A Bounded Sub-Optimal Approach for Multi-Agent Combinatorial Path Finding

Zhongqiang Ren[1], Sivakumar Rathinam[2] and Howie Choset[1]

***Abstract*—Multi-Agent Path Finding (MAPF) seeks collision-free paths for multiple agents from start to goal locations. This paper considers a generalization of MAPF called Multi-Agent Combinatorial Path Finding (MCPF) where agents must collectively visit a set of intermediate target locations before reaching their goals. MCPF is challenging as it involves both planning collision-free paths for multiple agents and target sequencing, i.e., assigning targets to and computing the visiting order for each agent. A recent method Conflict-Based Steiner Search (CBSS) is developed to solve MCPF to optimality, which, however, does not scale well when the number of agents or targets is large (e.g. 50 targets). While MAPF research has developed methods to plan bounded sub-optimality paths for many agents, it remains unknown how to find bounded sub-optimal solutions in the presence of many targets. This paper fills this gap by developing a method AK* for target sequencing (A for Approximation and K* for K-best), which leverages approximation algorithms for traveling salesman problems. AK* is motivated by MCPF, but is a standalone method that can solve K-best routing problems in general. We prove that AK* has worst-case polynomial runtime complexity and finds bounded sub-optimal solutions. With AK*, we develop two CBSS variants that find bounded sub-optimal paths for MCPF. Our results verify the fast running speeds of our methods with up to 200 targets.**

***Note to Practitioners*—The motivation of this paper originates from the need to plan conflict-free paths for multiple mobile robots in cluttered environment in warehouse logistics, manufacturing and inspection. While existing methods for multi-agent planning typically consider finding paths from starts to goals, this paper investigates the case, where agents must collectively visit a set of intermediate target locations before reaching their goals, for the purpose of inspection, picking or placing parts, etc. To solve the problem, this paper first develops an algorithm to find K-best solutions for traveling salesman problems with bounded sub-optimality, which then leads to two multi-agent planners that can handle hundreds of targets and tens of agents. We provide a Gazebo simulation to showcase the usage of the planner in a warehouse like environment.**

I. Introduction

Multi-Agent Path Finding (MAPF) computes an ensemble of collision-free paths for multiple agents from their respective start locations to their respective goal locations while minimizing a cost function defined over the paths. This paper considers a generalization of MAPF where agents must also visit a given set of intermediate target locations before reaching their goals. This generalization called the Multi-Agent Combinatorial Path Finding (MCPF) problem aims to find "start-target-goal" paths for the agents rather than the "start-goal" paths present in MAPF (Fig. 1(a)). MAPF and MCPF arise in applications such as logistics [1] and surveillance [2]. Specifically, factory environments [3], [4] often use multiple mobile robots to visit target locations under a centralized management system [5] for inspection or manufacturing. Consider mobile robots that need to visit a set of target locations to collect finished parts from machine centers to the storage. These robots may have heterogeneous capabilities due to their different grippers or payloads, and are thus subject to agent-target assignment constraints. These robots share a cluttered environment and need to find collision-free paths. In such settings, MCPF naturally arises to optimize operations.

MCPF is challenging as it requires both collision avoidance between agents (as present in MAPF), and target sequencing, i.e., solving Traveling Salesman Problems (TSPs) to specify the allocation and visiting orders of targets for all agents. Both the TSP [6] and the MAPF [7] are NP-hard to solve to optimality, and so is MCPF. To handle the challenges in MCPF, our prior work [8] developed a method called Conflict-Based Steiner Search (CBSS), which finds an optimal solution to MCPF by leveraging the recent advances in the research of both MAPF [9]–[11] and TSPs [12]–[15]. CBSS is currently the only approach that can find an optimal or bounded sub-optimal solution for MCPF. It works as follows: First, it ignores the potential collision between the agents by solving a multi-agent TSP to find a *target sequence* that specifies the allocation and visiting order of the targets for each agent. Next, CBSS fixes the target sequence for each agent and plans the corresponding collision-free paths by using Conflict-Based Search (CBS) [11] from the MAPF literature. Finally, CBSS alternates between target sequencing and path planning by generating new target sequences when needed in order to ensure solution optimality bounds. To generate new target sequences, CBSS solves a K-best TSP [15], [16], which requires finding a set of K cheapest target sequences, as opposed to an optimal target sequence, with K being a variable that is determined during the search process (Fig. 1).

While CBSS finds an optimal solution to the MCPF, it does not scale well as the number of agents or targets increase. The main reason for the bottleneck is that as the number of agents or targets in the problem increases, solving the K-best TSP to optimality in the CBSS becomes computationally prohibitive. While the number of potential collisions between the agents increases as the number of agents increases, this issue can be addressed through the latest advances in MAPF [17]–[19]. On the other hand, as the number of targets increases, finding

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). Sivakumar Rathinam is with Texas A&M University in USA (email: srathinam@tamu.edu). Howie Choset is with Carnegie Mellon University in USA. (email: choset@andrew.cmu.edu).

Digital Object Identifier (DOI):

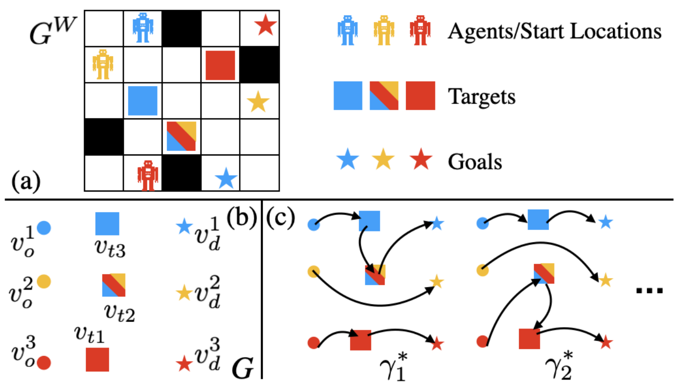

Fig. 1. (a) An example of MCPF. The color of a target indicates the assignment constraints, i.e., the subset of agents that are eligible to visit each target. (b) The corresponding target sequencing problem in a complete graph, where v_o^i, v_d^i are the start and goal vertices of agents $i = 1, 2, 3$ and v_{t1}, v_{t2}, v_{t3} are the target vertices that need to be visited. (c) A set of possible solutions to the target sequencing problem.

bounded sub-optimal solutions for the K-best TSP is challenging. This article focuses on developing new approximate methods for the K-best TSP, and as a result, new approximate versions of CBSS that can handle MCPF with a large number of targets and agents.

Conventionally, K-best TSPs are solved by a partition method [15], [16], which iteratively creates new TSPs in a systematic way and solves each of them to optimality by invoking an exact TSP algorithm. However, since each generated TSP is computationally heavy to solve to optimality, this partition method is computationally burdensome especially for a large graph or a large K. To address this challenge, this paper seeks to solve K-best TSPs by leveraging approximation algorithms for TSPs [20]–[24] due to their fast running speeds and solution quality guarantees. Specifically, approximation algorithms often address TSPs by first solving a corresponding simpler problem, such as finding a minimum spanning tree, and then converting the result to a solution tour or path for TSPs. Approximation algorithms enjoy worst-case polynomial runtime complexities and are able to return bounded sub-optimal solutions.

Although approximation algorithms for TSPs have been studied for decades [20], [24], combining them with the partition method for K-best TSPs is under-explored and challenging for the following reason. Most approximation algorithms for TSPs require that the graph is metric, so that the edge costs satisfy the triangle inequality. However, the partition method [15], [16] needs to impose special edge constraints on the graph, which breaks the triangle inequality, and makes these approximation algorithms no more applicable. To handle this difficulty, we develop a new partition method called AK*, where "A" stands for Approximation and "K*" stands for K-best. AK* can properly handle these edges constraints and can thus fuse a variety of approximation algorithms with the partition method to solve a family of K-best TSPs.

We prove that AK* can convert an approximation algorithm for TSPs into its K-best version, while preserving the sub-optimality bound and polynomial runtime guarantees. We compare our AK* against an exact method for TSPs for both single and multiple agents. Our results show that, AK* requires down to 1% of the runtime of the corresponding exact method in practice, while finding solutions that are about 50% more expensive than the solutions found by the exact method. Furthermore, AK* can quickly solve K-best TSPs with up to 20 agents and 200 targets, which are not solvable within a minute using the existing exact method for K-best TSPs [8].

We then combine AK* with CBSS and develop two variants of CBSS. The first variant is called CBSS-A ("A" stands for Approximation), which is able to bypass the challenge in target sequencing in the presence of many targets, by replacing the exact method for K-best TSPs with our AK*. The second variant is called CBSS-AF ("F" stands for focal search), which combines CBSS-A with the technique of focal search [25] in a similar way as in Enhanced Conflict-Based Search (ECBS) [17] to improve the scalability with respect to the number of agents by intelligently avoiding collision between agents. We show that both CBSS-A and CBSS-AF are guaranteed to find bounded sub-optimal collision-free paths for the agents. We test CBSS-A and CBSS-AF on instances with up to 30 agents and 80 targets with one minute runtime limit for each instance, and both algorithms often double or triple the success rates of the existing CBSS. Finally, we simulate the planned path with mobile robots in a warehouse in Gazebo/ROS and outline possible future work.

The main contribution of this paper is AK* that is able to convert an approximation algorithms for TSPs to its K-best counterpart, while preserving the solution sub-optimality bounds and the polynomial runtime complexity. While our main motivation is to improve CBSS to solve MCPF, AK* is itself a standalone method that is independent from CBSS. AK* has the potential to be used to solve other K-best sequencing and routing problems that arise in robotics, such as UAV routing [26], surveillance [27], and multi-agent robotic assembly planning [4]. Besides AK*, additional contributions are CBSS-A and CBSS-AF, two multi-agent path planners that rely on AK*, to find bounded sub-optimal collision-free paths for MCPF problems, while bypassing the challenge in the presence of many targets or agents.

The rest of this paper reviews the related work in Sec. II and describes the problem in Sec. III. We present AK* in Sec. IV and prove its properties in Sec. V. We then present CBSS-A and CBSS-AF in Sec. VI. Finally, we discuss the test result in VII and conclude in VIII. We summarize the frequently used notations and abbreviations in Table I.

II. Related Work

A. Traveling Salesman Problems

The Traveling Salesman Problem (TSP), which seeks to find a shortest tour for a single agent to visit each vertex in a graph, is one of the most well-known NP-hard problems [6]. Closely related to TSP, the Hamiltonian Path Problem (HPP) requires finding a shortest path that visits each vertex in the graph from a fixed start vertex to either a fixed or a unfixed goal vertex. The multi-agent version of the TSP and HPP (denoted as mTSP and mHPP, respectively) are harder to solve compared to the (single-agent) TSP and HPP, since the vertices in the graph must be allocated to each agent in addition to computing the optimal visiting order of vertices for each agent. Without causing confusion, we refer to all of TSP, HPP, mTSP and mHPP simply as TSPs, and we distinguish between them when needed during the presentation.

To solve TSPs, a variety of algorithms have been developed ranging from exact methods (branch and bound, branch and price) [6] to heuristics [14] and approximation algorithms [20], trading off solution optimality for runtime efficiency. This paper is particularly interested in approximation algorithms for TSPs [20]–[24] due to both their worst-case polynomial runtime complexity and sub-optimality bounds on solutions. Approximation algorithms typically solve TSPs by first solving a simpler problem and then convert the result to a desired solution path or tour [20], [22]. In practice, the approximation algorithms can be used to either directly solve the TSPs, or to provide an initial guess to start a heuristic algorithm for TSPs.

Most approximation algorithms for TSPs require the graph to be metric or allow a vertex to be visited at least once (as opposed to exactly once), since otherwise, there is no polynomial-time constant-factor (i.e., constant sub-optimality bound) approximation algorithm unless P=NP [28], [29]. Therefore, this paper considers TSPs on metric graphs and allows each vertex to be visited at least once. As we will see, in a MCPF problem and its corresponding target sequencing problem, these assumptions can be readily satisfied.

B. Multi-Agent Path Finding and Target Sequencing

While being able to find a sequence of targets to visit for the agents, algorithms for TSPs [12], [13], [30]–[32] typically do not consider the collision avoidance constraints between the agents, which are important for robotics especially when the workspace is cluttered and crowded. To find collision-free paths for multiple agents, a family MAPF algorithms have been developed, which fall on a spectrum from coupled [33] to decoupled [34], trading off completeness and optimality for scalability. In the middle of this spectrum lies the popular dynamically-coupled methods such as subdimensional expansion [10] and CBS [11], which begin by planning for each agent a shortest path from the start to the goal ignoring any potential collision and then couple agents for planning only when necessary to resolve agent-agent collision. These dynamically-coupled methods have been improved and extended in many ways [35]–[38].

Some recent research combines MAPF with target assignment and sequencing [8], [39]–[51]. Most of them either consider target assignment only (without the need for computing visiting orders of targets) [39]–[41], or compute the visiting order given that each agent is pre-allocated a set of targets [42]–[44]. Additionally, the multi-agent pick-up and delivery problems [46]–[48], which finds collision-free paths to fulfill a set of pick-up and delivery tasks, also require assigning a sequence of tasks to each agent. For MCPF, on the one hand, one can enumerate all possible target sequences in a brute-force manner during the path planning in order to ensure solution optimality [49], which is computational burdensome. On the other hand, one can leverage heuristics, such as a sequential method that first finds a target sequence and then plans paths with the sequence fixed [47], [52], which often enjoy good scalability but fail to provide solution quality guarantees. The recent approach CBSS [8] lies in their middle to provide solution optimality guarantees while avoiding the expensive enumeration of target sequences, by generating new target sequences only when necessary during path planning.

III. Problem Definitions

A. Multi-Agent Combinatorial Path Finding Problem

Let 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.

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 $v^i_o, v^i_d \in V^W$ denote the *initial/original* vertex and the *goal/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 $V_t \subset V^W$ 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*.

Let $\pi^i(v^i_1, v^i_\ell)$ denote a path for agent i between vertices v^i_1 and v^i_ℓ, which is a list of adjacent vertices $(v^i_1, v^i_2, \dots, v^i_\ell)$ in G^W. Let $g(\pi^i(v^i_1, v^i_\ell))$ denote the cost of the path, which is the sum of the costs of all edges present in the path: $g(\pi^i(v^i_1, v^i_\ell)) = \Sigma_{j=1,2,\dots,\ell-1} c^W(v^i_j, v^i_{j+1})$.

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 $t + 1$.[1]

Problem 1 (MCPF Problem [8])**.** *The MCPF problem seeks to find a set of conflict-free paths for the agents such that (1) each target $v \in V_t$ is visited[2] at 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 $u \in V_d$ such that $i \in f_A(u)$, and (3) the sum of the cost of all agents' paths reaches the minimum.*

[1]Here, the edge cost can be an arbitrary positive real number while the traversal time of each edge must be one. It is common in MAPF and its variants [9] to assume that the cost of an edge is always equal to its traversal time and is one for each edge.

[2]In MCPF, the notion that 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. In other words, if a target v is in the middle of the path of agent i and agent i does not claim v is visited, then v is not considered as visited. Additionally, a visited target v can appear in the path of another agent. For the rest of the paper, when we say an agent or a path "visits" a target, we always mean the agent "visits and claims" the target.

Notation	Section	Meaning
I, N	Sec. III	The index set $I = \{1, 2, \cdots, N\}$ that represents a set of N agents.
$G^W = (V^W, E^W, c^W)$	Sec. III	A (workspace) graph with vertex set V^W and edge set E^W. Each edge $(u, v) \in E^W$ has a cost $c^W(u, v)$.
V_o, V_t, V_d	Sec. III	A set of initial vertices, a set of target vertices and a set of destination vertices, respectively.
π^i,$\pi^{i'}$	Sec. III	A path of agent i in G^W.
π,π'	Sec. III	A joint path of all agents in G^W.
g,g'	Sec. III	The cost values.
$G = (V, E, c)$	Sec. III	A (target) graph with vertex set V and edge set E. Each edge $(u, v) \in E$ has a cost $c(u, v)$.
γ^i, γ	Sec. III	A target sequence of agent i, a joint (target) sequence of all agents, respectively.
$\{\gamma^*_k, k = 1, 2, \cdots, K\}$	Sec. III	A set of minimum cost joint sequences.
$\{\gamma_k, k = 1, 2, \cdots, K\}$	Sec. III	A set of joint sequences.
f_A	Sec. III	Assignment constraints.
mHPP, HPP	Sec. III	Multi-Depot Multi-Terminal Hamiltonian Path Problem, (Single-Agent) Hamiltonian Path Problem.
SPPFP	Sec. IV	Shortest Path Problem with Forbidden Paths.
AK*	Sec. IV	An Approximation Approach for K-Best TSPs.
I_e,I'_e	Sec. IV	A set of edges that must be included in the solution (joint sequence).
O_e,$\{o'_e\}$	Sec. IV	A set of forbidden edges, i.e., edges that must be excluded from the solution (joint sequence).
π_{fp},Π_{fp}	Sec. IV	A forbidden path, a set of forbidden paths.
R_1,R_k,R,R'	Sec. IV	A node for AK* search, which represents a restricted mHPP instance.
$D(R_k)$	Sec. V	The domain of R_k, i.e., the set of all solutions (joint sequences) to the restricted mHPP represented by R_k.
CBSS	Sec. VI	Conflict-Based Steiner Search.
P,P'	Sec. VI	A C-node for CBSS search.

TABLE I
FREQUENTLY USED NOTATIONS AND ABBREVIATIONS.

B. Target Sequencing Problems

Let $G = (V, E, c)$ denote a complete undirected metric (target) graph, where V is the set of all vertices, E is the set of edges connecting any pair of vertices in V, and $c : E \to (0, \infty)$ is a cost function that maps each edge $e = (u, v) \in E$ to a positive cost value $c(u, v)$ that represents the traversal cost from u to v of any agent. G is a metric graph and the edges satisfy the triangle inequality: $c(u, v) + c(v, w) \geq c(u, w)$ for all $u, v, w \in V$. Here, G is a target graph and is different from the aforementioned workspace graph G^W. The vertex set V of G is always the same as $V_o \cup V_t \cup V_d$, and there can be vertices in G^W that do not belong to any one of V_o, V_t, V_d and thus do not belong to G. The edge costs in the G^W and G are related as follows. The cost $c(u, v)$ of an edge (u, v) in the target graph G is equal to the cost of a minimum cost path in G^W between u and v ignoring any possible collision.

Let $\gamma^i(v^i_1, v^i_\ell)$ denote a *target sequence* for agent i that connects vertices v^i_1 and v^i_ℓ via a sequence of vertices $(v^i_1, v^i_2, \ldots, v^i_\ell)$ in G. Let $cost(\gamma^i(v^i_1, v^i_\ell))$ denote the cost of $\gamma^i(v^i_1, v^i_\ell)$, which is the sum of costs of all edges in the target sequence, i.e., $cost(\gamma^i(v^i_1, v^i_\ell)) = \Sigma_{j=1,2,\ldots,\ell-1} c(v^i_j, v^i_{j+1})$. Similarly, let $\gamma = \gamma(V_o, V_d) = \{\gamma^i(v^i_o, v^i_d), \forall i \in I\}$ denote a *joint (target) sequence* that consists of N target sequences, one target sequence for each agent $i \in I$. We denote $\gamma(V_o, V_d)$ and $\gamma^i(V_o, V_d)$ simply as γ and γ^i when there is no ambiguity. In this paper, when referring to a graph G, unless specified otherwise, we mean a target graph with vertex set $V_o \cup V_t \cup V_d$, and "a joint sequence γ in G" means a joint sequence that starts from V_o, visits V_t, and ends at V_d while satisfying the assignment constraint f_A.

Problem 2 (Multi-Depot Multi-Terminal Weighted Hamiltonian Path Problem (mHPP))**.** *Given a complete undirected metric graph $G = (V, E, c)$ and f_A, the mHPP seeks to find a joint sequence γ in G such that $\sum_{i \in I} cost(\gamma^i)$ reaches the minimum.*

Remark 1. *As a special case of mHPP, if $v^i_d = v^i_o, \forall i \in I$ (i.e., each agent has a depot vertex at which its target sequence starts and ends), this special case of mHPP is called a Multi-Depot Multiple Traveling Salesman Problem (mTSP) [12], [13]. If all agents share the same depot (i.e., $v^i_d = v^i_o = v \in V$), the variant is called (Single-Depot) Multiple Traveling Salesman Problem [12], [13]. Finally, if there is only one agent (i.e., $N = 1$), the corresponding mHPP and mTSP becomes the Traveling Salesman Problem (TSP) and Hamiltonian Path Problem (HPP), respectively.*

Problem 3 (K-Best mHPP)**.** *Given a complete undirected metric graph $G = (V, E, c)$ and the assignment constraints f_A, the K-Best mHPP seeks to find a set of K minimum cost joint sequences $\{\gamma^*_1, \gamma^*_2, \cdots, \gamma^*_K\}$ in G with $cost(\gamma^*_1) \leq cost(\gamma^*_2) \leq \cdots \leq cost(\gamma^*_K)$.*

Problem 4 (Approximated K-Best mHPP)**.** *Given a complete undirected metric graph $G = (V, E, c)$, $v^i_o, v^i_d \in G, \forall i \in I$ and f_A, the Approximated K-Best mHPP seeks to find a set of K joint sequences $\{\gamma_1, \gamma_2, \cdots, \gamma_K\}$ in G such that $cost(\gamma_k) \leq \alpha \cdot cost(\gamma^*_k)$ for $k = 1, 2 \cdots, K$, where γ^*_k is a k-th best solution as defined in Problem 3 and α is a positive constant real number. γ_k is called the approximated k-th best solution.*

In Problem 4, the joint sequences $\gamma_1, \gamma_2, \cdots, \gamma_K$ are not required to have monotonically non-decreasing costs. α is often called the approximation factor, which is the same as the sub-optimality bound of the solution returned. α is a number that may vary when different approximation algorithms are used to solve the problem.

IV. METHOD

This section presents AK* that solves Problem 4. We first introduce the notions of restricted problems in Sec. IV-A, which are required by AK*, before presenting AK*.

A. Restricted Problems

Problem 5 (Restricted mHPP)**.** *Given a complete undirected metric graph $G = (V, E, c)$, let $I_e, O_e \subseteq E$ denote two disjoint subsets of edges, the Restricted mHPP seeks to find a minimum cost joint sequence γ^* in G such that $I_e \subseteq \gamma^* \subseteq E \backslash O_e$.*

Here, "restricted" means there is a set of edges I_e that must be included into the solution and another set of edges O_e that must be excluded from the solution. For example, the leftmost plot in Fig. 2(c) seeks a solution with $I_e = \emptyset$ and $O_e = \{e_1\}$.

Problem 6 (Approximated Restricted mHPP)**.** *Given a complete undirected metric graph $G = (V, E, c)$, let $I_e, O_e \subseteq E$ denote two disjoint subsets of edges in G, the Approximated Restricted mHPP seeks to find a joint sequence γ in G such that (i) $I_e \subseteq \gamma \subseteq E \backslash O_e$, and (ii) $cost(\gamma) \leq \alpha \cdot cost(\gamma^*)$, where γ^* is a minimum cost solution to the Restricted mHPP defined in Problem 5.*

Finally, we introduce the Shortest Path Problem with Forbidden Paths (SPPFP) [53], [54], which will be used in AK*.

Problem 7 (Shortest Path Problem with Forbidden Paths (SPPFP))**.** *Given a undirected graph $G = (V, E, c)$ (which is not required to be complete or metric), a start vertex $v_{start} \in V$, a goal vertices $v_{goal} \in V$, and a set of forbidden paths Π_{fp}, where each path $\pi_{fp} \in \Pi_{fp}$ is a sequence of adjacent vertices in G connecting two vertices $u_1, u_2 \in V$, the SPPFP problem seeks to find a minimum cost path $\pi(v_{start}, v_{goal})$ in G such that (i) the cost of $\pi(v_{start}, v_{goal})$ is minimized and (ii) no forbidden path in Π_{fp} appears as a sub-path in $\pi(v_{start}, v_{goal})$.*

B. Two New Techniques in AK*

Our new partition method AK* (Alg. 1) runs a best-first search by iteratively (i) partitioning the set of possible solutions into subsets, (ii) finding an approximated solution in each partitioned subset, and (iii) picking the minimum cost one among these approximated solutions as the k-th solution, where k increases from 1 to K as the search proceeds. Fig. 2 shows the workflow of an iteration in AK*. AK* is based on the existing partition method [15] and has two new techniques that handles I_e and O_e respectively to use approximation algorithms for TSPs.

The first new technique (Sec. IV-D) is to use an Iterative Graph Completion (IGC) process to handle the edges in O_e that must be excluded from the solution. With IGC, the resulting graph G' is complete and metric and a constant-factor approximation algorithm can be applied on G'.

The second new technique (Sec. IV-E) is to identify a property (i.e., Lemma 1) of the edges I_e that must be included into the solution. Lemma 1 allows AK* to (i) create a sub-graph G'' out of G', (ii) invoke an approximation algorithm to find a joint sequence γ'' in G'', and (iii) combine γ'' with I_e to form a joint sequence γ' in G', and γ' is the desired solution that includes I_e.

C. Technical Overview of AK*

1) Node: Let $R = (G, I_e, o_e, \pi_c, \gamma)$ denote a *node*, which corresponds to a Restricted mHPP instance and consists of

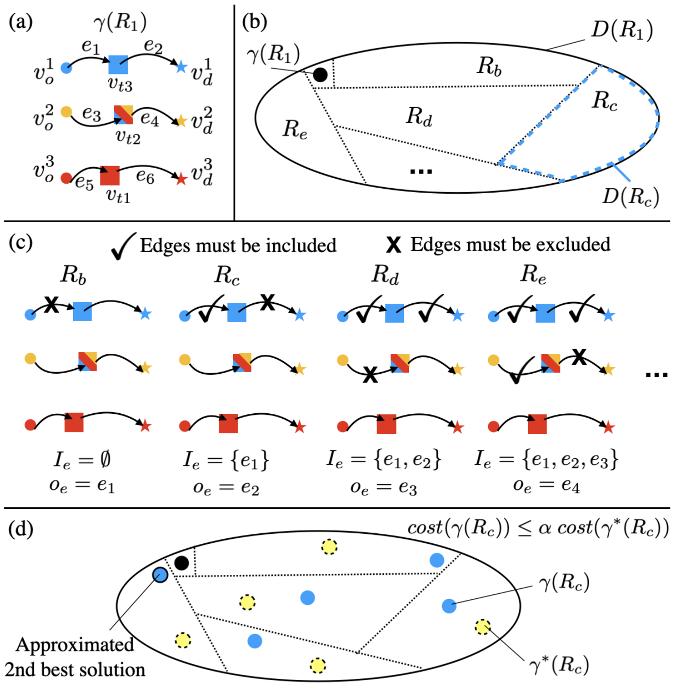

Fig. 2. An illustration of the partition method. In (a), the first best joint sequence is obtained. (b) uses the oval to denote the domain $D(R_1)$ of the mHPP (i.e., the set of all possible joint sequences that solve the mHPP), and shows the partition, where $D(R_1)$ is divided into multiple sets and each set is the domain of a new restricted mHPP represented by $R_b, R_c, \cdots$. (c) shows the restricted mHPP corresponding to $R_b, R_c, \cdots$, which requires finding a joint sequence that includes the edges in I_e and excludes the edge o_e. In (d), a bounded sub-optimal joint sequence γ (blue dot) is computed for each restricted mHPP, and each yellow dot represents a minimum cost joint sequence for that restricted mHPP. Among all blue dots, the minimum cost one (with black solid circle) is the approximated 2nd best solution.

five elements, where G is a graph, I_e, o_e, π_c and γ are a set of edges, an edge, a path and a joint sequence in G respectively. The graphs in different nodes can be different as we will see later. Let $G(R), I_e(R), o_e(R), \pi_c(R), \gamma(R)$ denote the respective elements in node R, and let $\gamma^i(R)$ denote the target sequence of agent i contained in $\gamma(R)$. We use an edge $o_e(R)$ instead of a set of edge $O_e(R)$ in a node because AK* only needs to exclude one edge from the solution at a time. Let $cost(R)$ denote the cost of node R, which is the same as the cost of the joint sequence $cost(\gamma(R))$. Let *mHPP-Solve*(G, I_e, O_e) denote the procedure that uses an approximation algorithm to solve a Restricted mHPP (Problem 5) in graph G with two sets of edges I_e and O_e that must be included into and excluded from the solution respectively. Here, *mHPP-Solve* is *transparent* to AK* in a sense that different algorithms for mHPP can be used to implement *mHPP-Solve*. AK* can also be used to solve problems different from mHPP, such as mTSP, and TSP by implementing *mHPP-Solve* with a different approximation algorithm.

2) Initialization: AK* takes as input a complete undirected graph, which is denoted as G_1 for clarity. The goal of AK* is to solve the Approximated K-best mHPP on G_1. AK* begins by creating an initial node R_1 (Lines 1-2) with $G(R_1) \leftarrow G_1$, $I_e(R_1) \leftarrow \emptyset$, $o_e(R_1) \leftarrow NULL$, $\pi_c \leftarrow \emptyset$ and $\gamma(R_1) \leftarrow$*mHPP-*

Algorithm 1 Pseudocode for AK*

```
    INPUT: $G_1 = (\{V_o, V_t, V_d\}, E_1, c_1)$, $f_A$, and $K$.
    OUTPUT: a set of approximated K-best joint sequences in $G_1$.
1:  $\gamma_1 \leftarrow$ mHPP-Solve$(G_1, \emptyset, \emptyset)$
2:  $R_1 \leftarrow (G_1, I_e = \emptyset, o_e = NULL, \pi_c = \emptyset, \gamma \leftarrow \gamma_1)$
3:  $parent(R_1) \leftarrow NULL$, $\mathcal{S} \leftarrow \emptyset$
4:  Add $R_1$ into OPEN$_R$
5:  while OPEN$_R \neq \emptyset$ do
6:      $R_k \leftarrow$ OPEN$_R$.pop()
7:      $\gamma_k \leftarrow$ReconstructSol$(R_k, \gamma(R_k))$, add $\gamma_k$ into $\mathcal{S}$
8:      if $k = K$ then
9:          return $\mathcal{S}$
10:     Index the edges in $\gamma(R_k)$ as $\{e_1, e_2, \dots, e_\ell\}$
11:     for all $p \in \{1, 2, \dots, \ell\}$ do
12:         $o'_e \leftarrow e_p$
13:         $I'_e \leftarrow I_e(R_k) \bigcup \{e_1, e_2, \dots, e_{p-1}\}$
14:         if $o'_e \in I'_e(R_k)$ then
15:             continue
16:         $\Pi_{fp} \leftarrow \emptyset$
17:         $\Pi_{fp} \leftarrow$GetForbidPaths$(R_k, o'_e, \Pi_{fp})$
18:         $(u, v) \leftarrow o'_e$
19:         $\pi'_c \leftarrow$ SPPFP-Solve$(G_1, \Pi_{fp}, u, v)$
20:         $G' \leftarrow$CompleteGraph$(G(R_k), o'_e, \pi'_c)$
21:         $\gamma' \leftarrow$ mHPP-Solve$(G', I'_e, \emptyset)$
22:         $R' \leftarrow (G', I'_e, o'_e, \pi'_c, \gamma')$
23:         $parent(R') \leftarrow R_k$
24:         Add $R'$ into OPEN$_R$
25: return failure
```

Solve$(G(R_1), \emptyset, \emptyset)$. Let $parent$ denote the parent pointer of a node, and $parent(R_1)$ is set to $NULL$ to indicate that R_1 has no parent. The solution set $\mathcal{S}$ stores the approximated K-best solutions found during the search. R_1 is added to OPEN$_R$, a priority queue of nodes, where nodes are prioritized based on their costs from the minimum to the maximum (Line 4).

3) Iteration: Lines 5-24 is called an *iteration* of AK*. In the k-th ($k = 1, 2, \cdots, K$) iteration, AK* first pops a node R_k from OPEN$_R$, and the joint sequence $\gamma(R_k)$ is the approximated k-th best solution. $\gamma(R_k)$ cannot be directly added to the solution set $\mathcal{S}$, because $\gamma(R_k)$ is a joint sequence in graph $G(R_k)$ and needs to be reconstructed in graph G_1 (see Sec. IV-F) before being added into $\mathcal{S}$ (Line 7). If $k = K$, then $\mathcal{S}$ contains the approximated K-best solutions and AK* terminates (Line 9). Otherwise, AK* indexes the edges in the joint sequence $\gamma(R_k)$ from 1 to ℓ, where ℓ is the total number of edges in $\gamma(R_k)$, i.e., the sum of number of edges in each target sequence $\gamma^i(R_k), i = 1, 2, \cdots, N$. Then, AK* iterates these edges to create new nodes as follows.

4) Edges Must Be Included: When iterating the edges $e_p, p = 1, 2, \cdots, \ell$ (Lines 11-24), for every e_p, the subset of edges $\{e_1, e_2, \dots, e_{p-1}\}$ is unioned with the set $I_e(R_k)$, to form I'_e, a new set of edges that must be included into the solution (Line 13). Here, (i) $\gamma(R_k)$ is a joint sequence that includes all edges in $I_e(R_k)$ and (ii) $\{e_1, e_2, \dots, e_{p-1}\}$ are the first $p-1$ edges in $\gamma(R_k)$ based on the index. Therefore, the new set I'_e is guaranteed to be a set of paths where each path is a complete or partial target sequence in $\gamma(R_k)$ (as opposed to trees, cycles, stars, etc.). Fig. 3(a) and 3(c) show an example, where the set $I'_e = (e_1, e_2, e_3)$ forms a set of two paths $\{(e_1, e_2), (e_3)\}$. This property of I'_e is summarized in the following lemma.

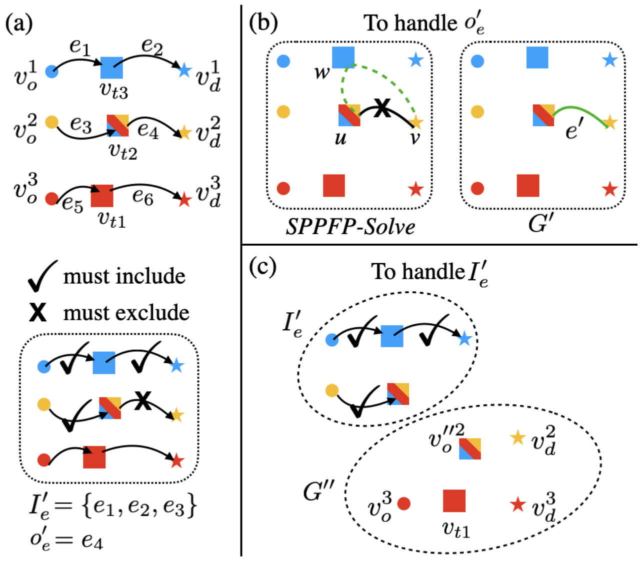

Fig. 3. An illustration of the two new techniques in AK*. (a) shows a restricted mHPP where edges in I'_e must be included into and the edge o'_e must be excluded from the solution to be computed. (b) shows that AK* removes o'_e and then completes the graph by adding a new edge e', which corresponds to a minimum cost path that does not contain o'_e. (c) shows that AK* creates a new complete directed metric graph G'' and finds a joint sequence γ'' in G'' by solving a new mHPP using an approximation algorithm. Then γ'' can always be combined with I'_e to form a joint sequence that is desired.

Lemma 1. *In the k-th iteration of Alg. 1, on Line 13, the set I'_e always forms a set of paths $\{\gamma^i_{I'_e}, i = 1, 2, \cdots, N\}$ in graph $G(R_k)$, where each path corresponds to a target sequence with the following features.*

- *If $\gamma^i_{I'_e}$ is not empty, then $\gamma^i_{I'_e}$ starts with v^i_o.*
- *If $\gamma^{i+1}_{I'_e}$ is not empty, then $\gamma^i_{I'_e}$ ends with v^i_d.*
- *If $\gamma^i_{I'_e}$ does not end with v^i_d, then every $\gamma^j_{I'_e}, j \in I, j \geq i$ is empty.*

5) Edges Must Be Excluded: The edge e_p must be excluded from the solution, which is re-denoted as o'_e on Line 12 and is called a *forbidden edge*. If o'_e is already contained in $I'_e(R_k)$ (Line 14), then the corresponding Restricted mHPP, which will be generated and solved on Line 21, is not solvable since the same edge o'_e cannot be both included into and excluded from a solution at the same time, and the current iteration ends (Line 15). Otherwise (i.e., $o'_e \notin I'_e(R_k)$), a Restricted mHPP is created and solved (Line 17-21). Here, to exclude o'_e from the solution, deleting o'_e from the graph $G(R_k)$ would lead to a graph that is not fully connected, and the triangle inequality does not hold any more.[3]

6) Basic Graph Completion: To handle this difficulty, one can complete the graph by finding a minimum cost path π' connecting both vertices of edge o'_e without using edge o'_e, and add a new edge e' to replace o'_e while setting the cost of

[3]In other words, deleting o'_e from the graph is equivalent to modifying the cost of the forbidden edge $o'_e = (u, v)$ to infinity from the computational perspective, and the triangle inequality $cost(u, w) + cost(w, v) \geq cost(u, v)$ does not hold for any $w \in V, w \neq u, w \neq v$, since $cost(u, w) + cost(w, v)$ is finite and $cost(u, v) = \infty$. Take Fig. 3(a) and Fig. 3(b) for example, if edge $e_4 = (v_{t2}, v_d^2)$ is modified to have an infinity cost, then $cost(v_{t2}, v_{t3}) + cost(v_{t3}, v_d^2) \ngeq cost(v_{t2}, v_d^2) = \infty$.

e' to be the same as $cost(\pi')$. By doing so, the resulting graph G' is complete, undirected and metric, and approximation algorithms can be applied to find a joint sequence γ' in G'. Later, if the computed γ' includes e', then the joint sequence γ in G_1 can be reconstructed by replacing e' with π'. For example, in Fig. 3(b), the black edge (u, v) is o'_e and must be excluded. The green dashed line $((u, w), (w, v))$ is π', which becomes e' the green solid line, and e' completes the graph after the removal of o'_e.

7) Iterative Graph Completion (IGC): AK* has to conduct the aforementioned graph completion process iteratively. We explain the intuition with the example in Fig. 4: First, in an iteration of Alg. 1, edge e_1 must be excluded and the graph is completed by finding path (e_2, e_3) as shown in Fig. 4(a). Then, in a future iteration of Alg. 1, edge e' in Fig. 4(b) must be excluded. At this moment, the forbidden edge e' corresponds to a path (e_2, e_3) in Fig. 4(a), and to complete the graph, a path connecting u, v without using both edge e_1 and path (e_2, e_3) is needed, as shown in Fig. 4(c). To find such a path π'', we need to solve a SPPFP as defined in Problem 7 with two forbidden paths $\Pi_{fp} = \{(e_1), (e_2, e_3)\}$, and Fig. 4(d) shows π'' in brown dotted lines. We now introduce the IGC (Line 17-20 in Alg. 1), which is detailed in Sec. IV-D. IGC begins by creating a set of forbidden paths in G_1 using *GetForbidPaths*, which iteratively traces the parent nodes to reconstruct all the forbidden paths in G_1. After obtaining the forbidden paths, AK* creates a SPPFP instance, which is solved in *SPPFP-Solve* (Line 19) to find a minimum cost path π'_c connecting both vertices u, v of edge o'_e. Then, in *CompleteGraph*, an edge e' corresponding to path π'_c is added to $G(R_k)$ to replace o'_e, and the resulting graph is denoted as G', which is an undirected complete metric graph.

8) New Node Generation: After IGC, the procedure *mHPP-Solve* is called to solve the restricted mHPP in G' by leveraging an approximation algorithm to find a joint sequence γ', while ensuring all edges in I_e are included in γ', which is elaborated in Sec. IV-E. Then, a corresponding new node R' (Line 22) is created and added to OPEN$_R$, and the parent of R' is pointed to R_k. AK* iterates until K joint sequences are found (Line 9).

D. Handle The Edge That Must Be Excluded

We revisit IGC (i.e., Lines 17-20 in Alg. 1) with more details and explain the pseudo-code of the key procedures.

1) Forbidden Paths Reconstruction: AK* invokes *GetForbidPaths* on Line 17 in Alg. 1 and the inputs are: a node R_k, a forbidden edge o_e, and a set of forbidden paths Π_{fp}. As shown in Alg. 2, *GetForbidPaths* is a recursive procedure that aims to eventually reconstruct the forbidden paths in G_1. The path set Π_{fp} contains the forbidden paths that are reconstructed during the recursive calls of Alg. 2.

GetForbidPaths first treats the edge o_e as a path that contains a single edge, and adds o_e to the path set Π_{fp} (Line 2). Then, if R_k is $NULL$ (Line 4), there is no further reconstruction and Alg. 2 terminates. At this moment, all forbidden paths in Π_{fp} have been reconstructed in graph G_1. Otherwise (i.e., $R_k \neq NULL$), *GetForbidPaths* iterates each path $\pi \in \Pi_{fp}$ (Line 5) and each edge $e_j \in \pi$ (Line 7). If $e_j = o_e(R_k)$, then e_j needs to be replaced with $\pi_c(R_k)$ in path π (Lines 9-11). After Line 11, *GetForbidPaths* ensures that any edge $o_e(R_k)$ that appears in any $\pi \in \Pi_{fp}$ is now replaced with $\pi_c(R_k)$. Finally, *GetForbidPaths* makes a recursive call on itself (Line 12) to further reconstruct the path in graph $G(parent(R_k))$.

Algorithm 2 Pseudocode for *GetForbidPaths*(R_k, o_e, Π_{fp})

```
INPUT: $R_k$ is a node in Alg. 1, $o_e$ is an edge, $\Pi_{fp}$ is a set of paths.
1:  if $o_e \neq NULL$ then
2:      add $o_e$ into $\Pi_{fp}$
3:  if $R_k = NULL$ then
4:      return $\Pi_{fp}$
5:  for all $\pi \in \Pi_{fp}$ do
6:      $(e_1, e_2, \cdots, e_m) \leftarrow \pi$
7:      for all $e_j \in \{e_1, e_2, \cdots, e_m\}$ do
8:          if $o_e(R_k) \neq NULL$ and $e_j = o_e(R_k)$ then
9:              $\pi' \leftarrow$ make a copy of $\pi$
10:             replace $e_j$ in $\pi'$ with $\pi_c(R_k)$
11:             replace $\pi$ in $\Pi_{fp}$ with $\pi'$
12: return GetForbidPaths($parent(R_k)$, $o_e(R_k)$, $\Pi_{fp}$)
```

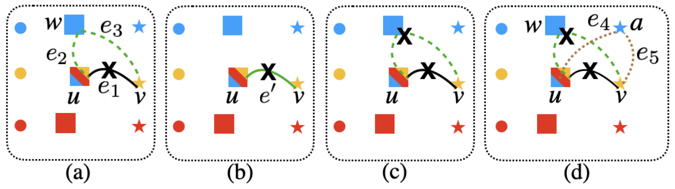

Fig. 4. An illustration of why AK* needs to solve SPPFP for iterative graph completion. In (a), edge e_1 is forbidden (i.e., must be excluded from the solution), and a minimum cost path (e_2, e_3) is found. In (b), a new edge e' corresponding to the path (e_2, e_3) in (a) is created. In a future iteration, e' is forbidden again. (c) shows that, at this moment, between vertices (u, v), both the edge e_1 and the path (e_2, e_3) are forbidden, and to complete the graph, AK* needs to solve a SPPFP to find a minimum cost path between (u, v) without using e_1 and (e_2, e_3). (d) shows a minimum cost solution to the SPPFP which is the path (e_4, e_5) in brown color.

2) Minimum Cost Path Subject to Forbidden Paths: After getting all the forbidden paths Π_{fp} (Line 17), a minimum cost path between (u, v) that does not use any path in Π_{fp} as a sub-path is then computed by solving a SPPFP as defined in Problem 7. This SPPFP is always considered in graph G_1 with Π_{fp} being the set of forbidden paths. To solve this SPPFP, this paper uses a dynamic programming algorithm [54].

E. Handle The Edges That Must Be Included

In Alg. 1, when AK* reaches Line 21, the forbidden edge o'_e has been handled by IGC and the obtained graph G' is a complete undirected metric graph. In other words, any joint sequence in G' do not use the forbidden edge o'_e. Then, on Line 21, AK* seeks to find an approximated joint sequence in G' that includes all edges in I'_e. AK* calls *mHPP-Solve*, which is described in Alg. 3 and consists of three steps.

1) New Graph Creation: *mHPP-Solve* first creates a new graph G'' by (i) removing the sub-graph induced by I'_e in G', i.e., removing the vertices from G' that appear in the edges in I'_e, and (ii) updating the initial/target/goal vertices in G''. An example can be found in Fig. 3(a) and Fig. 3(c), and we explain this updating step as follows.

Algorithm 3 Pseudocode for *mHPP-Solve*$(G', I'_e, O'_e = \emptyset)$

```
INPUT: $G'$ is a complete undirected metric graph, $I'_e$ is a set of edges that must be included into the solution, the third argument $O_e$ is always an empty set.
1: $V''_o, V''_t, V''_d, G'' \leftarrow$ RemoveEdges($G'$, $I'_e$)
2: $\gamma_{G''} \leftarrow$ApproximationAlgorithm($G''$)
3: return CombineJointTargetSequence($\gamma_{G''}$, $I'_e$)
```

Specifically, I'_e has features that were summarized in Lemma 1. Let $\{\gamma^i_{I'_e}, i = 1, 2, \cdots, N\}$ denote the list of paths formed by I'_e. For each $\gamma^i_{I'_e}, i = 1, 2, \cdots, N$, let $\gamma^i_{I'_e}(j)$ denote the j-th vertex in $\gamma^i_{I'_e}$, and let $\gamma^i_{I'_e}(last)$ denote the last vertex in $\gamma^i_{I'_e}$. For each $i = 1, 2, \cdots, N$, remove the vertices in $\gamma^i_{I'_e}(j), j = 1, 2, \cdots, |\gamma^i_{I'_e}(j)| - 1$ (i.e., from the first to the second last vertices), as well as the edges incident on these vertices, from G'. If the last vertex $\gamma^i_{I'_e}(last)$ is the same as v^i_d, then remove $\gamma^i_{I'_e}(last)$ from G' as well. In this new graph G'', we create a new set of initial and goal vertices for the agents. Let $A \subseteq I$ denote the set of agents whose corresponding v^i_d remains in G'. Intuitively, agents that are not in A are the ones whose target sequences are fully contained in I'_e and are thus fixed, since I'_e are edges that must be included in the solution. Let V''_o denote a set of new initial vertices of agents that is constructed as follows. For each $i \in A$, if $\gamma^i_{I'_e}$ is empty, then add v^i_o into V''_o. Otherwise (i.e., $\gamma^i_{I'_e}$ is not empty), add $\gamma^i_{I'_e}(last)$ into V''_o. Finally, we create a new set of goal vertices V''_d by copying v^i_d for each agent $i \in A$. Let $V''_t \subseteq V_t$ denote the subset of targets that remains in G''.

2) Approximation Algorithm: mHPP-Solve then invokes an existing approximation algorithm to find a joint target sequence in G''. Different approximation algorithms can be used in the procedure *ApproximationAlgorithm* on Line 2 in Alg. 3. We showcase two examples in Sec. IV-G.

3) Inclusion of Edges: Finally, *mHPP-Solve* combines the solution (denoted as $\gamma_{G''}$) computed by the approximation algorithm in graph G'' with the edges in I'_e. Note that $\gamma_{G''} = \{\gamma^i_{G''}, i \in A\}$, and each $\gamma^i_{G''}$ must start from the corresponding new initial vertex of agent i in V''_o, which is either $\gamma^i_{I'_e}(last)$ or v^i_o. Therefore, $\gamma^i_{G''}$ can be concatenated with $\gamma^i_{I'_e}$ so that a target sequence γ^i from v^i_o to v^i_d is obtained for each agent. For agents $i \in I, i \notin A$, we know $\gamma^i = \gamma^i_{I'_e}$, which is a target sequence from v^i_o to v^i_d. All these $\gamma^i, i \in I$ then form a joint sequence γ that is returned by *mHPP-Solve*.

F. Solution Reconstruction

When a node R_k is popped from OPEN in Alg. 1, the corresponding joint sequence $\gamma(R_k)$ is the k-th solution (Line 7) to be returned. Here, $\gamma(R_k)$ is a joint sequence in $G(R_k)$ and it needs to be reconstructed in G_1 before being added into the solution set $\mathcal{S}$. To reconstruct $\gamma(R_k)$ in G_1, as shown in Alg. 4, $\gamma(R_k)$ is recursively reconstructed. Specifically, if $\pi_c(R_k)$ of the input node R_k is $NULL$, then *ReconstructSol* reaches the base case where $G(R_k)$ is G_1 and the current joint sequence γ is returned. Otherwise (Line 3-9 in Alg. 4), every edge e_j in γ is checked if e_j is the same as $o_e(R_k)$ (Line 6). If yes, then e_j needs to be replaced with $\pi_c(R_k)$

Algorithm 4 Pseudocode for *ReconstructSol*(R_k, γ)

```
INPUT: $R_k$ is a node in Alg. 1
1: if $R_k = NULL$ then
2:     return $\gamma$
3: for all $\gamma^i \in \gamma$ do
4:     $(e_1, e_2, \cdots, e_m) \leftarrow \gamma^i$
5:     for all $e_j \in \{e_1, e_2, \cdots, e_m\}$ do
6:         if $e_j = o_e(R_k)$ then
7:             $\gamma^{i'} \leftarrow$ make a copy of $\gamma^i$
8:             replace $e_j$ in $\gamma^{i'}$ with $\pi_c(R_k)$
9:             replace $\gamma^i$ in $\gamma$ with $\gamma^{i'}$
10: return ReconstructSol($parent(R_k)$, $\gamma$)
```

and the joint sequence γ is updated correspondingly. After this replacement, a recursive call is made (Line 10) where the current γ is further reconstructed in $G(parent(R_k))$, i.e., the graph corresponding to the parent node of R_k.

G. Examples

In AK*, the *ApproximationAlgorithm* in Alg. 3 can be implemented by using different approximation algorithms. We discuss the following two examples.

The first example uses the algorithm in [22], which is guaranteed to provide an 11/3 approximation ratio, i.e., the returned solution γ has a cost $cost(\gamma) \leq \frac{11}{3}\, cost(\gamma^*)$ where γ^* is a minimum cost joint sequence. This 11/3-algorithm can handle a special class assignment constraints: (i) Each destination $v \in V_d$ can only be visited by a unique agent $i \in I$, i.e., $f_A(v) = \{i\}, i \in I$). (ii) Each target $v \in V_t$ can either be visited by all agents (i.e., $f_A(v) = I$), or can only be visited by a specific agent (i.e., $f_A(v) = \{i\}, i \in I$). This implementation of AK* is referred to 11/3-AK* and will later be used in Sec. VI-A.

The second example considers a (single-agent) TSP problem: where $N = 1$, $V_o = \{v_o\}$, $V_d = \{v_d\}$ and $v_o = v_d$. There are many approximation algorithms for the TSP. A popular one among them is Christofides algorithm [20], which has an approximation ratio of 3/2, and can be used to implement *ApproximationAlgorithm*. This implementation of AK* is referred to as Christofides-AK*.

V. Analysis

AK* (i.e., Alg. 1) has worst-case polynomial runtime complexity since (i) all procedures used in every iteration has a polynomial runtime compleixty, (ii) AK* makes a finite number of calls of these procedures in each iteration, and (iii) AK* terminates within a finite number of iterations. Let $C_{approxi}$ denote the worst-case runtime complexity of the *ApproximationAlgorithm*, which depends on the specific approximation algorithm that is used.

Theorem 1. *If K is finite and $C_{approxi}$ is a polynomial compleixty, then the worst-case runtime complexity of AK* is polynomial.*

For the rest of the analysis, we consider that K is given as part of the problem input. In AK*, each node R_k corresponds to a Restricted mHPP in graph $G(R_k)$. Let $D(R_k)$ denote the *domain* of R_k with respect to the graph $G(R_k)$, i.e., the set of all possible joint sequences in $G(R_k)$ that solve the Restricted mHPP corresponding to R_k. In other words, every $\gamma \in D(R_k)$ is a joint sequence in $G(R_k)$ that satisfies $I_e(R_k) \subseteq \gamma \subseteq E \backslash \{o_e(R_k)\}$ where E is the edge set in $G(R_k)$.

In every iteration of AK*, after a node R_k is popped from OPEN$_R$, its domain $D(R_k)$ is divided into multiple subsets on Lines 10-13 in Alg. 1, where each subset is the domain of a new Restricted mHPP that corresponds to a node R' with $I_e(R') = I'_e$ and $o_e(R') = o'_e$ (Line 22). Let $L = \{R'_1, R'_2, \cdots, R'_\ell\}$ denote the list of nodes that are generated on Lines 11-24 in Alg. 1. If R'_p for some $p \in \{1, 2, \cdots, \ell\}$ is not generated due to Line 14-15, let $R'_p = NULL$ and $D(R'_p) = \emptyset$. We show that this division of $D(R_k)$ is a partition of $D(R_k) \backslash \{\gamma(R_k)\}$, i.e., $\bigcup_{p=1,2,\cdots,\ell} D(R'_p) = D(R_k) \backslash \{\gamma(R_k)\}$ and $p, j = 1, 2, \cdots, \ell, p \neq j, D(R'_p) \cap D(R'_j) = \emptyset$, which is equivalent to the following lemma.

Lemma 2. *For each $\gamma \in D(R_k) \backslash \{\gamma(R_k)\}$, there exists a unique $D(R'_j), j = 1, 2, \cdots, \ell$ such that $\gamma \in D(R'_j)$.*

Proof. For an arbitrary joint sequence γ in $G(R_k)$ that is different from $\gamma(R_k)$, we can introduce a binary vector $b \in \mathbb{B}^\ell \backslash \mathbf{1}$ of length ℓ to indicate if each edge in $\gamma(R_k)$ is included in γ, where $\mathbb{B}$=$\{0, 1\}$ and $\mathbf{1}$ is a vector with all component being one. Specifically, the j-th component $b(j)$ is one (or zero) if the j-th edge in $\gamma(R_k)$ is included in (or excluded from) γ. Obviously, b cannot be equal to $\mathbf{1}$, since γ must be different from $\gamma(R_k)$. Therefore, for each $\gamma \in D(R_k) \backslash \{\gamma(R_k)\}$, there exists a corresponding vector $b \in \mathbb{B}^\ell \backslash \mathbf{1}$ (Claim 1). Lines 10-13 in Alg. 1 divides $\mathbb{B}^\ell \backslash \mathbf{1}$ into the following list of sets: $L_B = \{(0, \mathbb{B}, \cdots, \mathbb{B}), (1, 0, \mathbb{B}, \cdots, \mathbb{B}), (1, 1, 0, \mathbb{B}, \cdots, \mathbb{B}), \cdots, (1, 1, \cdots, 1, 0)\}$, where $\mathbb{B}$ means the corresponding component can be either zero or one. These sets are mutually disjoint to each other (Claim 2). The j-th set in L_B corresponds to $D(R'_j)$ in L. To show that this division is a partition, we need to further show that every $b \in \mathbb{B}^\ell \backslash \mathbf{1}$ belongs to one of the sets (Claim 3), which is shown as follows. For each $b \in \mathbb{B}^\ell \backslash \mathbf{1}$, we can find the first component in b that is zero. Without losing generality, let $b(j) = 0$ with j being a specific number ranging from 1 to ℓ. Then the j-th set in L_B contains b. Putting these claims together, by Claim 1, for each $\gamma \in D(R_k) \backslash \{\gamma(R_k)\}$, there is a corresponding $b \in \mathbb{B}^\ell \backslash \mathbf{1}$. By Claim 3, this b belongs to one of the set in L_B, let this set be the j-th set, which corresponds to $D(R'_j)$. Therefore, $\gamma \in D(R'_j)$. By Claim 2, the sets in L are mutually disjoint to each other. The lemma is thus proved. $\square$

Following the previous notation that $L = \{R'_1, R'_2, \cdots, R'_\ell\}$ is the list of nodes that are generated on Lines 11-24 in Alg. 1. For an arbitrary $p = 1, 2, \cdots, \ell$, let γ^* denote the minimum cost joint sequence that solves the Restricted mHPP represented by R'_p, and let γ' denote the joint sequence computed by Alg. 3. Same as the notation in Alg. 3, let G', I'_e denote the input to *mHPP-Solve* (i.e., Alg. 3). Additionally, *ApproximationAlgorithm* is invoked on G'', a graph that is constructed on Line 1 in Alg. 3. Let $\gamma^*_{G''}$ denote a minimum cost joint sequence in G'', and let $\gamma_{G''}$ denote the joint sequence computed by *ApproximationAlgorithm* in G''. Let $\alpha \geq 1$ denote the approximation ratio of the approximation algorithm that implements *ApproximationAlgorithm*. As a result, $cost(\gamma_{G''}) \leq \alpha\, cost(\gamma^*_{G''})$.

Lemma 3. *The joint sequence γ' returned by mHPP-Solve satisfies $cost(\gamma') \leq \alpha\, cost(\gamma^*)$.*

Proof. Edges in I'_e must be included into the computed joint sequence. γ^* must be the concatenation of I'_e and $\gamma^*_{G''}$, and $cost(\gamma^*) = cost(I'_e) + cost(\gamma^*_{G''})$. This holds because if there exists another γ^{**} that is cheaper than γ^*, then by removing I'_e from γ^{**}, we get a new joint sequence $\gamma^{**}_{G''}$ in graph G'' whose cost must be cheaper than $\gamma^*_{G''}$, which contradicts that $\gamma^*_{G''}$ is a minimum cost joint sequence in G''.

Let $cost(I'_e)$ denote the sum of costs of edge in I'_e. Then, $cost(\gamma) = cost(I'_e) + cost(\gamma_{G''}) \leq cost(I'_e) + \alpha\, cost(\gamma^*_{G''}) \leq \alpha(cost(I'_e) + cost(\gamma^*_{G''})) = \alpha\, cost(\gamma^*)$. The first and the second inequalities hold because $\alpha \geq 1$. $\square$

Lemma 4. *Let $\{\gamma_1, \gamma_2, \cdots, \gamma_k\}$ denote the set of joint sequences stored in $\mathcal{S}$ in AK*. Then, at the beginning of each iteration of AK* (i.e., before Line 6), the following equation holds: $(\bigcup_{R_j \in OPEN} D(R_j)) \cup \mathcal{S} = D(R_1)$.*

Proof. We prove this by induction. Base case: at the beginning of the first iteration, $\mathcal{S} = \emptyset$ and the only node in OPEN is R_1, so the lemma holds.

Assumption: at the beginning of the k-th ($k > 1$) iteration, the lemma holds.

Induction: during the k-th iteration, between Lines 5-24, a joint sequence $\gamma(R_k)$ is added to $\mathcal{S}$. By Lemma 2, a partition of $D(R_k) \backslash \{\gamma(R_k)\}$ is created and all corresponding nodes $R'_j, j = 1, 2, \cdots, \ell$ are added to OPEN. This lemma therefore holds at the beginning of the $(k+1)$-th iteration. $\square$

Theorem 2. *Let $\{\gamma^*_1, \gamma^*_2, \cdots, \gamma^*_K\}$ denote a set of K-best joint sequences in G_1, let $\mathcal{S} = \{\gamma_1, \gamma_2, \cdots, \gamma_K\}$ denote the set of joint sequences returned by AK*, then $cost(\gamma_k) \leq \alpha\, cost(\gamma^*_k), k = 1, 2, \cdots, K$ (i.e., AK* solves the Approximated mHPP defined in Problem 4).*

Proof. We prove by induction. Base case: during the initialization, AK* computes γ_1 based on G_1 without any edge constraints (i.e., $I_e = \emptyset$, $o_e = NULL$). γ_1 is added to the solution set $\mathcal{S}$ on Line 7 in Alg. 1 and $cost(\gamma_1) \leq \alpha\, cost(\gamma^*_1)$.

Assumption: in the k-th iteration, after AK* finishes Line 7 in Alg. 1, this theorem holds, i.e., $\mathcal{S}$ contains $\{\gamma_1, \gamma_2, \cdots, \gamma_k\}$ and $cost(\gamma_j) \leq \alpha\, cost(\gamma^*_j), j = 1, 2, \cdots, k$.

Induction: at the beginning of the $(k+1)$-th iteration (before Line 6), by Lemma 4, we know that $(\bigcup_{R_j \in \text{OPEN}} D(R_j)) \cup \{\gamma_1, \gamma_2, \cdots, \gamma_k\} = D(R_1)$, which means every possible joint sequence in $D(R_1)$ must be either in $(\bigcup_{R_j \in \text{OPEN}} D(R_j))$ or in $\mathcal{S}$. Since the size of set $\{\gamma^*_1, \gamma^*_2, \cdots, \gamma^*_{k+1}\}$ is larger than the size of $\mathcal{S} = \{\gamma_1, \gamma_2, \cdots, \gamma_k\}$ by one, there must be at least one joint sequence $\gamma^*_j, j \leq k+1$ in $\{\gamma^*_1, \gamma^*_2, \cdots, \gamma^*_{k+1}\}$ that is not contained in $\mathcal{S}$ but is contained in $(\bigcup_{R_j \in OPEN} D(R_j))$. Without losing generality, let $D(R_j), R_j \in OPEN$ denote the set that contains γ^*_j. Then, by Lemma 3, $cost(\gamma(R_j)) \leq \alpha\, cost(\gamma^*_j)$. Since the popped node R_{k+1} from OPEN on Line 6 is the minimum cost node in OPEN, $cost(R_{k+1}) = cost(\gamma(R_{k+1})) \leq cost(\gamma(R_j)) \leq \alpha\, cost(\gamma^*_j) \leq \alpha\, cost(\gamma^*_{k+1})$.

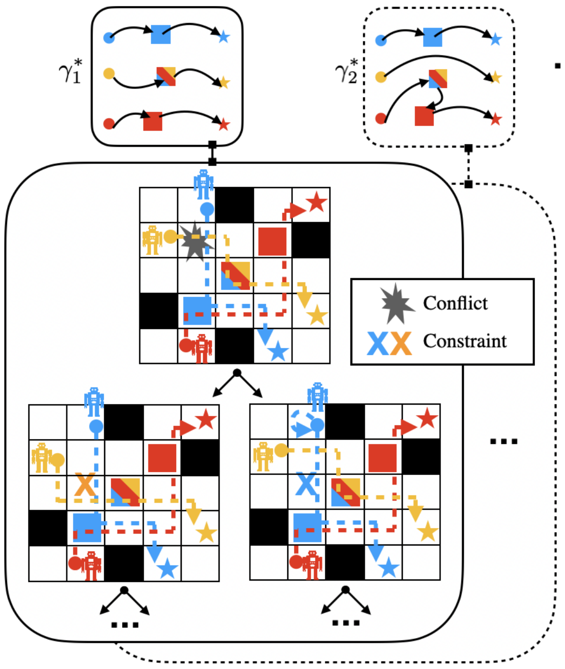

Fig. 5. An illustration of the search process in CBSS [8]. CBSS finds a joint sequence γ_1^*, ignoring any agent-agent conflicts, and then runs Conflict-Based Search. Here, the blue and the yellow agent runs into a vertex conflict. To resolve the conflict, a constraint is created for either the blue or the yellow agent and a minimum cost path satisfying all constraints for the agent is replanned. During the conflict resolution, if the cost of the current search exceeds a threshold value, a next-best joint sequence γ_2^* is created and the search continues. CBSS considers multiple joint sequences when needed and the entire search is organized in a best-first manner.

Therefore, when $\gamma(R_{k+1})$ is added to $\mathcal{S}$ on Line 7 in the $(k+1)$-th iteration, this theorem still holds. $\square$

Corollary 1. *Given a mHPP instance and a positive integer K, the 11/3-AK* computes a set of K joint sequence $\{\gamma_1, \gamma_2, \cdots, \gamma_K\}$ and $cost(\gamma_k^*) \leq \frac{11}{3} cost(\gamma_k^*), k = 1, 2, \cdots, K$, where γ_k^* is a k-th best solution for the given mHPP instance.*

Corollary 2. *Given a TSP instance and a positive integer K, the Christofides-AK* computes a set of K tours $\{\gamma_1, \gamma_2, \cdots, \gamma_K\}$ and $cost(\gamma_k^*) \leq \frac{3}{2} cost(\gamma_k^*), k = 1, 2, \cdots, K$, where γ_k^* is a k-th best solution for the given TSP instance.*

VI. Variants of Conflict-Based Steiner Search

A. *CBSS-A Algorithm*

CBSS-A is conceptually visualized in Fig. 5 and shown in Alg. 5. To solve MCPF, CBSS either creates a new joint sequence, or plan paths based on the existing joint sequences. Let $P = (\pi, g, \Omega)$ denote a *C-node* (C stands for CBSS), which differs from the "node" in AK* in Sec. IV-C. Each C-node consists of three elements:

- $\pi = (\pi^1, \pi^2, \ldots, \pi^N)$ is a joint path that connects the $v_o^i \in V_o$ and $v_d^i \in V_d$ for each agent $i \in I$.
- g is the scalar cost value of π, i.e., $g = g(\pi) = \Sigma_{i \in I} g(\pi^i)$.

Algorithm 5 Pseudocode for CBSS-A

```
INPUT: $G^W = (V^W, E^W, c^W)$, a workspace graph.
OUTPUT: a conflict-free joint path $\pi$ in $G^W$.
1:  $G_1 = (V_1, E_1, c_1) \leftarrow$ *CreateTargetGraph*($G^W$)
2:  $\gamma_1^* \leftarrow$ AK* ($G_1$,$f_A$,$K = 1$)
3:  $\Omega \leftarrow \emptyset$, $\pi, g \leftarrow$ *LowLevelSearch*($\gamma_1^*$, $\Omega$)
4:  Add $P_{root,1} = (\pi, g, \Omega)$ to OPEN$_C$
5:  **while** OPEN$_C \neq \emptyset$ **do**
6:      $P = (\pi, g, \Omega) \leftarrow$ OPEN$_C$.pop()
7:      $P' = (\pi', g', \Omega') \leftarrow$ *CheckNewRoot*($P$, OPEN$_C$)
8:      $cft \leftarrow$ *DetectConflict*($\pi'$)
9:      **if** $cft = NULL$ **then return** $\pi'$
10:     $\Omega \leftarrow$ *GenerateConstraints*($cft$)
11:     **for all** $\omega^i \in \Omega$ **do**
12:         $\Omega'' = \Omega' \cup \{\omega^i\}$
13:         $\pi'', g'' \leftarrow$ *LowLevelSearch*($\gamma(P')$, $\Omega''$)  $\triangleright$ only plan for agent $i$.
14:         Add $P'' = (\pi'', g'', \Omega'')$ to OPEN$_C$
15: **return** failure
```

- Ω is a set of collision constraints. A collision constraint is of form (i, v, t) (or (i, e, t)), which indicates agent i is prevented from occupying vertex v (or traversing edge e) at time t.

1) CBSS-A Overview: Both CBSS-A and CBSS are based on Conflict-Based Search (CBS) [11], a two-level search algorithm that includes a low-level search and a high-level search. The low-level search (*LowLevelSearch*) typically runs A* to find a minimum cost path for an agent based on a target sequence γ^i while satisfying a set of constraints on agent i. A path π^i *follows* a target sequence γ^i if π^i visits all targets in the same order as in γ^i. A joint path π *follows* a joint sequence γ if every path $\pi^i \in \pi$ follows $\gamma^i \in \gamma$.

In Alg. 5, to initialize (Lines 1-4), CBSS-A first creates a complete undirected metric graph G_1, which will be used as the input to AK*. CBSS-A then invokes AK* with $K = 1$ to compute γ_1, the first joint sequence. Afterwards, CBSS-A calls *LowLevelSearch* with an empty constraint set to plan a minimum cost path for each agent $i \in I$ by following $\gamma_1^i \in \gamma_1$, ignoring any possible conflict between agents. Finally, CBSS-A creates an initial C-node $P_{root,1}$ and add it to OPEN$_C$, a priority queue of C-nodes where C-nodes are prioritized based on their g-values from the minimum to the maximum. CBSS-A resolves conflicts by creating a constraint-tree (CT) for each joint sequence, and CBSS-A may create multiple CTs. This initial C-node is denoted as $P_{root,1}$, which is the *root C-node* of the first CT.

CBSS-A conducts the high-level search by first extracting a minimum cost C-node $P = (\pi, g, \Omega)$ from OPEN (Line 6). CBSS-A then determines whether a new root C-node should be created (Line 7) by calling *CheckNewRoot*, which is elaborated in the next paragraph. The returned C-node $P' = (\pi', g', \Omega')$ is either the input C-node P or a new root C-node $P_{root,2}$ (or $P_{root,3}, \cdots$). CBSS-A then checks P' for a conflict (i, j, v, t) (or (i, j, e, t), which is handled in a similar way and is thus omitted hereafter). If no conflict is detected, π' is returned and CBSS-A terminates. Otherwise, two constraints $\omega^i = (i, v, t)$ and $\omega^j = (j, v, t)$ are created based on the conflict (i, j, v, t) (Line 10), which means either

Algorithm 6 Pseudocode for *CheckNewRoot*

```
    INPUT: $P = (\pi, g, \Omega)$, OPEN$_C$
    OUTPUT: a C-node
 1: $r \leftarrow$ number of root C-nodes generated.
 2: $cost_{\gamma,max} \leftarrow \max\{cost(\gamma_1), cost(\gamma_2), \cdots, cost(\gamma_r)\}$
 3: if $g \leq (1+\epsilon)cost_{\gamma,max}$ then
 4:     return $P$
 5: $\gamma^*_{r+1} \leftarrow$ AK* $(G_1, f_A, K = r+1)$
 6: $\pi', g' \leftarrow$ LowLevelSearch$(\gamma^*_{r+1}, \emptyset)$
 7: $P_{root,r+1} = (\pi', g', \emptyset)$
 8: if $g \leq g'$ then
 9:     Add $P_{root,r+1}$ to OPEN$_C$
10:     return $P$
11: Add $P$ to OPEN$_C$
12: return $P_{root,r+1}$
```

agent i or agent j is not allowed to occupy vertex v at time t in G^W. For each $\omega^{i'} \in \{\omega^i, \omega^j\}$, CBSS-A then creates a new corresponding constraint set Ω'', which is the union of Ω' and $\{\omega^{i'}\}$, and invokes the *LowLevelSearch* (Line 13) to find a minimum cost path $\pi^{i'}$ for agent i' that satisfies Ω'' while following the target sequence $\gamma^{i'} \in \gamma(P')$ where $\gamma(P')$ denotes the joint sequence of the root node of the CT that contains P'. A new joint path π'' is created based on $\pi^{i'}$ where all other agents' paths in π'' remain the same as in π'. A new C-node $P'' = (\pi'', g(\pi''), \Omega'')$ is created and added to OPEN$_C$ (Line 14) for future search.

2) Check for New Root: As shown in Alg. 6, *CheckNewRoot* takes a C-node $P = (\pi, g, \Omega)$ and OPEN$_C$ as input and determines whether a new root C-node needs to be generated. *CheckNewRoot* first finds the maximum cost $cost_{\gamma,max}$ of all joint sequences that have been computed so far, and compares g against $(1+\epsilon)cost_{\gamma,max}$. Here, $\epsilon \geq 0$ is a user-defined hyper-parameter that tunes the behaviour of the search. When ϵ increases, CBSS-A tends to deter the generation of the next joint sequence during conflict resolution. If $g \leq (1+\epsilon)cost_{\gamma,max}$, the C-node P is returned for expansion. Otherwise, a new joint sequence is generated by calling AK* with $K = r+1$ and a new root C-node $P_{root,r+1}$ is created correspondingly. Then, the cheaper node among P and $P_{root,r+1}$ is returned while the other is added to OPEN$_C$.

3) Relationship to CBSS: CBSS-A is similar to CBSS [8] with the following two differences. First, while CBSS uses an exact algorithm to compute K-best joint sequences, CBSS-A uses AK* to compute approximated K-best joint sequences. Second, in CBSS, the generated K-best joint sequences have non-decreasing costs (as defined in Problem 3) and CBSS can always use $cost(\gamma_r)$ as $cost_{\gamma,max}$ for comparison on Line 3 in Alg. 6. In CBSS-A, the computed approximated K-best joint sequences are not guaranteed to have non-decreasing costs. *CheckNewRoot* needs to use the maximum cost of the joint sequence among $\{\gamma_1, \gamma_2, \cdots, \gamma_r\}$ for comparison.

B. *CBSS-AF Algorithm*

CBSS-AF differs from the CBSS-A by employing focal search [25] in both the high-level and the low-level search as in ECBS [17], which is summarized as follows.

1) A Review of Focal Search: Focal search [25] is an A*-like search algorithm that finds bounded sub-optimal solutions. While A* uses an OPEN list to store and prioritize search nodes for future expansion based on their f-values, focal search further introduces another list called FOCAL. Let f_{min} denote the minimum f-value of search nodes in OPEN at any time during the search, FOCAL contains search nodes n in OPEN that have $f(n) \leq (1+w)f_{min}$ with $w \in [0, \infty]$ being a weight parameter that is defined by the users. The FOCAL list then prioritizes the contained search nodes based on a different value $d(n)$ which estimates the "distance-to-go", i.e., the number of hops from n to the goal. Since f_{min} is a lower bound on the optimal solution cost g^*, focal search guarantees finding a solution whose cost is at most $(1+w)g^*$.

2) Low-Level Search: When the low-level search is called for an agent i with a constraint set Ω in a C-node $P = (\pi, g, \Omega)$, the low-level search finds a bounded sub-optimal path for agent i while satisfying all constraints in Ω that is related to i by using the focal search. This focal search defines d as the number of conflicts with respect to the paths of all other agents in π, and seeks to minimize the number of conflicts with other agents' paths. When a solution path π^i is found, the low-level search returns both π^i and f^i_{min}, i.e., the minimum f-value over all nodes in OPEN when the search terminates, which is a lower bound on the cost of an optimal solution path g^{*i} for agent i in that low-level search. It is guaranteed that the cost of the returned path satisfies: $g(\pi^i) \leq (1+w)f^i_{min} \leq (1+w)g^{*i}$. The focal search minimizes the conflicts to be resolved by the high-level search and therefore expedites the entire computation, rather than speeding up the low-level search itself.

3) High-Level Search: The high-level search employs another focal search to expedite the computation by selecting a bounded sub-optimal C-node with the fewest number of conflicts for expansion in every iteration. Specifically, the high-level search stores C-nodes P in OPEN and prioritizes them based on a lower bound value $lb(P) := \sum_{i \in I} f^i_{min}(P)$ where $f^i_{min}(P)$ indicates the f^i_{min} returned by the low-level search for agent i when generating the C-node P. The FOCAL list then stores C-nodes $P = (\pi, g, \Omega)$ in OPEN whose g-value satisfies $lb(P) \leq g \leq (1+w)lb(P)$, and prioritizes these nodes P based on the number of conflicts in π from the minimum to the maximum. Note that the number of conflicts indicates the "distance-to-go" of the high-level search. Let $\gamma(P)$ denote the joint sequence that the joint path π in P follows, and let $\pi^*_{\gamma(P)}$ indicates the minimum cost conflict-free joint path that follows $\gamma(P)$. Since the paths returned by the low-level search is bounded sub-optimal for each agent, for a C-node $P = (\pi, g, \Omega)$, we know that $lb(P) \leq g(\pi) \leq (1+w)lb(P) \leq (1+w)g(\pi^*_{\gamma(P)})$. Therefore, every node in the FOCAL list is bounded sub-optimal, and the cost of the solution joint path returned is at most $(1+w)g(\pi^*_{\gamma(P)})$.

C. *Bounded Sub-Optimal Joint Path*

Let α denote the approximation ratio provided by the approximation algorithm used in AK*. Let ϵ denote the parameter that defers the generation of the next-best joint target sequence.

Let w denote the weight parameter of the FOCAL list in CBSS-AF.

Theorem 3. *For a solvable MCPF instance, let π^* denote a minimum cost conflict-free joint path of this MCPF instance. Then, the joint path π returned by CBSS-A is guaranteed to have a cost $g(\pi) \leq \alpha(1+\epsilon)g(\pi^*)$.*

The same proof in [8] can be applied here and we only summarize the main idea. Same as CBSS, CBSS-A conducts search along two directions: CBSS-A either computes a new joint sequences, or resolving conflicts along joint paths that follow the generated joint sequences. By Theorem 2, the computed new joint sequence γ_{r+1} is bounded sub-optimal with respect to a $(r+1)$-th best joint sequence γ^*_{r+1} at any time during the search. For conflict resolution, a CT is created for each computed joint sequence, and C-nodes in CT that are expanded have non-decreasing costs. The entire search is conducted in a best-first manner by always extracting the cheapest C-node from OPEN, and the first conflict-free joint path found is thus guaranteed to be bounded sub-optimal. For unsolvable instance, both CBSS and CBSS-A cannot terminate in finite time since the underlying CBS search [11] cannot terminate in finite time.

Corollary 3. *When using 11/3-AK* to implement AK* in CBSS-A, for a solvable MCPF instance, the computed conflict-free joint path π by CBSS-A has a cost $g(\pi) \leq \frac{11}{3}(1+\epsilon)g(\pi^*)$*

Theorem 4. *For a solvable MCPF instance, let π^* denote a minimum cost conflict-free joint path of this MCPF instance. Then, the joint path π returned by CBSS-AF is guaranteed to have a cost $g(\pi) \leq \alpha(1+\epsilon)(1+w)g(\pi^*)$.*

VII. Experimental Results

We implement AK*, CBSS-A, CBSS-AF and the baselines in Python. When implementing AK*, we use the graph data structure and algorithms (e.g. minimum spanning tree) in NetworkX, a popular Python package. We run all tests on a laptop with an Intel Core i7-11800H CPU and 16GB RAM.

A. Experimental Results for AK*

We use a random 32x32 four-neighbor grid (Fig. 6). We create instances by (i) randomly sampling N initial vertices V_o, N goal vertices V_d and M target vertices V_t in the grid, (ii) finding the shortest path between each pair of vertices $(u,v), u,v \in V_o \cup V_t \cup V_d$, and (iii) using the path length as the edge cost $c(u,v)$. The assignment constraints are: each goal vertex $v_d \in V_d$ is assigned to a unique agent (i.e., $f_A(v_d)=\{i\}, i \in I$), and all targets $v_t \in V_t$ are anonymous (i.e, $f_A(v_t)=I$). The resulting target graph G_1 is the input graph to 11/3-AK*, and $K=10$ for all tests. We generate 25 instances for each combination of $N \in \{1,2,5,10\}$ and $M \in \{10,20,30,40,50\}$. We implement a transformation algorithm (TFA) for mHPP [8], [13] as a baseline. This TFA first transforms an mHPP to a single-agent asymmetric TSP, then invokes LKH 2.0.9, a solver that can handle asymmetric TSP, to obtain a tour τ to the asymmetric TSP, and finally un-transforms the tour τ into a joint sequence γ. TFA ensures that when τ is a minimum cost tour for the asymmetric TSP, then the corresponding γ is a minimum cost joint sequence for mHPP. We use TFA to compute γ^*_1, an optimal[4] solution to mHPP and use 11/3-AK* to compute approximated K-best ($K=10$) joint sequences $\{\gamma_1,\gamma_2,\cdots,\gamma_{10}\}$.

1) Solution Costs: For each instance, let $cost_{avg}$ denote the average of $\{cost(\gamma_k), k=1,2,\cdots,10\}$ and define the cost ratio as $CR := cost_{avg}/cost(\gamma^*_1)$. We report the minimum, median and maximum of CR over all instances for each combination of N,M in Fig. 6(a) using the black error bars against the left vertical axis. As shown in Fig. 6(a), CR is always less than 2 in our tests, and is often less than 1.5, which means 11/3-AK* can often compute solutions that are closer to the optimum than the sub-optimality bound 11/3 (≈ 3.67).

2) Runtime: For each instance, let RT_{TFA} denote the runtime of TFA to compute γ^*_1 and let RT_{AK} denote the runtime of 11/3-AK* to compute γ_1. Let $RR := RT_{AK}/RT_{TFA}$ denote the runtime ratio, and we report the minimum, median and maximum of RR over all instances for each combination of N,M in Fig. 6(a) using the red error bars against the right vertical axis. As shown in Fig. 6(a), in our tests, when N,M are small, RR is close to 1, which means 11/3-AK* does not have runtime advantage in comparison with TFA. As N,M increases, RR decreases obviously, which means 11/3-AK* runs much faster than TFA and can handle more agents and more targets in general. For example, when $N=10$ and $M=50$, RR is less than 10^{-2}, which means 11/3-AK* takes less than 1% of the runtime required by TFA to compute γ_1.

3) Runtime for SPPFP: SPPFP-Solve is an important procedure in 11/3-AK*. We report the average runtime per instance (in seconds) of both the entire 11/3-AK* and of the procedure *SPPFP-Solve* in 11/3-AK*. Here, we run an additional set of tests with $N=20$ and $M \in \{40,80,120,160,200\}$ for 11/3-AK* to verify its runtime when the number of agents and targets further increases. As shown in Fig. 6(b), on average, 11/3-AK* takes less than a second to handle instances with $N \leq 10$ and $M \leq 50$ and takes up to around 50 seconds to handle instances with $N=20$ and $M=200$. In these tests, we observe that procedure SPPFP takes from 23% to 48% of the total runtime of 11/3-AK*.

To summarize, 11/3-AK* often runs much faster than the baseline TFA when computing a solution to mHPP, especially when N,M are large. The solutions computed by 11/3-AK* are bounded sub-optimal and can be closer to the true optimum than the theoretic sub-optimality bound. Finally, we acknowledge that our implementation can be further expedited by optimizing the software or using C/C++ instead of Python.

B. Experimental Results for CBSS-A and CBSS-AF

We then test CBSS-A and compare it against CBSS. We fix the the number of agents $N=10$. We use $\epsilon=0.01$ for both CBSS-A and CBSS. Additionally, we use CBSS with $\epsilon=0.5$ as an additional baseline for comparison. The

[4]LKH is a heuristic algorithm for TSP and does not guarantee solution optimality. In practice, LKH often computes an optimal solution.

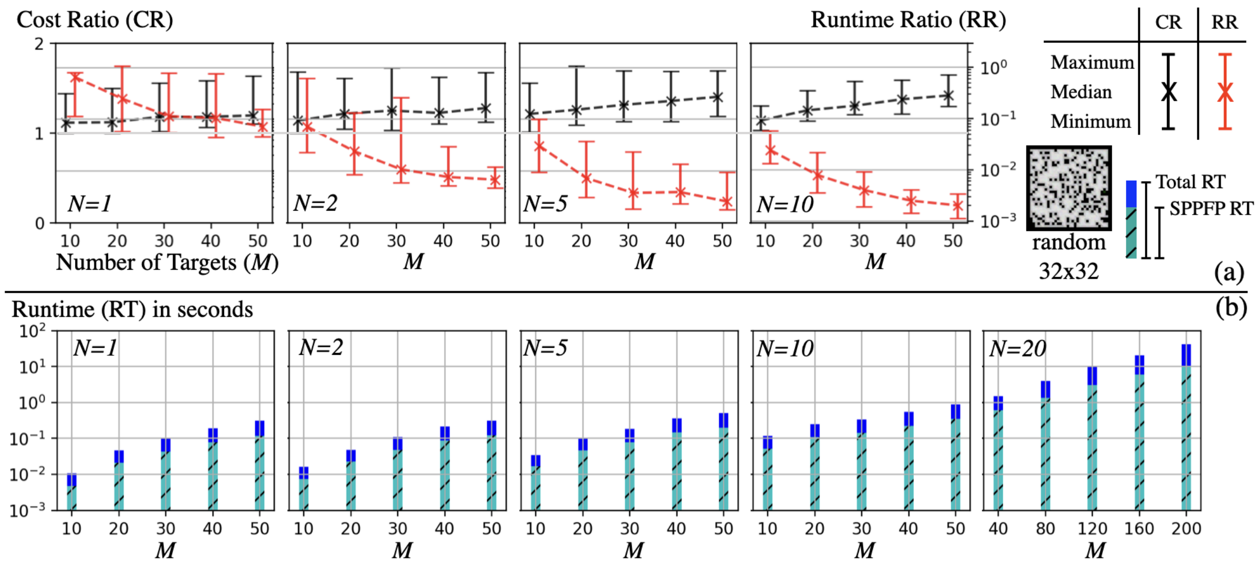

Fig. 6. Experimental results for K-best mHPP. (a) shows the minimum, median and maximum of cost ratios and runtime ratios over 25 instances for each M and N. (b) shows the average runtime in seconds of both the entire 11/3-AK* and of the procedure *SPPFP-Solve* in 11/3-AK*.

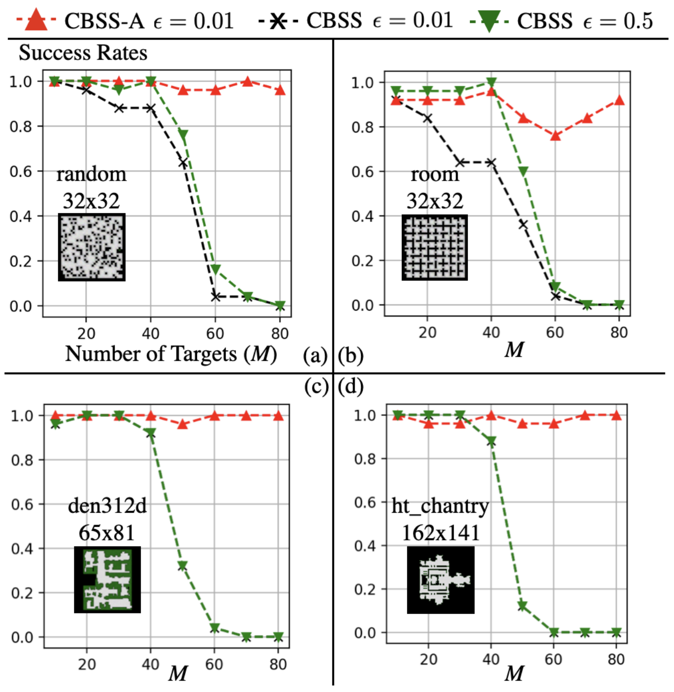

Fig. 7. Experimental results of success rates. The curves are against show the success rates of the algorithms. Our CBSS-A achieves higher success rates than the baselines.

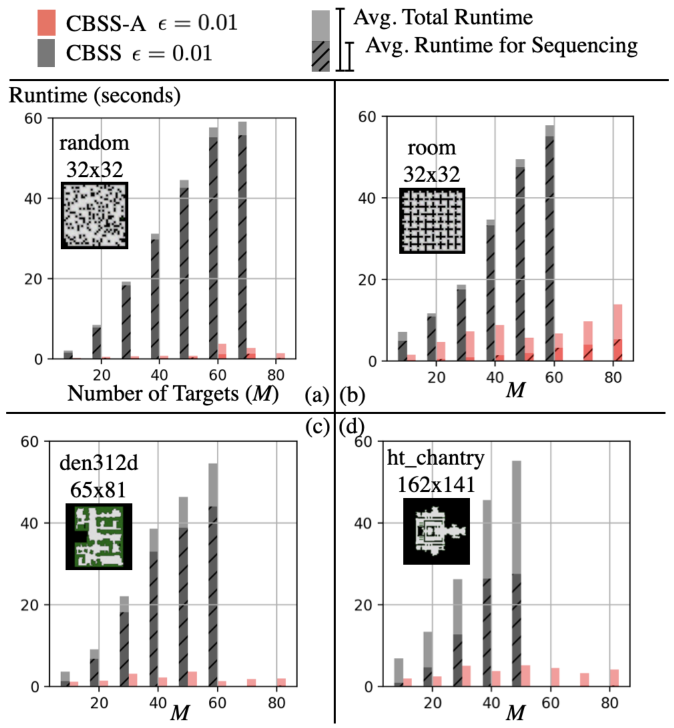

Fig. 8. Experimental results of runtime. The bars show the average total runtime and the average runtime for compute joint sequences (i.e., target sequencing). CBSS-A reduces the runtime for target sequencing.

selection of $\epsilon = 0.5$ is based on the observation that CBSS-A often computes up to 50% more expensive solutions than CBSS in practice (which will be discussed in Sec. VII-B2). We use four different grid maps of various sizes from [9]. For each $M \in \{10, 20, 30, \cdots, 80\}$, there are 25 instances. The assignment constraints are: each destination $v_d \in V_d$ is assigned to a unique agent (i.e., $f_A(v_d) = \{i\}, i \in I$), and all targets $v_t \in V_t$ are anonymous (i.e, $f_A(v_t) = I$). We set a runtime limit of 60 seconds for each instance.

1) Success Rates and Runtime for Sequencing: As shown by the red, black and green curves in Fig. 7, as M increases, CBSS-A achieves higher success rates than CBSS $\epsilon = 0.01$ and $\epsilon = 0.5$. Furthermore, the black and red bars in Fig. 8 show the average total runtime and the average runtime for target sequencing over the succeeded instances of each algorithm (i.e., instances that the algorithm successfully solves within the runtime limit). We omit the bar plot for CBSS

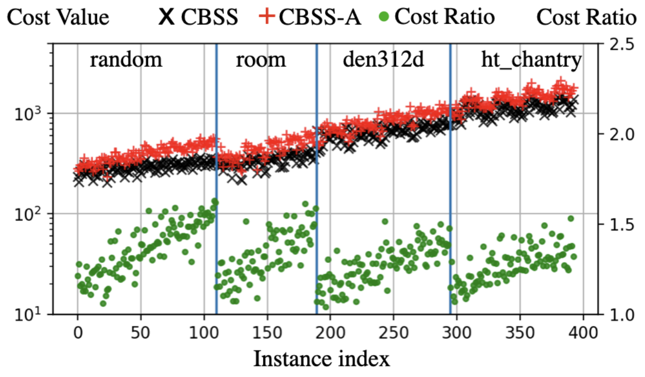

Fig. 9. Solution cost comparison between CBSS-A and CBSS. The solution computed by CBSS-A are often 10%-50% more expensive than the solutions computed by CBSS.

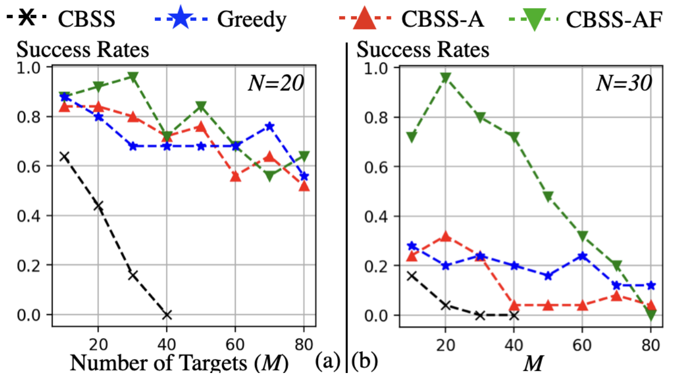

Fig. 10. Success rates of algorithms with $N = 20, 30$ in Random 32x32 map. Our CBSS-A and CBSS-AF have higher success rates than the baselines.

$\epsilon = 0.5$ to make the figure readable, and the bar plot for CBSS $\epsilon = 0.5$ is similar to the one for CBSS $\epsilon = 0.01$. CBSS-A requires less runtime than CBSS for solving mHPP to find joint sequences. As M increases (e.g. when $M = 80$), CBSS fails to compute the first joint sequence within the runtime limit. In contrast, CBSS-A can still quickly compute joint sequences and plan conflict-free paths. The reason is CBSS-A employs AK* to bypass the computational challenge in target sequencing when M is large. Additionally, we observe from Fig. 7(b) that, as M increases, the success rate of CBSS-A sometimes increases. A possible reason is that all the targets are anonymous and having more targets can change the paths of the agents. The changed paths may lead to fewer conflicts, which may speed up the overall search.

2) Solution Costs: For each instance, let $cost_{\text{CBSS}}$ and $cost_{\text{CBSS-A}}$ denote the solution cost computed by the respective algorithm. Let $cost_{\text{CBSS-A}}/cost_{\text{CBSS}}$ be the cost ratio. To compare the solution quality, we report $cost_{\text{CBSS}}$, $cost_{\text{CBSS-A}}$ and the cost ratio for each instance that are successfully solved within the runtime limit by both CBSS-A and CBSS. As shown in Fig. 9, $cost_{\text{CBSS-A}}$ is often up to 50% more expensive than $cost_{\text{CBSS}}$.

3) Varying Number of Agents: We test our CBSS-A ($\epsilon = 0.01$), CBSS-AF ($\epsilon = 0.01$, $w = 0.1$), the existing CBSS and a greedy baseline in the random 32x32 map with $N = 20$ and $N = 30$, and the results are shown in Fig. 10. This greedy baseline first assigns the targets to the nearest agent in a greedy manner, then uses LKH to find the optimal visiting order for each agent, and finally uses CBS to find conflict-free paths. As N increases, the success rates of all algorithms decrease. The algorithms spend more time in conflict resolution between agents. CBSS-AF achieves higher success rates than all other algorithms, and the reason is the employment of focal search in CBSS-AF, which helps CBSS-AF resolve conflicts more efficiently than the other algorithms. For example, when $N = 30, M = 20, 30, 40$, CBSS-AF often triples the success rates of other algorithms.

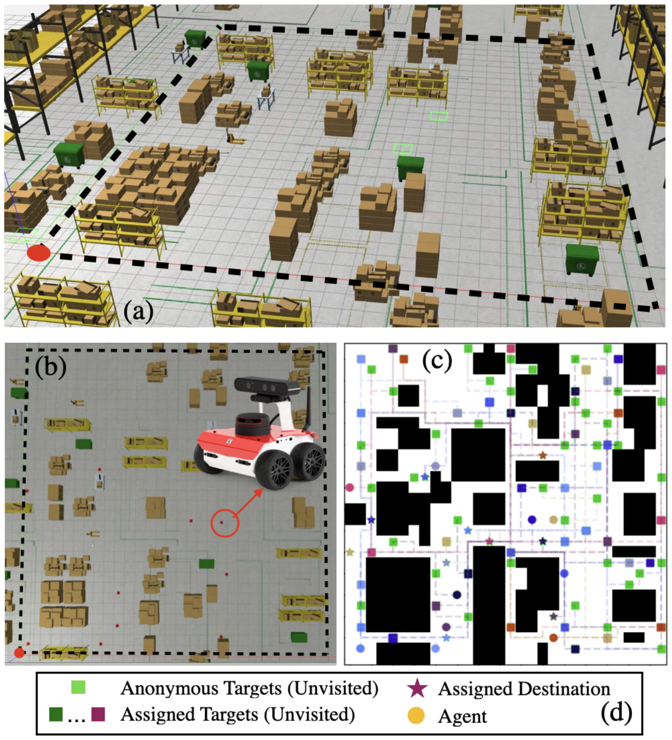

Fig. 11. Gazebo simulation. (a) shows an oblique view of the simulated warehouse. (b) shows both a top-down view of the simulated region and the ROSBot model used in the simulation. (c) shows a four-neighbor grid representation of the simulated region. (d) explains the markers used in (c).

To summarize, CBSS-A trades off solution quality for runtime efficiency. CBSS-A can reduce the runtime for target sequencing due to the usage of 11/3-AK*, and can achieve higher success rates than CBSS in general. The solution cost computed by CBSS-A is often up to 50% more expensive than the solution cost computed by CBSS. CBSS-AF further improves the success rates of CBSS-A especially when the number of agents is large.

C. Gazebo Simulation

We demonstrate the usage of our planner in a simulated warehouse in Gazebo/ROS, where the planned paths are executed by differential drive robots to visit a set of target locations. Fig. 11(a) and 11(b) show an oblique view and a top-down view of the simulated warehouse, where the black dashed lines mark the region that is used in the simulation. The region is of size 30m$\times$30m, containing ten agents ($N = 10$)

and 80 targets ($M = 80$). There are 10 starts, 10 goals and 80 targets, in the corresponding target graph. We set the assignment constraints as follows: each goal is assigned to a unique agent; 40 targets are evenly assigned to all agents; the remaining 40 targets are anonymous. Fig. 11(c) shows an occupancy grid representation of the region and the planned paths of the agents. The grid is of size 30×30, where each cell is of size 1m×1m. This simulation uses the ROSBot model, a differential drive robot. The uncertainty and disturbance in robot motion is simulated using the default setting in Gazebo but is not considered by the planner. We execute the planned paths of the agents in a centralized manner where each agent is required to follow the waypoints in its path while respecting a shared global clock. The intelligent and robust execution of a joint path is itself a research topic [55] and is beyond the scope of this paper. Our multi-media attachment provides a video of this simulation.

VIII. Conclusion and Future Work

This paper considers K-best TSPs. First, we develop a new partition method AK* that is able to convert an approximation algorithms for TSPs to its K-best counterpart, while preserving the solution sub-optimality bounds and the polynomial runtime complexity. Our experimental results verify the fast running speed of AK* and the bounded sub-optimality of the computed solution. Second, based on AK*, we develop CBSS-A and CBSS-AF, algorithms that can find bounded sub-optimal collision-free paths for MCPF problems. CBSS-A and CBSS-AF are able to bypass the challenge in target sequencing with a large number of targets in MCPF problems, and CBSS-AF is able to handle more agents than CBSS-A and CBSS by leveraging the focal search technique.

There are several future work directions. First, one can consider combining AK* with heuristic algorithms for TSPs by first using AK* to find a set of approximated K-best solutions, and then improving these K-best solutions via local search. Heuristic algorithms can typically compute high-quality solutions in practice but lacks worst-case guarantees. Combining the approximation method such as AK* with heuristic algorithms may obtain both theoretic sub-optimality bounds and high-quality solutions in practice. Second, the developed CBSS-AF shows that the existing conflict resolution techniques for MAPF can be leveraged to improve the scalability of CBSS when the number of agents increases in MCPF. Other techniques in MAPF such as [18] can be potentially combined with CBSS and is worthwhile further investigation. We note from our simulation that, the disturbance in robot motion may affect the execution of the planned path. To handle the disturbance, one can use a similar approach in [55] to ensure collision-free execution of the planned paths, which is part of our future work.

Acknowledgments

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.

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. 9–9, 2008.

[2] J. Keller, D. Thakur, M. Likhachev, J. Gallier, and V. Kumar, "Coordinated path planning for fixed-wing uas conducting persistent surveillance missions," *IEEE Transactions on Automation Science and Engineering*, vol. 14, no. 1, pp. 17–24, 2016.

[3] M. Schneier, M. Schneier, and R. Bostelman, *Literature review of mobile robots for manufacturing*. US Department of Commerce, National Institute of Standards and Technology ..., 2015.

[4] K. Brown, O. Peltzer, M. A. Sehr, M. Schwager, and M. J. Kochenderfer, "Optimal sequential task assignment and path finding for multi-agent robotic assembly planning," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 441–447.

[5] C. Liu and A. Kroll, "A centralized multi-robot task allocation for industrial plant inspection by using a* and genetic algorithms," in *Artificial Intelligence and Soft Computing: 11th International Conference, ICAISC 2012, Zakopane, Poland, April 29-May 3, 2012, Proceedings, Part II 11*. Springer, 2012, pp. 466–474.

[6] 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.

[7] 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.

[8] Z. Ren, S. Rathinam, and H. Choset, "CBSS: A new approach for multiagent combinatorial path finding," *IEEE Transactions on Robotics*, pp. 1–15, 2023.

[9] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. Kumar *et al.*, "Multi-agent pathfinding: Definitions, variants, and benchmarks," *arXiv preprint arXiv:1906.08291*, 2019.

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

[11] 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.

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

[13] P. Oberlin, S. Rathinam, and S. Darbha, "Today's traveling salesman problem," *IEEE robotics & automation magazine*, vol. 17, no. 4, pp. 70–77, 2010.

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

[15] E. S. van der Poort, M. Libura, G. Sierksma, and J. A. van der Veen, "Solving the k-best traveling salesman problem," *Computers & Operations Research*, vol. 26, no. 4, pp. 409–425, 1999.

[16] H. W. Hamacher and M. Queyranne, "K best solutions to combinatorial optimization problems," *Annals of Operations Research*, vol. 4, no. 1, pp. 123–143, 1985.

[17] M. Barer, G. Sharon, R. Stern, and A. Felner, "Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem," in *Seventh Annual Symposium on Combinatorial Search*, 2014.

[18] J. Li, D. Harabor, P. J. Stuckey, H. Ma, G. Gange, and S. Koenig, "Pairwise symmetry reasoning for multi-agent path finding search," *Artificial Intelligence*, vol. 301, p. 103574, 2021.

[19] S.-H. Chan, J. Li, G. Gange, D. Harabor, P. J. Stuckey, and S. Koenig, "Flex distribution for bounded-suboptimal multi-agent path finding," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 9, 2022, pp. 9313–9322.

[20] 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.

[21] J. Hoogeveen, "Analysis of christofides' heuristic: Some paths are more difficult than cycles," *Operations Research Letters*, vol. 10, no. 5, pp. 291–295, 1991.

[22] S. Yadlapalli, J. Bae, S. Rathinam, and S. Darbha, "Approximation algorithms for a heterogeneous multiple depot hamiltonian path problem," in *Proceedings of the 2011 American Control Conference*. IEEE, 2011, pp. 2789–2794.

[23] G. Even, N. Garg, J. Könemann, R. Ravi, and A. Sinha, "Min–max tree covers of graphs," *Operations Research Letters*, vol. 32, no. 4, pp. 309–315, 2004.
[24] G. Laporte, "The traveling salesman problem: An overview of exact and approximate algorithms," *European Journal of Operational Research*, vol. 59, no. 2, pp. 231–247, 1992.
[25] J. Pearl and J. H. Kim, "Studies in semi-admissible heuristics," *IEEE transactions on pattern analysis and machine intelligence*, no. 4, pp. 392–399, 1982.
[26] R. Beard, T. McLain, M. Goodrich, and E. Anderson, "Coordinated target assignment and intercept for unmanned air vehicles," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 6, pp. 911–922, 2002.
[27] P. Sujit and D. Ghose, "Search using multiple uavs with flight time constraints," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 40, no. 2, pp. 491–509, 2004.
[28] L. Trevisan, "Combinatorial optimization: exact and approximate algorithms," *Standford University*, 2011.
[29] S. Sahni and T. Gonzalez, "P-complete approximation problems," *Journal of the ACM (JACM)*, vol. 23, no. 3, pp. 555–565, 1976.
[30] W. Malik, S. Rathinam, and S. Darbha, "An approximation algorithm for a symmetric generalized multiple depot, multiple travelling salesman problem," *Operations Research Letters*, vol. 35, pp. 747–753, 2007.
[31] K. Sundar and S. Rathinam, "Generalized multiple depot traveling salesmen problem-polyhedral study and exact algorithm," *Computers & Operations Research*, vol. 70, pp. 39–55, 2016.
[32] J. Li, M. Zhou, Q. Sun, X. Dai, and X. Yu, "Colored traveling salesman problem," *IEEE transactions on cybernetics*, vol. 45, no. 11, pp. 2390–2401, 2014.
[33] T. S. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.
[34] D. Silver, "Cooperative pathfinding." 01 2005, pp. 117–122.
[35] E. Boyarski, A. Felner, R. Stern, G. Sharon, D. Tolpin, O. Betzalel, and E. Shimony, "Icbs: improved conflict-based search algorithm for multi-agent pathfinding," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.
[36] L. Cohen, T. Uras, T. Kumar, and S. Koenig, "Optimal and bounded-suboptimal multi-agent motion planning," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 10, no. 1, 2019, pp. 44–51.
[37] 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.
[38] ——, "A conflict-based search framework for multiobjective multiagent path finding," *IEEE Transactions on Automation Science and Engineering*, pp. 1–13, 2022.
[39] 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.
[40] 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.
[41] 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.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *Conference on Autonomous Agents & Multiagent Systems*, 2017.
[47] M. Liu, H. Ma, J. Li, and S. Koenig, "Task and path planning for multi-agent pickup and delivery," in *2019 AAMAS*, 2019, pp. 1152–1160.
[48] Q. Xu, J. Li, S. Koenig, and H. Ma, "Multi-goal multi-agent pickup and delivery," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 9964–9971.
[49] C. Henkel, J. Abbenseth, and M. Toussaint, "An optimal algorithm to solve the combined task allocation and path finding problem," *arXiv preprint arXiv:1907.10360*, 2019.
[50] Z. Ren, A. Nandy, S. Rathinam, and H. Choset, "Dms*: Towards minimizing makespan for multi-agent combinatorial path finding," *IEEE Robotics and Automation Letters*, vol. 9, no. 9, pp. 7987–7994, 2024.
[51] Y. Zhang, H. Wang, and Z. Ren, "A short summary of multi-agent combinatorial path finding with heterogeneous task duration (extended abstract)," in *Seventeenth International Symposium on Combinatorial Search, SOCS 2024, Kananaskis, Alberta, Canada, June 6-8, 2024*, A. Felner and J. Li, Eds. AAAI Press, 2024, pp. 301–302.
[52] J. Li, X. Meng, M. Zhou, and X. Dai, "A two-stage approach to path planning and collision avoidance of multibridge machining systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 7, pp. 1039–1049, 2016.
[53] D. Villeneuve and G. Desaulniers, "The shortest path problem with forbidden paths," *European Journal of Operational Research*, vol. 165, no. 1, pp. 97–107, 2005.
[54] O. J. Smith and M. W. Savelsbergh, "A note on shortest path problems with forbidden paths," *Networks*, vol. 63, no. 3, pp. 239–242, 2014.
[55] 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.

Zhongqiang (Richard) Ren (Member, IEEE) received the dual B.E. degree from Tongji University, Shanghai, China, and FH Aachen University of Applied Sciences, Aachen, Germany, and the M.S. and Ph.D. degrees from Carnegie Mellon University, Pittsburgh, PA, USA. He is currently an assistant Professor at the UM-SJTU Joint Institute, Shanghai Jiao Tong University, Shanghai, China.

Sivakumar Rathinam (Senior Member, IEEE) received the Ph.D. degree from the University of California at Berkeley in 2007. He is currently a Professor with the Mechanical Engineering Department, Texas A&M University. His research interests include motion planning and control of autonomous vehicles, collaborative decision making, combinatorial optimization, vision-based control, and air traffic control.

Howie Choset (Fellow, IEEE) received the undergraduate degrees in computer science and business from the University of Pennsylvania, Philadelphia, PA, USA, and the M.S. and Ph.D. degrees in mechanical engineering from Caltech, Pasadena, CA, USA. He is a Professor in the Robotics Institute, Carnegie Mellon, Pittsburgh, PA, USA.