Subdimensional Expansion Using Attention-Based Learning For Multi-Agent Path Finding

Lakshay Virmani\textsuperscript{1*}, Zhongqiang Ren\textsuperscript{1*}, Sivakumar Rathinam\textsuperscript{2} and Howie Choset\textsuperscript{1}

Abstract—Multi-Agent Path Finding (MAPF) finds conflict-free paths for multiple agents from their respective start to goal locations. MAPF is challenging as the joint configuration space grows exponentially with respect to the number of agents. Among MAPF planners, search-based methods, such as CBS and M*, effectively bypass the curse of dimensionality by employing a dynamically-coupled strategy: agents are planned in a fully decoupled manner at first, where potential conflicts between agents are ignored; and then agents either follow their individual plans or are coupled together for planning to resolve the conflicts between them. In general, the number of conflicts to be resolved decides the run time of these planners and most of the existing work focuses on how to efficiently resolve these conflicts. In this work, we take a different view and aim to reduce the number of conflicts (and thus improve the overall search efficiency) by improving each agent’s individual plan. By leveraging a Visual Transformer, we develop a learning-based single-agent planner, which plans for a single agent while paying attention to both the structure of the map and other agents with whom conflicts may happen. We then develop a novel multi-agent planner called LM* by integrating this learning-based single-agent planner with M*. Our results show that for both “seen” and “unseen” maps, in comparison with M*, LM* has fewer conflicts to be resolved and thus, runs faster and enjoys higher success rates. We empirically show that MAPF solutions computed by LM* are near-optimal. Our code is available at https://github.com/lakshayvirmani/learning-assisted-mstar.

I. INTRODUCTION

Multi-Agent Path Finding (MAPF) aims to find collision-free paths for a team of agents from their respective start to goal locations while optimizing some path criteria, such as the sum of individual path lengths. MAPF arises in many applications ranging from automated warehouses [37] to aircraft towing [21]. MAPF is challenging as the joint configuration space of agents grows exponentially with respect to the number of agents, and solving MAPF to optimality is NP-hard [38].

Among MAPF planners, search-based methods such as CBS [27] and M* [33] effectively bypass this curse of dimensionality by employing a dynamically-coupled strategy: At first, agents are planned in a fully decoupled manner, where potential conflicts between agents are ignored; Then, agents either follow their individual plans or are coupled together for planning to resolve conflicts between them. In general, the number of conflicts to be resolved between agents decides the run time of these dynamically-coupled MAPF planners and many of the existing work focus on how to efficiently resolve these conflicts [6], [12], [13].

In this work, we take a different perspective and aim to reduce the number of potential conflicts between agents by improving their individual plans apriori in the first step. Instead of planning for each agent in a fully decoupled manner (i.e. ignoring other agents), we take a learning-based approach to let each agent consider the potential conflicts with other agents (rather than ignoring them) and plan its individual path to avoid these potential conflicts (Fig. 1).

Specifically, by leveraging a Visual Transformer [36], we develop an attention-based model that takes an agent’s observation at each time step as input and output the (predicted) action of that agent. This model can pay attention to the structure of the map and other agents’ information (such as start, goals, etc.), in order to avoid potential conflicts. The actions of an agent at all time steps computed by this model form an individual path from the start to the goal of

\textsuperscript{1}Lakshay Virmani, Zhongqiang Ren and Howie Choset are with Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA.

\textsuperscript{2}Sivakumar Rathinam is with Texas A&M University, College Station, TX 77843-3123.

\textsuperscript{*}The two authors contributed equally to this work.
that agent, and this model can thus be used as an individual planner for an agent. With that in hand, we then develop a novel MAPF planner called LM* (Learning-assisted M*): LM* begins by using this attention-based model to plan an individual path for each agent, and then couples agents together when needed by planning in their joint configuration spaces (just as in M*) to resolve conflicts.

To verify the proposed LM*, we first leverage an expert MAPF planner ODrM* [13] to provide labeled data for training, and then test LM* against the original M* (i.e., baseline) in several different maps with various obstacle densities and different structures such as room, maze, etc. Our results show that, (1) LM* can effectively reduce the number of conflicts among agents in comparison with M*, (2) LM* runs faster than M* in general (due to the reduced number of conflict), and (3) LM* empirically computes near-optimal solutions in a sense that the solution cost computed by LM* is less than 10% more expensive than the one computed by M* (and M* is guaranteed to provide an optimal solution).

The rest of the article is organized as follows: Sec. II reviews related work and Sec. III describes the method in detail. We then discuss the training and the testing settings in Sec. IV along with the numerical results. We conclude and list our future work in Sec. V.

II. PRIOR WORKS

A. Multi-Agent Path Finding (MAPF)

MAPF planners tend to fall on a spectrum from coupled [28], [29] to decoupled [7], trading off completeness and optimality for scalability. In the middle of the spectrum, several planners take a dynamically-coupled strategy to efficiently compute conflict-free paths with optimality guarantees. Among them, Conflict-Based Search (CBS) and its variants [27], [4], [6], [24], [1] employs a two-level search, where on the high level, conflicts between agents are detected and constraints on agents are generated, while on the low level, an individual path satisfying the added constraints for each agent is planned.

Subdimensional expansion [33], as another dynamically-coupled planner, begins by planning for each agent in a decoupled manner and then plans in the joint configuration space of agents to resolve conflicts. Subdimensional expansion bypasses the curse of dimensionality by modifying the dimension of the search space based on agent-agent conflicts. It inherits completeness and optimality if the underlying algorithm already has these features. While being general to many planners [34], [25], [23], most of the prior work focus on applying subdimensional expansion to A*, which results in M* [33]. In this work, we combine M* with attention-based learning [32], [36] to avoid conflicts when planning individual paths for agents.

In addition to these dynamically-coupled MAPF planners, another set of related MAPF planners are the reinforcement learning (RL)-based methods [26], [19], [9]. These RL-based planners use multi-agent reinforcement and imitation learning to learn fully-decentralized “end-to-end” policies, which maps a partially observed world as well as other agents’ information into actions for each agent for execution. Different from RL-based planners that learn end-to-end policies, our LM* combines learning and search-based planner by “embedding” attention-based model into M* for the purpose of reducing the number of conflicts and improving the search efficiency of M*.

B. Learning-Assisted Single-Agent Search

Using learning techniques to improve single-agent path planning algorithms has been investigated recently [35], [2], [31], [16]. For single-agent search-based planners, learning techniques are often leveraged to predict heuristic values [5], [16], [31], [18] of search states so that the search can be better guided towards the goal and the search effort can thus be reduced. In these papers, the models are typically trained to predict the heuristic value of all possible states within the search space in order to guide the search.

However, all these methods are limited to a single-agent. Applying them directly to a multi-agent system is non-trivial for the following two reasons. On one hand, the interactions between agents are not considered in the single-agent model, which is important for multi-agent problems. On the other hand, if we treat all agents as a single “meta-agent” and apply these single-agent methods to the corresponding joint configuration space, the curse of dimensionality makes these methods scale poorly. To handle this challenge, by leveraging the self-attention mechanism [32], we choose to train a single-agent model that can predict the next move for an agent while paying attention to the structure of the map and other agents with whom interactions may happen.

C. Attentions and Transformers

Transformers [32] were first introduced as a new attention-based method for machine translation. They introduced self-attention [3] layers which scan through each element of a sequence and aggregate information from the whole sequence. They are now widely used in various applications [20], [15]. Transformers, and more specifically, the self-attention mechanisms are able to model the relations among per-element such as tokens (for sequence data [32], [10]) and pixels (for image data [8]). The capability to model relations inspires us to leverage transformers to describe interactions (i.e., collision avoidance) between agents for MAPF.

Specifically, we take the view that transformers can help each agent to pay attention to the structure of the map and the subset of other agents that they have to interact with in order to avoid conflicts. However, applying transformer to a multi-agent system is challenging due to the curse of dimensionality of multi-agent systems as well as the computational burden of self-attention operations. To circumvent this challenge, we leverage the recent Visual Transformer [36] which operates in a semantic token space, judiciously attending to parts of the input based on the “context”. Using token space helps preserve the most important features and considerably reduces the number of parameters for the self-attention operation in Transformers.
III. OUR APPROACH

A. Problem Description

Let index set $I = \{1, 2, \ldots, N\}$ denote a set of $N$ agents. All agents move in a workspace represented as a finite graph $G = (V, E)$, where the vertex set $V$ represents the possible locations for agents and the edge set $E = V \times V$ denotes the set of all the possible actions that can move an agent between any two vertices in $V$. An edge between $u, v \in V$ is denoted as $(u, v)$, and the cost of an edge $e \in E$ is a non-negative real number cost($e$) $\in \mathbb{R}^+$. Let $\pi^i(v^t_i, v^t_{i'})$ denote a path for agent $i$ that connects vertices $v^t_i$ and $v^t_{i'}$ via a sequence of vertices $(v^t_1, v^t_2, \ldots, v^t_\ell)$ in the graph $G$. Let $g^i(\pi^i(v^t_i, v^t_{i'}))$ denote the cost associated with the path. This path cost is the sum of the costs of all the edges present in the path, i.e., $g^i(\pi^i(v^t_i, v^t_{i'})) = \sum_{j=1,2,\ldots,\ell-1} \text{cost}(v^t_j, v^t_{j+1})$.

All agents share a global clock. Each action, either wait or move, requires one unit of time for any agent. Any two agents $i, j \in I$ are claimed to be in conflict if one of the following two cases happens. The first case is a “vertex conflict” where two agents occupy the same vertex at the same time. The second case is an “edge conflict” (also called swap conflict) where two agents travel through the same edge from opposite directions between times $t$ and $t + 1$ for some $t$.

Let $v^t_{i_1}, v^t_{i_2} \in V$ denote the start and goal vertex of agent $i$. The Multi-Agent Path Finding (MAPF) problem aims to compute conflict-free paths for all agents while the sum of path costs reaches the minimum.

B. Approach Overview

As shown in Fig. 1, we develop an attention-based model that is able to “plan path” for a single agent while taking other agents and the map into consideration. Specifically, the model takes $O_i^t$, the observation of agent $i$ at time step $t$ (see Sec. III-E) as input, outputs the next action $A_i^t$ of that agent, and all of the computed actions $A^t_i, t = 0, 1, \ldots, T$, where $T$ is the time step where all agents have arrived at their respective goals, forms an individual path $\pi^i$ for agent $i$. During the training phase, the model uses data generated by solving various MAPF instances using an expert MAPF planner ODrlM* [13]. During the testing phase, each agent shares its observation $O_i^t$ as input to the model and the model predicts the action $A_i^t$ for the agent.

The model begins with a number of convolutional layers to extract the low-level features from $O_i^t$. The resulting output feature map then passes through a Visual Transformer (VT). The VT first uses a tokenizer to group pixels (of the feature map) into a small number of visual tokens; with each token representing a semantic concept in the image, Transformers are then applied to model relationships between these tokens. The attended visual tokens are then used as input to fully connected layers, which then output the predicted actions for agents.

C. Learning-Assisted M*

In this section, we describe how our attention-based model is integrated with M*. The regular M* [33] begins by running exhaustive backwards A* searches to compute an individual optimal policy $\phi^i$ for each agent $i$, where $\phi^i$ takes the current location of the agent and returns the next optimal move. The search process of M* is guided by these policies in a sense that agents either follow their individual policies if there is no conflict or are coupled together by planning in the joint configuration space to resolve conflicts.

In this work, instead of running a backwards A* search for each agent (which ignores any other agents), we compute these policies using the aforementioned attention-based model, which takes each agent’s observations and returns a next move for that agent. Since the output of both $\phi^i$ and the attention-based model are the same (i.e. the next move of an agent), the model can be readily integrated into M*. We name our approach learning-assisted M* (LM*) since learning-based method (i.e. the attention-based model) is leveraged to learn to predict and thus avoid conflicts when planning individual paths for each agent. For the rest of this section, we present this attention-based model in details.

D. Action Space

In this work, we consider the graph $G$ to be a four-connected grid in which agents are allowed to move in one of the four cardinal directions or to wait in place at each time step. Each action, either wait or move takes a unit time and incurs a unit cost. Moving into an obstacle is considered to be an invalid move, and if an agent selects to move into an obstacle during testing, it instead waits in place for that time step. In practice, after training, agents rarely choose an invalid move, which indicates that they effectively learn the set of valid actions at each location.

E. Observation Space

The observation $O_i^t$ of an agent consists of ten channels, where each channel comprises of a 32x32 size matrix.

1) Obstacles: Channel 1 contains a binary matrix which represents the free space and obstacles in the grid graph. Specifically, entries corresponding to free spaces have value zeros while entries of obstacles have value ones.

2) Agent $i$ Start, Goal, and Cost-to-go: Channel 2 and 3 consists of two matrices describing agent $i$’s start and goal respectively. In these matrices, the entry corresponding to the agent $i$’s start (or goal) has value one while all other entries have value zeros. We then generate another matrix (Channel 4) in which each entry has a value of the minimum cost-to-go from the corresponding location to agent $i$’s goal $v^t_{i'}$. These values are computed by running Dijkstra search backwards from $v^t_{i'}$ while ignoring any other agents. We scale this matrix so that all values lie between 0 and 1 (i.e. normalize).

3) Other Agents’ Starts, Goals, and Sum of Cost-to-go: Similarly, channel 5 and 6 of $O_i^t$ are two binary matrices that represent the start and goal locations of all other agents $(j \in \{1, 2, \ldots, N\}, j \neq i)$ respectively, where the entries corresponding to any other agents’ starts (or goals) have value ones while all other entries have value zeros. We also compute the cost-to-go for each agent $j$, sum up these cost values for each location and normalize, which results in a
H, W are the height and width of the feature map, and the feature map can be represented by X. Formally, a solution is then passed through a Visual Transformer (VT) model (Fig. 3). The output feature map from these convolutions is inherited from ResNet [14] as the CNN backbone of our fully connected layers for action classification.

Distributed, higher-order semantic concepts, and (3) finally learn sparsely-distributed, low-level patterns, (2) followed by components: (1) a convolutional neural network (CNN) to transform spatial dimensions into one. Visual tokens can be represented by a reshaped matrix of X obtained by merging the two channel 7. Intuitively speaking, these 3 matrices (i.e. channel 5, 6, 7) together provide the model the context about where the other agents in the world are headed and what route they might take.

4) Other Agents’ Future Positions: As shown in PRIMAL2 [9], an RL-based MAPF planner, considering the future positions of other agents can help avoid conflicts. In this work, similar to PRIMAL2 [9], we use three binary matrices to provide the future position of other agents, one per time step. These matrices constitute channel 8, 9 and 10 in O_t. It’s worthwhile to point that, in our experiments, we observed that adding channels 6-10 in O_t can help the model to scale to a larger number of agents, and the model achieves higher train and test accuracy.

F. Model Architecture

As shown in Figure 1, our model comprises of 3 main components: (1) a convolutional neural network (CNN) to learn densely-distributed, low-level patterns, (2) followed by a Visual Transformer to learn and relate more sparsely-distributed, higher-order semantic concepts, and (3) finally fully connected layers for action classification.

We use 3 residual learning building blocks (BasicBlock) inherited from ResNet [14] as the CNN backbone of our model (Fig. 3). The output feature map from these convolutions is then passed through a Visual Transformer (VT) (Fig. 2). VT uses a static tokenizer module to convert the feature map into a compact set of visual tokens. Formally, a feature map can be represented by \( X \in \mathbb{R}^{H \times W \times C} \), where \( H, W \) are the height and width of the feature map, and \( C \) is the channel size of the feature map. Consider \( X \in \mathbb{R}^{H \times W \times C} \) to be a reshaped matrix of X obtained by merging the two spatial dimensions into one. Visual tokens can be represented by \( T \in \mathbb{R}^{L \times C_T} \) where \( L \) is the number of visual tokens, and \( C_T \) is the channel size of the visual tokens. The static tokenization can be described by:

\[
A = \text{softmax}(XW_A^T) \quad (1)
\]

\[
V = XW_V \quad (2)
\]

\[
T = A^T V = \sum_i A[i,:]^T V[i,:] \quad (3)
\]

Here, \( A \in \mathbb{R}^{HW \times L} \) are normalized token coefficients, and the value of \( A[i;l] \in \mathbb{R} \) determines the contribution of the \( i \)-th pixel \( V[i;l] \) to the \( l \)-th token \( T[l,:] \). \( W_A \in \mathbb{R}^{L \times C} \) and \( W_V \in \mathbb{R}^{C \times C_T} \) are learnable weights used to compute A and to convert the feature map X into \( V \in \mathbb{R}^{HW \times C_T} \) respectively.

In order to perform classification, we use the approach of adding an extra learnable “classification token” to the visual tokens extracted from the feature map. Position embeddings are also added to retain positional information as done in [11]. The resulting vectors are then fed to the transformer [11], which can be described as:

\[
T_{out} = T_{in} + \text{softmax}(T_{in}K(T_{in}Q)^T)(T_{in}V)F, \quad (4)
\]

where \( T_{in}, T_{out} \in \mathbb{R}^{(L+1) \times C_T} \) are the input and output tokens, \( K \in \mathbb{R}^{C_T \times C_T/2} \), \( Q \in \mathbb{R}^{C_T \times C_T/2} \), \( V \in \mathbb{R}^{C_T \times C_T} \), \( F \in \mathbb{R}^{C_T \times C_T} \) are learnable weights used to compute keys, queries, values, and output tokens. We then use the attended classification tokens and cross-entropy loss to train the model for action classification.

IV. Experiments

A. Implementation Detail

As shown in Fig. 5, we stack 3 BasicBlock [14] with an output channel size of 32, 64, and 128 respectively, followed by a Visual Transformer (VT). The input to the VT is a feature map of size \( H = 8, W = 8 \) and \( C = 128 \). We extract \( L = 16 \) tokens from the feature map with a channel size of \( C_T = 256 \). The visual tokens are attended to using a transformer comprising of 16 encoder layers each containing multi-headed attention modules with 16 attention heads and multi-layer perceptron with a dimension of 512.

We implement the model in Python using the Pytorch [22] library. The model is trained and tested with a 3.30 GHz Intel(R) Core(TM) i9-9820X CPU and a NVIDIA GeForce RTX 2080 Ti GPU. During the training, we minimize the cross-entropy loss using the Adam optimizer [17] for 10 epochs. We use batch size of 64 and start with a learning rate of 0.003 which is decayed by a factor of 0.992 after every 10k steps. We obtain a top-1% accuracy of 91.3%
on the training set and 91.7% on the testing set. Such high train and validation accuracy indicates that the model is capable to select an optimal action for each agent during the planning process. To compare M* and LM* during the test, both algorithms are given 300 seconds to solve each MAPF instance. We test with three different inflation rates: 1.0, 1.1 and 10.0, which are used in M* [33] and PRIMAL [26].

### B. Data Generation

For training, we generate 10k (k stands for thousand) random maps. Each map has a size of 32x32 and the obstacles are placed randomly, where the probability of each cell being marked as an obstacle is randomly selected from \(\{0\%, 10\%, 20\%, 30\%, 40\%, 50\%\}\). In each map, we generate a MAPF instance with \(N = 2, 3, \ldots, 50\) respectively, where \(N\) is the number of agents in that instance. Thus, we generate 490k MAPF instances, 10k for each number of agents from 2 to 50. For each test instance, a unique start and goal is selected randomly for each agent, and it is ensured that a path from the start to the goal exists.

To provide labeled training data \(\{\{O_i^k, A_i^k\}\}\) (i.e. the observation of an agent at a certain time step and an action that should be selected by the agent at that time step), we use ODrM* [13] with an inflation of 1.1 and a time limit of 300 seconds to solve these MAPF instances. Out of the 490k MAPF instances, ODrM* is able to solve approximately 295k instances. Fig. 4 shows the number of solved instances with respect to the number of agents, under different obstacle probability.

![Fig. 4: The number of solved MAPF instances in the training set with respect to the number of agents, under different obstacle probability.](image)

To test on maps that appear in the training set, and hence the name “seen maps”, as LM* has “seen” these maps during the training. We randomly choose 100 maps from the train set and generate test instances with \(N = 5, 10, 15\ldots60\) agents (similar to Sec. IV-B). The success rates and the average run time to solution are shown in Fig. 5 (a), (b). LM* enjoys higher success rates than M* and shorter average run times in general, within the seen maps.

![Fig. 5: Success rates and the average run time to solution for different maps. “OBS PROB” stands for the obstacle probability of the map.](image)

#### TABLE I: This tables shows the percentage of decreases offered by LM* in comparison with M* in “seen” maps against the metrics: (left) the maximum size of the collision set during the planning, (middle) the number of nodes generated, and (right) the number of nodes expanded.

| Agents | \(\epsilon = 1.0\) | \(\epsilon = 1.1\) | \(\epsilon = 10.0\) |
|--------|----------------------|----------------------|----------------------|
|        | Max Collision Set Size | Nodes Generated | Nodes Expanded |
| 5      | 45 / 42 / 42         | 71 / 10 / 16       | 78 / -1 / -1       |
| 10     | 43 / 33 / 26         | 69 / 80 / 90       | 91 / 85 / 76       |
| 15     | 66 / 37 / 30         | 95 / 66 / 70       | 96 / 53 / 75       |
| 20     | 41 / 17 / 20         | 86 / -1 / 83       | 78 / -21 / -1      |
| 25     | - / 15 / 17          | - / 53 / 84        | - / 25 / 70        |
| 30     | - / 8 / 6            | - / 54 / 79        | - / 2 / 51         |
| 35     | - / - / 8            | - / - / 74         | - / - / 13         |

C. Seen Maps, Unseen Agent Positions

To compare the performance of M* and LM*, we first start to test on maps that appear in the training set, and hence the name “seen maps”, as LM* has “seen” these maps during the training. We randomly choose 100 maps from the train set and generate test instances with \(N = 5, 10, 15\ldots60\) agents (similar to Sec. IV-B). The success rates and the average run time to solution are shown in Fig. 5 (a), (b). LM* enjoys higher success rates than M* and shorter average run times in general, within the seen maps.

![C. Seen Maps, Unseen Agent Positions](image)
reduces the average run time. Finally, as shown in Table III for most (> 90%) of the test instances, the solution cost computed by LM* is less than 10% more expensive than the solution cost computed by M*. It shows that, empirically, LM* is able to produce near-optimal solutions.

D. Unseen Maps, Same Type

This section verifies whether LM* is able to generalize to unseen maps that are of the same type (i.e. random) as the maps in the training set. We generate the test instances using the convention as one in M* [33]: the grid map is of size 32x32 and each cell has a 20% probability to be occupied by an obstacle. Unique start and goal locations for each agent are chosen randomly and the existence of a path connecting the start and the goal is ensured.

![Image 1](image1.png)

**Table II**: This tables shows the percentage of decreases offered by LM* in comparison with M* in “unseen” maps against the same metrics as in Table I.

| Agents | $\epsilon = 1.0$ | $\epsilon = 1.1$ | $\epsilon = 10.0$ |
|--------|-----------------|-----------------|-----------------|
| Max Collision Set Size | Nodes Generated | Nodes Expanded |
| 5      | 76 / 73 / 71    | 82 / 27 / 49    | 91 / 10 / 8     |
| 10     | 52 / 52 / 49    | 49 / 47 / 90    | 61 / 4 / 17     |
| 15     | 57 / 43 / 99    | 85 / 38 / 99    | 74 / 8 / 34     |
| 20     | 94 / 25 / 37    | 99 / 59 / 99    | 92 / 51 / 49    |
| 25     | - / 19 / 28    | - / 28 / 89    | - / 16 / 43     |
| 30     | - / 17 / 17    | - / 45 / 87    | - / 52 / 50     |
| 35     | - / - / 12     | - / - / 88     | - / - / 52      |

E. Unseen Maps, Different Types

Finally, we verify whether LM* can generalize to unseen map of different types (such as room-like grid) as the training set (random grid). Here, we use the room (room-32-32-4) and maze maps (‘maze-32-32-2’, ‘maze-32-32-4’) from a online data set [30]. Due to the space limit, we omit the plots and summarize the results: In the room map, LM* achieves better success rates and shorter run time than M* in general, but in the maze-like maps, LM* fails to solve any instances when $N > 10$, while M* can still solve some of the instances. The possible reason is that: room-like map is “similar” to the random maps that appears in the training set while maze-like maps are drastically different as compared to the random maps in the training set. This result leads us to further explore how to improve LM* so that it can handle unseen maps with very different structures, in our future work.

V. CONCLUSION

In this work, we introduce a novel MAPF planner called Learning-assisted M* (LM*) by leveraging both attention-based learning [36] and M* [33]. LM* begins by running an attention-based model to plan for each agent, and couples agents together to plan in their joint configuration space to resolve conflicts between agents. Our results show that LM* is able to circumvent conflicts between agents and thus achieves higher success rates and shorter run time.

For future work, one can investigate whether the developed attention-based model can be fused with other MAPF planners such as CBS [27]. Additionally, one can also consider further improve this attention-based model to handle environments that are very different to the ones appeared in the training set.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. 2120219 and 2120529. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
