection strategy that assigns $\pi_i(\phi(N))$ to CT node N as its d-value, where $\phi(N)$ is the feature vector of N and π_i is a learned ranking function. Therefore, a desirable ranking function is one that assigns smaller d-values to CT nodes that are closer to a w-approximate conflict-free solution and larger d-values to those CT nodes that are farther away from one.

Our training algorithm is a curriculum learning algorithm, as shown in Algorithm 2. Algorithm 2 takes $\{k_1, \ldots, k_m\}$ and m sets of training instances $\{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$ as input and outputs $\{\pi_1, \ldots, \pi_m\}$. Each instance in $\mathcal{I}_i$ includes k_i agents, where the start and goal vertices of the agents are drawn i.i.d. from a given distribution. π_0 is set to the ranking function π^*, that corresponds to an initial node-selection strategy (e.g., node-selection strategy h_1, h_2 or h_3) (line 2). To obtain π_1 for instances with k_1 agents, we use DAgger$(\pi^*, \mathcal{I}_1)$ [25] as a training algorithm that learns a ranking function from solving the training instances in $\mathcal{I}_1$ using π^* as a starting point. To obtain π_i (for $i > 1$), instead of starting from π^* again, we start learning from π_{i-1}. We obtain π_i $(1 < i \leq m)$ by calling DAgger$(\pi_{i-1}, \mathcal{I}_i)$, that learns a ranking function using π_{i-1} as the ranking function of the initial node-selection strategy (line 3-4) until a stopping criterion is met (line 5-7) or $i = m$. If the stopping criterion is met before $i = m$, we terminate training (line 7) and simply set π_j to π_i for all $i < j \leq m$ (line 6). If DAgger$(\pi_{i-1}, \mathcal{I}_i)$ returns π_{i-1}, then it cannot find a better ranking function than the initial one. This situation typically occurs at some point in time during training for hard instances with many agents since only data collected from solved instances during data collection contributes to the training data, and, for hard instances, it is difficult to collect a sufficient amount of training data, which makes it difficult to improve on the initial ranking function. When the training algorithm observes this situation (line 5), it stops training and uses the last obtained ranking function for all instances with larger numbers of agents than the one for which we could not improve the ranking function.

DAgger$(\pi^{(0)}, \mathcal{I})$, shown in Algorithm 3, is an imitation learning algorithm. The inputs $\pi^{(0)}$ and $\mathcal{I}$ are the ranking function of the initial node-selection strategy and the set of training instances. DAgger repeatedly determines a ranking function that makes better decisions in those situations that were encountered when running ECBS with previous versions of the ranking function. Initially, the training data D is set to $\emptyset$ (line 1). Let R be the number of iterations for which the algorithm runs (line 2). In iteration j, it collects training data by solving the instances in $\mathcal{I}$ with the ranking function $\pi^{(j-1)}$ obtained in iteration $j-1$, aggregates it with D (line 4) and learns a new ranking function $\pi^{(j)}$ from D that minimizes a loss function over D (line 5). When collecting training data using $\pi^{(j-1)}$ in ECBS, we set a runtime limit for each instance. We record the success rate (i.e., the fraction of instances solved within the given runtime limit) on $\mathcal{I}$ (line 6) and the average runtime for the solved instances in $\mathcal{I}$ (line 7). Finally, DAgger returns the ranking function

Algorithm 2 Training Algorithm: Curriculum Learning

```
1: Input: Training instance sets {I_1, ..., I_m}
2: π_0 ← π*
3: for i = 1 to m do
4:     π_i ← DAgger(π_{i−1}, I_i)                ▷ Call Algorithm 3
5:     if π_i = π_{i−1} then                    ▷ Stopping criterion met
6:         ∀i < j ≤ m, π_j ← π_i
7:         break
8: return {π_1, ..., π_m}
```

Algorithm 3 DAgger$(\pi^{(0)}, \mathcal{I})$

```
1: D = ∅
2: for j = 1 to R do
3:     for I in training instance set 𝓘 do
4:         D ← D ∪ CollectData(I, π^(j−1))          ▷ Call ECBS
5:     π^(j) ← train a ranking function using D
6:     r_{j−1} ← success rate on 𝓘 using π^(j−1) in ECBS
7:     c_{j−1} ← average runtime on solved instances in 𝓘 using
       π^(j−1) in ECBS
8: L ← arg max_{0≤l′<R} {r_{l′}}
9: l ← an element from arg max_{l′∈L} {c_{l′}}
10: return π^(l)
```

that achieves the highest success rate on the instances in $\mathcal{I}$ in all R iterations (line 8), breaking ties in favor of the lowest average runtime for the solved instances (line 7).

We explain in details how we collect data and learn a ranking function in the following two subsections.

4.1.1 *Collecting Data.*

Given an instance I and a ranking function π, the subroutine CollectData(I, π) for data collection called in DAgger runs ECBS using the node-selection strategy given by the ranking function π and returns the entire CT $\mathcal{T}$. For each CT node $N \in \mathcal{T}$, it computes the following atomic features $f_1, \ldots, f_9$:

(1) features related to the conflicts: the number of conflicts $|N_{\text{Conf}}|$ (f_1), the number of pairs of agents that have at least one conflict with each other (f_2) and the number of agents that have at least one conflict with other agents (f_3);
(2) features related to N_{Cost}: $f_4 := N_{\text{Cost}}, f_5 := \frac{N_{\text{Cost}}}{\text{LB}}, f_6 := N_{\text{Cost}} - \text{LB}, f_7 := N_{\text{Cost}} - S$ and $f_8 := N_{\text{Cost}}/S$, where S is the sum of costs of the individually cost-minimal paths of all agents; and
(3) the depth of N in the CT (f_9).

From these atomic features, we obtain interaction features $f_i f_j$ (for $i \le j$), which are the pairwise products of the atomic features. The final feature vector $\phi(N) \in \mathbb{R}^p$ ($p = 54$) is obtained by concatenating all atomic features and interaction features, resulting in the degree-2 polynomial kernel in the space of atomic features. Features f_2 and f_3 can be computed in time $O(|N_{\text{Conf}}|)$, and the other features can be computed in time $O(1)$. Therefore, the overall time complexity for computing all 54 features is $O(|N_{\text{Conf}}|)$,

During data collection, we run ECBS until either T solutions are found, the search exceeds the runtime limit or the search terminates. If the search exceeds the runtime limit without finding any solution, we return an empty set of training data for instance I. Otherwise, for each CT node N, we assign a label y_N to it based on the minimum distance N_d between N and any solution found within the subtree rooted at N. We assign $N_d = \infty$ if no solution was found within its subtree. Since we want to assign smaller d-values to CT nodes that are closer to a solution, we need to label N in a way such that the closer N is to a solution, the smaller y_N is:

$$y_N = \begin{cases} 0, & \text{if } N_d < 10, \\ 1, & \text{if } 10 \le N_d < 30, \\ 2, & \text{if } 30 \le N_d < 60, \\ 3, & \text{if } 60 \le N_d < \infty, \\ \infty, & \text{otherwise.} \end{cases}$$

Our labeling scheme allows us to learn a ranking function that focuses on pairs of CT nodes that have large differences in N_d and, different from using $y_N = N_d$, avoids having to rank CT nodes correctly that are almost equally good or bad, which is irrelevant for making good node selections.

4.1.2 *Learning a Ranking Function.*

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

$$\pi : \mathbb{R}^p \to \mathbb{R} : \pi(\phi(N)) = \boldsymbol{w}^\mathsf{T} \phi(N)$$

and minimize the loss function

$$L(\boldsymbol{w}) = \sum_{\mathcal{T} \in D} l(y_\mathcal{T}, \hat{y}_\mathcal{T}) + \frac{C}{2} ||\boldsymbol{w}||_2^2$$

over the training data D, where $y_\mathcal{T}$ is the ground-truth label vector of all CT nodes in $\mathcal{T}$, $\hat{y}_\mathcal{T}$ is the vector of predicted values resulting from applying π to the feature vector $\phi(N)$ of every CT node $N \in \mathcal{T}$, $l(\cdot, \cdot)$ is a loss function that measures the difference between the ground truth label vector and the predicted value vector, and $C > 0$ is a regularization parameter. The loss function $l(\cdot, \cdot)$ is based on a weighted pairwise loss. Specifically, we consider the set of ordered CT node pairs

$$\mathcal{P}_\mathcal{T} = \{(N_i, N_j) \in \mathcal{Q}_\mathcal{T} : y_{N_i} > y_{N_j}\},$$

where $\mathcal{Q}_\mathcal{T}$ is the set of ordered CT node pairs (N_i, N_j) such that neither N_i nor N_j is an ancestor of the other CT node in $\mathcal{T}$. Restricting the loss function to take into account only pairs in $\mathcal{Q}_\mathcal{T}$ helps avoid the task of learning to rank two CT nodes that will never be in the focal list at the same time. The weight w_{N_i,N_j} of each pair $(N_i, N_j) \in \mathcal{P}_\mathcal{T}$ is set to $e^{-(d_i+d_j)/rd_{\max}}$, where d_i and d_j are the depths of N_i and N_j in $\mathcal{T}$ respectively, $d_{\max}$ is the depth of $\mathcal{T}$, and r is a damping factor. The weight w_{N_i,N_j} can be understood as the product of the weights of N_i and N_j, where the weight of N_i is $e^{-d_i/rd_{\max}}$. The loss function $l(\cdot, \cdot)$ is the weighted fraction of swapped pairs, defined as

$$l(y_\mathcal{T}, \hat{y}_\mathcal{T}) = \frac{\sum_{(N_i,N_j) \in \mathcal{P}_\mathcal{T} : \pi(\phi(N_i)) \le \pi(\phi(N_j))} w_{N_i,N_j}}{\sum_{(N_i,N_j) \in \mathcal{P}_\mathcal{T}} w_{N_i,N_j}}. \quad (1)$$

To learn π, we use an open-source solver LIBLINEAR [10] that implements a Support Vector Machine approach that minimizes an upper bound on the loss, since the loss itself is NP-hard to minimize.

Grid Map	Random	Warehouse	Maze	Game	City
w	1.1	1.05	1.01	1.005	1.005
m	10	11	9	16	13
k_1	75	140	45	80	160
k_m	125	240	125	305	400
$\|V\|$	819	5,699	14,818	28,178	47,240

Table 1: Parameters for each grid map. w is the suboptimality factor, m is the number of different numbers of agents we train and test on, k_1 is the number of agents that we start training on, k_m is the largest number of agents that we test on, and $|V|$ is the number of unblocked cells on the grid map. $k_2, \cdots, k_{m-1}$ are evenly distributed on $[k_1, k_m]$, i.e., $k_i = (i-1)(k_m - k_1)/(m-1) + k_1$.

4.2 ML-Guided Search

After learning the ranking functions $\{\pi_1, \ldots, \pi_m\}$, we deploy them in ECBS. Given an instance with k agents and the same underlying graph as used during training, we run ECBS with ranking function π_j, where $j \in \arg\min_{i \in [m]}\{|k - k_i|\}$, i.e. the one trained on the most similar number of agents. When a CT node N is generated, we compute its feature vector $\phi(N)$ and set its d-value to $\pi_j(\phi(N))$. The overall time complexity of computing the d-value is $O(|N_{\text{Conf}}|)$ because of the time complexity of computing the features. Even though the time complexity of computing the d-value for node-selection strategy h_1 is $O(1)$, we will show in experimentally that ECBS+ML outperforms ECBS with h_1 in terms of both the success rate and the runtime.

4.3 Discussion

Our main motivation to train multiple ranking functions for different numbers of agents is that, as will be shown in Section 5, the ranking functions learned with DAgger$(\cdot, \cdot)$ do not generalize well to instances with different numbers of agents, especially when those numbers are substantially larger than the one we train on. There are two reasons for this issue: (1) We are not able to normalize the feature values since we need to compute the d-value of a CT node immediately during the search when it is generated; and (2) different features are important for instances with different numbers of agents. Therefore, we learn node-selection strategies specific to the number of agents, and we use curriculum learning to learn them efficiently.

There are heuristic components in our training algorithm. The first component is the design of the stopping criterion for curriculum learning (line 5 in Algorithm 2). If we cannot improve on π_{i-1} in iteration i of the training algorithm, we are not able to improve on it in subsequent iterations either. The other component is the criterion for choosing the best ranking function in DAgger. One could argue that the best ranking function should be chosen based on the performance on a set of instances for validation drawn from the same distribution as for training (lines 8-9 in Algorithm 3). However, this would approximately double the runtime of DAgger and thus, also the training algorithm if the numbers of instances for validation and training were the same. Our criterion allows the training algorithm to select a good ranking function efficiently.

Grid Map	Random	Warehouse	Maze	Game	City
l_1	.0075	.0088	.0092	.0085	.0051
$l_{\lfloor m/2 \rfloor}$	.0330	.0166	.0192	.0131	.0068
l_m	.0653	.0283	.0318	.0204	.0107

Table 2: Loss $l_i \in [0,1]$ of ranking function π_i for k_i agents evaluated by Equation (1) averaged over all CTs in the training data.

5 EXPERIMENTAL RESULTS

We now demonstrate the efficiency and effectiveness of ECBS+ML through extensive experiments. We implement ECBS+ML in C++ and conduct our experiments on 2.4 GHz Intel Core i7 CPUs with 16 GB RAM. During testing, we compare against ECBS with the node-selection strategies h_1, h_2 and h_3 (ECBS+h_1, ECBS+h_2 and ECBS+h_3). We also compare against two versions of ECBS+ML, one that stops early, denoted by ECBS+ML(ES), and the other that uses only imitation learning, denoted by ECBS+IL. We set the runtime limit to 5 minutes per instance for running ECBS in both data collection and testing. The number of solutions T collected during data collection is set to 10. The number of iterations R for DAgger is also set to 10. The damping factor r for weight w_{N_i,N_j} is set to 0.3727. r is chosen so that a CT node at depth $0.6d_{\max}$ has weight $e^{-0.6/r} = 0.2$. Since we are using a pairwise loss, we suffer from a quadratic time complexity ($O(|\mathcal{T}|^2)$) for the loss computation. Therefore, we record only the first 10,000 CT nodes generated for each instance during data collection. We use the default values for all parameters in LIBLINEAR where the regularization parameter C is set to 1.

We evaluate ECBS+ML on five grid maps of different sizes and structures from the MAPF benchmark[1] [30], including: (1) a random map "random-32-32-20", which is a 32×32 grid map with 20% randomly blocked cells; (2) a warehouse map "warehouse-10-20-10-2-1", which is a 163×63 grid map with 200 10×2 rectangular obstacles; (3) a maze map "maze-128-128-10", which is a 128×128 grid map with ten-cell-wide corridors; (4) a game map "den520d", which is a 257×256 grid map from the video game *Dragon Age: Origins*; and (5) a city map "Paris_1_256", which is a 256×256 grid map of Paris. For testing, we use the "random" scenarios in the benchmark, yielding 25 instances for each number of agents on each grid map. For training, we generate another 25 instances drawn from the same distribution as the "random" scenarios for each $\mathcal{I}_i$ in the training algorithm. The parameters related to each grid map are listed in Table 1. k_1 is chosen such that at least one of ECBS+h_i (i=1,2,3) has a success rate of 88% or higher. We fix the increment between k_i and k_{i+1}, and k_m is chosen such that either the success rate of ECBS+ML falls below 20% or all ECBS+h_i (i=1,2,3) have 0% success rates. We fix the suboptimality factor w, following the reasoning in previous work [3], where small w values are chosen for large grid maps and vice versa. Our objective is to obtain a ranking function π_i for each number of agents k_i in $\{k_1, \ldots, k_m\}$. The training loss of the learned ranking functions is shown in Table 2. It is small. We now test the node-selection

[1] All data is publicly available at: https://movingai.com/benchmarks/mapf/index.html.

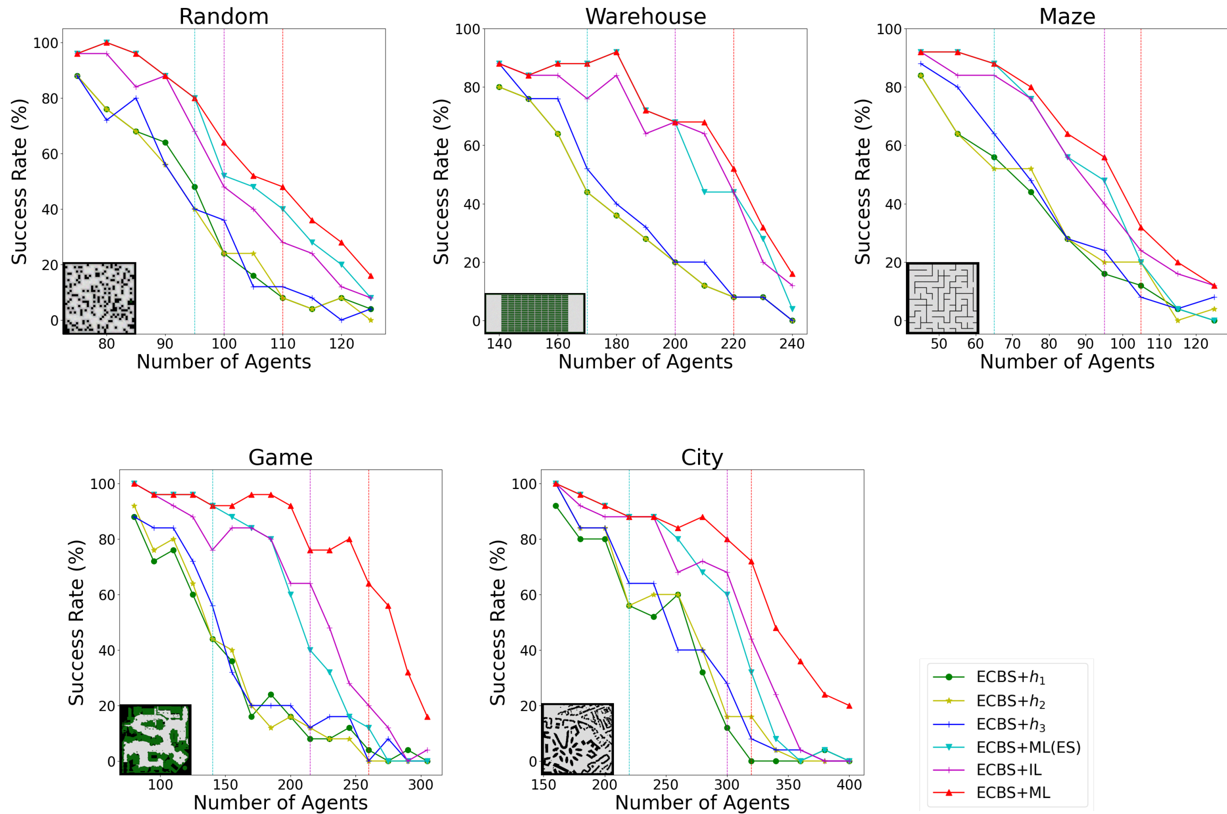


Figure 1: Success rates within the runtime limit of 5 minutes as a function of the number of agents. For ECBS+ML, ECBS+ML(ES) **and** ECBS+IL**, the vertical line of the same color indicates the number of agents in the last iteration that a ranking function is learned in the training algorithm. In the figure for the warehouse map, the graph of ECBS+h_1 is hidden entirely by the one of ECBS+h_2.**

strategies that correspond to those ranking functions on unseen instances with $k_1, \ldots, k_m$ agents.

5.1 Success Rate and Runtime

Figure 1 plots the success rates on all grid maps. Overall, ECBS+ML significantly outperforms the three baselines, ECBS+h_1, ECBS+h_2 and ECBS+h_3, in all grid maps. On the game map, in particular, the success rates of the baselines drop below 20% when the number of agents increases to 170, and the baselines could hardly solve instances with more than 245 agents, while the success rates of ECBS+ML stay above 76% for up to 245 agents and ECBS+ML can solve 16% of the instances with 305 agents. Overall, the success rates of ECBS+ML are 52% to 80% when those of the baselines begin to drop below 20%. When the success rates of the baselines are all below 8%, ECBS+ML can still solve instances with 9% to 17% more agents with success rates around 12% to 20%. To demonstrate the efficiency of ECBS+ML further, we show the success rates for different runtime limits in Figure 2. Due to space limits, we show only one figure for each grid map with a fixed number of agents, namely the smallest number of agents k_i where at least one of the baselines has a success rate below 50%. In these cases, ECBS+ML has a success rate above 80% and still significantly outperforms the baselines for shorter runtime limits, e.g., of 1 or 2 minutes.

Figure 3 shows the success rates of ECBS+ML and the baselines within the runtime limit of 5 minutes as a function of the suboptimality factor w on the random map for 95 agents. We use ECBS+ML with the ranking function obtained in the previous experiment that is trained on 95 agents and $w = 1.1$. To show that the success rates of ECBS+ML can be improved with curriculum learning, we use the same training algorithm (Algorithm 2) but, instead of using a fixed value for w and different numbers of agents, we use a fixed number of agents and give different values of w, namely, $w_i \in \{1.09, 1.08, \ldots, 1.05\}$. We then obtain a ranking function for each w_i. Figure 3 shows the success rates of the resulting solvers ECBS+ML(w). ECBS+ML(w) achieves higher success rates by applying curriculum learning on different values of w than ECBS+ML, which just generalizes the ranking function for $w = 1.1$ to other values of w.

5.2 Ablation Analysis

To assess the effect of curriculum learning, we perform two ablation analyses. First, we experiment with ECBS+ML(ES). ECBS+ML(ES)

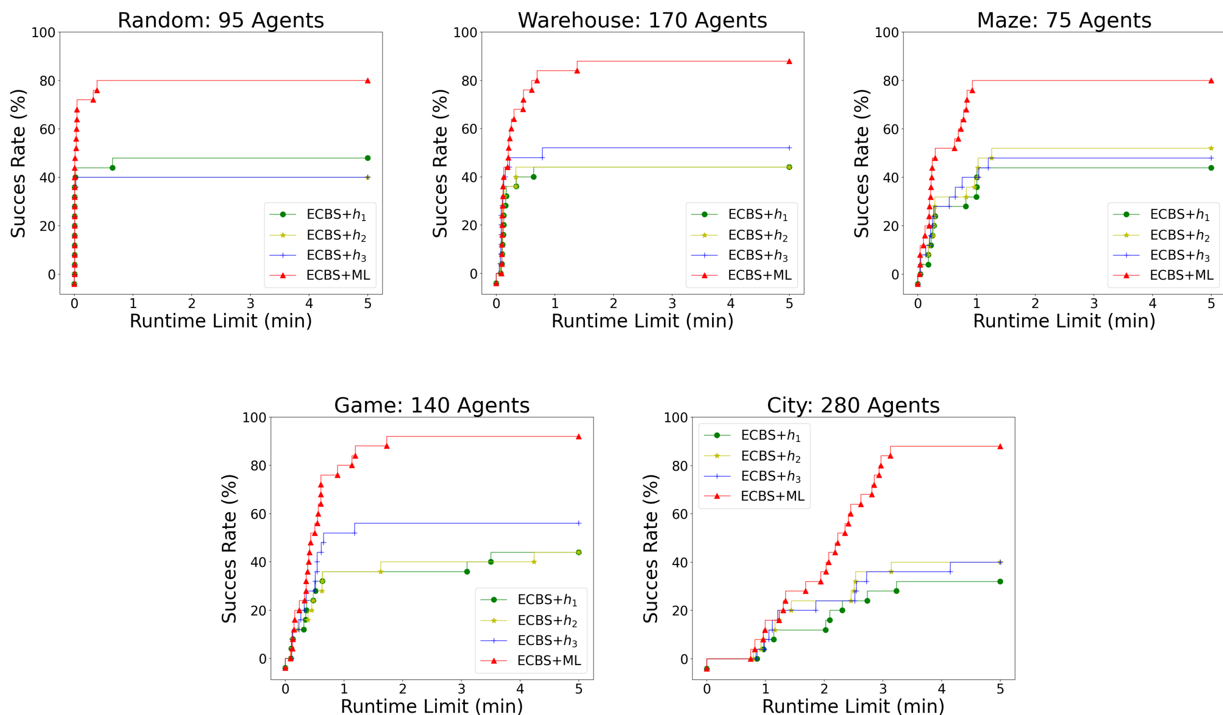


Figure 2: Success rates for a fixed number of agents as a function of the runtime limit for each grid map.

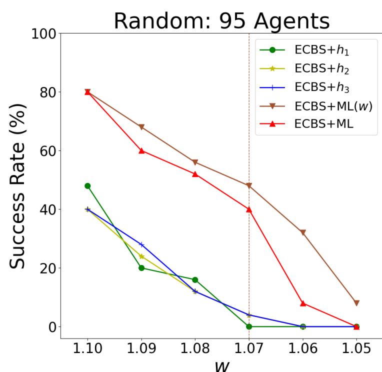


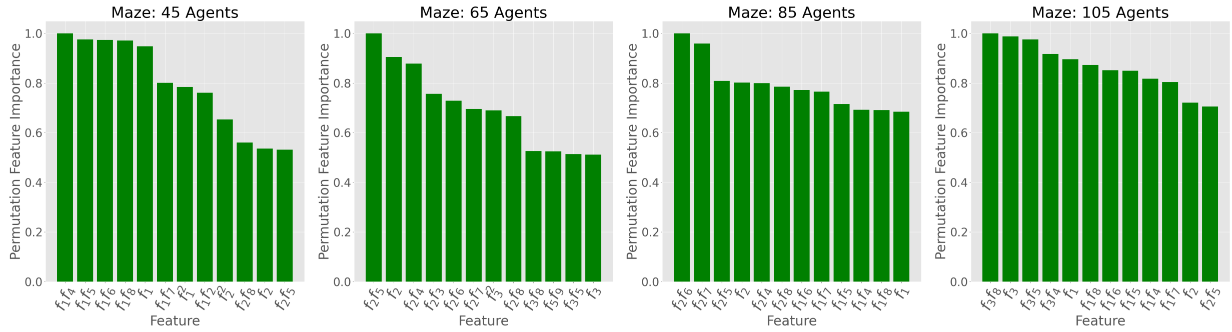
Figure 3: Success rates within the runtime limit of 5 minutes as a function of the suboptimality factor w on the random map for 95 agents. The vertical brown line indicates the value of w in the last iteration that a ranking function is learned for ECBS+ML(w).

uses the same training algorithm as ECBS+ML except that it stops training earlier than ECBS+ML. The number of agents in the last iteration of training in the training algorithm is the one where the success rate of ECBS+h_1 first drops below 60%. The success rates of ECBS+ML(ES) are shown in Figure 1. ECBS+ML(ES) is competitive with ECBS+ML and outperforms all baselines on the random, warehouse and maze maps, but its success rates on the city and game map drop dramatically beyond the number of agents that ECBS+ML(ES) stopped training at. The results imply that the learned node-selection strategy does not generalize well to larger numbers of agents on some grid maps and curriculum learning helps to find better strategies in those cases.

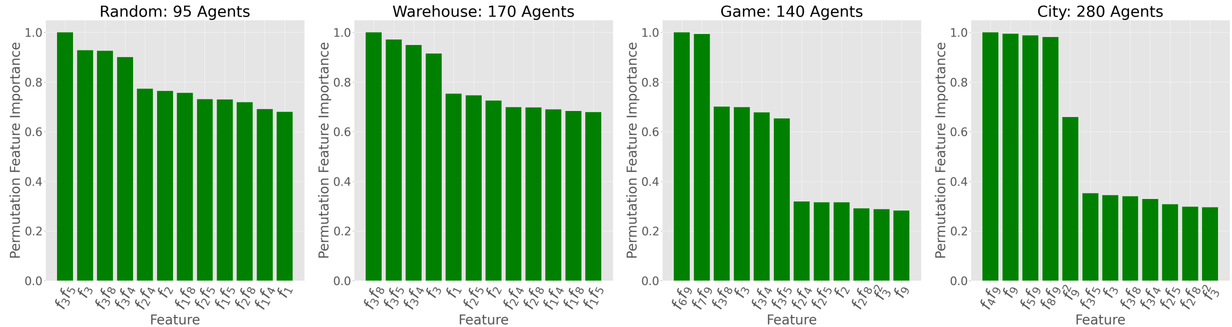
Second, we experiment with ECBS+IL. ECBS+IL uses the same training algorithm as ECBS+ML except that, for each number of agents, it learns a ranking function starting from the given initial ranking function without relying on the previously-learned one. We replace line 4 in the training algorithm with "$\pi_i \leftarrow \text{DAgger}(\pi^*, \mathcal{I}_i)$" and "$\pi_i = \pi_{i-1}$" on line 5 with "$\pi_i = \pi^*$". The success rates of ECBS+IL are shown in Figure 1. ECBS+IL outperforms the baselines but not as significantly as ECBS+ML. The results show another two advantages of curriculum learning: (1) It enables learning for one to three more iterations than ECBS+IL by enabling DAgger to collect more data for training due to being provided with better node-selection strategies for this purpose; and (2) it obtains better node-selection strategies based on the previously-learned strategies, as opposed to ECBS+IL that learns the node-selection strategy from the given initial ranking function in every iteration.

5.3 Feature Importance

Next, we study the feature importance of the learned ranking functions of ECBS+ML, measured by the permutation feature importance [1] of each feature, which is the increase in the loss on the training data after randomly permuting the values of that feature across all CT nodes for each CT in the training data. In Figure 4, we plot the normalized permutation feature importance of the top



(a) Permutation feature importance of the learned ranking functions for different numbers of agents on the maze maps.



(b) Permutation feature importance of the learned ranking functions for different grid maps.

Figure 4: Feature importance plots. We restate the definitions of some atomic features here (see Section 4.1.1 for the full list): f_1 is the number of conflicts, f_2 is the number of pairs of agents that have at least one conflict with each other, f_3 is the number of agents that have at least one conflict with other agents, and f_9 is the depth of the CT node.

12 features of the ranking functions for some numbers of agents and some grid maps. We first study the important features of the ranking functions when varying the numbers of agents for a single grid map , as shown in Figure 4a. We choose the maze map as a representative example to show that the learned node-selection strategies change as the number of agents increases. For 45 agents, the most important features are related to f_1 (the number of conflicts), followed by some features related to f_2 (the number of pairs of agents that have at least one conflict with each other). For both 65 and 85 agents, the top 6 features are those related to f_2, followed by some features related to f_3 (the number of agents that have at least one conflict with other agents) for 65 agents and f_1 for 85 agents. For 105 agents, the most important features are related to f_3, followed by some features related to f_1. To show that the set of important features varies across grid maps, we study the feature importance of the ranking functions for the random, warehouse, game and city maps, as shown in Figure 4b. The ranking functions are for the numbers of agents used in Figure 2. For the random and warehouse maps, the most important features are related to f_3, and the feature importance drops after the 4th feature. For the city map, the most important features include five features related to f_9 (the depth of the CT node). For the game map, the two most important features are also related to f_9, followed by some features related to f_3.

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed the first ML framework for learning node-selection strategies for ECBS. We deployed advanced ML techniques such as imitation learning and curriculum learning. Our extensive experimental results showed that ECBS+ML substantially improves on the current state-of-the-art bounded suboptimal MAPF solvers on different types of grid maps. It is future work to conduct feature selection to see whether we are still able to achieve good performance when reducing the feature set. It is also future work to apply our framework to instances that progressively increase in difficulty via problem parameters other than the number of agents, such as the suboptimality factor w or the obstacle density of the grid map.

ACKNOWLEDGMENTS

We thank Jiaoyang Li for helpful discussions. The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, 1837779 and 1935712, the U.S. Department of Homeland Security under Grant Award No. 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

[1] André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. 2010. Permutation importance: a corrected feature importance measure. *Bioinformatics* 26, 10 (2010), 1340–1347.

[2] Jacopo Banfi, Nicola Basilico, and Francesco Amigoni. 2017. Intractability of time-optimal multirobot path planning on 2d grid graphs with holes. *IEEE Robotics and Automation Letters* 2, 4 (2017), 1941–1947.

[3] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Annual Symposium on Combinatorial Search*.

[4] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *International Conference on Machine Learning*. 41–48.

[5] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Eyal Shimony. 2015. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *International Joint Conference on Artificial Intelligence*. 740–746.

[6] Liron Cohen, Tansel Uras, and Sven Koenig. 2015. Feasibility study: Using highways for bounded-suboptimal multi-agent path finding. In *Annual Symposium on Combinatorial Search*.

[7] Liron Cohen, Tansel Uras, TK Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. 2016. Improved solvers for bounded-suboptimal multi-agent path finding.. In *International Joint Conference on Artificial Intelligence*. 3067–3074.

[8] Hal Daumé, John Langford, and Daniel Marcu. 2009. Search-based structured prediction. *Machine Learning* 75, 3 (2009), 297–325.

[9] Kurt Dresner and Peter Stone. 2008. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research* 31 (2008), 591–656.

[10] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research* 9, Aug (2008), 1871–1874.

[11] Ariel Felner, Meir Goldenberg, Guni Sharon, Roni Stern, Tal Beja, Nathan R Sturtevant, Jonathan Schaeffer, and Robert Holte. 2012. Partial-Expansion A* with Selective Node Generation.. In *AAAI Conference on Artificial Intelligence*. 180–181.

[12] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, TK Satish Kumar, and Sven Koenig. 2018. Adding heuristics to conflict-based search for multi-agent path finding. In *International Conference on Automated Planning and Scheduling*. 83–87.

[13] He He, Hal Daume III, and Jason M. Eisner. 2014. Learning to search in branch and bound algorithms. In *Advances in Neural Information Processing Systems*. 3293–3301.

[14] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. 2019. Persistent and robust execution of MAPF schedules in warehouses. *IEEE Robotics and Automation Letters* 4, 2 (2019), 1125–1131.

[15] Taoan Huang, Bistra Dilkina, and Sven Koenig. 2021. Learning to Resolve Conflicts for Multi-Agent Path Finding with Conflict-Based Search. In *AAAI Conference on Artificial Intelligence*.

[16] Elias B Khalil, Bistra Dilkina, George L. Nemhauser, Shabbir Ahmed, and Yufen Shao. 2017. Learning to Run Heuristics in Tree Search.. In *International Joint Conference on Artificial Intelligence*. 659–666.

[17] Elias B. Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra Dilkina. 2016. Learning to branch in mixed integer programming. In *AAAI Conference on Artificial Intelligence*. 724–731.

[18] Edward Lam and Pierre Le Bodic. 2020. New Valid Inequalities in Branch-and-Cut-and-Price for Multi-Agent Path Finding. In *International Conference on Automated Planning and Scheduling*. 184–192.

[19] Edward Lam, Pierre Le Bodic, Daniel Damir Harabor, and Peter J Stuckey. 2019. Branch-and-Cut-and-Price for Multi-Agent Pathfinding.. In *International Joint Conference on Artificial Intelligence*. 1289–1296.

[20] Jiaoyang Li, Eli Boyarski, Ariel Felner, Hang Ma, and Sven Koenig. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search: Preliminary Results. In *Annual Symposium on Combinatorial Search*.

[21] Hang Ma, Jiaoyang Li, TK Satish Kumar, and Sven Koenig. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *International Conference on Autonomous Agents and MultiAgent Systems*. 837–845.

[22] Hang Ma, Jingxing Yang, Liron Cohen, TK Satish Kumar, and Sven Koenig. 2017. Feasibility study: Moving non-homogeneous teams in congested video game environments. In *Artificial Intelligence and Interactive Digital Entertainment*.

[23] Judea Pearl and Jin H. Kim. 1982. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 4 (1982), 392–399.

[24] Stéphane Ross and Drew Bagnell. 2010. Efficient reductions for imitation learning. In *International Conference on Artificial Intelligence and Statistics*. 661–668.

[25] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *International Conference on Artificial Intelligence and Statistics*. 627–635.

[26] Guillaume Sartoretti, Justin Kerr, Yunfei Shi, Glenn Wagner, TK Satish Kumar, Sven Koenig, and Howie Choset. 2019. PRIMAL: Pathfinding via reinforcement and imitation multi-agent learning. *IEEE Robotics and Automation Letters* (2019), 2378–2385.

[27] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219 (2015), 40–66.

[28] Jialin Song, Ravi Lanka, Albert Zhao, Aadyot Bhatnagar, Yisong Yue, and Masahiro Ono. 2018. Learning to search via retrospective imitation. *arXiv preprint arXiv:1804.00846* (2018).

[29] Trevor Scott Standley. 2010. Finding Optimal Solutions to Cooperative Pathfinding Problems.. In *AAAI Conference on Artificial Intelligence*. 28–29.

[30] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Satish Kumar, Eli Boyarski, and Roman Bartak. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Annual Symposium on Combinatorial Search*.

[31] Glenn Wagner and Howie Choset. 2011. M*: A complete multirobot path planning algorithm with performance bounds. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 3260–3267.

[32] Jingjin Yu and Steven M. LaValle. 2013. Planning optimal paths for multiple robots on graphs. In *IEEE International Conference on Robotics and Automation*. 3612–3617.