Robotics: Science and Systems 2022
New York City, NY, USA, June 27-July 1, 2022

Conflict-Based Steiner Search for Multi-Agent Combinatorial Path Finding

Zhongqiang Ren
Carnegie Mellon University
Pittsburgh, PA 15213
Email: zhongqir@andrew.cmu.edu

Sivakumar Rathinam
Texas A&M University
College Station, TX 77843-3123
Email: srathinam@tamu.edu

Howie Choset
Carnegie Mellon University
Pittsburgh, PA 15213
Email: choset@andrew.cmu.edu

***Abstract*—Conventional Multi-Agent Path Finding (MAPF) problems aim to compute an ensemble of collision-free paths for multiple agents from their respective starting locations to pre-allocated destinations. This work considers a generalized version of MAPF called Multi-Agent Combinatorial Path Finding (MCPF) where agents must collectively visit a large number of intermediate target locations along their paths before arriving at destinations. This problem involves not only planning collision-free paths for multiple agents but also assigning targets and specifying the visiting order for each agent (i.e. multi-target sequencing). To solve the problem, we leverage the well-known Conflict-Based Search (CBS) for MAPF and propose a novel framework called Conflict-Based Steiner Search (CBSS). CBSS interleaves (1) the conflict resolving strategy in CBS to bypass the curse of dimensionality in MAPF and (2) multiple traveling salesman algorithms to handle the combinatorics in multi-target sequencing, to compute optimal or bounded sub-optimal paths for agents while visiting all the targets. Our extensive tests verify the advantage of CBSS over baseline approaches in terms of computing shorter paths and improving success rates within a runtime limit for up to 20 agents and 50 targets. We also evaluate CBSS with several MCPF variants, which demonstrates the generality of our problem formulation and the CBSS framework.**

I. Introduction

Multi-Agent Path Finding (MAPF), as its name suggests, computes a set of collision-free paths for multiple agents from their respective starting locations to designated destinations. This article addresses a generalization of MAPF, referred to as Multi-Agent Combinatorial Path Finding (MCPF), where the agents are also required to visit a collection of target locations before reaching their destinations while satisfying additional agent-target assignment constraints (see Fig. 1 (a) for a toy example). MAPF and its generalizations such as MCPF arise in applications in logistics [41] and surveillance [12]. For example, in a hazardous material warehouse, multiple mobile robots equipped with various sensors need to collectively measure temperature, humidity and detect potential leakage of various hazardous chemicals at many predefined target locations. These robots need to plan their paths such that each target location is visited at least once by a robot and the paths are collision-free. In addition, since robots may carry different sensors, only a subset of robots may have the sensors to measure the desired data at a target location; this introduces agent-target assignment constraints that must be respected while planning paths. Simpler versions of the MCPF without

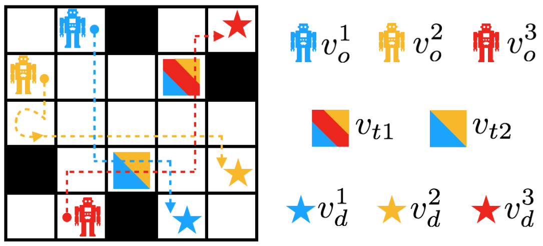

Fig. 1: A feasible solution to a MCPF problem. There are three agents; their starting locations are ($v_o^i, i = 1, 2, 3$) and destinations are ($v_d^i, i = 1, 2, 3$). The two targets are denoted as v_{t1} and v_{t2}. The color of the targets and destinations indicates the assignment constraints, (i.e., the subset of agents that are eligible to visit the target or destination). For example, the target v_{t2} can be visited by either the yellow or the blue agent. Feasible paths for agents are shown using dashed lines. The circular part of the yellow path indicates a wait-in-place action.

the robot-robot collision constraints have been addressed in [20, 35], motivated by unmanned vehicle applications.

Solving MCPF with optimality guarantees is quite challenging as it requires handling both the curse of dimensionality in multi-agent path planning as well as the multi-target sequencing. If the set of (intermediate) targets to visit is empty and each destination is pre-assigned to a unique agent, MCPF reduces to MAPF [34] which is NP-hard [42]. On the other hand, if the collisions between agents are ignored and no agent-target constraints are present, MCPF reduces to the Multiple Traveling Salesman Problem (mTSP) [20] which is also NP-hard [21]. As such, solving MCPF to optimality involves simultaneously addressing the challenges in both MAPF and mTSP.

Unlike MAPF, where ignoring the conflicts between the agents leads to a decoupled shortest path problem for each agent, in MCPF, ignoring the conflicts between the agents leads to an mTSP where the agents' paths are still coupled. In the special case where there are no agent-target assignment constraints, an approach called MS* [26] has been proposed based on the subdimensional expansion framework [39]. In this paper, we present a new framework called Conflict-Based Steiner Search (CBSS) for the general case that attempts to bypass both the combinatorics in mTSP and the curse of dimensionality in MAPF to solve MCPF. CBSS *interleaves* mTSP and MAPF algorithms by alternating between (1) generating new target sequences for the agents, and (2) generating conflict-free paths for agents given the target sequences. To compute conflict-free paths, CBSS conducts a two-level search like CBS [31]: On the high level, CBSS generates a search forest (of multiple trees) where each tree follows a fixed target sequence for each agent. The key contribution of this paper is in the generation of the search forest where we leverage the transformation [20] and K-best partition [38] methods for the TSP; the transformation method allows us to find a solution for the mTSP by posing an equivalent TSP on a larger graph, and the K-best partition method allows us to incrementally generate K least-cost mTSP solutions. Each mTSP solution allocates the targets among the agents and specifies a path (thereby, also fixing the target sequence) for each agent to visit. On the low level, CBSS runs constrained single-agent search to plan a path for each agent following the target sequences while satisfying the collision avoidance constraints.

We show that CBSS is guaranteed to compute an optimal or an ϵ-bounded sub-optimal solution, where ϵ is a pre-defined bound specified by the user. By varying ϵ from zero to infinity, CBSS moves along a spectrum from computing an optimal solution with heavy computational burden to a naive sequential method that efficiently computes a feasible solution without any theoretic optimality bounds.

To verify CBSS, we generate test instances with various forms of assignment constraints based on an online dataset [34]. We compare CBSS in various maps with several baselines including a greedy method and the recent MS* [26]. We observe that CBSS computes shorter paths than the greedy method and doubles the success rates in comparison with MS*. By varying the form of assignment constraints, we show that CBSS is widely applicable to solve different cases of MCPF.[1] Finally, we carry out a multi-robot experiment to validate that the planned paths are executable on physical robots.

The rest of the article is organized as follows. In Sec. II, we review related problems and methods. We then formulate the MCPF problem in Sec. III. The CBSS framework is introduced in Sec. V with proofs in Sec. VI. Extensive numerical results are presented in Sec. VII. Finally, we conclude our work, summarize the contributions and outline possible future directions in Sec. VIII.

II. Related Work

Multi-Agent Path Finding algorithms tend to fall on a spectrum from coupled methods [33] to decoupled methods [32], trading off completeness and optimality for scalability. In the middle of this spectrum lies the popular dynamically-coupled methods such as subdimensional expansion [39] and Conflict-Based Search (CBS) [31]. These methods have been improved and extended in many ways [1, 4, 7, 25, 27, 29], to name a few. All of them aim to navigate each agent to its pre-assigned destination without visiting any intermediate targets along the path, which differs from MCPF.

[1]Our implementation is at https://github.com/wonderren/public_pymcpf.

The Traveling Salesman Problem (TSP), which aims to find a shortest path/tour for a single agent to visit each node in a given graph, is one of the most well known NP-hard problems in the literature [2]. A spectrum of methods have been developed ranging from exact techniques (branch and bound, branch and price) [2] to heuristics [9, 10] and approximation algorithms [6] which trade off optimality for computational efficiency.

The Multiple Traveling Salesman Problem (mTSP) [3] is harder to solve compared to the (single-agent) TSP as the nodes in the given graph must be allocated to each agent in addition to finding an optimal sequence of the assigned targets for each agent to visit. A variant of mTSP that is also related to this work is the multiple-Steiner TSP[2] [30] where the agents are required to visit a subset of nodes in a graph. While focusing on allocating and computing the visiting order of targets for agents, mTSP methods [3, 17, 20, 24, 35] do not consider the collision avoidance constraints among the agents. In the MCPF problem investigated in this work, agent-agent conflicts are also considered.

Combined Target Assignment/Sequencing and Path Finding problems are investigated from different perspectives very recently [11, 14, 15, 18, 36]. However, these methods either consider target assignment only (without the need for computing visiting orders of targets) [11, 14, 18], or require computing the visiting order only as each agent is pre-allocated a set of targets [36]. In addition, the multi-agent pick-up and delivery problem [13, 16], which computes a set of conflict-free paths for agents that fulfill a set of pick-up and delivery tasks, also requires assigning a sequence of tasks for each agent. However, these approaches [13, 16] address this problem in two stages; in the first stage, a special TSP is solved to find a suitable sequence of tasks for each agent, and in the second stage, given the sequences of tasks, a conflict-free path is found for each agent. This work aims to simultaneously sequence the targets while finding conflict-free paths in order to compute solutions with optimality (or bounded sub-optimality) guarantees.

Other related problems that involve both target sequencing and path planning for multiple agents have also been investigated by the robotics community [12, 37]. However, these methods either over-simplify the collision avoidance, target allocation/sequencing requirements, or provide no theoretical guarantees on the solution quality. The developed CBSS framework in this work is able to compute solutions with optimality or bounded sub-optimality guarantees.

[2]The origin of Steiner problems are ascribed to mathematician Jakob Steiner [5] where agents are not required to visit each and every vertex in the given graph. The Steiner TSP has many variants which depend on whether an agent is required to return to its initial location or end its path at a pre-determined location; we will use Steiner TSP to refer to all these variants.

III. Problem Description

Let index set $I = \{1, 2, \dots, N\}$ denote a set of N agents. All agents move in a workspace represented as a graph $G = (V, E)$ where the vertex set V represents the possible locations for agents and the edge set $E = V \times V$ denotes the set of all the possible actions that can move an agent between any two vertices in V. An edge between $u, v \in V$ is denoted as $(u, v) \in E$ and the cost of an edge $e \in E$ is a positive real number $cost(e) \in (0, \infty)$.

In this article, we use superscript $i \in I$ over a variable to represent the specific agent to which the variable belongs (e.g. $v^i \in V$ means a vertex corresponding to agent i). Let $v^i_o \in V$ denote the *initial* vertex (also called the start) of agent i and V_o denote the set of all initial vertices of the agents. There are N *destination* vertices in G denoted by the set $V_d \subseteq V$. In addition, let $V_t \subseteq V \setminus \{V_o \bigcup V_d\}$ denote the set of M *target*[3] vertices that must be visited by at least one of the agents along its path. For each vertex $v \in V_t \bigcup V_d$, 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 that connects vertices v^i_1 and v^i_ℓ via a sequence of vertices $(v^i_1, v^i_2, \dots, v^i_\ell)$ in the graph G. Let $g^i(\pi^i(v^i_1, v^i_\ell))$ denote the cost associated with the path. This path cost is the sum of the costs of all the edges present in the path (i.e., $g^i(\pi^i(v^i_1, v^i_\ell)) = \Sigma_{j=1,2,\dots,\ell-1} cost(v^i_j, v^i_{j+1})$).

All agents share a global clock. Each action, either wait or move, 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* where two agents occupy the same vertex at the same time. The second case is an *edge conflict* where two agents go through the same edge from opposite directions between times t and $t + 1$ for some t.

The MCPF problem aims to find a set of conflict-free paths for the agents such that (1) each target $v \in V_t$ is visited 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 destination $u \in V_d$ such that $i \in f_A(u)$, and (3) the sum of the cost of the paths is a minimum.

Remark. MCPF generalizes several existing problems. When $M = 0$ (i.e., no target is present) and f_A maps each destination to a single distinct agent, MCPF reduces to the standard MAPF. When $f_A(v) = I, \forall v \in V_t \bigcup V_d$, we get the fully anonymous version of MCPF which has been solved by our prior work using MS* [26]. Finally, if the conflicts between the agents are ignored, and the destination of each agent is same as its starting location (which is often called a "depot" in TSP literature), then MCPF reduces to a variant of mTSP [22].

IV. Review of Conflict-Based Search

Conflict-Based Search (CBS) [31] is a two-level search algorithm that computes a collision-free optimal solution (joint path) to a MAPF problem. On the high level, every search node P is defined as a tuple of (π, g, Ω), where:

- $\pi = (\pi^1, \pi^2, \dots, \pi^N)$ is a joint path that connect starts and destinations of agents respectively.
- g is the scalar cost value of π (i.e., $g = g(\pi) = \Sigma_{i \in I} g^i(\pi^i)$).
- Ω is a set of (collision) constraints.[4] Each constraint is of form (i, v, t) (or i, e, t), which indicates agent i is forbidden from entering node v (or edge e) at time t.

CBS constructs a search tree $\mathcal{T}$ with the root node $P_{root} = (\pi_o, g(\pi_o), \emptyset)$, where the joint path π_o is constructed by running the low level (single-agent) planner, such as A*, for every agent respectively with an empty set of constraints while ignoring any other agents. P_{root} is added to OPEN, a queue that prioritizes nodes based on their g-values.

In each search iteration, a node $P = (\pi, g, \Omega)$ with the minimum g-value is popped from OPEN for expansion. To expand P, every pair of individual paths in π is checked for vertex conflict (i, j, v, t) (and edge conflict (i, j, e, t)). If no conflict is detected, π is conflict-free and is returned as an optimal solution. Otherwise, the detected conflict (i, j, v, t) is *split* into two constraints (i, v, t) and (j, v, t) respectively and two new constraint sets $\Omega \bigcup \{i, v, t\}$ and $\Omega \bigcup \{j, v, t\}$ are generated. (Edge conflict is handled in a similar manner and is thus omitted.) Then, for the agent i in each split constraint (i, v, t) and the corresponding newly generated constraint set $\Omega' = \Omega \bigcup \{i, v, t\}$, the low level planner is invoked to plan an individual optimal path π'^i of agent i subject to all constraints related to agent i in Ω'. The low level planner typically runs A*-like search in a time-augmented graph with constraints marked as obstacles. A new joint path π' is then formed by first copying π and then updating agent i's individual path π^i with π'^i. Finally, for each of the two split constraints, a corresponding high level node is generated and added to OPEN for future expansion. CBS [31] is guaranteed to compute an optimal solution for a given MAPF problem, if there exists a feasible solution.

V. Conflict-Based Steiner Search

A. Basic Concepts

In MCPF, a path for agent i may also visit a sequence of intermediate targets before reaching its destination. Let $\gamma^i = \{v^i_o, u^i_1, u^i_2, \dots, u^i_\ell, v^i_d\}$ denote the sequence of targets visited by agent $i \in I$ where v^i_o and v^i_d are initial and destination vertices of agent i, and u^i_j is the j^{th} intermediate target visited by agent i for $j = 1, \cdots, \ell$. Let $\gamma = \{\gamma^i : i \in I\}$ denote a *joint (target) sequence*, which is a collection of the individual target sequences of the agents. The cost incurred in traveling between any two adjacent nodes in γ^i is simply the minimum-cost path cost between the nodes in G. The cost of an individual sequence $cost(\gamma^i)$ is defined as the total cost incurred in traversing all the nodes in γ^i. Similarly, the cost of a joint sequence is defined as $cost(\gamma) := \Sigma_{i \in I} cost(\gamma^i)$. Note

[3]The term "targets" in this work represent static target locations, which are also called waypoints within the robotics community.

[4]For the rest of the work, we refer to collision constraints simply as constraints, which differs from the aforementioned assignment constraint.

that $cost(\gamma^i)$ is computed *ignoring all the conflicts* between the agents.

Following the same notations as in CBS, let $P = (\pi, g, \Omega)$ denote a high level search node. We say a path π^i *follows* γ^i if π^i visits all the assigned targets in the same order as specified in γ^i. Similarly, a joint path π *follows* γ if each $\pi^i \in \pi$ follows the corresponding $\gamma^i \in \gamma$.

Conflict-Based Steiner Search (CBSS) is a two-level search framework similar to CBS, which is conceptually visualized in Fig. 2. The key differences in the CBSS as compared to CBS are in the high level. Specifically, in the high level, CBSS constructs a search forest[5] (rather than a single search tree) where each tree $\mathcal{T}_j$ in the forest corresponds to a fixed joint sequence γ^*_j. In other words, the joint path π for any high level node $P = (\pi, g, \Omega)$ within the tree $\mathcal{T}_j$ follows the same joint sequence γ^*_j. The joint sequences $(\gamma^*_1, \gamma^*_2, \gamma^*_3 \cdots)$ are generated *ignoring the agent-agent conflicts* using a sequencing[6] procedure which ensures that the costs of the joint sequences are monotonically increasing: $cost(\gamma^*_1) \leq cost(\gamma^*_2) \leq cost(\gamma^*_3) \dots$. Given a joint sequence γ^*_j and its corresponding tree $\mathcal{T}_j$, conflicts are resolved in the high level nodes through a similar branching and low level search as in CBS.

Initially, CBSS starts with a joint sequence γ^*_1. A high level node corresponding to γ^*_1 is created first and forms the root node for $\mathcal{T}_1$. If γ^*_1 does not lead to any conflicts between the agents, then the search outputs the joint path corresponding to γ^*_1 as an optimal solution and terminates. On the other hand, if a conflict is present between any two agents following γ^*_1, then new high level search nodes are created as in CBS, and are added to OPEN. If the cost of the cheapest, high level node, say $P = (\pi, g, \Omega)$, in OPEN is less than the cost of γ^*_2, the search continues to check for conflicts in the P and expand P if necessary. On the contrary, if γ^*_2 is cheaper, a new tree denoted by $\mathcal{T}_2$ is created and a root node corresponding to γ^*_2 is added to OPEN, and the search continues. Since $cost(\gamma^*_1)$ is a lower bound to the optimal cost of MCPF (because γ^*_1 ignores the conflicts), and that the subsequent high level nodes are systematically generated using a best-first search procedure, the above process finds an optimal solution to the MCPF (a detailed proof is later presented in section VI).

There are some crucial parts to this high level search process. Generating a set of joint sequences with monotonically increasing costs is non-trivial, especially for a mTSP with agent-target assignment constraints. To address this, we leverage a partition procedure given in for TSPs (i.e., K-best partition [38]) which systematically forces a solution to include some edges and exclude another set of edges to generate the desired set of joint sequences. Given some edges to include and another set of edges to exclude, the mTSP

[5]CBSS is not the first method that extends CBS to a search forest. For example, CBS-TA [11] uses a search forest to combine task assignment and path finding. In MO-CBS [27], a search forest is constructed to find the Pareto-optimal front for multi-objective MAPF problems.

[6]In Fig. 2, this refers to the K-best Sequencing procedure. Here, K is a parameter that specifies the number of the cheapest K solutions to be found.

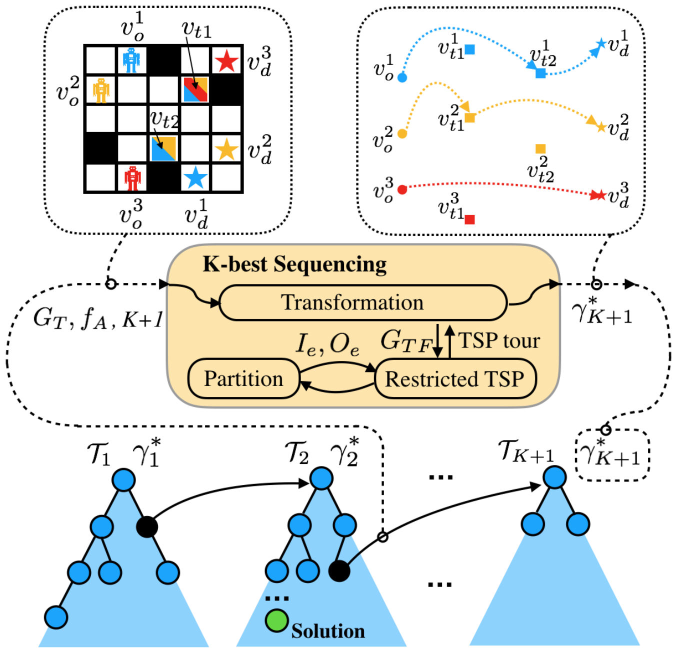

Fig. 2: A conceptual visualization of the CBSS framework.

with agent-target assignment constraints is solved leveraging a transformation method given in [20] that converts a mTSP to a single-agent TSP. This transformation method guarantees solution optimality while being able to leverage the state-of-the-art single-agent TSP solvers. As finding an optimal solution for MCPF can be time consuming, we also provide a way to find bounded sub-optimal solutions which can handle more agents and targets. Specifically, the user can specify an approximation parameter ϵ prior to solving the problem and the proposed approach will find a feasible solution (if one exists) whose cost is at most $(1+\epsilon)$ times the optimum. More details on the above methods are provided in the ensuing subsections.

B. *CBSS Overview*

Let $G_T = (V_T, E_T, C_T)$ denote a *target graph*, which is a complete undirected graph with vertex set $V_T = V_o \bigcup V_t \bigcup V_d$ ($|V_T| = 2N + M$) and edge set E_T. Here, C_T represents a symmetric cost matrix of size $(2N + M) \times (2N + M)$ that defines the cost of each edge in E_T. Each edge $(u, v) \in E_T$ represents a minimum-cost path connecting u, v in the (workspace) graph G ignoring agent-agent conflicts and the corresponding entry $C_T(u, v)$ in the cost matrix stores the cost value of that path. Using G_T and the assignment constraints, the K-best partition procedure (Sec. V-C) is used to compute K joint sequences with monotonically increasing costs.

An overview of the CBSS is shown in Alg. 1. To start with, CBSS finds an optimal joint sequence γ^*_1 (lines 1-2) using G_T. CBSS then invokes the low level planner (lines 3-4) for all agents $i \in I$ with an empty set of constraints, which computes π following γ^*_1, as well as the cost value g of π. Finally, the first root node $P_{root,1}$ is initialized and inserted into OPEN, a queue that prioritizes high level nodes based on their g-costs.

Algorithm 1 Pseudocode for CBSS

```
1: Compute $G_T = (V_T, E_T, C_T)$
2: $\gamma_1^* \leftarrow$ *K-best-Sequencing*($G_T$,$f_A$,$K = 1$)
3: $\Omega \leftarrow \emptyset$
4: $\pi, g \leftarrow$ *LowLevelPlan*($\gamma_1^*$, $\Omega$)
5: Add $P_{root,1} = (\pi, g, \Omega)$ into OPEN
6: **while** OPEN not empty **do**
7:     $P_l = (\pi_l, g_l, \Omega_l) \leftarrow$ OPEN.pop()
8:     $P_k = (\pi_k, g_k, \Omega_k) \leftarrow$ *CheckNewRoot*($P_l$, OPEN)
9:     *DetectConflict*($\pi_k$)
10:    **if** no conflict detected in $\pi_k$ **then**
11:        **return** $\pi_k$
12:    $\Omega \leftarrow$ Split the detected conflict
13:    **for all** $\omega^i \in \Omega$ **do**
14:        $\Omega'_k = \Omega_k \cup \{\omega^i\}$
15:        $P'_k \leftarrow$ *LowLevelPlan*($\gamma(P_k)$, $\Omega'_k$)
16:        // In this LowLevelPlan, only agent $i$'s path is planned.
17:        Add $P'_k$ to OPEN
18: **return** failure
```

In each iteration of the high level search (lines 6-17), a node P_l with the least g value is popped from OPEN. Before expanding node P_l, a procedure *CheckNewRoot* (see Sec. V-E, Alg. 3) is invoked, where the cost g_l of node P_l is compared against some threshold to decide whether the next best root node needs to be generated.

- If the next root does not need to be generated, the input node P_l is returned by *CheckNewRoot*.
- If the next root (denoted as the r-th root $P_{root,r}$) needs to be generated, a next best joint sequence γ_r^* is computed by using the K-best Sequencing procedure with $K = r$. The corresponding joint path as well as the path cost is computed by calling *LowLevelPlan*(γ_r^*, $\emptyset$), which runs the low level planner to plan an individual path for each agent by following $\gamma_r^{*,i} \in \gamma_r^*$. All these individual paths together form a joint path that follows γ_r^*. The resulting new root node $P_{root,r}$ is then returned by *CheckNewRoot* and P_l (i.e., the input node to *CheckNewRoot*) is inserted back into OPEN for future expansion.

The node returned by *CheckNewRoot* is denoted as P_k and the joint path in P_k is then checked for conflicts (line 9). If no conflict is detected, the algorithm terminates and an optimal solution (joint path) is returned. Otherwise, CBSS splits the detected conflict in the same way as CBS by generating two constraints. Here, we abuse the notation a bit to simplify our exposition: let $\gamma(P_k)$ (line 15) denote the joint sequence that π_k (the joint path in P_k) follows. This can be computed by first finding the root node $P_{root,r}$ of the tree to which P_k belongs, and then returning the joint sequence γ_r^* related to root $P_{root,r}$. For each newly generated constraint ω^i, CBSS updates the constraint set (line 14) and invokes the low level planner for agent i (line 15) to recompute its individual path that satisfies the new set of constraints. Finally, the newly generated high level nodes are inserted into OPEN for future expansion (line 17).

C. *K-best Joint Sequences*

To find the K cheapest solutions to the mTSP with assignment constraints, we build on a partition-based approach for the TSP in [38]. The main idea here is to transform the mTSP with assignment constraints into an equivalent TSP on an augmented graph and then apply the partition-based approach to find the K cheapest solutions. We will first discuss the partition-based approach for the TSP, and the transformation method will be presented in the next subsection.

Definition 1 (Restricted TSP (rTSP)): Given a graph $G' = (V', E', C')$, let $I_e, O_e \subseteq E'$ denote two disjoint subsets of edges in G', an rTSP requires computing an optimal tour τ^* such that $I_e \subseteq \tau^* \subseteq E' \backslash O_e$ and τ^* visits each node in G'.[7]

Intuitively, the rTSP is defined by two sets of edges I_e, O_e, and requires computing an optimal tour with all edges in I_e being included in the tour and the edges in O_e being excluded from the tour. In practice, the rTSP can be solved by modifying the cost of edges in I_e to a relatively small value while modifying the cost of edges in O_e to a large positive value, and then running a TSP solver in graph G' with the modified costs. Since the costs of the edges in I_e and O_e can be chosen suitably, an optimal tour can be readily found for the rTSP. We denote this solution process of the rTSP as $\tau^* \leftarrow$ *rTSP-Solve*(G', I_e, O_e).

The partition-based method [38] solves a K-best TSP via a best-first search by iteratively partitioning the set of feasible tours while finding the optimal tour in each partitioned subset (refer to Alg. 2). In Alg. 2, $I_e(k), O_e(k), \tau^*(k)$ keeps track of I_e, O_e and τ^* as a function of the iteration number (k) of the algorithm. At the start, $k = 1$, an optimal tour $\tau^*(1)$ in the graph G' with $I_e(1), O_e(1)$ is first computed (lines 1-4). The tuple $(I_e(1), O_e(1), \tau^*(1))$ with tour cost $cost(\tau^*(1))$ is then inserted into OPEN$_{rTSP}$ which denotes a queue of tuples that are prioritized based on their tour costs. Initially, the set of K-best solutions $\mathcal{S}$ is empty.

In the k-th while-iteration, a tuple $(I_e(k), O_e(k), \tau^*(k))$ is popped from OPEN$_{rTSP}$ (line 6). The tour $\tau^*(k)$ is a k-th best solution and is thus added into $\mathcal{S}$ (line 7). If k is equal to K, then $\mathcal{S}$ contains the K-best solutions and the algorithm terminates (lines 8-9). Otherwise, the algorithm continues by indexing the edges in the tour $\tau^*(k)$ from 1 to ℓ, where ℓ is the length of the tour. Then, the algorithm runs an inner for-loop over all the edges iteratively to partition the set of remaining feasible tours and generate a corresponding rTSP for each partition. Specifically, we use notation $[p], p \in \{1, 2, \ldots, \ell\}$ to denote the p-th for-loop iteration (line 11). For each edge e_p, the subset of edges $\{e_1, e_2, \ldots, e_{p-1}\}$ are then added to the

[7]We use G' to differentiate the graph for rTSP from the graph G that represents the workspace. Also note the difference between a set of edges I_e and the index set I that represents all agents. Additionally, there are two possible definitions of the rTSP problem: one requires finding a tour that visits each node in G' at least once (i.e., with repetition) and another that requires visiting each node exactly once (i.e., without repetition). We will show in the Appendix that these two versions are equivalent within the framework of CBSS. For the rest of the presentation, we focus on finding a tour that visits each node exactly once.

Algorithm 2 Pseudocode for K-best-TSP

```
1: $I_e(1), O_e(1) \leftarrow \emptyset$
2: $\tau^*(1) \leftarrow$ rTSP($G', I_e(1), O_e(1)$)
3: Add $(I_e(1), O_e(1), \tau^*(1))$ into OPEN$_{rTSP}$
4: $\mathcal{S} \leftarrow \emptyset$
5: while OPEN$_{rTSP}$ not empty do
6:     $(I_e(k), O_e(k), \tau^*(k)) \leftarrow$ OPEN$_{rTSP}$.pop()
7:     Add $\tau^*(k)$ into $\mathcal{S}$
8:     if $k = K$ then
9:         return $\mathcal{S}$
10:    Index the edges in $\tau^*(k)$ as $\{e_1, e_2, \dots, e_\ell\}$
11:    for all $p \in \{1, 2, \dots, \ell\}$ do
12:        $I_e(k+1)[p] \leftarrow I_e(k) \bigcup \{e_1, e_2, \dots, e_{p-1}\}$
13:        $O_e(k+1)[p] \leftarrow O_e(k) \bigcup \{e_p\}$
14:        $\tau^*(k+1)[p] \leftarrow$ rTSP-Solve($G', I_e(k+1)[p], O_e(k+1)[p]$)
15:        if $\tau^*(k+1)[p]$ is feasible then
16:            Add $(I_e(k+1)[p], O_e(k+1)[p], \tau^*(k+1)[p])$ into OPEN$_{rTSP}$
17: return failure
```

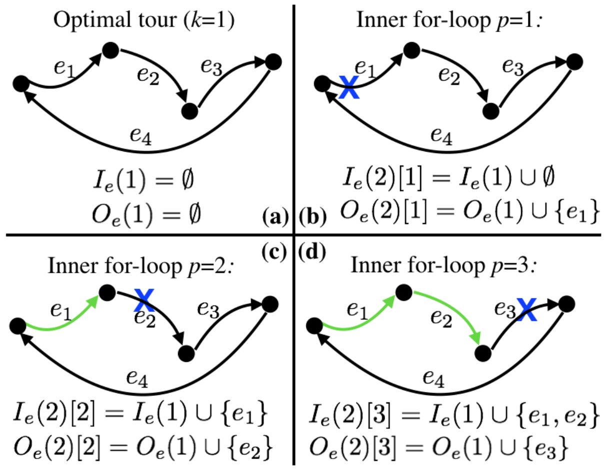
Optimal tour (k=1)
Inner for-loop p=1:
Inner for-loop p=2:
Inner for-loop p=3:
e1
e2
e3
e4
$I_e(1) = \emptyset$
$O_e(1) = \emptyset$
$I_e(2)[1] = I_e(1) \cup \emptyset$
$O_e(2)[1] = O_e(1) \cup \{e_1\}$
$I_e(2)[2] = I_e(1) \cup \{e_1\}$
$O_e(2)[2] = O_e(1) \cup \{e_2\}$
$I_e(2)[3] = I_e(1) \cup \{e_1, e_2\}$
$O_e(2)[3] = O_e(1) \cup \{e_3\}$
(a)
(b)
(c)
(d)

Fig. 3: An illustration of the running process of Alg. 2. (a) shows an optimal solution to a TSP, whose edges are indexed. To compute the second optimal solution ($K = 2$), (b), (c) and (d) show the the I_e and O_e sets of the inner for-loop iterations with $p = 1, 2, 3$ respectively. The iteration $p = 4$ is not shown.

set $I_e(k)$, to form a new set of edges (denoted as $I_e(k+1)[p]$) that must be included in the tour. Additionally, a new set of edges to be excluded (denoted as $O_e(k+1)[p]$) is formed by taking the union of $\{e_p\}$ and $O_e(k)$ (line 13). An illustration can be found in Fig. 3. Alg. 2 then solves the corresponding rTSP defined by $I_e(k+1)[p]$ and $O_e(k+1)[p]$, verifies the feasibility[8] of the computed tour and adds the resulting tuple into OPEN$_{rTSP}$ if the tour is feasible (lines 14-16). Alg. 2 has the following property.

Theorem 1: Given a graph G', Alg. 2 is guaranteed to compute a set of K-best tours, if there exists one.

The correctness of this theorem relies on that (i) the partition is complete, and (ii) the search runs in a best-first manner. We refer the reader to [8, 38] for more details.

Remark. Solving a K-best TSP is expensive. In each iteration of the while loop (lines 5-16), the algorithm needs to solve ℓ rTSPs, where ℓ is the length of the tour. To find K-best tours, the algorithm requires solving $1 + (K-1)\ell$ rTSPs. For implementation details, a couple of techniques can be used to improve the runtime efficiency, which is discussed in the Appendix.

D. Transformation Method for Solving mTSP

In this subsection, we present our approach which takes the target graph G_T and assignment constraints f_A as input, and returns an optimal joint sequence for the mTSP.[9] Our approach is based on the transformation used in [20]. We note that the transformation method described in our work differs from the one in [20] in the following aspects: (1) Destinations are explicitly introduced and they can be different from agents' start locations in this work; (2) Agent-target assignment constraints are introduced in this work; (3) The concept about the "heterogeneous costs" in [20] is not relevant in this work and thus removed. The main steps in this transformation method are the following:

- Step-1 converts the target graph G_T into a *transformed graph* G_{TF} (subscript TF stands for "transformation") based on the assignment constraints f_A, and defines the edge costs in G_{TF} with a set of rules which are elaborated in the ensuing paragraphs;
- Step-2 invokes an off the shelf TSP solver to compute an optimal (single-agent) *tour* in G_{TF} (i.e., a path with both ends being the same vertex while visiting all the vertices in G_{TF} exactly once);
- Step-3 divides the (single-agent) tour into N segments by removing some special edges (that are defined in Step-1) in the tour where each segment corresponds to a target sequence for an agent.

We now elaborate these three steps and provide a toy example in Fig. 4 to illustrate. **Step-1** generates $G_{TF} = (V_{TF}, E_{TF}, C_{TF})$ based on G_T and f_A. The vertex set is defined as $V_{TF} := V_o \bigcup U$, where U is an "augmented" set of targets and destinations: for each target or destination $v \in V_t \bigcup V_d$, make a copy v^i of v for agent i if $i \in f_A(v)$. The purpose of this augmentation is to suitably represent assignment constraints. Let U^i denote the set of all copies of targets and destinations that agent i is eligible to visit. Clearly, $U = \bigcup_{i \in I} U^i$. Additionally, let $U(v), v \in V_t \bigcup V_d$ denote an ordered list of all copies v^i of vertex v where the order is specified by sorting i from the smallest to the largest. Based on this order in $U(v)$, let $Next(v^i), v^i \in U(v)$ denote the next copy of v in $U(v)$ and let $Prev(v^i) \in U(v)$ denote the previous copy of v in $U(v)$. As an edge case, when v^i

[8]Feasibility here means that the solution tour includes all edges in I_e and excludes all edges in O_e. As an implementation detail, the feasibility check is helpful in practice as we set the costs of edges in O_e to a large number, which means the edges in O_e may still be used in the solution tour.

[9]Using the terms from the TSP community, the sequencing problem involved in MCPF is a Multi-Depot Multi-Terminal Hamiltonian Path Problem. To simplify presentation without causing confusion, we simply refer to it as a mTSP or a sequencing problem.

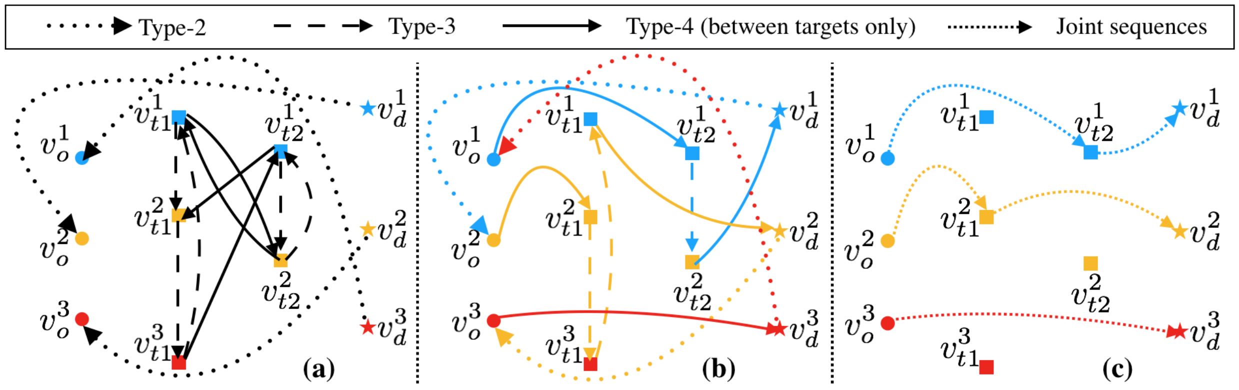

Fig. 4: A visualization of the target sequencing procedure for the toy example in Fig. 1. (a) Transformed graph G_{TF} with edges from the starts and edges connected to destinations omitted to make the plot readable. (b) The solution TSP tour $\{v_o^1, v_{t2}^1, v_{t2}^2, v_d^1, v_o^2, v_{t1}^2, v_{t1}^3, v_{t1}^1, v_d^2, v_o^3, v_d^3, v_o^1\}$ in graph G_{TF}. (c) The joint sequence $\gamma = \{\gamma^i, i = 1, 2, 3\}$ returned by the sequencing procedure, where $\gamma^1 : \{v_o^1, v_{t1}^1, v_{t2}^1, v_d^1\}$, $\gamma^2 : \{v_o^2, v_{t1}^2, v_d^2\}$, $\gamma^3 : \{v_o^3, v_d^3\}$.

is the last one in $U(v)$, let $Next(v^i)$ indicate the first copy in $U(v)$; And when v^i is the first one in $U(v)$, let $Prev(v^i)$ indicate the last copy in $U(v)$. As an example, in Fig. 4 (a), set $U = \{v_{t1}^1, v_{t1}^2, v_{t1}^3, v_{t2}^1, v_{t2}^2, v_d^1, v_d^2, v_d^3\}$ and $U^2 = \{v_{t1}^2, v_{t2}^2\}$. For target v_{t1}, $U(v_{t1}) = \{v_{t1}^1, v_{t1}^2, v_{t1}^3\}$ (an ordered list) with $Next(v_{t1}^3) = v_{t1}^1$ and $Prev(v_{t1}^1) = v_{t1}^3$.

The edge set E_{TF} consists of several different types of *directed* edges. The cost value of these edges are stored in the corresponding cost matrix C_{TF}.

- **Type-1 Edges:** the start v_o^i of agent i is connected to any other node $u^i \in U^i$ with a cost value equal to the minimum path cost in the workspace graph G, which is the same value as $C_T(v_o^i, u^i)$.
- **Type-2 Edges:** the copy of each destination of agent i is connected to the start of agent $(i+1)$ for $i = 1, 2, \dots, (N-1)$ with zero-cost edges, and the copy of each destination of agent $i = N$ is connected to v_o^1, the starting node of agent $i = 1$, with zero-cost edges. The intuition behind these zero-cost edges is to make sure the optimal tour of G_{TF} uses these zero-cost edges to connect two subsequent agents' destinations and starts, and the tour can therefore be readily divided into N individual segments by removing these zero-cost edges.
- **Type-3 Edges:** for each $v \in V_t \bigcup V_d$, an edge is connected from $v^i \in U(v)$ to $Next(v^i)$ with cost Z, a large constant number that is an over-estimate of the cost of the optimal joint sequence. For example, $Z = 2(N+M) \max_{(u,v) \in E_T} C_T(u, v)$.
- **Type-4 Edges:** for each $v \in V_t \bigcup V_d$ and for each agent $i \in f_A(v)$, connect an edge from $Prev(v^i)$ to agent i's copy u^i of all other targets and destinations, (i.e., $\forall u^i \in U^i, u^i \neq v^i$), with cost $Z + C_T(v^i, u^i)$. In Fig. 4 (a), take v_{t2}^1 as an example: $i = 1$, $Prev(v_{t2}^1) = v_{t2}^2$, and "agent-$i$'s copies of all other targets and destinations" are $\{v_{t1}^1, v_d^1\}$; Thus, v_{t2}^2 is connected with each of $\{v_{t1}^1, v_d^1\}$. Note that in Fig. 4 (a), all edges connected to destinations are omitted to make the figure readable.

Type-3 and Type-4 edges are related to Noon-Bean transformation [19], which solves the so-called one-in-a-set TSPs. The intuition behind the Type-3 and Type-4 edges as well as their cost values is to ensure that when an optimal tour in G_{TF} visits a copy v^i of a target or destination $v \in V_t \bigcup V_d$, the tour must visit all other copies $v^j, j \neq i$ before arriving at the copy u^i of a next vertex $u \in V_t \bigcup V_d, u \neq v$. By doing so, the copies of the same target can later be removed from the tour to extract individual sequences (as shown in Fig. 4 (b) and (c)).

So far, we have finished the presentation about Step-1, which generates a transformed graph G_{TF}. In **Step-2**, a regular (single-agent) TSP solver is invoked on G_{TF} and a minimum cost tour is computed. As shown in Fig. 4 (b), an optimal tour is computed and visualized for the toy example in Fig. 1. In **Step-3**, the computed optimal tour is post-processed to obtain the corresponding joint sequence. First, all zero-cost edges from destinations to starts are removed, which breaks the tour into N segments. (In Fig. 4 (b), the dotted lines, which denote the zero-cost edges, are removed.) Second, the copies of the same targets are shortcut. (In Fig. 4 (b), the dashed lines in yellow and blue are shortcut.) Third, the cost of the resulting joint sequence can be obtained by sum up the cost of edges present in the joint sequence with respect to the corresponding values in cost matrix C_T of the target graph G_T. The resulting joint sequence for the toy example is shown in Fig. 4 (c). The property of the sequencing procedure is summarized with the following theorem.

Theorem 2: Given G_T and f_A, the target sequencing procedure computes an optimal (i.e., minimum-cost) joint sequence that visits all targets and ends at destinations while satisfying all assignment constraints.

We refer the reader to [20, 21] for a detailed proof, which can be readily adapted to the transformation in this work.

Remark. As a special case of MCPF, if all targets and destinations are anonymous (i.e., $f_A(v) = I, \forall v \in V_t \bigcup V_d$), then this fully anonymous MCPF problem is called a MSMP problem in [26]. For a MSMP problem, the sequencing procedure can be simplified: There is no need to make copies of targets and destinations for each eligible agent and the edge of the third type is unnecessary. We refer the reader to [20, 26] for a detailed description about the approach for MSMP.

E. Generation of New Roots

To compute an optimal solution, CBSS determines whether a new root needs to be generated in the *CheckNewRoot* procedure (line 8 in Alg. 1) during the search. Let r denote the number of roots that have been generated during the search. Note that each root corresponds to a joint sequence and all joint sequences are generated with monotonically increasing costs. Thus γ_r^* is a joint sequence with the largest cost value $cost(\gamma_r^*)$ among all roots that have been generated. Let $\epsilon \in [0, \infty]$ denote a sub-optimality bound, a hyper-parameter of CBSS: When $\epsilon = 0$, CBSS is required to compute an optimal solution; When $\epsilon = \infty$, there is no optimality bound required for the computed solution.

Algorithm 3 Pseudocode for CheckNewRoot

```
1:  Input: $P_l = (\pi_l, g_l, \Omega_l)$, OPEN
2:  $r \leftarrow$ number of roots generated so far.
3:  if $g_l \leq (1+\epsilon)cost(\gamma_r^*)$ then
4:      return $P_l$          ▷ Defer the generation of the next root
5:  $\gamma_{r+1}^* \leftarrow$ K-best-Sequencing($G_T, f_A, K = r+1$)
6:  $\pi, g \leftarrow$ LowLevelPlan($\gamma_{r+1}^*, \emptyset$)
7:  $P_{root,r+1} = (\pi, g, \emptyset)$
8:  if $g_l \leq (1+\epsilon)cost(\gamma_{r+1}^*)$ then
9:      Add $P_{root,r+1}$ into OPEN
10:     return $P_l$          ▷ Defer the expansion of the next root
11: Add $P_l$ into OPEN
12: return $P_{root,r+1}$
```

As shown in Alg. 3, *CheckNewRoot* first checks if the cost of the input node exceeds $(1+\epsilon)cost(\gamma_r^*)$. If not, P_l is returned (for expansion) as the cost of P_l is still within the sub-optimality bound. Otherwise, *CheckNewRoot* generates a next best joint sequence γ_{r+1}^* via procedure *K-best-Sequencing* (line 5).

With γ_{r+1}^*, *LowLevelPlan* computes a joint path π as well as its cost g following γ_{r+1}^*, and the next root node $P_{root,r+1}$ is generated. Finally, line 8 checks if the cost of the input node P_l exceeds $(1+\epsilon)cost(\gamma_{r+1}^*)$. If not, P_l is returned for expansion as its cost is still within the sub-optimality bound. Otherwise, P_l is inserted back to OPEN for future expansion and the newly generated root $P_{root,r+1}$ is returned.

As shown in Alg. 1, CBSS invokes *CheckNewRoot* right before the expansion of a node to defer the expensive computation of a next best joint sequence until needed. CBSS intentionally defers this computation by first comparing the cost of the node to be expanded against the sub-optimality bound and then computing a next best joint sequence until absolutely necessary. Finally, it's worthwhile to note that, when $\epsilon = \infty$, CBSS becomes a "sequential" method in a sense that the optimal joint sequence γ_1^* is computed at first and then CBSS plans a joint path following γ_1^* without generating a second best joint sequence (since $\epsilon = \infty$). A collision-free path may still be found but no optimality guarantee can be provided (i.e., a bound of ∞).

VI. Analysis

This section provides sketch proofs to show that CBSS is guaranteed to find a solution if one exists (Theorem 3) and the returned solution is guaranteed to be an ϵ-bounded sub-optimal solution (Theorem 4). A solution (joint path) is *feasible* if it (i) visits all the targets and ends at destinations while satisfying all the assignment constraints, and (ii) is conflict-free. A MCPF problem instance is feasible if there exists a feasible solution.

Theorem 3: For a feasible MCPF problem instance, CBSS returns a feasible solution.

Proof: By Theorem 2 and the construction of the algorithm (line 10 in Alg. 1), when CBSS terminates, the returned solution is guaranteed to be feasible. During the search, CBSS either generates a new root with monotonically increasing costs (by Theorem 1), or expands a node within a certain tree. There is a finite number of possible roots to be generated. Additionally, within each tree, CBSS expands nodes with non-decreasing costs (i.e., in a best-first manner), and there are only a finite number of possible nodes with costs no larger than a certain cost value [31]. Therefore, if there is a feasible solution, CBSS terminates in finite time and finds a feasible solution. ■

Theorem 4: Let g^* denote the cost value of an optimal solution for a MCPF problem instance. When $\epsilon < \infty$, CBSS is guaranteed to return a solution π with cost value g that is no larger than $(1+\epsilon)g^*$.

Proof: Let $P^* = (\pi^*, g^*, \Omega^*), P = (\pi, g, \Omega)$ denote the search node corresponding to g^* (the optimal solution cost) and g (the cost of the solution returned by CBSS) respectively. Also, let $P_{root,r'} = (\pi', g', \Omega')$ denote the root node of the tree that contains P^*. When P is expanded and solution π is returned, the root node $P_{root,r'}$ is either (1) generated (i.e., $r' \leq r$) or (2) not generated (i.e., $r' > r$).

For case (1), CBSS searches in a best-first manner, which guarantees that P is the first conflict-free node being expanded and g is the minimum cost among all un-expanded nodes in OPEN. Node P^* must be an un-expanded node because otherwise CBSS should have expanded it. Thus, $g \leq g^* \leq (1+\epsilon)g^*$.

For case (2), by Alg. 3, any nodes that are expanded by CBSS cannot have a cost value that is greater than $(1+\epsilon)g'$, because otherwise root $P_{root,r'}$ would have been generated. Therefore, $g^* \geq g' \geq \frac{g}{(1+\epsilon)}$. The first inequality holds because node $P_{root,r'}$ has an empty constraint set Ω' while node P^* has a constraint set Ω^* that is a super set of Ω'. In summary, for either case, we have $g \leq (1+\epsilon)g^*$. ■

VII. Results

A. *Implementation, Baselines and Test Settings*

We implement CBSS framework in Python. We use LKH-2.0.9[10] [10] as the single-agent TSP solver required by the transformation method (Sec. V-D). We implement the low level planner in CBSS by using SIPP [23] to search a time-augmented (workspace) graph $G \times \{0, 1, 2, \dots, T\}$ subject to vertex and edge constraints. It has been shown that SIPP runs faster than A* as the low level planner for CBS-like algorithms [28]. We use different grid maps from a online data set [34] and make each of them four-connected with unit-cost edges. All tests are run on a computer with an Intel Core i7-11800H CPU and 16GB RAM. Each test instance has a time limit of **one** minute. For subsequent sections, let N denote the number of agents and M denote the number of targets. The number of destinations are not included in M.

We select three baselines for comparison. The **first** one is using A* to search the joint configuration space of agents, where each search state encodes both the location of agents as well as the visiting status of the targets. This A* method is guaranteed to find an optimal solution. The **second** baseline is MS* [26], which is a state-of-the-art multi-agent planner leveraging subdimensional expansion [39] to solve the fully anonymous version of MCPF problems, (i.e., each target or destination can be assigned to any agent). The **third** baseline is a greedy method. It begins by assigning targets and destinations in a greedy manner, which is elaborated later. Then, LKH is invoked for each agent to compute the visiting order of the assigned targets. The computed joint sequence is then used within the CBSS framework, and ϵ is set to ∞. As a result, this greedy baseline planner runs in a sequential manner by first computing a (greedy) joint sequence γ and then planning conflict-free paths following γ. This greedy method can handle arbitrary forms of assignment constraints.

The aforementioned greedy assignment procedure consists of the following steps: (i) In each iteration, the procedure loops over all unassigned targets v and the corresponding eligible agents $i \in f_A(v)$ to find a pair (v, i) that has the minimum path cost between the agent-i's "current location" (which is initialized as v_o^i for each agent i) and the target v; (ii) The procedure assigns target v to agent i and updates agent-i's current location to v; (iii) The procedure repeats (i) and (ii) until all targets are assigned, and then runs the same greedy assignment process for all destinations while ensuring that each agent is assigned to a unique destination.

B. *CBSS vs MS**

We begin our tests with fully anonymous MCPF problems. We compare CBSS ($\epsilon = 0$) against both MS* [26] and A* for

[10]LKH is a popular heuristic algorithm for the TSP, which has been shown to be able to find an optimal solution for numerous TSP instances. More details about the LKH solver can be found in http://akira.ruc.dk/~keld/research/LKH. This work uses LKH as the TSP solver due to its computational efficiency. Other TSP solvers can also be used. Note that, since the solution tour returned by LKH is not guaranteed to optimal, the resulting implementation of CBSS (with $\epsilon = 0$) is not guaranteed to return an optimal solution joint path.

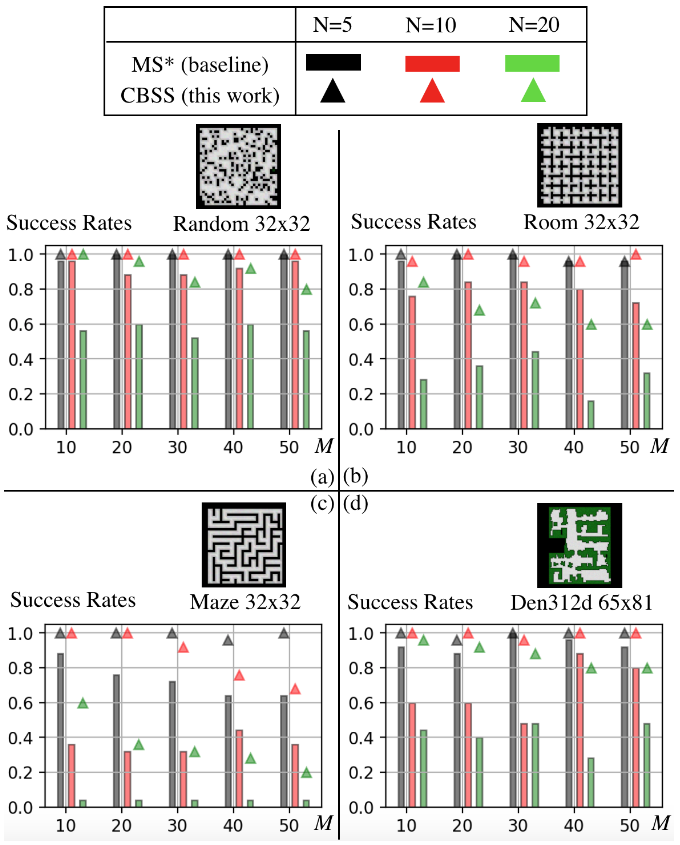

Fig. 5: Numerical results of CBSS (this work) and MS* (baseline). Color indicates different number of agents N and the x-axis represents the number of targets M. CBSS (the triangles markers) achieves up to 60% higher success rates than MS* (the bars) within the one minute runtime limit.

$N \in \{5, 10, 20\}$ and $M \in \{10, 20, 30, 40, 50\}$. We report the success rates within the runtime limit.

The A* method can not solve any instances with $N = 5$ within the time limit due to the exponential growth of the joint configuration space with respect to the number of agents, and is thus omitted from the figure. For CBSS and MS*, as shown in Fig. 5, CBSS achieves higher success rates than MS* in all settings, and doubles the success rates in some settings. For example, in Fig. 5 (c), when $N = 10$ (the red markers) and $M = 20$, the success rate of MS* is less than 40% while the rate of CBSS is 100%.

C. *CBSS with Different Sub-optimality Bounds*

This section investigates the effect of ϵ on CBSS. We use the most challenging setting from the previous test (i.e., the maze map with $N = 20$) and vary the ϵ among $\{0, 0.01, 0.1\}$. All test instances are fully anonymous, same as the instances in the previous section. All statistics in Fig. 6 are taken over all instances (both solved and unsolved within the time limit).

1) Success Rates: As shown in Fig. 6 (b), increasing ϵ from 0 to 0.01 can significantly increase the success rate. One reason behind is that a small ϵ can help with tie-breaking:

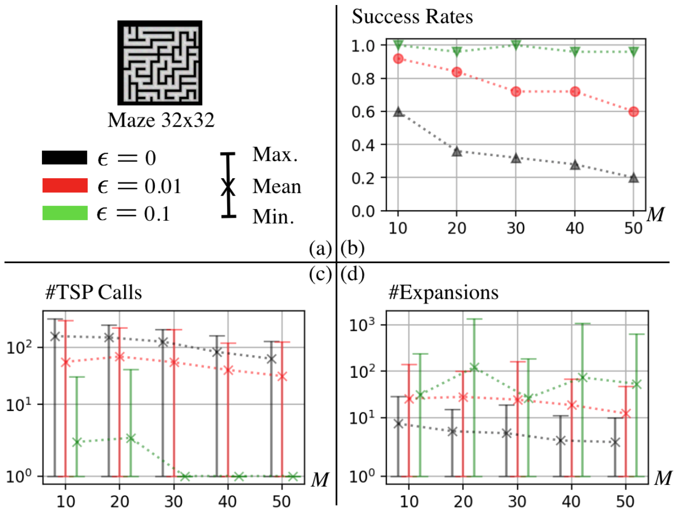

Fig. 6: Numerical results of CBSS with different sub-optimality bounds ϵ. (a) shows the success rates. (b) shows the number of TSP solver calls. (c) shows the number of high-level nodes expanded by CBSS. There is a trade-off between solution optimality bound and runtime efficiency: larger ϵ tends to defer the generation of the next best joint sequence and can lead to higher success rates within the time limit.

for problem instances with many equal-cost joint sequences, setting ϵ to a smaller number (such as 0.01) can defer the generation of those equal-cost joint sequences with tiny loss in the optimality bound (1% loss). Similarly, increasing ϵ from 0.01 to 0.1 further improves the success rates.

2) Number of Calls on the TSP Solver: As shown in Fig. 6 (c), increasing ϵ leads to fewer TSP solver calls. This is expected as a larger ϵ can defer the generation of the next best joint sequence (Sec. V-E). For example, when $\epsilon = 0.1$ and $M = 30, 40, 50$, only one joint sequence is generated (Fig. 6 (c)).

3) Number of High Level Nodes Expanded: As shown in Fig. 6 (d), increasing ϵ leads to more high level nodes expansion in general. Combined with Fig. 6 (b) and (c), it indicates that, with a large ϵ, CBSS tries to find conflict-free paths by following the joint sequences that have been generated and defers the (expensive) computation of the next best joint sequence, which consequently leads to higher success rates in general. However, note that using a larger ϵ to defer the generation of the next best joint sequence can not theoretically guarantee a higher success rate, since it's possible that the generated joint sequences lead to many conflicts between agents and result in a large runtime. We will revisit this in the next subsection.

To summarize, there is a trade-off between solution optimality bound and runtime efficiency, and a larger ϵ often leads to higher success rates empirically.

D. CBSS for MCPF with Various Assignment Constraints

In the previous sections, CBSS is evaluated with fully anonymous MCPF problems. This section evaluates CBSS with various types of assignment constraints. Since the problem formulation of MCPF is very general, it's impossible to evaluate all different forms of assignment constraints within this paper. We select the following cases to investigate.

- (Case-1) Each destination is assigned to a unique agent while all targets are anonymous. This type is a strict generalization of MAPF.
- (Case-2) Each agent has a pre-assigned target while the remaining targets and all destinations are anonymous.
- (Case-3) A combination of Type-1 and Type-2: every destination is assigned to a unique agent, and each agent has a pre-assigned targets while the remaining targets are anonymous.

All tests are run in the aforementioned random map with $N = 10$, $\epsilon = 0.01$. We compare CBSS and the aforementioned greedy baseline (which can also handle arbitrary forms of assignment constraints). We report in Fig. 7 both the success rate and the cost ratio, which is defined as

$$\text{cost ratio} = \frac{cost(\pi_{Greedy}) - cost(\pi_{CBSS})}{cost(\pi_{CBSS})}, \quad (1)$$

where $cost(\pi_{Greedy})$ is the solution cost computed by the greedy method and $cost(\pi_{CBSS})$ is the solution cost computed by CBSS. The statistics of the cost ratio are computed over all the instances that are solved by both methods within the runtime limit.

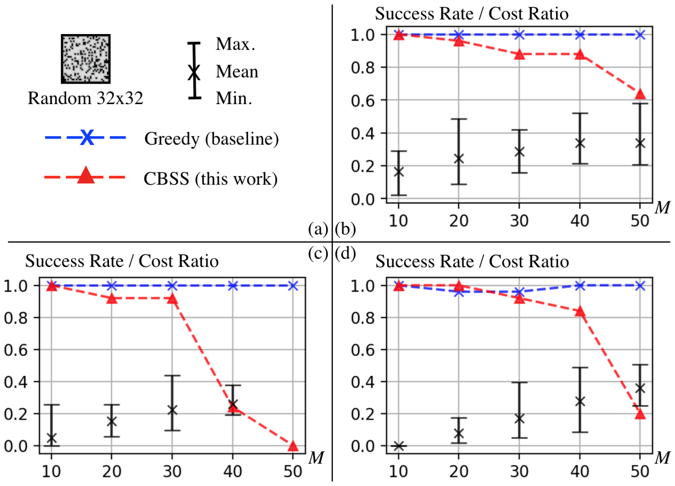

Fig. 7: Numerical results of CBSS and the greedy baseline. (b) (c) and (d) correspond to Case-1, Case-2, Case-3 of assignment constraints respectively as explained in the text. The greedy method achieves higher success rates than CBSS in general but suffers from higher solution costs.

1) Success Rates: The greedy methods achieves higher success rates than CBSS in general, especially when M is large (e.g. in Fig. 7 (c) and (d)). This is expected since a larger M leads to TSPs with more nodes, which is in general computationally more expensive to solve. We observe from the data that, in Fig. 7 (b) with $M = 50$, CBSS cannot finish computing the first optimal joint sequence within the time limit, which demonstrates the computational burden of sequencing the targets. In contrast, the greedy method can sequence targets quickly and can achieve higher success rates.

There are also cases where CBSS achieves higher success rates than the greedy method (e.g. in Fig. 7 (d) when $M = 20$). The reason is that the greedy method computes only one joint sequence and then run CBS-like search by following this (fixed) joint sequence. If this joint sequence leads to many conflicts between agent, the greedy method has to resolve all of them to return a solution. In CBSS, the next best joint sequence is generated when needed, which can help bypass the large number of conflicts caused by following only one joint sequence.

2) Solution Cost: The black error bars in Fig. 7 show the distribution of the cost ratios. Although the greedy baseline runs fast, it can lead to solutions that are up to 50% more expensive than the solutions computed by CBSS. It shows the trade-off between solution quality and runtime: solving computationally expensive TSPs in CBSS can lead to solutions with cheaper cost.

E. Physical Robot Tests

To demonstrate the applicability of CBSS on physical robots, we run CBSS using Robotarium [40], a remotely accessible multi-robot testbed, with four mobile robots and 12 target locations with assignment constraints.[11]

VIII. Conclusion

This paper formulates a problem called Multi-Agent Combinatorial Path Finding (MCPF), which requires both planning collision-free paths for multiple agents as MAPF does as well as sequencing targets for agents as mTSP requires. The work develops a framework called Conflict-Based Steiner Search (CBSS) to solve MCPF with optimality guarantees. CBSS is a general and flexible framework in the following sense. First, by varying the sub-optimality bound $\epsilon \in [0, \infty]$, CBSS moves along a spectrum from computing optimal solutions with high computational burden, to computing ϵ-bounded sub-optimal solutions within a smaller amount of time, to computing unbounded sub-optimal solutions efficiently as a naive sequential approach does. Second, different sequencing procedures (e.g. greedy sequencing, K-best sequencing) can be used within the CBSS framework, trading off solution quality for runtime efficiency. This work also provides extensive numerical results to corroborate the performance of CBSS in various settings with different numbers of agents and targets.

For future work, one can extend CBSS to address the uncertainty in the robot motion or the perturbation in the environment, which are common in some applications but not considered in the current work. In addition, one can also expedite CBSS by leveraging approximation or heuristic target sequencing methods, or by incorporating CBS-related techniques to improve the multi-agent path planning process.

[11]https://youtu.be/xwLoCiJ2vJY

Acknowledgment

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

References

[1] Anton Andreychuk, Konstantin Yakovlev, Dor Atzmon, and Roni Stern. Multi-agent pathfinding with continuous time. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 39–45, 2019.

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

[3] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *omega*, 34(3):209–219, 2006.

[4] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Eyal Shimony. Icbs: improved conflict-based search algorithm for multi-agent pathfinding. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[5] Marcus Brazil, Ronald L. Graham, Doreen A. Thomas, and Martin Zachariasen. On the history of the euclidean steiner tree problem. *Archive for History of Exact Sciences*, 68(3):327–354, 2014.

[6] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[7] Liron Cohen, Tansel Uras, TK Satish Kumar, and Sven Koenig. Optimal and bounded-suboptimal multi-agent motion planning. In *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[8] Horst W Hamacher and Maurice Queyranne. K best solutions to combinatorial optimization problems. *Annals of Operations Research*, 4(1):123–143, 1985.

[9] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[10] Keld Helsgaun. General k-opt submoves for the lin–kernighan tsp heuristic. *Mathematical Programming Computation*, 1(2):119–163, 2009.

[11] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph Durham, and Nora Ayanian. Conflict-based search with optimal task assignment. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 2018.

[12] James Keller, Dinesh Thakur, Maxim Likhachev, Jean Gallier, and Vijay Kumar. Coordinated path planning for fixed-wing uas conducting persistent surveillance missions. *IEEE Transactions on Automation Science and Engineering*, 14(1):17–24, 2016.

[13] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. Task and path planning for multi-agent pickup and delivery. In *2019 AAMAS*, pages 1152–1160, 2019.

[14] Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 1144–1152, 2016.

[15] Hang Ma, Sven Koenig, Nora Ayanian, Liron Cohen, Wolfgang Hönig, TK Kumar, Tansel Uras, Hong Xu, Craig Tovey, and Guni Sharon. Overview: Generalizations of multi-agent path finding to real-world scenarios. *arXiv preprint arXiv:1702.05515*, 2017.

[16] Hang Ma, Jiaoyang Li, T K Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Conference on Autonomous Agents & Multiagent Systems*, 2017.

[17] W. Malik, S. Rathinam, and S. Darbha. An approximation algorithm for a symmetric generalized multiple depot, multiple travelling salesman problem. *Operations Research Letters*, 35:747–753, 2007.

[18] Van Nguyen, Philipp Obermeier, Tran Cao Son, Torsten Schaub, and William Yeoh. Generalized target assignment and path finding using answer set programming. In *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[19] Charles E Noon and James C Bean. An efficient transformation of the generalized traveling salesman problem. *INFOR: Information Systems and Operational Research*, 31(1):39–44, 1993.

[20] P. Oberlin, S. Rathinam, and S. Darbha. Today's traveling salesman problem. *IEEE Robotics and Automation Magazine*, 17(4):70–77, December 2010.

[21] Paul Oberlin, Sivakumar Rathinam, and Swaroop Darbha. A transformation for a heterogeneous, multiple depot, multiple traveling salesman problem. In *2009 American Control Conference*, pages 1292–1297. IEEE, 2009.

[22] Paul Oberlin, Sivakumar Rathinam, and Swaroop Darbha. Today's traveling salesman problem. *IEEE robotics & automation magazine*, 17(4):70–77, 2010.

[23] Mike Phillips and Maxim Likhachev. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*, pages 5628–5635. IEEE, 2011.

[24] S. Rathinam, R. Ravi, J. Bae, and K. Sundar. Primal-Dual 2-Approximation Algorithm for the Monotonic Multiple Depot Heterogeneous Traveling Salesman Problem. volume 162 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 2020.

[25] Zhongqiang Ren, Sivakumar Rathinam, and Howie 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.

[26] Zhongqiang Ren, Sivakumar Rathinam, and Howie 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.

[27] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. Multi-objective conflict-based search for multi-agent path finding. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021.

[28] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. Multi-objective conflict-based search using safe-interval path planning. *arXiv preprint arXiv:2108.00745*, 2021.

[29] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. Subdimensional expansion for multi-objective multi-agent path finding. *IEEE Robotics and Automation Letters*, 6(4):7153–7160, 2021.

[30] Jessica RodrÃguez-Pereira, Elena FernÃ¡ndez, Gilbert Laporte, Enrique Benavent, and Antonio MartÃnez-Sykora. The steiner traveling salesman problem and its extensions. *European Journal of Operational Research*, 278(2):615 – 628, 2019.

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

[32] David Silver. Cooperative pathfinding. pages 117–122, 01 2005.

[33] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[34] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. Multi-agent pathfinding: Definitions, variants, and benchmarks. *arXiv preprint arXiv:1906.08291*, 2019.

[35] Kaarthik Sundar and Sivakumar Rathinam. Generalized multiple depot traveling salesmen problem-polyhedral study and exact algorithm. *Computers & Operations Research*, 70:39–55, 2016.

[36] Pavel Surynek. Multi-goal multi-agent path finding via decoupled and integrated goal vertex ordering. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, pages 197–199, 2021.

[37] Matthew Turpin, Nathan Michael, and Vijay Kumar. Capt: Concurrent assignment and planning of trajectories for multiple robots. *The International Journal of Robotics Research*, 33(1):98–112, 2014.

[38] Edo S Van der Poort, Marek Libura, Gerard Sierksma, and Jack AA van der Veen. Solving the k-best traveling salesman problem. *Computers & operations research*, 26(4):409–425, 1999.

[39] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24, 2015.

[40] Sean Wilson, Paul Glotfelter, Li Wang, Siddharth Mayya, Gennaro Notomista, Mark Mote, and Magnus Egerstedt. The robotarium: Globally impactful opportunities, challenges, and lessons learned in remote-access, distributed

control of multirobot systems. *IEEE Control Systems Magazine*, 40(1):26–44, 2020.

[41] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1):9–9, 2008.

[42] Jingjin Yu and Steven M LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

Appendix

A. Improved K-best partition for CBSS

A few techniques can be introduced to improve the computational efficiency of Alg. 2 within the CBSS framework. First, the transformed graph G_{TF} (which is the graph G' in Alg. 2) contains special types of edges (the second and the third type of edges as defined in Sec. V-D), which are auxiliary edges that help with the transformation. These edges should be skipped during the partition for-loop (lines 11-16 in Alg. 2). For example in Fig. 4, only the solid edges in Fig 4 (b) (i.e., $\{(v_o^1, v_{t2}^1), (v_{t2}^2, v_d^1), (v_o^2, v_{t1}^2), (v_{t1}^1, v_d^2), (v_o^3, v_d^3)\}$) need to be indexed and looped over for partitioning.

Second, Alg. 2 can be readily modified to an *incremental* version by saving OPEN$_{rTSP}$ and $\mathcal{S}$ computed in the K-th call for the future $(K+1)$-th call. In other words, when a set of K-best tours is computed and a $(K+1)$-th best tour is required, the search process can be resumed by reusing OPEN$_{rTSP}$ and $\mathcal{S}$. This incremental version is helpful for CBSS since CBSS always requires a next best joint sequence incrementally.

B. Visiting a node exactly once vs. at least once

Within the CBSS framework, the K-best sequencing procedure only needs to compute joint sequences that visit each node in G_T (the target graph) exactly once, instead of at least once. In the ensuing paragraphs, we show that CBSS is able to find an $(1+\epsilon)$-bounded sub-optimal solution joint path (if one exists) by considering only the joint sequences that visit each node in G_T (the target graph) exactly once (i.e., Theorem 4 holds).

Let $L^* = \{\gamma_1^*, \gamma_2^*, \dots, \gamma_n^*\}$ denote the (finite) list of all joint sequences where each $\gamma_i^* \in L^*$ visits each node in G_T exactly once. The sequences in L^* are ordered such that their costs are non-decreasing (i.e., $cost(\gamma_i^*) \leq cost(\gamma_{i+1}^*)$ for $i = 1, 2, \cdots, n-1$). Similarly, let $L' = \{\gamma_1', \gamma_2', \dots\}$ denote the (infinite) list of joint sequences where each $\gamma_j' \in L'$ visits each node in G_T for at least once. Since each $\gamma_i^* \in L^*$ also visits each node in G_T for at least once, it follows that $L^* \subset L'$. The sequences in L' are ordered such that their costs are non-decreasing. In addition, if the cost of any two sequences $\gamma_i', \gamma_j' \in L'$ are the same, they are ordered such that

- γ_i' appears before γ_j' in L', if $\gamma_i', \gamma_j' \in L^*$ and γ_i' appears before γ_j' in L^*;
- γ_i' appears before γ_j' in L', if $\gamma_i' \in L^*$ and $\gamma_j' \notin L^*$.

Lemma 1: For each $\gamma_i^* \in L^*$, there exists a corresponding $\gamma_j' \in L'$ such that $\gamma_j' = \gamma_i^*$ and $j \geq i$.

For presentation purposes, we now use notation (γ_i^*, γ_j') to denote such a pair of joint sequences in both lists as described in Lemma 1. Let (γ_i^*, γ_j') and $(\gamma_{i+1}^*, \gamma_{j+\ell}')$ denote two adjacent pairs such that $cost(\gamma_{i+1}^*) > cost(\gamma_i^*)$. Let L_i^* denote the list of the i-best joint sequences $\{\gamma_1^*, \gamma_2^*, \dots, \gamma_i^*\}$ (and $i \leq n-1$). Then, we have the following lemma.

Lemma 2: For any joint sequence $\gamma_{k'}', k' = j+1, j+2, \dots, j+\ell-1$, $\gamma_{k'}'$ can be converted into a target sequence γ_k^* for some $k = 1, 2, \cdots, i$ by taking shortcuts in G_T so that γ_k^* visits each node in G_T exactly once.

This lemma holds because G_T (the target graph) satisfies triangle inequality (we can thus take shortcuts for nodes that are visited multiple times), and γ_k^* must be the same as one of the joint sequence in L_i^*, because otherwise L_i^* cannot be the i-best joint sequences. Additionally, for the case where $i = n$, the ℓ in Lemma 2 becomes infinity. In other words, let $(\gamma_n^*, \gamma_{n'}')$ denote the last pair. All joint sequences after $\gamma_{n'}'$ in L' can be shortcut to one of the sequence in L^*.

Next, we want to show that, by using L^* (instead of L') to generate root nodes during the CBSS search (Alg. 3), Theorem 4 still holds.

Definition 2 (CV-set): For a high level node $P = (\pi, g, \Omega)$, let $CV(P)$ be a set of joint paths, such that for each $\pi \in CV(P)$, π (i) is conflict-free, (ii) follows $\gamma(P)$, and (iii) satisfies all constraints in Ω.
Additionally, if $\pi \in CV(P)$, we say node P *permits* π.

Let $\gamma_{k'}'$ denote a joint sequence as discussed in Lemma 2 and let γ_k^* denote a the corresponding sequence after the shortcut as stated in Lemma 2. Let $\pi_{k'}$ denote a joint path that follows $\gamma_{k'}'$ while ignoring any agent-agent conflicts (and obviously $cost(\pi_{k'}) = cost(\gamma_{k'}')$), and let $P_{k'} = (\pi_{k'}, cost(\pi_{k'}), \emptyset)$ denote a high level node. Similarly, let π_k denote a joint path that follows γ_k^* while ignoring any agent-agent conflicts, and let $P_k = (\pi_k, cost(\pi_k), \emptyset)$ denote a root node γ_k^*.

Lemma 3: For each π that is permitted by P', π is also permitted by P_k.

To show this lemma, we need to verify the three conditions in Def. 2. Condition (i) and (iii) are obvious given that π is permitted by P'. We now show condition (ii). π is permitted by P', which means π follows $\gamma_{k'}'$ (where some of the nodes are visited multiple times). Since $\gamma_{k'}'$ can be shortcut to γ_k^* (i.e., skip the nodes that are visited multiple times), π also follows the joint sequence γ_k^*. This justifies the condition (ii) in Def. 2.

Lemma 3 shows that, during the CBSS search process, by only generating root nodes that correspond to joint sequences in L^*, Theorem 4 holds.