DMS*: Towards Minimizing Makespan for Multi-Agent Combinatorial Path Finding

Zhongqiang Ren, Anushtup Nandy, Sivakumar Rathinam and Howie Choset

***Abstract*—Multi-Agent Combinatorial Path Finding (MCPF) seeks collision-free paths for multiple agents from their start to goal locations, while visiting a set of intermediate target locations in the middle of the paths. MCPF is challenging as it involves both planning collision-free paths for multiple agents and target sequencing, i.e., solving traveling salesman problems to assign targets to and find the visiting order for the agents. Recent work develops methods to address MCPF while minimizing the sum of individual arrival times at goals. Such a problem formulation may result in paths with different arrival times and lead to a long makespan, the maximum arrival time, among the agents. This paper proposes a min-max variant of MCPF, denoted as MCPF-max, that minimizes the makespan of the agents. While the existing methods (such as MS*) for MCPF can be adapted to solve MCPF-max, we further develop two new techniques based on MS* to defer the expensive target sequencing during planning to expedite the overall computation. We analyze the properties of the resulting algorithm Deferred MS* (DMS*), and test DMS* with up to 20 agents and 80 targets. We demonstrate the use of DMS* on differential-drive robots.**

***Index Terms*—Path Planning for Multiple Mobile Robots or Agents, Motion and Path Planning, Task and Motion Planning**

I. Introduction

MULTI-Agent Path Finding (MAPF) seeks a set of collision-free paths for multiple agents from their respective start to goal locations. This paper considers a generalization of MAPF called Multi-Agent Combinatorial Path Finding (MCPF), where the agents need to visit a pre-specified set of intermediate target locations before reaching their goals. MAPF and MCPF arise in applications such as logistics [1]. For instance, factories use a fleet of mobile robots to visit a set of target locations to load machines for manufacturing. These robots share a cluttered environment and follow collision-free paths. In such settings, MAPF problems and their generalizations naturally arise to optimize operations.

MCPF is challenging as it involves both collision avoidance among the agents as in MAPF, and target sequencing, i.e., solving Traveling Salesman Problems (TSPs) [2], [3] to specify the assignment and visiting orders of targets for all agents. Both the TSP and the MAPF are NP-hard to solve to optimality [2], [4], and so is MCPF. A few methods were developed [5], [6] to address MCPF, and they often formulate the problem as a min-sum optimization problem, denoted as MCPF-sum, where the objective is to minimize the sum of individual arrival times. Such a formulation may result in an ensemble of paths where some agents arrive early while others arrive late, which leads to long execution times before all agents finish their paths. This paper thus proposes a min-max variant of MCPF (Fig. 1), denoted as MCPF-max, where the objective is to minimize the maximum arrival time, which is also called the makespan, of all agents.

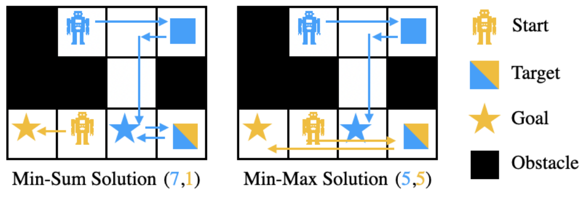

Fig. 1. MCPF-max and MCPF-sum. MCPF-max seeks a set of collision-free paths while minimizing the maximum arrival time of the agents. The color of a target or goal indicates the assignment constraints, i.e., the subset of agents that are eligible to visit that target or goal.

To solve MCPF-max, we first adapt our prior MS* algorithm [5], which was designed for MCPF-sum, to address MCPF-max. Then, we further develop two new techniques to expedite the planning, and we call the resulting new algorithm Deferred MS* (DMS*). Specifically, the existing MS* is a heuristic search approach (such as A*) by iteratively generating, selecting and expanding states to construct partial solution paths from the initial state to the goal state. MS* uses Traveling Salesman Problem (TSP) algorithms to compute target sequences for the agents. When solving the TSP, agent-agent collision are ignored, and the cost of resulting target sequences are thus lower bounds of the true costs to reach the goals. Therefore, the cost of target sequences provides an admissible heuristic to guide the state selection and expansion as in A*. Furthermore, MS* leverages the idea in M* [7] to first use the target sequences to build a low-dimensional search space, and then grow this search space by coupling agents together for planning only when collision happens. By doing so, MS* interleaves TSP (target sequencing) and MAPF (collision resolution) techniques using a heuristic search approach.

The first technique developed in this paper is applicable to MS* for both MCPF-max and MCPF-sum. When expanding a state, a set of successor states are generated, and for each of them, MS* needs to invoke the TSP solver to find the target sequence and the heuristic value of this successor state.

Manuscript received February, 15, 2024; Revised June, 5, 2024; Accepted July, 11, 2024. This paper was recommended for publication by Editor M. Ani Hsieh upon evaluation of the Associate Editor and Reviewers' comments. This work was supported by National Science Foundation under Grant No. 2120219 and 2120529. *(Corresponding author: Zhongqiang Ren.)*

Zhongqiang Ren is with Shanghai Jiao Tong University in China. (email: zhongqiang.ren@sjtu.edu.cn). Anushtup Nandy and Howie Choset are with Carnegie Mellon University in USA. (email: anandy@andrew.cmu.edu; choset@andrew.cmu.edu). Sivakumar Rathinam is with Texas A&M University in USA (email: srathinam@tamu.edu).

Digital Object Identifier (DOI):

Since the number of successor states can be large for each expansion, MS* needs to frequently invokes the TSP solver which slows down the computation. To remedy this issue, for each generated successor, we first use a fast-to-compute yet roughly estimated cost-to-go as the heuristic value, and defer calling TSP solver for target sequencing until that successor is selected for expansion.

The second technique is only applicable to MS* when solving MCPF-max, and does not work for MCPF-sum. Since the goal here is to minimize the makespan, during the search, agents with non-maximum arrival time naturally have "margins" in a sense that they can take a longer path without worsening the makespan of all agents. We take advantage of these margins to let agents re-use their previously computed target sequences and defer the expensive calls of TSP solvers until the margin depletes.

We analyze the conditions under which DMS* finds an optimal solution. To verify the methods, we conduct both simulation in various maps with up to 20 agents and 80 targets, as well as a simple real robot experiment. The simulation shows that: (i) DMS* finds paths that are up to 50% cheaper than an iterative greedy baseline method, and (ii) the new techniques in DMS* help triple the success rates and reduce the average runtime to solution comparing to MS*. The robot experiments show that the planned path are executable, and inspire future work.

II. Related Work

Multi-Agent Path Finding algorithms fall on a spectrum from coupled [8] to decoupled [9], trading off completeness and optimality for scalability. In the middle of this spectrum lie the dynamically-coupled methods such as M* [7] and CBS [10], which begin by planning for each agent a shortest path from the start to the goal ignoring any potential collision with the other agents, and then couple agents for planning only when necessary to avoid agent-agent collision.

Traveling Salesman Problems determine both the assignment and visiting order of the targets for the agents, where there are multiple intermediate targets to visit. For a single agent, the Traveling Salesman Problem (TSP) seeks a shortest tour that visits every vertex in a graph, and is a well-known NP-hard problem [2]. Closely related to TSP, the Hamiltonian Path Problem (HPP) requires finding a shortest path that visits each vertex in the graph from a start vertex to a goal vertex. The multi-agent version of the TSP and HPP (denoted as mTSP and mHPP, respectively) are more challenging since the vertices in the graph must be allocated to each agent in addition to finding the optimal visiting order of vertices. We refer to all these problems simply as TSPs. Different methods have been developed [2], [11], [12] to solve TSPs, trading off solution optimality for runtime efficiency. This paper does not develop new TSPs solvers and leverage the existing ones.

Target Assignment, Sequencing and Path Finding were recently combined in different ways [5], [6], [13]–[18]. Most of them either consider target assignment only (without the need for computing visiting orders of targets) [13]–[15], or consider the visiting order only given that each agent is pre-allocated a set of targets [16]–[18]. Our prior work [5], [6] seeks to handle the challenge in target assignment and ordering and the collision avoidance in MAPF simultaneously. These work uses the MCPF-sum formulation and the developed planners minimize the sum of individual arrival times, while this paper investigates MCPF-max. We do not extend our prior method CBSS [6] for MCPF-sum to MCPF-max, because CBSS requires solving a min-sum K-best sequencing problem, and it is not obvious how to handle the min-max variant of this K-best sequencing problem.

III. Problem

Let the index set $I = \{1, 2, \dots, N\}$ denote a set of N agents. All agents share a workspace that is represented as an undirected graph $G^W = (V^W, E^W, c^W)$, where W stands for workspace. Each vertex $v \in V^W$ represents a possible location of an agent. Each edge $e = (u, v) \in E^W \subseteq V^W \times V^W$ represents an action that moves an agent between u and v. $c^W : E^W \to (0, \infty)$ maps an edge to its positive cost value. In this paper, the cost of an edge is equal to its traversal time, and each edge has a unit cost.[1]

Let the superscript $i \in I$ over a variable denote the specific agent to which the variable belongs (e.g. $v^i \in V^W$ means a vertex corresponding to agent i). Let $v_o^i, v_d^i \in V^W$ denote the *initial* (or original) vertex and the *goal* (or destination) vertex of agent i respectively. Let $V_o, V_d \subset V^W$ denote the set of all initial and goal vertices of the agents respectively, and let $V_t \subset V^W \backslash \{V_o \cup V_d\}$ denote the set of *target* vertices. For each target $v \in V_t$, let $f_A(v) \subseteq I$ denote the subset of agents that are eligible to visit v; these sets are used to formulate the (agent-target) *assignment constraints*.[2]

Let $\pi^i(v_1^i, v_\ell^i)$ denote a path for agent i between vertices v_1^i and v_ℓ^i, which is a list of vertices $(v_1^i, v_2^i, \dots, v_\ell^i)$ in G^W with $(v_k^i, v_{k+1}^i) \in E^W, k = 1, 2, \cdots, \ell - 1$. Let $g(\pi^i(v_1^i, v_\ell^i))$ denote the cost of the path, which is the sum of the costs of all edges present in the path: $g(\pi^i(v_1^i, v_\ell^i)) = \Sigma_{j=1,2,\dots,\ell-1} c^W(v_j^i, v_{j+1}^i)$.

All agents share a global clock and start to move along their paths from time $t = 0$. Each action of the agents, either wait or move along an edge, requires one unit of time. Any two agents $i, j \in I$ are in *conflict* if one of the following two cases happens. The first case is a *vertex conflict* (i, j, v, t) where two agents $i, j \in I$ occupy the same vertex v at the same time t. The second case is an *edge conflict* (i, j, e, t), where two agents $i, j \in I$ go through the same edge e from opposite directions between times t and $t + 1$.

Definition 1 (MCPF-max Problem): The MCPF with Min-Max Objective (MCPF-max) seeks to find a set of conflict-free paths for the agents such that (1) each target $v \in V_t$ is visited at

[1]Here, the edge cost is the same as the edge traversal time, which is one unit for each edge. When the traversal time of edges are not unitary, it leads to continuous-time MAPF, and we refer the reader to [19], [20].

[2]An agent i "visits" a target $v \in V_t$ means (i) there exists a time t such that agent i occupies v along its path, and (ii) the agent i claims that v is visited. If a target v is in the middle of the path of i and i does not claim v is visited, then v is not considered as visited. A visited target v can appear in the path of another agent. When we say an agent or a path "visits" a target, we always mean the agent "visits and claims" the target. The assignment constraints do not forbid any agent $j \notin f_A(v)$ to use v in its path, and only forbid agent j to claim visiting v.

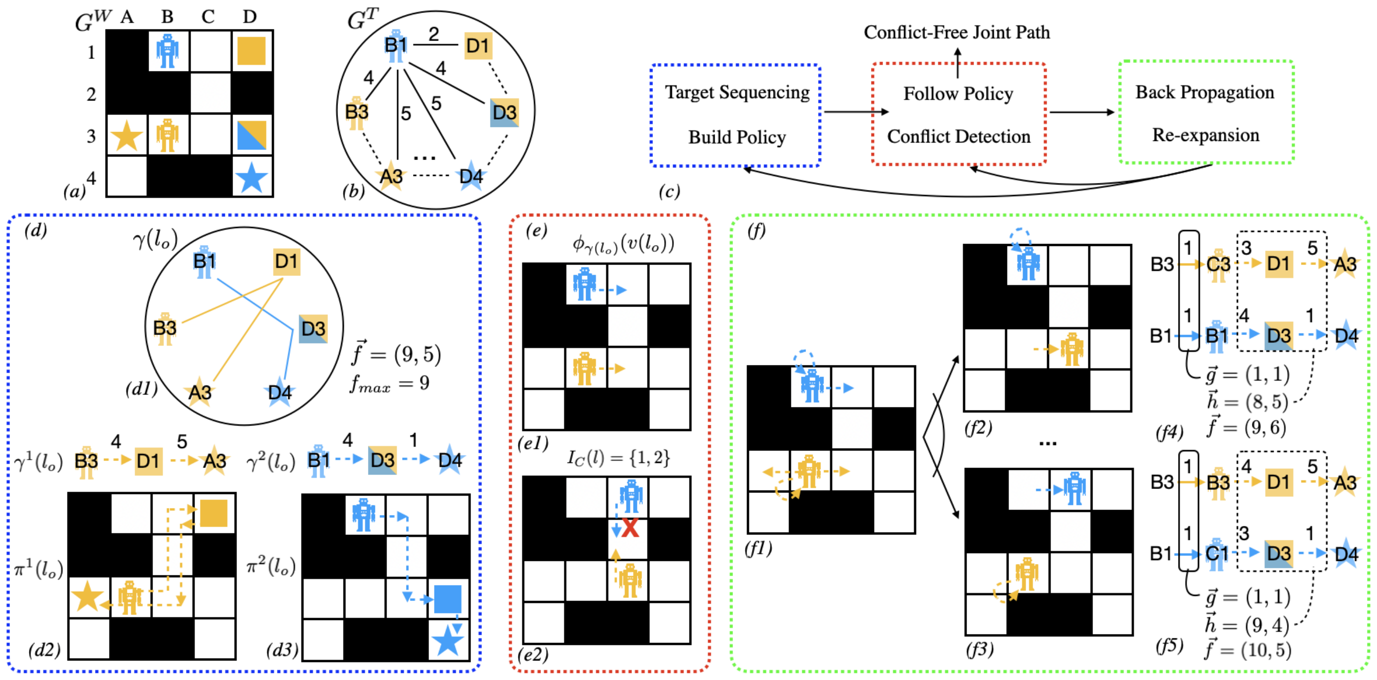

Fig. 2. An illustration of DMS* and related concepts. (a) shows the workspace graph G^W. (b) shows the target graph G^T where each edge in G^T corresponds to a minimum cost path in G^W between the respective vertices. (c) shows the workflow of DMS* in Alg. 1. (d) shows DMS* first ignores any agent-agent conflict and solves a corresponding mHPP, which provides a joint sequence $\gamma(l_o)$ (d1). This joint sequence $\gamma(l_o)$ can be converted to a joint path $\pi(l_o)$ (d2,d3), whose corresponding makespan f_{max} is 9. (e) The joint path $\pi(l_o)$ leads to a policy $\phi_{\gamma(l_o)}$ that maps a joint vertex to the next joint vertex along $\pi(l_o)$. Since conflicts are ignored in this policy, agents may run into conflict (e2). When a conflict is detected, the subset of agents that are in conflict $I_C(l)$ is back propagated to the ancestor labels. (f) After the back propagation, these ancestor labels and re-opened and re-expanded while considering all possible actions of all agents in conflict (f1,f2,f3). During the re-expansion, for each of the successors, DMS* first uses a fast-to-compute yet roughly estimated cost-to-go as the heuristic of the generated labels to avoid solving a mHPP. For both labels (f2) and (f3), this rough heuristic is $(9,5)-(1,1)=(8,4)$, where $(9,5)$ is the heuristic of (f1), the parent of (f2) and (f3), and $(1,1)$ means each agent can move at most one step towards their goals. When either (f2) or (f3) is popped from OPEN and before being expanded, DMS* re-computes a new heuristic and checks if the popped label should be expanded or re-added to OPEN for future expansion. To obtain this new heuristic, DMS* first attempts to let the agents follow the previously computed target sequences of the parent label and check if this worsens the makespan. For the successor shown in (f2), the resulting makespan is 9 (f4), which is no worse than the previous makespan 9, and DMS* will not call mHPP solver to save computational effort. For the successor shown in (f3), the resulting makespan is 10 (f5), which is worse than the previous makespan 9, and DMS* have to call mHPP solver to find a new joint sequence from that successor (f3) to the goals. Finally, with the joint sequence, new policy for the successors can be built and the search continues as in (e).

least once by some agent in $f_A(v)$, (2) the path for each agent $i \in I$ starts at its initial vertex and terminates at a unique goal vertex $v_d^i \in V_d$ such that $i \in f_A(v_d^i)$, and (3) the maximum of the cost of all agents' paths (i.e., $\max_{i\in I} g(\pi^i(v_o^i, v_d^i))$) reaches the minimum.

IV. Method

This section begins with an example in Fig. 2 to illustrate the planning process of DMS*. Then, Sec. IV-A introduces some concepts that will be used during the search process, before the presentation of the search algorithm in Sec. IV-B.

A. Concepts and Notations

1) Joint Graph: Let $\mathcal{G}^{\mathcal{W}} = (\mathcal{V}^{\mathcal{W}}, \mathcal{E}^{\mathcal{W}}) = \underbrace{G^W \times G^W \times \cdots \times G^W}_{N \text{ times}}$ denote the *joint graph* of the agents which is the Cartesian product of N copies of G^W, where each $v \in \mathcal{V}^{\mathcal{W}}$ represents a *joint vertex* and $e \in \mathcal{E}^{\mathcal{W}}$ represents a *joint edge* that connects a pair of joint vertices. Let $v_o = (v_o^1, v_o^2, \cdots, v_o^N)$ denote the initial joint vertex, which contains the initial vertices of all the agents. Let $\pi(u,v), u, v \in \mathcal{V}^{\mathcal{W}}$ denote a *joint path*, which is a tuple of N (individual) paths of the same length, i.e., $\pi(u,v) = (\pi^1(u^1,v^1), \cdots, \pi^N(u^N,v^N))$. A joint path π can be viewed as a list of joint vertices $(v_0, v_1, \cdots, v_\ell)$, and the time for the agents to arrive at each joint vertex $v \in \pi$ is specified by the index of v in π. Given two subsequent joint vertices u, v along a joint path π, let *CheckConflict*(u,v) denote a procedure that checks for vertex conflicts at u, v and edge conflicts during the transition from u to v among all pairs of agents, and *CheckConflict* returns a set $I_C \subseteq I$ of agents that are in conflict. We use a "path" to denote an "individual path". DMS* searches the joint graph $\mathcal{G}^{\mathcal{W}}$ for a joint path that solves the MCPF-max.

2) Binary Vector: For any $v \in \mathcal{V}^{\mathcal{W}}$, there can be multiple joint paths, e.g. $\pi_1(v_o,v), \pi_2(v_o,v)$, from v_o to v with different sets of targets visited, and we need to differentiate between them. First, without losing generality, let all targets in V_t be arranged as an ordered list $V_t = \{u_m, m = 1,2,\cdots,M\}$, where subscript m indicates the index of a target in this list.[3]

[3]In this paper, for a vector related to targets (e.g. a binary vector $\vec{a}$ of length M), we use subscripts (e.g. a_m) to indicate the elements in the vector. For a vector or joint vertex that is related to agents (e.g. $\vec{g}$ of length N, or $v \in \mathcal{V}^{\mathcal{W}}$), we use a superscript (e.g. i in g^i, v^i) to indicate the element in the vector or joint vertex corresponding to an agent.

Let $\vec{a} \in \{0,1\}^M$ denote a *binary vector* of length M that indicates the visiting status of all targets in V_t by a joint path during the search, where the m-th component of $\vec{a}$ is denoted as a_m, and $a_m = 1$ if u_m is visited, and $a_m = 0$ otherwise.

3) Label: Let $l = (v, \vec{a}, \vec{g})$ denote a *label*, where v is a joint vertex, $\vec{a}$ is a binary vector of length M, and $\vec{g}$ is a *cost vector* of length N. Here, each component $g^i, i \in I$ is the path cost of agent i. During the search, each label l identifies a joint path from v_o to v that visits a subset of targets as described by $\vec{a}$ and with path cost $\vec{g}$. Given l, let $v(l), \vec{a}(l), \vec{g}(l)$ denote the corresponding component in l, and let $v^i(l)$ denote the vertex of agent i in $v(l)$, $i \in I$. Let $g_{max}(l) := \max_{i \in I} \vec{g}(l)$ denote the maximum path cost over all agents in $\vec{g}(l)$. To solve the MCPF-max in Def. 1, the planner searches for a label l, whose corresponding joint path leads all agents to visit all targets and eventually reaches the goals, and $g_{max}(l)$ is the *objective value* to be minimized.

4) Label Comparison: To compare two labels at the same joint vertex, we compare both $\vec{a}$ and $\vec{g}$.

Definition 2 (Binary Dominance): For any two binary vectors $\vec{a}$ and $\vec{b}$, $\vec{a}$ dominates $\vec{b}$ ($\vec{a} \succeq_b \vec{b}$), if both the following conditions hold: (i) $\forall m \in \{1, 2, \cdots, M\}$, $a_m \geq b_m$; (ii) $\exists m \in \{1, 2, \cdots, M\}$, $a_m > b_m$.
Intuitively, $\vec{a} \succeq_b \vec{b}$ if $\vec{a}$ visits all targets that are visited in $\vec{b}$, and $\vec{a}$ visits at least one more target than $\vec{b}$. Two binary vectors are equal to each other ($\vec{a} = \vec{b}$) if both vectors are component-wise same to each other.

Definition 3 (Label Dominance): For two labels l_1, l_2 with $v(l_1) = v(l_2)$, l_1 dominates l_2 ($l_1 \succeq_l l_2$) if either (i) $\vec{a}(l_1) \succeq_b \vec{a}(l_2)$, $g_{max}(l_1) \leq g_{max}(l_2)$; or (ii) $\vec{a}(l_1) = \vec{a}(l_2)$, $g_{max}(l_1) < g_{max}(l_2)$ holds.
Intuitively, if $l_1 \succeq_l l_2$, then the joint path identified by l_1 is guaranteed to be better than the joint path identified by l_2. If l_1 does not dominate l_2, l_2 is then non-dominated by l_1. Any two labels are non-dominated (with respect to each other) if each of them is non-dominated by the other. Two labels are said to be *equal to* (or *same to*) each other (notationally $l_1 = l_2$) if $v(l_1) = v(l_2), a(l_1) = a(l_2), g_{max}(l_1) = g_{max}(l_2)$. There is no need to compare $\vec{g}(l_1)$ and $\vec{g}(l_2)$ when comparing labels l_1, l_2 since the problem in Sec. III seeks to minimize the maximum path cost over the agents. Finally, for each joint vertex $v \in \mathcal{V}^{\mathcal{W}}$, let $\mathcal{F}(v)$ denote a set of labels that are non-dominated to each other during the search.

5) Target Sequencing: Let $\gamma^i = \{v_o^i, u_1, u_2, \cdots, u_k, v_d^i\}$ denote an (individual) target sequence, where each $u_j, j = 1, 2, \cdots, k$ is a target vertex (i.e., $u_j \in V_t$). Let $\gamma = \{\gamma^1, \gamma^2, \cdots, \gamma^N\}$ denote a *joint sequence*, which specify the assignment and visiting order of all targets for all agents. Given l, $\vec{a}(l)$ specifies the set of targets that are visited and unvisited. We introduce the notation $\gamma(l) = \{\gamma^1(l), \gamma^2(l), \cdots, \gamma^N(l)\}$, a joint sequence *based on* l in the sense that $\gamma(l)$ visits all unvisited targets in $\vec{a}(l)$: (i) each $\gamma^i(l) \in \gamma(l)$ starts with $v^i(l)$ ($v^i(l)$ is not necessarily an origin vertex), visits a set of unvisited targets $\{u_m\}$ (where the m-th component of $\vec{a}(l)$ is zero), and ends with a goal vertex v_d^i; (ii) all $\gamma^i(l), i \in I$ together visits all unvisited targets in $\vec{a}(l)$. Intuitively, $\gamma(l)$ is a joint sequence that is meant to complete the joint path represented by l.

6) Target Graph: Let $\pi^*(u, v), u, v \in G^W$ denote a minimum-cost path between u, v in G^W, and let $c_{\pi^*}(u, v)$ denote the cost of path $\pi^*(u, v)$. The cost of a target sequence $c(\gamma^i(l))$ is equal to the sum of $c_{\pi^*}(u, v)$ for any two adjacent vertices u, v in $\gamma^i(l)$. Given l, to find $\gamma(l)$, a corresponding min-max multi-agent Hamiltonian path problem (mHPP) needs to be solved as follows. First, a target graph $G^T = (V^T, E^T, c^T)$ is created based on G^W. The V^T includes the current vertices of agents $v(l)$, the unvisited targets $\{u_m \in V_t | a_m(l) = 0\}$ and goals V_d. G^T is fully connected and $E^T = V^T \times V^T$. The edge cost in G^T of any pair of $u, v \in V^T$ is denoted as $c^T(u, v)$, which is the cost of a minimum cost path $\pi^*(u, v)$ in the G^W. An example G^T is shown in Fig. 2(b). A target sequence $\gamma^i(l)$ for an agent i is a path in G^T, and a $\gamma(l)$ is a set of paths that starts from $v(l)$, visits all unvisited targets in V_t as in $\vec{a}(l)$, and ends at goals V_d, while satisfying the assignment constraints. The procedure *SolveMHPP*(l) can be implemented by various existing algorithms for mHPP.

7) Heuristic and Policy: Given $\gamma(l)$, let $h^i(l) := c(\gamma^i(l))$ be a *heuristic* value. The vector $\vec{h} := \{h^i(l) | i \in I\}$ estimates the cost-to-go for each agent $i \in I$. When *SolveMHPP*(l) solves the mHPP to optimality, since conflicts are ignored along the target sequences, the corresponding $\vec{h}$ provides lower bounds of the cost-to-go for all agents, which is an admissible heuristic for the search. For any label l, let $\vec{f}(l) := \vec{g}(l) + \vec{h}(l)$ be the f-vector of l, which is an estimated cost vector of the entire joint path from v_o to goals for all agents by further extending the joint path represented by l. Let $f_{max}(l) := \max_{i \in I}(f^i(l))$ denote the maximum element in $\vec{f}(l)$, which provides an estimate of the objective value related to l.

Given l and $\gamma(l)$, a *joint policy* $\phi_{\gamma(l)}$ is built out of $\gamma(l)$, mapping one label to another as follows. First, a joint path π is built based on $\gamma(l)$ by replacing any two subsequent vertices $u, v \in \gamma^i(l), i \in I$ with a corresponding minimum cost path $\pi^*(u, v)$ in G^W. Then, along this joint path $\pi = (v_0, v_1, v_2, \cdots, v_\ell)$, all agents move from v_k to v_{k+1}, $k = 0, 1, \cdots, \ell - 1$. The corresponding binary vectors $\vec{a}_k$ for each $v_k \in \pi$ are built by first making $\vec{a}_0 = \vec{a}(l)$, and then updating $\vec{a}_k, k = 1, 2, \cdots, \ell$ based on $\vec{a}_{k-1}$ and v_k by checking if v_k visits any new targets. The corresponding cost vectors $\vec{g}_k$ are computed similarly as a_k by starting from $\vec{g}_0 = \vec{g}(l)$. As a result, a joint policy $\phi_{\gamma(l)}$ is built by mapping one label $l = (v, \vec{a}, \vec{g})$ to the next label $l' = (v', \vec{a}', \vec{g}')$ along the target sequence $\gamma(l)$. For a vertex v^i, let $\phi^i_{\gamma(l)}(v^i)$ denote the next vertex of agent i in the joint policy $\phi_{\gamma(l)}$. An example of $\phi^i_{\gamma(l)}$ is shown in Fig. 2(d). A label l is *on-policy* if its next label is known in $\phi_{\gamma(l')}$. Otherwise, l is *off-policy*, i.e., the next label is unknown and a mHPP needs to be solved for l to find the policy $\phi_{\gamma(l)}$. Let $\gamma(l), \vec{h}(l), \phi_{\gamma(l)} \leftarrow$*SolveMHPP*$(l)$ denote the process of computing the joint sequence, heuristic values and joint policy.

B. DMS* Algorithm

To initialize (Lines 1-3), DMS* creates an initial label l_o and calls *SolveMHPP* to compute $\gamma(l_o), \vec{h}(l_o), \phi_{\gamma(l_o)}$ for l_o. For each label l, DMS* uses two f-values: $f_{temp}(l)$ and

Algorithm 1 Pseudocode for DMS*

```
1: $l_o \leftarrow (v_o, \vec{a} = 0^M, \vec{g} = 0^N)$
2: $parent(l_o) \leftarrow NULL$, $f_{temp}(l_o) \leftarrow 0$
3: add $l_o$ to OPEN with $f_{temp}(l_o)$ as the priority
4: add $l_o$ to $\mathcal{F}(v_o)$
5: while OPEN is not empty do
6:     $l = (v, \vec{a}, \vec{g}) \leftarrow$ OPEN.pop()
7:     TargetSeq($parent(l)$, $l$, $I_C(parent(l))$)
8:     $f_{max}(l) \leftarrow \max_{i \in I}\{\vec{g}(l) + w \cdot \vec{h}(l)\}$
9:     if $f_{max}(l) > f_{temp}(l)$ then
10:        $f_{temp}(l) \leftarrow f_{max}(l)$
11:        add $l$ to OPEN with $f_{temp}(l)$ as the priority
12:        continue
13:    if CheckSuccess($l$) then
14:        return Reconstruct($l$)
15:    $L_{succ} \leftarrow$ GetSuccessors($l$)
16:    for all $l' \in L_{succ}$ do
17:        $I_C(l') \leftarrow$ CheckConflict($v(l), v(l')$)
18:        BackProp($l$, $I_C(l')$)
19:        if $I_C \neq \emptyset$ continue then
20:        if IsDominated($l'$) then
21:            DomBackProp($l, l'$)
22:            continue
23:        $f_{temp}(l') \leftarrow \max_{i \in I}\{\vec{g}(l') + w \cdot$ SimpleHeu$(l')\}$
24:        $parent(l') \leftarrow l$
25:        add $l'$ to $\mathcal{F}(v(l'))$ and back_set($l'$)
26:        add $l'$ OPEN with $f_{temp}(l')$ as the priority
27: return Failure (no solution)
```

Algorithm 2 Pseudocode for *TargetSeq*(l, l', $I_C(l)$)

```
1: if $l'$ is on-policy then
2:     $\gamma(l') \leftarrow \gamma(l)$, return
3: $\gamma(l') \leftarrow \gamma(l)$
4: Compute $\phi_{\gamma(l')}$ and $\vec{h}(l')$ based on $\gamma(l')$
5: if $\exists i \in I_C(l), g^i(l') + h^i(l') > f_{max}(l')$ then
6:     $\gamma(l'), \vec{h}(l'), \phi_{\gamma(l')} \leftarrow$ SolveMHPP($l'$)
7: return
```

Algorithm 3 Pseudocode for BackProp($l, I_C(l')$)

```
1: if $I_C(l') \not\subseteq I_C(l)$ then
2:     $I_C(l) \leftarrow I_C(l') \bigcup I_C(l)$
3:     if $l \notin$ OPEN then add $l$ to OPEN
4:     for all $l'' \in$ back_set($l$) do
5:         BackProp($l''$, $I_C(l)$)
```

Algorithm 4 Pseudocode for DomBackProp(l, l')

```
1: for all $l'' \in \mathcal{F}(v(l'))$ do
2:     if $l'' \succeq_l l'$ or $l'' = l'$ then
3:         BackProp($l$, $I_C(l'')$)
4:         add $l$ to back_set($l''$)
```

$f_{max}(l)$, where $f_{temp}(l)$ is a fast-to-compute yet roughly estimated cost-to-go, which does not require computing any joint sequence from l to the goals; and $f_{max}(l)$ is an estimated cost-to-go based on a joint sequence, which is computationally more expensive to obtain than $f_{temp}(l)$. Similarly to A* [21], let OPEN denote a priority queue storing labels and prioritizing labels based on their f-values from the minimum to the maximum. In Alg. 1, we point out which f-value (either f_{temp} or f_{max}) is used when a label is added to OPEN. Finally, l_o is added to $\mathcal{F}(v_o)$ since l_o is non-dominated by any other labels at v_o, and l_o is added to OPEN for future search.

In each iteration (Lines 5-26), DMS* pops a label l from OPEN. DMS* calls *TargetSeq* for l, which takes the parent label of l (denoted as $parent(l)$), l itself, and the conflict set of $parent(l)$. *TargetSeq* either calls *SolveMHPP* to find a joint sequence $\gamma(l)$, or re-uses the joint sequence $\gamma(parent(l))$ that is previously computed for $parent(l)$. We elaborate *TargetSeq* in Sec. IV-C. After *TargetSeq*, $\vec{h}(l)$ may change since a joint sequence may be computed for l within *TargetSeq*, DMS* thus computes $f_{max}(l)$ and compare it against $f_{temp}(l)$. When computing f_{max} out of $\vec{g}$ and $\vec{h}$ on Line 8, a heuristic inflation factor $w \in [1, \infty)$ is used, which scales each component in $\vec{h}$ by the factor w. Heuristic inflation is common for A* [22] and M*-based algorithms [7] that can often expedite the computation in practice while providing a w-bounded sub-optimal solution [22]. If $f_{max}(l) > f_{temp}(l)$, then l should not be expanded in the current iteration since there can be labels in OPEN that have smaller f-value than $f_{max}(l)$. DMS* thus updates $f_{temp}(l)$ to be $f_{max}(l)$, re-adds l to OPEN with the updated $f_{temp}(l)$, and ends the iteration. In a future iteration, when this label l is popped again, the condition on Line 9 will not hold since $f_{temp}(l) = f_{max}(l)$, and l will be expanded.

Afterwards, DMS* checks if l leads to a solution using *CheckSuccess*(l), which verifies if every component in $\vec{a}(l)$ is one and if every component of $v(l)$ is a unique goal vertex while satisfying the assignment constraints. If *Check-Success*(l) returns true, a solution joint path π^* is found and can be reconstructed by iterative tracking the parent pointers of labels from l to l_o in *Reconstruct*(l). DMS* then terminates. If *CheckSuccess*(l) returns false, l is expanded by considering its limited neighbors [7] described as follows. The limited neighbors of l is a set of successor labels of l. For each $i \in I$, if $i \notin I_C(l)$, agent i is only allowed to move to its next vertex $\phi^i_{\gamma(l)}(v^i(l))$ as defined in the joint policy $\phi_{\gamma(l)}$. If $i \in I_C(l)$, agent i is allowed to visit any adjacent vertex of $v^i(l)$ in G^W. The successor vertices of $v^i(l)$ are:

$$u^i \leftarrow \begin{cases} \phi^i_{\gamma(l)}(v^i(l)) & \text{if } i \notin I_C(l) \\ u^i \mid (v^i(l), u^i) \in E^W & \text{if } i \in I_C(l) \end{cases} \quad (1)$$

Let V^i_{succ} denote the set of successor vertices of $v^i(l)$, which is either of size one or equal to the number of edges incident on $v^i(l)$ in G^W. The successor joint vertices V_{succ} of $v(l)$ is then the combination of $v^i(l)$ for all $i \in I$, i.e., $V_{succ} := V^1_{succ} \times V^2_{succ} \times \cdots \times V^N_{succ}$. For each joint vertex $v' \in V_{succ}$, a corresponding l' is created and added to L_{succ}, the set of successor labels of l. When creating l', the corresponding $\vec{g}(l')$ and $\vec{a}(l')$ are computed based on $\vec{g}(l)$ $\vec{a}(l)$ and v'. In other words, the element in $\vec{g}(l')$ is one unit larger than the corresponding element in $\vec{g}(l)$ (unless the agent has reached the goal and stays there) since every agents takes an action, and the element in $\vec{a}(l')$ changes its value from 0 to 1, if v' visits any targets that are unvisited as in $a(l)$.

After generating the successor labels L_{succ} of l, for each $l' \in L_{succ}$, DMS* checks for conflicts between agents during the transition from $v(l)$ to $v(l')$, and store the subset of agents in conflict in the conflict set $I_C(l')$. DMS* then invokes *BackProp* (Alg. 3) to back propagate $I_C(l')$ to its ancestor labels recursively so that the conflict set of these ancestor labels are modified, and labels with modified conflict set are re-added to OPEN and will be re-expanded. DMS* maintains a back_set(l) for each l, which is a set of pointers pointing to the predecessor labels to which the back propagation should be conducted. Intuitively, similarly to [5], [7], the conflict set of labels are dynamically enlarged during planning when agents are detected in conflict. The conflict sets of labels determine the sub-graph within the joint graph $\mathcal{G}^{\mathcal{W}}$ that can be reached by DMS*, and DMS* always attempts to limit the search within a sub-graph of $\mathcal{G}^{\mathcal{W}}$ as small as possible.

Afterwards, if $I_C(l') \neq \emptyset$, l' leads to a conflict and is discarded. Otherwise, l' is checked for pruning by using dominance (Def. 3) against any existing labels in $\mathcal{F}(v(l'))$. If l' is pruned, any future joint path from l' can be cut and paste to $l'' \in \mathcal{F}(v(l'))$ that dominates l' without worsening the cost to reach the goals. Furthermore, for each $l'' \in \mathcal{F}(v(l'))$ that dominates or is equal to l', *DomBackProp*(Alg. 4) is invoked so that the conflict set $I_C(l')$ is back propagated to l, and l is added to the back_set of l''. By doing so, DMS* is able to keep updating the conflict set of the predecessor labels of l' after l' is pruned. This ensures the predecessor labels of l' will also be re-expanded after l' is pruned. If l' is not pruned, *SimpleHeu* is invoked for l' to compute a heuristic, which can be implemented by first copying $\vec{h}(l)$, where l is the parent of l', and then reduce each component of the copied vector by one except for the components that are already zero. This heuristic is an underestimate of the cost-to-go since all agents can move at most one step closer to their goals in each expansion. Then, $f_{temp}(l')$ is computed and l' is added to OPEN with $f_{temp}(l')$ as its priority. Other related data structure including $\mathcal{F}(v(l'))$, back_set, $parent$ are also updated correspondingly, and the iteration ends.

When DMS* terminates, it either finds a conflict-free joint path, or returns failure when OPEN is empty if the given instance is unsolvable.

C. Deferred Target Sequencing

DMS* introduces two techniques to defer the target sequencing until needed. As aforementioned, the first one uses a fast-to-compute yet roughly estimated cost-to-go as the heuristic when a label is generated, and invokes *TargetSeq* only when that label is popped from OPEN for expansion.

We now focus on the second technique in DMS*. In the previous MS* [5], every time when the search encounters a new label l that is off-policy, inside *TargetSeq*, *SolveMHPP* is invoked for l to find a joint sequence and policy from l, which burdens the overall computation, especially when a lot of new labels are generated due to the agent-agent conflict. Different from MS*, DMS* seeks to defer the call of *SolveMHPP* inside *TargetSeq*. For a label l', DMS* attempts to avoid calling *SolveMHPP* for l' by re-using the joint sequence $\gamma(l)$ of its parent label l (l is the parent of l'). As presented in Alg. 2, on Lines 3-4, DMS* first attemps to build a policy from l' by following the joint sequence of its parent label $\gamma(l)$ and computes the corresponding cost-to-go $\vec{h}(l')$. Agents $i \notin I_C(l)$ are still along their individual paths as in the previously computed policy $\phi_{\gamma(l)}$. Agents $i \in I_C(l)$ consider all actions as in Equation (1) and may deviate from the individual paths specified by the previously computed policy $\phi_{\gamma(l)}$. Therefore, DMS* needs to go through a check for these agents $i \in I_C(l)$: DMS* first computes the f-vector by summing up $\vec{g}(l')$ and $\vec{h}(l')$. Then, if $f^i(l')$ of some agent $i \in I_C(l)$ is no larger than $f_{max}(l)$, DMS* can avoid calling *SolveMHPP*, since letting the agents follow $\gamma(l)$ in the future will not worsen the objective value, the makespan. Otherwise, there exists an agent $i \in I_C(l)$, whose corresponding $f^i(l')$ is greater than $f_{max}(l)$, and in this case, DMS* cannot avoid calling *SolveMHPP* since there may exists another joint sequence from l', which leads to a solution joint path with better (smaller) objective value.

D. Properties of DMS*

This section discusses the solution optimality of DMS* and more analysis can be found in [23]. DMS* only finds an optimal solution for MCPF-max under the following two assumptions: (A1) *SolveMHPP* returns an optimal joint sequence for the given mHPP instance. (A2) Line 5 in Alg. 1 never returned true during the search of DMS*.

Theorem 1: For a solvable instance, when assumptions A1 and A2 hold, DMS* returns an optimal solution joint path.

Proof 1: The proof follows the analysis in [5], [7]. The policies in DMS* defines a sub-graph $\mathcal{G}^W_{sub}$ of the joint graph $\mathcal{G}^{\mathcal{W}}$. DMS* first expands labels in $\mathcal{G}^W_{sub}$. If no conflict is detected when following the policy, the resulting joint path is conflict-free and optimal due to A1: With A1, $h_{max}(l)$ for any label l computed by *SolveMHPP* is an estimated cost-to-go that is admissible, i.e., $h_{max}(l)$ is no larger than the true optimal cost-to-go. If conflicts are detected, DMS* updates (i.e., enlarges) $\mathcal{G}^W_{sub}$ by growing the conflict set and back propagating the conflict set. When A2 holds, the enlarged $\mathcal{G}^W_{sub}$ still ensures that an optimal conflict-free joint path π_* is contained in the updated $\mathcal{G}^W_{sub}$ [5], [7]. DMS* systematically search over $\mathcal{G}^W_{sub}$ and finds π_* at termination.

We now explain why DMS* loses the optimality guarantee if A2 is violated. In MCPF-max, all agents are "coupled" when assigning the targets, since a target visited by one agent does not need to be visited by any other agents. When one agent changes its target sequence and visits a target that is previously assigned to another agent, all agents may need to be re-planned to ensure solution optimality. When Line 5 in Alg. 1 returns true for label l', *SolveMHPP* needs to be called for l' and *SolveMHPP* may return a new joint sequence γ'. In this new sequence γ', it is possible that agents within $I_C(l)$ visit targets that are previously assigned to agents that are not in $I_C(l)$. However, DMS* does not let $i \notin I_C(l)$ to consider all possible actions in Equation (1). An optimal solution joint path may lie outside $\mathcal{G}^W_{sub}$ and DMS* thus loses the solution optimality guarantee when A2 is violated. [4]

Additionally, when A1 and A2 hold, DMS* can use the conventional heuristic inflation technique [22] and provide

[4]To ensure solution optimality when A2 is violated, same as in [5], one way is to back propagate the entire $I = \{1, 2, \cdots, N\}$ as the conflict set when calling *BackProp*. This ensures DMS* considers all actions of all agents when conflicts are detected. In practice, this is computationally burdensome for large N.

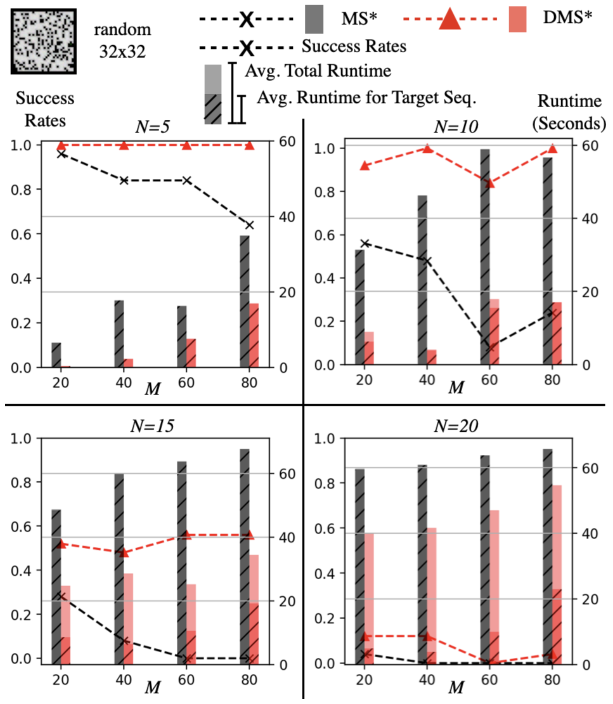

Fig. 3. The success rates and runtime of DMS* and MS* (baseline) with varying number of agents and targets in a random 32x32 map. DMS* has higher success rates and less runtime on average than MS*.

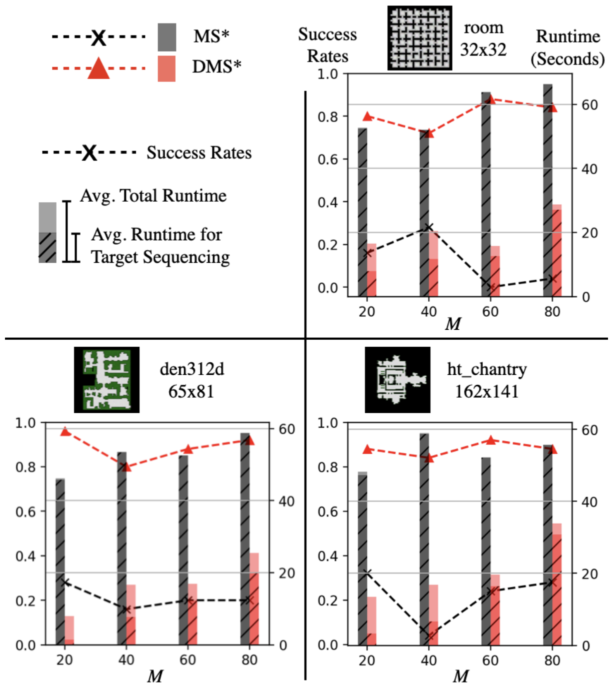

Fig. 4. The success rates and runtime of both DMS* and MS* (baseline) with varying number of targets in maps of different sizes. DMS* achieves higher success rates than MS* while requiring less runtime on average.

bounded sub-optimal solution. Additionally, when A2 holds, bounded sub-optimal joint sequences (e.g. by using an approximation algorithm to solve the MHPP) lead to inflated heuristic in DMS*, which leads to bounded sub-optimal solutions [22].

V. Experimental Results

We implement DMS* in Python, with Google OR-Tool as the mHPP solver. Limited by our knowledge on Google OR-Tool, we only consider the following type of assignment constraints, where any agent can visit all targets and goals. We set up a 60-second runtime limit for each instance, where each instance contains the starts, targets and goals in a grid map. All tests run on a MacBook Pro with a Apple M2 Pro CPU and 16GM RAM. It is known from [7], M*-based algorithms with inflated heuristics can often scale to more agents, and we set $w = 1.1$ in our tests. We set the number of targets $M = 20, 40, 60, 80$. We compare DMS*, which includes the two proposed technique to defer target sequencing, against two baselines. The first one is MS*, which does not have these two techniques. In other words, MS* here is a naive adaption of the existing MS* [5] algorithm to solve MCPF-max, by using the aforementioned Google OR-Tool to solve min-max mHPP for target sequencing. The second baseline is a iterative greedy approach which assigns one unvisited target to an agent to minimize the makespan of all agents in the current iteration while planning collision free paths.

A. Simulation Results

1) Varying Number of Agents: We first fix the map to Random 32x32, and vary the number of agents $N = 5, 10, 15, 20$. We measure the success rates, the average runtime to solution and the average runtime for target sequencing per instance. The averages are taken over all instances, including both succeeded instances and instances where the algorithm times out. As shown in Fig. 3, DMS* achieves higher success rates and lower runtime than the baseline MS*. As M increases from 20 to 80, both algorithms require more runtime for target sequencing. As N increases from 5 to 20, agents have higher density and are more likely to run into conflict with each other. As a result, both algorithms time out for more instances. Additionally, MS* spends almost all of its runtime in target sequencing, which indicates the computational burden caused by the frequent call of mHPP solver. In contrast, for DMS*, when $N = 5, 10$, DMS* spends most of the runtime in target sequencing, while as N increases to $15, 20$, DMS* spends more runtime in path planning.

2) Different Maps: We then fixed $N = 10$ and test in maps of different sizes (Fig. 4). DMS* outperforms MS* in success rates due to the alleviated computational burden for target sequencing. Larger maps do not lead to lower success rates since larger maps can reduce the density of the agents and make the agents less likely to run into conflicts with each other. The Room 32x32 map is more challenging than the other two maps since there are many narrow corridors which often lead to conflicts between the agents.

3) Solution Quality: We use the random 32x32 map and fixed the number of agents to $N = 5$ where DMS* achieves 100% success rate. We compare the solution cost ratio of DMS* over the greedy baseline. As shown in Table I, the solution of DMS* is up to around 50% cheaper than the solution of this greedy baseline.

M	min. CR	median CR	max.CR
20	0.47	0.71	0.94
40	0.46	0.63	0.86
60	0.53	0.66	0.76
80	0.56	0.72	0.87

TABLE I
COST RATIOS (CR) OF DMS* OVER A GREEDY BASELINE. SOLUTION COSTS OF DMS* ARE UP TO 50% CHEAPER THAN THE GREEDY.

M	20	40	60	80
MS*	72.2	67.6	75.0	71.8
DMS*	77.2	84.4	93.4	99.9

TABLE II
THE AVERAGE NUMBER OF EXPANSION OF MS* AND DMS* AMONG THE INSTANCES WHERE BOTH ALGORITHMS SUCCEED.

4) Number of Expansions: Table II shows the average number of label expansion in both MS* and DMS* among the instances in the Random 32x32 map which are successfully solved by both algorithms within the runtime limit. Due to the deferred sequencing, DMS* tends to search in a less informed manner and needs more expansion.

B. Experiments with Mobile Robots

We test our method with two differential drive robots, which are shown in the multi-media attachment. This test verifies that the paths planned by DMS* are executable on real robots. When the robots have large motion disturbance, such as delay or deviation from the planned path, additional techniques (such as [24]) will be needed to ensure collision-free execution of the paths.

VI. Conclusion and Future Work

This paper investigates a min-max variant of Multi-Agent Combinatorial Path Finding problem and develops DMS* algorithm to solve this problem. We test DMS* with up to 20 agents and 80 targets and conduct simple robot experiments to showcase the usage of DMS*.

For future work, one can investigate extending DMS* to the case where edges have non-unitary traversal times [19], [20], or simultaneously optimizing both the maximum and the sum of individual arrival times [25]. We note from our experiments the disturbance in robot motion may affect the execution of the planned path, and one can develop fast online replanning version of DMS* to handle the disturbance.

References

[1] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI magazine*, vol. 29, no. 1, pp. 9–19, 2008.

[2] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton, NJ, USA: Princeton University Press, 2007.

[3] T. Bektas, "The multiple traveling salesman problem: an overview of formulations and solution procedures," *omega*, vol. 34, no. 3, pp. 209–219, 2006.

[4] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[5] Z. Ren, S. Rathinam, and H. Choset, "MS*: A new exact algorithm for multi-agent simultaneous multi-goal sequencing and path finding," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021.

[6] ——, "CBSS: A new approach for multiagent combinatorial path finding," *IEEE Transactions on Robotics*, vol. 39, no. 4, pp. 2669–2683, 2023.

[7] G. Wagner and H. Choset, "Subdimensional expansion for multirobot path planning," *Artificial Intelligence*, vol. 219, pp. 1–24, 2015.

[8] T. S. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[9] D. Silver, "Cooperative pathfinding," in *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, vol. 1, no. 1, 2005, pp. 117–122.

[10] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[11] K. Helsgaun, "General k-opt submoves for the lin–kernighan tsp heuristic," *Mathematical Programming Computation*, vol. 1, no. 2, pp. 119–163, 2009.

[12] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1976.

[13] W. Hönig, S. Kiesel, A. Tinka, J. Durham, and N. Ayanian, "Conflict-based search with optimal task assignment," in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 2018.

[14] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, 2016, pp. 1144–1152.

[15] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh, "Generalized target assignment and path finding using answer set programming," in *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[16] P. Surynek, "Multi-goal multi-agent path finding via decoupled and integrated goal vertex ordering," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 12, no. 1, 2021, pp. 197–199.

[17] X. Zhong, J. Li, S. Koenig, and H. Ma, "Optimal and bounded-suboptimal multi-goal task assignment and path finding," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 10 731–10 737.

[18] H. Zhang, J. Chen, J. Li, B. C. Williams, and S. Koenig, "Multi-agent path finding for precedence-constrained goal sequences," in *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, 2022, pp. 1464–1472.

[19] A. Andreychuk, K. Yakovlev, P. Surynek, D. Atzmon, and R. Stern, "Multi-agent pathfinding with continuous time," *Artificial Intelligence*, vol. 305, p. 103662, 2022.

[20] Z. Ren, S. Rathinam, and H. Choset, "Loosely synchronized search for multi-agent path finding with asynchronous actions," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2021.

[21] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[22] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[23] Z. Ren, A. Nandy, S. Rathinam, and H. Choset, "Dms*: Minimizing makespan for multi-agent combinatorial path finding," *arXiv preprint arXiv:2312.06314*, 2023.

[24] W. Hönig, T. S. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, and S. Koenig, "Multi-agent path finding with kinematic constraints," in *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

[25] Z. Ren, C. Zhang, S. Rathinam, and H. Choset, "Search algorithms for multi-agent teamwise cooperative path finding," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 1407–1413.