CBSS: A New Approach for Multi-Agent Combinatorial Path Finding

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

***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., target sequencing). To solve the problem, we leverage Conflict-Based Search (CBS) for MAPF and propose a novel approach called Conflict-Based Steiner Search (CBSS). CBSS interleaves (1) the collision resolution strategy in CBS to bypass the curse of dimensionality in MAPF and (2) multiple traveling salesman algorithms to handle the combinatorics in target sequencing, to compute optimal or bounded sub-optimal paths for agents while visiting all the targets. We also develop two variants of CBSS that trade off runtime against solution optimality. Our test results verify the advantage of CBSS over the baselines in terms of computing cheaper paths and improving success rates within a runtime limit for up to 20 agents and 50 targets. Finally, we run both Gazebo simulation and physical robot tests to validate that the planned paths are executable.**

***Index Terms*—Path Planning for Multiple Mobile Robots or Agents, Multi-Agent Path Finding, Traveling Salesman Problem**

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 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 intermediate target locations before reaching their destinations while satisfying additional agent-target assignment constraints (see Fig. 1 for a toy example). MAPF and its generalizations such as MCPF arise in applications in logistics [1] and surveillance [2]. For example, in a hazardous material warehouse, multiple mobile robots equipped with different 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 is visited at least once by a robot and the paths are collision-free. In addition, robots may carry different sensors, and only a subset of robots may have the sensors to measure the desired data at a target; this introduces agent-target assignment constraints that must be respected while planning paths. Simpler versions of the MCPF without the robot-robot collision constraints have been addressed in [3], [4], motivated by unmanned vehicle applications.

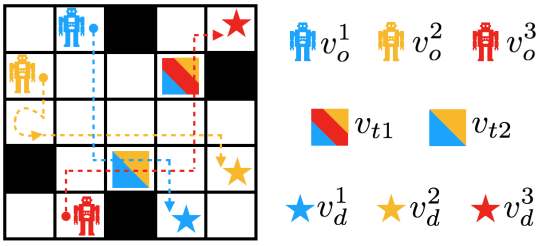

Fig. 1. An example of the MCPF problem. There are three agents; their initial locations are $(v_o^i, i = 1, 2, 3)$ and destinations are $(v_d^i, i = 1, 2, 3)$. The two targets are denoted as v_{t1} and v_{t2}. The color of the targets and destinations indicates the assignment constraints, (i.e., the 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. A solution (i.e., a collision-free joint path for the agents) is shown using dashed lines. The circular part of the yellow path indicates a wait-in-place action of the yellow agent.

Solving MCPF with optimality guarantees is challenging as it requires handling the difficulty in both MAPF and target sequencing. If the set of targets is empty and each destination is pre-assigned to a unique agent, MCPF reduces to MAPF [5], which is NP-hard [6]. On the other hand, if the collision between agents are ignored and no assignment constraints are present, MCPF reduces to a variant of Multiple Traveling Salesman Problem (mTSP) [3], [7], which is also NP-hard. As such, solving MCPF to optimality involves simultaneously addressing the challenges in both MAPF and mTSP.

Unlike MAPF, where ignoring the agent-agent collision leads to a decoupled shortest path problem for each agent, in MCPF, ignoring the collision leads to an mTSP where the agents' paths are still coupled. In the special case where there are no assignment constraints, our prior work developed an approach called MS* [8] based on the subdimensional expansion framework [9]. In this paper, we present a new approach called Conflict-Based Steiner Search (CBSS) for the general case of MCPF, which attempts to bypass the curses of dimensionality in both mTSP and MAPF to solve MCPF. CBSS *interleaves* mTSP and MAPF algorithms by alternating between (1) generating new target sequences for the agents, and (2) generating collision-free paths for agents based on the target sequences, while providing solution quality guarantees. To compute collision-free paths, CBSS conducts

(Corresponding author: Zhongqiang Ren.)
Zhongqiang Ren and Howie Choset are with Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA. (email: zhongqir@andrew.cmu.edu; choset@andrew.cmu.edu).
Sivakumar Rathinam is with Texas A&M University, College Station, TX 77843-3123. (email: srathinam@tamu.edu).

a two-level search like CBS [10]: For the high-level search, CBSS generates a search forest (of multiple trees) where each tree follows a fixed target sequence for the agents. A key contribution of this paper is in the generation of the search forest where we leverage the transformation [3] and K-best partition [11] 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. For the low-level search, CBSS runs constrained single-agent path planning to find a path for each agent following the target sequence 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 parameter that determines the sub-optimality bound of the solution returned. By varying ϵ from zero to infinity, CBSS moves along a spectrum from computing an optimal solution with heavy computational burden to computing a feasible solution quickly without any theoretic optimality bounds. Furthermore, when ϵ is infinity, CBSS becomes a "sequential" method that solves the problem in two sequential stages, where a target sequence is computed in the first stage, and collision-free paths are planned for the agents by fixing the target sequence in the second stage. To intelligently balance between runtime efficiency and solution quality, we further develop two variants of CBSS: (i) anytime CBSS, which can quickly compute a feasible solution and keep improving the quality of the solution before the runtime budget depletes; (ii) adaptive CBSS, which adjusts ϵ based on the difficulty of the problem instance.

To verify CBSS, we generate test instances with various forms of assignment constraints based on an online dataset [5]. We compare CBSS in various maps with several baselines including a greedy method, our prior MS* [8], and the aforementioned sequential method. We observe that CBSS computes shorter paths than the greedy method and the sequential method, and often 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 both Gazebo simulation and physical robot experiments to validate that the planned paths are executable.

To summarize, the main contribution of this article includes: (i) a novel method to compute K-best solutions for mTSPs; (ii) CBSS, a new approach to solve MCPF problems with various assignment constraints; (iii) two variants of CBSS that balance between runtime efficiency and solution quality guarantees; and (iv) simulation and physical robot tests that verify the planned paths are executable. Prior version of this work has appeared in [12], and this article differs from [12] by introducing the two variants of CBSS, detailed analysis, more numerical results against the baselines, and the Gazebo simulation. 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, and introduce the CBSS approach in Sec. V with proofs in Sec. VI. We present our test results in Sec. VII. Finally, we conclude and outline possible future directions in Sec. VIII.

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

II. Related Work

Multi-Agent Path Finding algorithms tend to fall on a spectrum from coupled [13] to decoupled [14], trading off completeness and optimality for scalability. In the middle of this spectrum lies the popular dynamically-coupled methods such as subdimensional expansion [9] and Conflict-Based Search (CBS) [10]. These methods have been improved and extended [15]–[18]. 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.

The Traveling Salesman Problem (TSP) seeks to find a shortest path/tour for an agent to visit each vertex in a graph, and is one of the most well known NP-hard problems [19]. A spectrum of methods have been developed ranging from exact techniques (branch and bound, branch and price) [19] to heuristics [20] and approximation algorithms [21], trading off solution optimality for runtime efficiency.

The Multiple Traveling Salesman Problem (mTSP) [7] is harder to solve compared to the (single-agent) TSP as the vertices in the graph must be allocated to each agent in addition to finding an optimal visiting order of the assigned targets for each agent. A variant of mTSP that is also related to this work is the multiple-Steiner TSP[2] [23] where the agents are required to visit a subset of vertices in a graph. While focusing on allocating and computing the visiting order of targets for agents, mTSP methods [3], [4], [7], [24] do not consider the collision avoidance constraints between the agents. In MCPF, agent-agent collision are also considered.

Combined Target Assignment/Sequencing and Path Finding problems are investigated from different perspectives very recently [25]–[29]. Closely related to this paper is the CBS-TA (Task Assignment) algorithm [25], which employs a strategy of using K-best task assignments to create multiple search trees. Our CBSS is similar to CBS-TA in terms of using K-best solutions to construct multiple trees. However, CBSS differs from CBS-TA by replacing the K-best task assignment method with a novel K-best sequencing approach. Additionally, since target sequencing (i.e., solving mTSPs) is in general computationally more expensive than solving task assignment problems [30], this paper further develops variants of CBSS to defer the target sequencing until absolutely necessary without losing solution quality guarantees.

Other methods that combine MAPF with target assignment and sequencing either consider target assignment only (without the need for computing visiting orders of targets) [25], [26], [28], or require computing the visiting order given that each agent is pre-allocated a set of targets [29], [31], [32]. In

[2]The origin of Steiner problems are ascribed to mathematician Jakob Steiner [22] where agents are not required to visit each and every vertex in a 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 use Steiner TSP to refer to all these variants.

addition, the multi-agent pick-up and delivery problem [33]–[35], which computes a set of collision-free paths to fulfill a set of pick-up and delivery tasks, also requires assigning a sequence of tasks for each agent. The existing approaches either enumerate all possible task sequences in a brute-force manner during the path planning in order to ensure solution optimality [36], which is computational burdensome, or leverage heuristics (such as the aforementioned sequential method) to ensure scalability [34], [37] without solution quality guarantees. This paper develops variants of CBSS to enjoy the fast running speed of the sequential method while providing tight sub-optimality bounds. In addition, MCPF allows the agents to be heterogeneous with various assignment constraints.

Finally, some recent work considers precedence constraints between tasks when planning collision-free paths for the agents [31], [38]. This paper does not consider precedence constraints, but focuses on assignment constraints to describe the heterogeneous capability of the agents.

III. Problem Description

Let index set $I = \{1, 2, \dots, N\}$ denote a set of N agents.[3] All agents move in a workspace represented as an finite undirected graph $G = (V, E)$ where the vertex set V represents the possible locations for agents and the edge set $E \subseteq V \times V$ denotes the set of all 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 paper, we use superscript $i \in I$ over a variable to represent the specific agent to which the variable relates (e.g. $v^i \in V$ means a vertex corresponding to agent i). Let $v_o^i \in V$ 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*[4] vertices that must be visited by at least one of the agents along its path. For each $v \in V_t \bigcup V_d$, 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 $\pi^i(v_1^i, v_\ell^i)$ denote a path for agent i between vertices v_1^i and v_ℓ^i via a list of vertices $(v_1^i, v_2^i, \dots, v_\ell^i)$ in G. Let $g(\pi^i(v_1^i, v_\ell^i))$ denote the cost of the path, which is the sum of the costs of all edges present in the path: $g(\pi^i(v_1^i, v_\ell^i)) = \Sigma_{j=1,2,\dots,\ell-1} cost(v_j^i, v_{j+1}^i)$. All agents share a global clock. 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 $t + 1$.

[3]Notations k [5], m [31] are also commonly used in the literature to indicate the number of agents. In this work, we reserve k to denote a general index and m to denote the index of the intermediate targets.

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

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 reaches the minimum.

Remark 1. *The notion that an agent i "visits" a target v 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. In this paper, when we say an agent or a path "visits" a target, we always mean the agent "visits and claims" the target.*

Remark 2. *MCPF generalizes several existing problems. When $M = 0$ (i.e., no target is present) and f_A maps each destination to an 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* [8]. Finally, if the conflict between agents is ignored, and the destination of each agent is the same as its starting location (which is often called a "depot" in TSP), then MCPF reduces to a variant of mTSP [3].*

IV. Review of Conflict-Based Search

Conflict-Based Search (CBS) [10] is a two-level search algorithm. On the high-level, every 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 connects the v_o^i and v_d^i for each agent $i \in I$.
- g is the scalar cost value of π (i.e., $g = g(\pi) = \Sigma_{i \in I} g(\pi^i)$).
- Ω is a set of (collision) constraints.[5] A constraint is of form (i, v, t) (or (i, e, t)), which indicates agent i is forbidden from occupying vertex v (or traversing edge e) at time t.

CBS constructs a 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 from the minimum to the maximum.

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 paths in π is checked for a vertex conflict (i, j, v, t) (and an edge conflict (i, j, e, t)). If no conflict is detected, π is conflict-free and is returned as a *solution* (i.e., a conflict-free joint path) and this solution is guaranteed to be optimal (i.e., has the minimum cost). 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

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

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 a minimum cost 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 space-time graph with constraints marked as inaccessible vertices and edges in this space-time graph. After the low-level planning, a new joint path π' is formed by first copying π and then replacing agent i's path π^i with π'^i. Finally, for each of the two split constraints, a corresponding new node is generated and added to OPEN for future expansion. CBS [10] is guaranteed to find a minimum cost solution for a solvable MAPF problem.

V. Conflict-Based Steiner Search

A. Basic Concepts and Overview

In MCPF, a path for agent i may visit a sequence of targets before reaching its destination. Let $\gamma^i = \{v_o^i, u_1^i, u_2^i, \dots, u_\ell^i, v_d^i\}$ denote a *target sequence* visited by agent $i \in I$ where v_o^i is the initial vertex of agent i, u_j^i is the j^{th} target visited by agent i with $j = 1, \cdots, \ell$, and $v_d^i \in V_d$ is a destination. Let $\gamma = \{\gamma^i : i \in I\}$ denote a *joint (target) sequence*, which is a collection of target sequences of agents. The cost incurred in traveling between any two subsequent vertices $u, v \in \gamma^i$ is simply the minimum-cost path cost between u and v in G. The cost of a target sequence $cost(\gamma^i)$ is defined as the total cost incurred in traversing all the vertices in γ^i. Similarly, the cost of a joint sequence is defined as $cost(\gamma) := \Sigma_{i \in I} cost(\gamma^i)$. Note that $cost(\gamma)$ is computed *ignoring all the conflicts* between the agents. Following the same notations as in CBS, let $P = (\pi, g, \Omega)$ denote a node for CBS search. 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) conducts a two-level search 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 search. Specifically, CBSS constructs a search forest[6] (rather than a single search tree) where each tree $\mathcal{T}_j$ in the forest corresponds to a joint sequence γ_j^* that is fixed. In other words, the joint path π for any 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 using a sequencing[7] procedure while ignoring any conflict between agents, which ensures that the cost of the joint sequences are monotonically non-decreasing: $cost(\gamma_1^*) \leq cost(\gamma_2^*) \leq cost(\gamma_3^*) \dots$. Given a joint sequence γ_j^* and its corresponding tree $\mathcal{T}_j$, conflicts are resolved through the same conflict splitting process as in CBS. Similar to the low-level search in CBS, the low-level search in CBSS iteratively plans a path for agent i from one vertex to another as specified in γ^i by using A*-like search, while satisfying the constraints.

Initially, CBSS starts with a joint sequence γ_1^*. A node corresponding to γ_1^* is created and forms the root node of $\mathcal{T}_1$. If the joint path π_o in this root node does not have any conflict between agents, then the search terminates and outputs π_o, which is an optimal solution to MCPF. If π_o has a conflict, then two new nodes are created as in CBS, and are added to OPEN. Then, during the search, a node $P = (\pi, g(\pi), \Omega)$ is popped from OPEN. If $g(\pi)$ is no larger than $cost(\gamma_2^*)$ (i.e., the cost of a second best joint sequence), the search continues to check for a conflict in π and expand P. Otherwise (i.e., γ_2^* is cheaper than $g(\pi)$), a new tree denoted by $\mathcal{T}_2$ is then created and the root node of γ_2^* is added to OPEN, and the search continues. Since $cost(\gamma_1^*)$ is a lower bound to the optimal solution cost of MCPF (because γ_1^* ignores conflicts), and that the nodes are systematically generated and expanded in a best-first search manner, CBSS can find an optimal solution to the MCPF (as proved in Sec. VI).

There are some crucial parts to this high-level search in CBSS. Generating a set of joint sequences with monotonically increasing costs is non-trivial, especially for a mTSP with agent-target assignment constraints. Formally, it requires solving a K-best Multi-Depot Multi-Terminal Hamiltonian Path Problem, which is referred to as a K-best sequencing problem for simplicity. Currently, there is no existing algorithm to solve this (multi-agent) K-best sequencing problem. We propose the following approach to solve it. We first leverage a transformation method [3] that can convert a mTSP to a (single-agent) TSP, then leverage a partition method [11] to solve the (single-agent) K-best TSP, and finally transform the obtained solutions back to the solutions to the original (multi-agent) K-best sequencing problem. Here, the transformation method [3] guarantees solution optimality while being able to leverage the state-of-the-art single-agent TSP solvers (Sec. V-C). The partition method [11] solves a K-best TSP by systematically forcing a solution to include some edges and exclude some other edges to find the desired K-best solutions (Sec. V-D).

As finding an optimal solution for MCPF can be computational expensive, 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 solution (if one exists) whose cost is at most $(1+\epsilon)$ times the optimum. With that in hand, we further develop anytime CBSS and adaptive CBSS that can trade off between solution quality and computational time.

B. CBSS Algorithm

Let $G_T = (V_T, E_T, C_T)$ denote a *target graph*, which is a complete undirected graph with the 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 conflicts and the corresponding entry $C_T(u, v)$ stores the cost of that path. Note

[6]CBSS is not the first method that extends CBS to a search forest. For example, CBS-TA [25] uses a search forest to combine task assignment and path finding. In MO-CBS [39], a search forest is constructed to find the Pareto-optimal front for multi-objective MAPF problems.

[7]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.

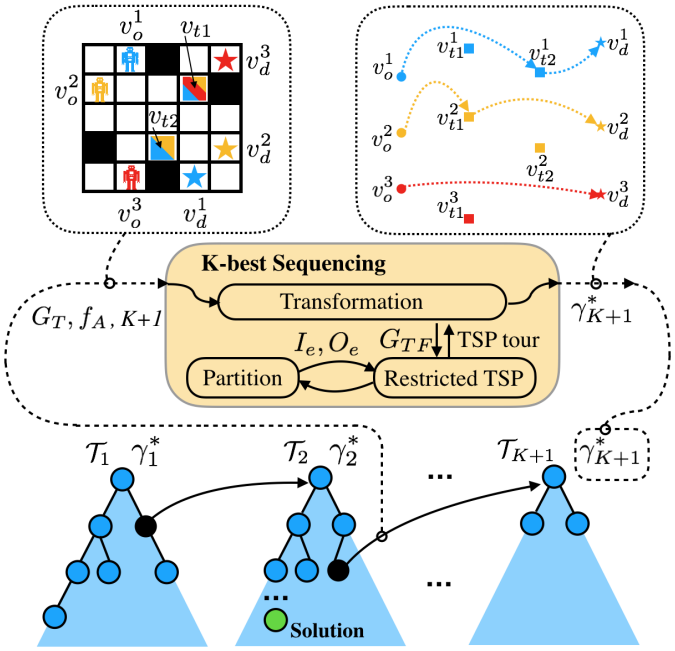

Fig. 2. A conceptual visualization of CBSS. Each tree $\mathcal{T}_k$ indicates that: given a joint sequence γ^*_k, CBSS plans a joint path by following γ^*_k and leverages Conflict-Based Search (CBS) to resolve conflicts between agents along their paths. During the CBS search, when the cost of the node to be expanded exceeds a threshold, the next-best joint sequence is created by using a K-best sequencing procedure. CBSS conducts the search in a best-first manner, which provides solution quality guarantees.

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*(γ^*_1, Ω)
5: Add $P_{root,1} = (\pi, g, \Omega)$ to OPEN
6: **while** OPEN is 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: $cft \leftarrow$ *DetectConflict*(π_k)
10: **if** $cft = NULL$ **then**
11: **return** π_k
12: $\Omega \leftarrow$ *GenerateConstraints*(cft)
13: **for all** $\omega^i \in \Omega$ **do**
14: $\Omega'_k = \Omega_k \cup \{\omega^i\}$
15: $\pi'_k, g'_k \leftarrow$ *LowLevelPlan*($\gamma(P_k)$, Ω'_k)
16: // In this LowLevelPlan, only agent i's path is planned.
17: Add $P'_k = (\pi'_k, g'_k, \Omega'_k)$ to OPEN
18: **return** failure

that G_T is a graph where the edge cost satisfies the triangle inequality, which will be used later in Sec. VI.

The CBSS algorithm is shown in Alg. 1. CBSS first finds a minimum cost joint sequence γ^*_1 (lines 1-2) in G_T. CBSS then invokes the low-level search (lines 3-4) for all agents $i \in I$ with an empty set of constraints, which computes π that follows γ^*_1, as well as the cost value $g(\pi)$. A root node $P_{root,1}$ is then created and added to OPEN, a priority queue where nodes are prioritized based on their g-values from the minimum to the maximum.

In each iteration of the high-level search (lines 6-17), a node P_l with the least g-value is popped from OPEN. A procedure *CheckNewRoot* (Alg. 3) is invoked, which compares the cost g_l of node P_l against some threshold to decide whether the next root node needs to be created. This threshold is based on both the cost of the current joint sequence, a next-best joint sequence, and a hyper-parameter ϵ that determines the sub-optimality bound of the computed solution, which is presented in Sec. V-E.

- If the next root node does not need to be created, the input node P_l is returned by *CheckNewRoot*.
- If the next root node (denoted as the r-th root $P_{root,r}$) needs to be created, 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*($\gamma^*_r, \emptyset$), which runs the low-level search to find a path for each agent by following $\gamma^{*,i}_r \in \gamma^*_r$. All these paths together form a joint path that follows γ^*_r. The resulting new root node $P_{root,r}$ is then returned by *CheckNewRoot*. Additionally, P_l (i.e., the input node to *CheckNewRoot*) is added back to OPEN for future expansion.

The node returned by *CheckNewRoot* is denoted as P_k and the joint path π_k in P_k is then checked for a conflict (line 9). If no conflict is detected, CBSS terminates and π_k is returned. Otherwise, CBSS splits the detected conflict (same as in CBS) by generating two constraints. Here, we abuse the notation 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 $P_{root,r}$. For each newly generated constraint ω^i, CBSS updates the constraint set (line 14) and invokes the low-level search for agent i (line 15) to recompute its path that satisfies the new set of constraints. Finally, the newly generated nodes are added to OPEN for future expansion (line 17).

C. Transformation Method for Sequencing

We now present our transformation method that takes the target graph G_T and assignment constraints f_A as input, and returns a minimum cost joint sequence. Our approach is based on the transformation used in [3] for mTSP, and the main idea is to convert a mTSP into a TSP while ensuring that the optimal solution to the generated TSP can be converted back to an optimal solution to the original mTSP. Our transformation method differs from the one in [3] as follows: (1) destinations are explicitly introduced and they can be different from agents' initial vertices; (2) assignment constraints are introduced in this paper; (3) the concept about the "heterogeneous costs" in [3] is not relevant in this paper and thus removed. The main steps in our transformation method are the following:

- Step-1 converts the target graph G_T into a directed *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 later;

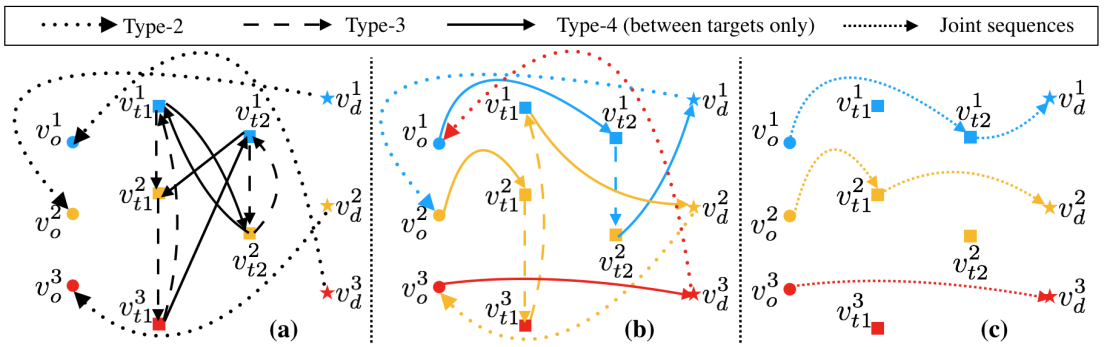

Fig. 3. A visualization of the transformation method for the toy example in Fig. 1. (a) shows the transformed graph G_{TF}, where edges from the initial vertices and the edges connected to destinations are omitted to make the plot readable. (b) shows a TSP tour $\{v_o^1, v_{t2}^1, v_{t2}^2, v_d^1, v_o^2, v_{t1}^2, v_{t1}^3, v_{t1}^1, v_d^2, v_o^3, v_d^3, v_o^1\}$ in G_{TF}. (c) shows the computed joint sequence $\gamma = \{\gamma^i, i = 1, 2, 3\}$, where $\gamma^1 : \{v_o^1, v_{t1}^1, v_{t2}^1, v_d^1\}$, $\gamma^2 : \{v_o^2, v_{t1}^2, v_d^2\}$, $\gamma^3 : \{v_o^3, v_d^3\}$.

- Step-2 invokes an existing asymmetric TSP[8] solver to compute a minimum cost (single-agent) *tour* in G_{TF} (i.e., a path that starts and ends at the same vertex while traversing all 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, and each segment corresponds to a target sequence for an agent.

We now elaborate these three steps and provide a toy example in Fig. 3. **Step-1** generates the directed graph $G_{TF} = (V_{TF}, E_{TF}, C_{TF})$ 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 $v \in V_t \bigcup V_d$, a copy v^i of v for agent i is made, if $i \in f_A(v)$. The purpose of this augmentation is to suitably represent the 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 is the last in $U(v)$, let $Next(v^i)$ indicate the first copy in $U(v)$; And when v^i is the first in $U(v)$, let $Prev(v^i)$ indicate the last copy in $U(v)$. As an example, in Fig. 3 (a), $U = \{v_{t1}^1, v_{t1}^2, v_{t1}^3, v_{t2}^1, v_{t2}^2, v_d^1, v_d^2, v_d^3\}$ 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 vertices $u^i \in U^i$ with a cost value equal to $C_T(v_o^i, u^i)$.
- **Type-2 Edges:** the copy of each destination of agent i is connected to the start of agent $(i+1)$ for $i = 1, 2, \ldots, (N-1)$ with an zero-cost edge, and the copy of each destination of agent $i = N$ is connected to v_o^1, the initial vertex of agent $i = 1$, with zero-cost edges. The intuition behind these zero-cost edges is to make sure the minimum cost tour of G_{TF} uses these zero-cost edges to connect two subsequent agents' destinations and initial vertices, and the tour can be divided into N segments by removing these zero-cost edges later.
- **Type-3 Edges:** for each $v \in V_t \bigcup V_d$, an zero-cost edge is connected from $v^i \in U(v)$ to $Next(v^i)$. We explain the intuition after introducing the Type-4 edges.
- **Type-4 Edges:** for each $v \in V_t \bigcup V_d$ and for each agent $i \in f_A(v)$, an edge is connected 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 $Z + C_T(v^i, u^i)$, where Z is a large constant number that is an over-estimate of the optimum (i.e., the cost of the minimum cost joint sequence). For example, $Z = 2(N+M) \max_{(u,v) \in E_T} C_T(u, v)$. In Fig. 3 (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 to each of $\{v_{t1}^1, v_d^1\}$. Note that in Fig. 3 (a), all edges connected to destinations are omitted to make the figure readable.

The intuition behind the Type-3, 4 edges and their costs is to ensure that when a minimum cost tour in G_{TF} visits a copy vertex v^i of $v \in V_t \bigcup V_d$, the tour must visit all other copies $v^j, j \neq i$ before arriving at the copy u^i of another vertex $u \in V_t \bigcup V_d, u \neq v$. Specifically, Type-3 edges have zero-cost to encourage a minimum cost tour to visit all copies in $U(v)$ for some $v \in V_t \bigcup V_d$ in a "loop" before visiting another vertex. Without the large cost Z for the Type-4 edges, it may be more expensive sometimes to visit all the copies in $U(v)$ one after another in a loop, and a cheaper tour may break from the loop, visit another nearest copy in $U(u)$ for some $u \in V_t \bigcup V_d, u \neq v$, and get back to the loop at some other copy in $U(v)$. With Z, a minimum cost tour uses totally $(|V_t| + |V_d|) = (M + N)$ edges to enter $U(v)$ exactly once for each $v \in V_t \bigcup V_d$, and any tour that breaks from the loop must use more than $(M + N)$ edges, which cannot lead to a minimum cost tour. By doing so, the copies of each $v \in V_t \bigcup V_d$ are always visited one after another in a loop, and this loop can later be removed from the tour to extract target sequences (Fig. 3 (b) and (c)).

We have finished the presentation about Step-1, which

[8]Asymmetric TSP means that the graph is directed and $cost(u, v)$ may not be the same as $cost(v, u)$ for any two vertices u, v in the graph. We use "edges" as opposed to "arcs" when referring to edges in G_{TF} for simplicity.

generates a transformed graph G_{TF}. In **Step-2**, a regular (single-agent) asymmetric TSP solver is invoked on G_{TF} and a minimum cost tour is computed. As shown in Fig. 3 (b), a minimum cost tour is visualized for the toy example in Fig. 1. In **Step-3**, the computed 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. 3 (b), the dotted lines, which denote the zero-cost edges, are removed.) Second, the copies of the same targets are shortcut. (In Fig. 3 (b), the dashed lines in yellow and blue are shortcut.) Third, the cost of the resulting joint sequence can be obtained by taking the sum of the cost of edges in the joint sequence based on C_T (as opposed to C_{TF}). The resulting joint sequence for the toy example is shown in Fig. 3 (c). The property of this transformation method is summarized with the following theorem. The proof in [40] can be readily adapted to the transformation in this paper.

Theorem 1. *Given G_T and f_A, the transformation method computes a minimum cost joint sequence that visits all targets and ends at destinations while satisfying all assignment constraints.*

Remark 3. *In 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 (Multi-Agent Simultaneous Multi-Goal Sequencing and Path Finding) problem in [8]. For MSMP, the transformation 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. Details can be found in [3], [8].*

D. K-best Joint Sequences

To find a set of K cheapest joint sequences, the main idea here is to first transform the multi-agent problem into an equivalent single-agent problem (i.e., a TSP) as described in the previous section, and then leverage the partition method [11], [41] to solve a K-best TSP to find K cheapest tours. The K-best TSP method in this section replaces the Step-2 in the transformation method in the previous section. Specifically, the Step-2 in the previous section finds a minimum cost tour in G_{TF} while the method in this section finds K-best tours in G_{TF}. To this end, we first introduce the Restricted TSP that is used in the partition method.

Definition 1 (Restricted TSP (rTSP))**.** *Given a directed graph $G' = (V', E', C')$, let $I_e, O_e \subseteq E'$ denote two disjoint subsets of edges in G', an rTSP seeks to find a minimum cost tour τ^* such that $I_e \subseteq \tau^* \subseteq E' \backslash O_e$.*[9]

Intuitively, an rTSP is defined by two sets of edges I_e, O_e, and requires computing a minimum cost tour with all edges in I_e included and the edges in O_e excluded. To solve an rTSP, one can first obtain a modified graph G'' by changing the cost of edges in I_e to a relatively small value (so that they must be part of a minimum cost tour), while removing the edges in O_e by modifying the cost of the edges to large positive values (so that the edges cannot be part of a minimum cost tour). Then a regular TSP solver can be invoked on G'' to solve rTSP. Let $\tau^* \leftarrow$ *rTSP-Solve*(G', I_e, O_e) denote this solution process.

The partition method [11] solves a K-best TSP via a best-first search by iteratively partitioning the set of possible tours while finding a minimum cost tour in each partitioned subset. As shown in Alg. 2, $I_e(k), O_e(k)$ and $\tau^*(k)$ keep track of I_e, O_e and τ^* as a function of the iteration number (k) of the algorithm. To initialize, Alg. 2 computes a minimum cost tour $\tau^*(1)$ in G' with $I_e(1), O_e(1)$ being empty sets (lines 1-4). The tuple $(I_e(1), O_e(1), \tau^*(1))$ with tour cost $cost(\tau^*(1))$ is then added to OPEN$_{rTSP}$, a priority queue where tuples are prioritized based on their tour costs. The set of K-best tours $\mathcal{S}$ is initialized to be an empty set.

[9]In this paper, G' is always the same as G_{TF}. We use G' instead of G_{TF} to highlight that the partition method is not limited to G_{TF}. Also note the difference between a set of edges I_e and the index set I representing all agents. Additionally, there are two possible definitions of the rTSP problem: one requires finding a tour that visits each vertex in G' at least once (i.e., with repetition) and another that requires visiting each vertex exactly once (i.e., without repetition). We show in Sec. VI-B that these two versions are equivalent within the framework of CBSS. We now focus on finding a tour that visits each vertex exactly once.

Algorithm 2 Pseudocode for K-best-TSP

1: $I_e(1) \leftarrow \emptyset, O_e(1) \leftarrow \emptyset$
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 $(I_e(k+1)[p], O_e(k+1)[p], \tau^*(k+1)[p])$ to OPEN$_{rTSP}$
17: **return** failure

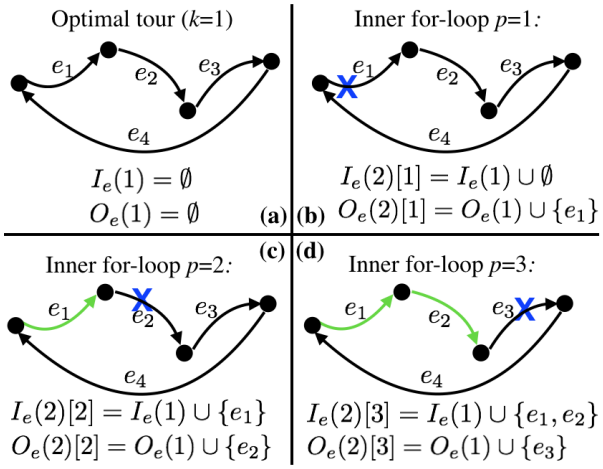

Fig. 4. An illustration of the partition process in Alg. 2. (a) shows a minimum cost tour for a TSP, whose edges are indexed. To compute the second best tour ($K = 2$), (b), (c) and (d) show the I_e and O_e sets of the inner for-loop iterations (lines 11-16 in Alg. 2) with $p = 1, 2, 3$ respectively. The iteration $p = 4$ is omitted.

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 tour and is added to $\mathcal{S}$ (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 $\tau^*(k)$ from 1 to ℓ, where ℓ is the number of edges in the tour. Then, Alg. 2 iterates all edges to partition the set of remaining possible tours and generate a corresponding rTSP for each partition. We use $[p], p \in \{1, 2, \ldots, \ell\}$ 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 added to the set $I_e(k)$ to form a new set of edges (denoted as $I_e(k+1)[p]$) that must be included into 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. 4. Alg. 2 then solves the resulting rTSP defined by $I_e(k+1)[p]$ and $O_e(k+1)[p]$ to obtain tour $\tau^*(k+1)[p]$. In practice, we set the cost of edges in O_e to a large number, and these edges may still be used. Therefore, Alg. 2 verifies the feasibility of $\tau^*(k+1)[p]$, where feasibility means the tour includes all edges in I_e and excludes all edges in O_e. If $\tau^*(k+1)[p]$ is feasible, Alg. 2 adds the resulting tuple to OPEN$_{rTSP}$ (lines 14-16).

Theorem 2. *Given a graph $G' = (V', E', C')$, Alg. 2 finds 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 provide an intuitive explanation about the completeness of the partition, which complements the analysis in [11], [41]. Given a minimum cost tour τ^*, index the edges in τ^* from 1 to ℓ. For an arbitrary tour τ' that is different from τ^*, we can introduce a binary vector $b \in \mathbb{B}^\ell \backslash \mathbf{1}$ of length ℓ to indicate if each edge in τ^* is included in τ', where $\mathbb{B}$=$\{0, 1\}$ and $\mathbf{1}$ is a vector with all component being one. Specifically, the j-th component $b(j)$ is equal to one (or zero) if the j-th edge in τ^* is included in (or excluded from) the tour τ'. Obviously, b cannot be equal to $\mathbf{1}$, since τ' must be different from τ^*. Alg. 2 partitions $\mathbb{B}^\ell \backslash \mathbf{1}$ into the following sets that are mutually disjoint to each other: $\{(0, \mathbb{B}, \cdots, \mathbb{B})$, $(1, 0, \mathbb{B}, \cdots, \mathbb{B})$, $(1, 1, 0, \mathbb{B}, \cdots, \mathbb{B})$, $\cdots$,$(1, 1, \cdots, 1, 0)\}$, where $\mathbb{B}$ means the corresponding component can be either zero or one. To show this partition is complete, we need to show that any $b \in \mathbb{B}^\ell \backslash \mathbf{1}$ belongs to one of the sets. We show it as follows: for an arbitrary $b \in \mathbb{B}^\ell \backslash \mathbf{1}$, find the first component in b that is zero. Without losing generality, say $b(j) = 0$ with j being a specific number ranging from 1 to ℓ. Then the j-th set in the above partition contains b. Therefore, the partition is complete.

Remark 4. *Solving a K-best TSP is computationally expensive. In a iteration of the while loop (lines 5-16), Alg. 2 needs to solve ℓ rTSPs. To find K-best tours, the algorithm requires solving $1 + (K-1)\ell$ rTSPs. For implementation, a couple of techniques can be used to improve the runtime efficiency, which is discussed in Sec. V-F.*

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
5: $\gamma_{r+1}^* \leftarrow$ *K-best-Sequencing*$(G_T, f_A, K = r+1)$
6: $\pi, g(\pi) \leftarrow$ LowLevelPlan$(\gamma_{r+1}^*, \emptyset)$
7: $P_{root,r+1} = (\pi, g(\pi), \emptyset)$
8: **if** $g_l \leq g(\pi)$ **then**
9: Add $P_{root,r+1}$ to OPEN
10: **return** P_l
11: Add P_l to OPEN
12: **return** $P_{root,r+1}$

E. Generation of New Root Nodes

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

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 and its cost $g(\pi)$ by following γ_{r+1}^*, and the next root node $P_{root,r+1}$ is created. Finally, line 8 checks if the cost of the input node P_l exceeds the cost of the new root node $P_{root,r+1}$ and the cheaper node is returned for expansion.

As shown in Alg. 1, CBSS invokes *CheckNewRoot* 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 is worthwhile to note that, when $\epsilon = \infty$, CBSS becomes a "sequential" method in a sense that the minimum cost 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 conflict-free joint path may still be found but no optimality guarantee can be provided (i.e., a bound of ∞).

F. Discussion on Implementation

A few techniques can be introduced in the implementation of CBSS. To improve the computational efficiency of Alg. 2, first, 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-C), which are auxiliary edges that help with the transformation. These edges can be skipped during the iterations (lines 11-16 in Alg. 2) to improve the computational ef-

ficiency. For example in Fig. 3, only the solid edges in Fig 3 (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 iterated for partition. Second, Alg. 2 can be implemented in an *incremental* fashion by reusing the OPEN$_{rTSP}$ and $\mathcal{S}$ computed in the K-th call of Alg. 2 in the future $(K+1)$-th call of Alg. 2. 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.

Additionally, the low-level search of CBSS needs to plan a path between each pair of subsequent vertices in a target sequence (line 4, 15 in Alg. 1). These paths can be cached and re-used for future low-level search as well as the generation of the target graph G_T.

G. Anytime CBSS

Finding a next-best joint sequence is often computationally expensive as it requires solving many rTSPs. A large ϵ can defer the generation of a next-best joint sequence during the CBSS search and thus improve the search efficiency.[10] However, a large ϵ means a loose sub-optimality bound. We therefore propose anytime CBSS: it begins by using a large ϵ so that CBSS can quickly find a solution, and then continues the search with a decreased ϵ to improve solution quality until a runtime limit is reached. Specifically, anytime CBSS modifies line 11 in Alg. 1 as follows. Instead of immediately returning the solution that is found, anytime CBSS stores the solution, reduces epsilon and continues the search until timeout. When timeout, among all solutions that are found, the one with the minimum cost is returned.

H. Adaptive CBSS

Adaptive CBSS seeks to allow CBSS to choose ϵ based on the difficulty of computing joint sequences for a specific problem instance. Given an instance, if computing target sequences is computationally expensive, adaptive CBSS uses a large ϵ to defer the generation of a next-best joint sequence during the CBSS search in order to find a solution within the runtime limit. If computing joint sequences is fast, adaptive CBSS chooses a small ϵ to find a solution with a tight sub-optimality bound. Specifically, adaptive CBSS differs from the regular CBSS by adding an additional line after line 2 in Alg. 1, where ϵ is set based on the runtime of line 2. Note that line 2 computes a minimum cost joint sequence γ_1^*, whose runtime is used as an indicator of the difficulty for computing target sequences. We present numerical results about both the anytime and adaptive CBSS in Sec. VII-E.

VI. Analysis

A. Solution Optimality

We begin by providing an intuitive explanation about the solution optimality guarantee and show the proof in the ensuing paragraphs. The CBSS search proceeds along two directions: CBSS either resolves conflicts between agents within a tree, or generate a next-best joint sequence to create the root node of a new tree. Along either direction, the cost of the node to be expanded is monotonically non-decreasing. In addition, the entire search is conducted in a best-first manner (i.e., by iteratively selecting the minimum cost node from OPEN for expansion), which ensures that the first solution returned is optimal (or bounded sub-optimal).

We now prove 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 MCPF problem instance is feasible if there exists a solution.

Theorem 3. *For a feasible MCPF problem instance, when $\epsilon < \infty$, CBSS returns a solution.*

Proof. During the search, CBSS either generates a new root node with monotonically non-decreasing costs (Theorem 2), or expands a node within a certain tree. There is a finite number of possible joint sequences in a finite graph and thus a finite number of root nodes to be generated. Additionally, within each tree, CBSS expands nodes with non-decreasing costs, and there is only a finite number of possible nodes with costs no larger than a certain cost value [10]. If the given instance is feasible, the correpsonding solution must have a finite cost. Therefore, after popping all these nodes from OPEN for expansion, CBSS terminates in finite time and returns solution, i.e., a conflict-free joint path that visits all targets and ends at destinations while satisfying the assignment constraints (Theorem 1). □

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 nodes corresponding to g^* (the optimal solution cost) and g (the cost of the solution returned by CBSS) respectively. Let $P_{root,r'} = (\pi', g', \Omega')$ denote the root node of the tree that contains P^*, and let r denote the number of root nodes that have been generated. 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 node with a conflict-free joint path that is popped from OPEN, and g is the smallest cost among the costs of all nodes in OPEN. Thus $g \leq g^*$. The joint path in node P^* is an optimal solution, and thus $g^* \leq g$. Therefore, $g = g^* \leq (1+\epsilon)g^*$.

For case (2), in 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'. Therefore, $g \leq (1+\epsilon)g^*$ for case (2). In summary, for either case, we have $g \leq (1+\epsilon)g^*$. □

[10]Note that a larger ϵ is not guaranteed to expedite the search, from the numerical results in Sec. VII-C, we observe that CBSS with a large ϵ tends to solve more instances within a fixed runtime limit.

B. Visiting Vertices Exactly Once and At Least Once

Within CBSS, the K-best sequencing procedure computes joint sequences that visit each vertex in G_T exactly once, instead of at least once. This section shows that CBSS can find an $(1+\epsilon)$-bounded sub-optimal solution (if one exists) by considering only the joint sequences that visit each vertex in G_T 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 vertex in G_T exactly once. The joint 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 vertex in G_T at least once. Since every $\gamma_i^* \in L^*$ visits each vertex in G_T for at least once, it follows that $L^* \subset L'$. The joint sequences in L' are ordered such that their costs are non-decreasing. In addition, if the cost of any two joint 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 $j \geq i$.*

Let (γ_i^*, γ_j') denote such a pair of joint sequences in L^*, L' 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 $i \leq n-1$). We then 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 joint sequence γ_k^* for some $k = 1, 2, \cdots, i$ by taking shortcuts in G_T so that γ_k^* visits each vertex in G_T exactly once.*

This lemma holds because G_T satisfies the triangle inequality, and we can thus take shortcuts for vertices that are visited multiple times. Furthermore, 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, if $i = n$, then 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 joint sequences in L^*.

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

Definition 2 (CV-set)**.** *For a node $P = (\pi, g, \Omega)$, let $CV(P)$ denote a set of joint paths (where CV stands for "consistent and valid"), such that for each $\pi \in CV(P)$, π (i) is conflict-free (i.e., valid), (ii) follows $\gamma(P)$, and (iii) satisfies all constraints in Ω (i.e., is consistent with 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 joint sequence after the shortcut as stated in Lemma 2. Let $\pi_{k'}$ denote a joint path that follows $\gamma_{k'}'$ while ignoring any conflict (and obviously $cost(\pi_{k'}) = cost(\gamma_{k'}')$), and let $P_{k'} = (\pi_{k'}, cost(\pi_{k'}), \emptyset)$ denote a root node. Similarly, let π_k denote a joint path that follows γ_k^* while ignoring any conflict, and let $P_k = (\pi_k, cost(\pi_k), \emptyset)$ denote a root node.

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 vertices in G_T are visited multiple times). Since $\gamma_{k'}'$ can be shortcut to γ_k^* (i.e., skip the vertices in G_T 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, by only generating root nodes that correspond to joint sequences in L^*, Theorem 4 holds.

VII. Results

A. Baselines and Test Settings

We implement CBSS in Python. We use LKH-2.0.9[11] [20] as the single-agent asymmetric TSP solver required by the transformation method (Sec. V-C). We implement the low-level search in CBSS by using SIPP [42] to search a space-time graph $G \times \{0, 1, 2, \dots, T\}$ subject to vertex and edge constraints. We learn from our prior work [43] that SIPP runs faster than A* as the low-level search for CBS-like algorithms. We use different grid maps from a online data set [5] and make each of them a four-neighbor graph 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 runtime limit of **one** minute. For the rest of the paper, 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 four baselines for comparison. The **first** one is using A* to search the joint configuration space of the 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* [8], which is a multi-agent planner leveraging subdimensional expansion [9] 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 (explained later) and then invokes LKH for each agent $i \in I$ to compute the visiting order of the assigned targets. The computed joint sequence is then used within the CBSS framework with $\epsilon = \infty$. As a result, this greedy baseline runs in a sequential manner by first computing a joint sequence γ and then planning a conflict-free joint path following γ. This greedy baseline can handle arbitrary forms of assignment constraints. Specifically, the greedy assignment procedure consists of the following steps: (i) In an iteration, the procedure iterates all

[11]LKH is a heuristic algorithm for TSP, which finds an optimal solution for numerous TSP instances [20] (http://akira.ruc.dk/~keld/research/LKH). This paper uses LKH as the TSP solver due to its computational efficiency. Other TSP solvers can also be used. Note that, since the tour returned by LKH is not guaranteed to have the minimum cost, the resulting implementation of CBSS (with $\epsilon = 0$) is not guaranteed to return an optimal solution.

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 vertex" (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 vertex to v; (iii) The procedure repeats (i) and (ii) until all targets are assigned, and then runs the same greedy assignment procedure for all destinations while ensuring that each agent is assigned to a unique destination. After the assignment, LKH is called to find the visiting order as follows. Let v_d^i denote the destination assigned to agent i. First, the edge cost $cost(v_d^i, v_o^i)$ is set to a small number so that (v_d^i, v_o^i) must be included into the tour. Then, LKH finds a tour that starts and ends at v_o^i while visiting all the assigned targets and v_d^i. Finally, the edge (v_d^i, v_o^i) is removed from the tour and the resulting path specifies the visiting order of the targets.

Finally, the **fourth** baseline is the aforementioned sequential method (Sec. V-E), which separates the target sequencing and path planning into two phases: this baseline first uses the target sequencing method in Sec. V-C to compute a minimum cost joint sequence γ, and then uses CBS-like search to plan a conflict-free joint path by following γ. This baseline is a special case of CBSS with ϵ being infinity (i.e., the second best joint sequence is never computed).

B. CBSS vs MS*

We begin our tests with fully anonymous MCPF problems. We compare CBSS ($\epsilon = 0$) against both MS* [8] and A* for $N \in \{5, 10, 20\}$ 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%. A possible reason is because MS* still needs to search the joint configuration space of the agents that are in conflict with each other while CBSS is able to bypass the search in the joint configuration space via multiple constrained single-agent search.

C. CBSS with Different Sub-optimality Bounds

We then investigate using different ϵs in 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 runtime limit).

1) Success Rates: As shown in Fig. 6 (b), increasing ϵ from 0 to 0.01 can increase the success rate. A reason is that a small ϵ can help with tie-breaking: for instances with many equal-cost joint sequences, a small ϵ 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 can further improve the success rate.

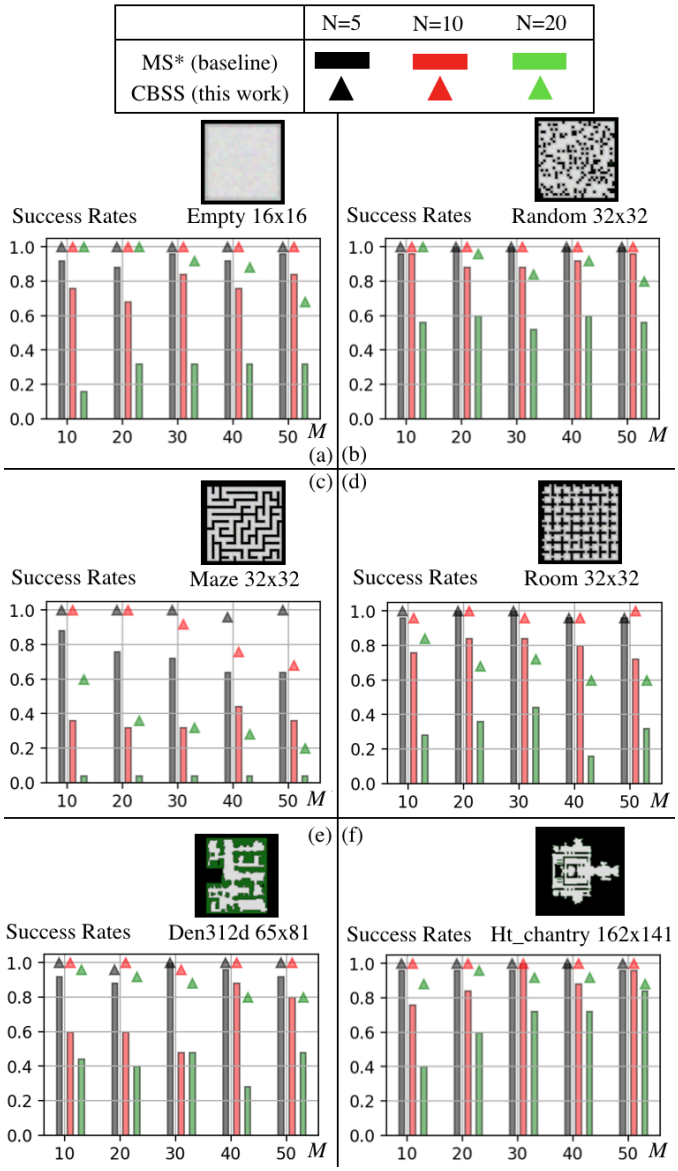

Fig. 5. Numerical results of CBSS (this work) and MS* (baseline). Color indicates different numbers of agents (N) and the x-axis represents the number of targets (M). CBSS (the triangles markers) achieves up to 60% higher success rates than MS* (the bars) within the one minute runtime limit.

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.

3) Number of Nodes Expanded: As shown in Fig. 6 (d), increasing ϵ leads to more nodes expansion in general. Combined with Fig. 6 (b) and (c), it indicates that, with a large ϵ, CBSS tries to find a solution by following the joint sequences that have been generated and defers the (expensive) computation of the next-best joint sequence, which 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 is possible that the generated joint sequences lead to many

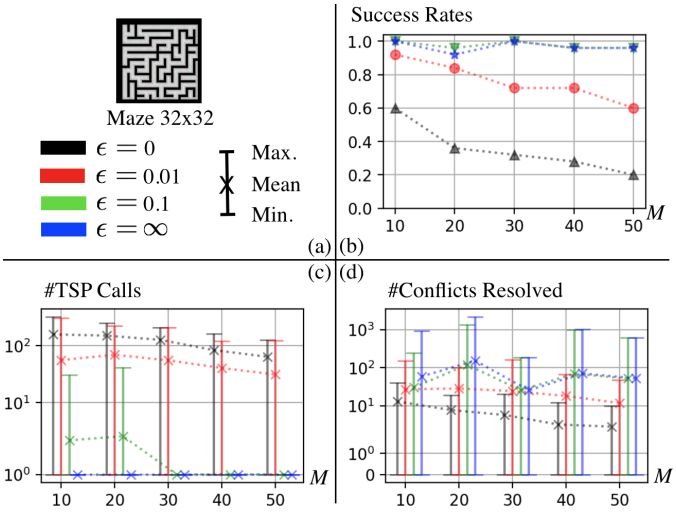

Fig. 6. Numerical results of CBSS with different sub-optimality bounds ϵ. CBSS with $\epsilon = \infty$ is the fourth baseline, the sequential method. (b) shows the success rates. (c) shows the number of TSP solver calls. (d) shows the number of nodes expanded by CBSS. There is a trade-off between solution 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 runtime limit.

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 vs Sequential Method

This section compares CBSS against the sequential method. As shown in Fig. 6, this baseline (in blue) performs similarly to CBSS with $\epsilon = 0.1$. In Fig. 6 (b) when $M = 20$, there is an instance where CBSS with $\epsilon = 0.1$ succeeds and the baseline fails. The reason is that this baseline method computes only one joint sequence and then plans paths by following the joint sequence, which may lead to a large number of conflicts. In contrast, CBSS with $\epsilon = 0.1$ (or other finite ϵ) can generate a next-best joint sequence when needed. It is also worthwhile to note that these cases exist but rarely occur. The condition under which these cases happen remain an open question and is worthwhile investigation in the future work.

Furthermore, we compare the solution cost computed by CBSS and the sequential method. As shown in Table I, CBSS with $\epsilon = 0.01$ often computes (slightly) cheaper solutions than the baseline in the Maze map, and in the Random map, both methods often compute solutions of the same cost. The result indicates that, for most instances in practice, the sequential method can compute optimal or near-optimal solutions. However, it fails to provide any solution quality guarantee, which motivates us to develop the anytime and adaptive variants of CBSS that can simultaneously achieve the relatively high success rates as this baseline does while providing tight sub-optimality bounds.

E. CBSS Variants

(a)	Maze 32x32, $N = 20$		
M	Total Succ. Inst.	% of Inst.	Median Cost Diff.
10	23	47.8%	4 (1.2%)
20	19	47.4%	4 (1.0%)
30	18	22.2%	7 (1.5%)
40	17	29.4%	3 (0.8%)
50	15	6.7%	10 (2.3%)

(b)	Random 32x32, $N = 20$		
M	Total Succ. Inst.	% of Inst.	Median Cost Diff.
10	25	16.0%	3 (1.5%)
20	25	8.0%	3 (1.3%)
30	25	4.0%	4 (1.6%)
40	25	8.0%	3 (1.1%)
50	23	0%	0 (1.2%)

TABLE I

THE SOLUTION COSTS COMPARISON BETWEEN CBSS WITH $\epsilon = 0.01$ (WHOSE SOLUTION COST IS DENOTED AS C_1) AND THE (BASELINE) SEQUENTIAL METHOD (WHOSE COST IS DENOTED AS C_2) IN TWO MAPS WITH 20 AGENTS ($N = 20$). FROM LEFT TO RIGHT, THE FIRST COLUMN SHOWS THE NUMBER OF TARGETS (M) AND THE SECOND COLUMN SHOWS THE TOTAL NUMBER OF INSTANCES WHERE BOTH METHODS SUCCEED. AMONG THESE COMMONLY SUCCEEDED INSTANCES, THE "% OF INST" COLUMN SHOWS THE PERCENTAGE OF INSTANCES WHERE $C_1 < C_2$. AMONG THESE $C_1 < C_2$ INSTANCES, THE LAST COLUMN SHOWS THE MEDIAN NUMBER OF THE ABSOLUTE SOLUTION COST DIFFERENCE AS WELL AS THE MEDIAN NUMBER OF THE RATIO $(C_2 - C_1)/C_1$ IN THE PARENTHESES.

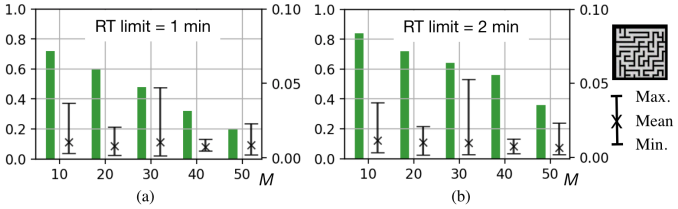

Fig. 7. Numerical results of anytime CBSS. The bar plots are against the left vertical axis and show the percentage of instances whose solutions are improved before timeout. The error bars are against the right vertical axis and show the amount of improvement. This result verifies that anytime CBSS can improve solution quality before the runtime limit is reached.

1) Anytime CBSS: To verify the idea of anytime CBSS, we implement the algorithm as follows. We begin by using $\epsilon = 0.1$ and reduce ϵ down to 0.01 after the first solution is found. If anytime CBSS improves its solution before timeout, the amount of improvement is measure by $(cost(\pi_{first}) - cost(\pi_{last}))/cost(\pi_{last})$, where π_{first} denote the first solution that is found while π_{last} denote the last solution that is returned when the algorithm terminates. We test in the Maze map and the result is shown in Fig. 7. When the runtime limit increases from 1 minute (Fig. 7 (a)) to 2 minutes (Fig. 7 (b)), there is an increasing percentage of instances, where the first solution computed by anytime CBSS is improved before termination, and the solution quality improvement is up to 0.05 (5%). This result verifies that anytime CBSS can improve solution quality before the time budget depletes.

2) Adaptive CBSS: To verify the idea of adpative CBSS, we count the runtime $T_{\gamma_1^*}$ of computing γ_1^* and set $\epsilon = T_{\gamma_1^*}/60$. We test in two maps as shown in Fig. 8. Overall, high success rates are maintained as the number of targets (M) increases,

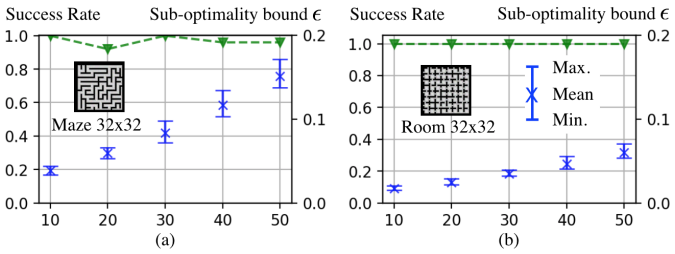

Fig. 8. Numerical results of the adaptive CBSS. The green curves are against the left vertical axis and show the success rates under a runtime limit of one minute. The error bars are against the right vertical axis and show the sub-optimality bound ϵ. This result verifies that adaptive CBSS can adjust the sub-optimality bound based on the difficulty of the instances while maintaining high success rates.

while the sub-optimality bound ϵ increases based on $T_{\gamma_1^*}$.

F. 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 is 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 and 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 target 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 handle arbitrary forms of assignment constraints). We report in Fig. 9 both the success rate and the cost ratio, which is defined as

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

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

1) Success Rates: The greedy method achieves higher success rates than CBSS in general, especially when M is large (e.g. in Fig. 9 (c) and (d)). This is expected since a larger M leads to TSPs with more vertices, which is in general computationally more expensive to solve. We observe from the data that, in Fig. 9 (b) with $M = 50$, CBSS cannot finish computing the first best joint sequence within the runtime limit, which demonstrates the computational burden for target sequencing. In contrast, the greedy method can compute a joint sequence quickly and achieve higher success rates.

There are also cases where CBSS achieves higher success rates than the greedy method (e.g. in Fig. 9 (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 agents, the greedy method has to resolve all conflicts before returning 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.

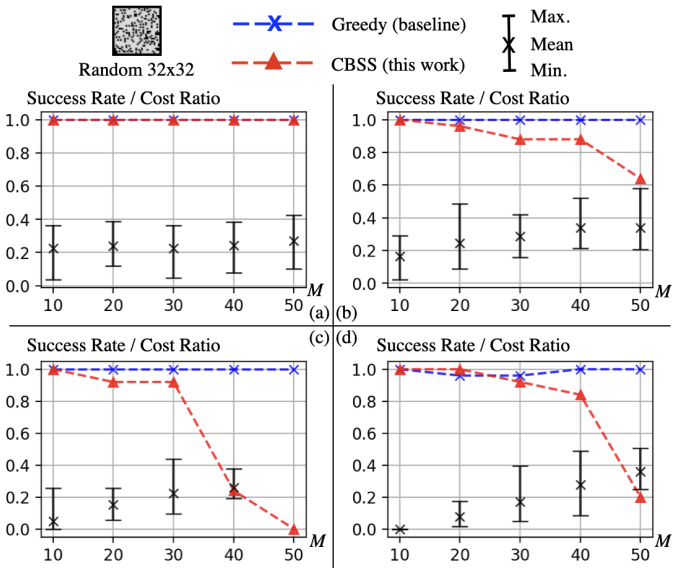

Fig. 9. Numerical results of CBSS and the greedy baseline. (a) shows the results of the anonymous case as described in Sec. VII-B. (b) (c) and (d) correspond to Case-1, Case-2, Case-3 of assignment constraints respectively as explained in the text. The greedy method achieves higher success rates than CBSS in general but suffers from higher solution costs.

2) Solution Cost: The black error bars in Fig. 9 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.

G. Gazebo Simulation and Physical Robot Tests

To verify that the planned joint path by CBSS can be executed on robots, we run simulation using Gazebo as well as physical robot tests using Robotarium [44], a remotely accessible multi-robot testbed. For Gazebo simulation, we describe the workspace using a four-neighbor grid-like graph and simulate $N = 10$ robots with $M = 20$ targets and 10 destinations as shown in Fig. 10.[12] For Robotarium, we test with $N = 4$ mobile robots and $M = 12$ targets. We demonstrate the test in the multi-media attachment. In this paper, we do not explicitly consider the uncertainty in the robot motion or the environment, which is itself an important research area (e.g. [45]) and is beyond the scope of this paper.

[12]The Gazebo warehouse environment is from https://github.com/belal-ibrahim/dynamic_logistics_warehouse, and the RosBot2 is from https://github.com/husarion/rosbot_description.

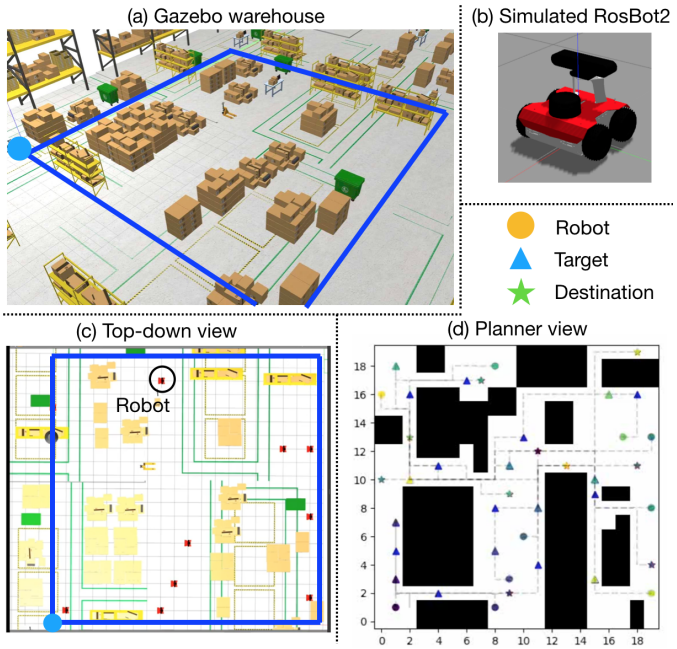

Fig. 10. Simulation using Gazebo and ROS. We describe the workspace using a four-neighbor grid-like graph and simulate 10 robots with 20 targets and 10 destinations.

VIII. Conclusion and Future Work

This paper formulates a problem called Multi-Agent Combinatorial Path Finding (MCPF), which requires both planning collision-free paths for multiple agents as MAPF requires, as well as allocating targets to agents and finding visiting order of targets for agents as mTSP requires. We develop a new approach Conflict-Based Steiner Search (CBSS) to solve MCPF with optimality guarantees. CBSS is a general and flexible approach 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 runtime, to computing unbounded sub-optimal solutions quickly. Second, different target sequencing procedures (e.g. greedy sequencing, K-best sequencing) can be used together with CBSS, trading off solution quality for runtime efficiency. Third, we develop the anytime and adaptive variants of CBSS, which trade off computational efficiency for solution quality. This paper 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 in the environment (e.g. by leveraging [45]), which is not considered in the current paper. In addition, one can also expedite CBSS by leveraging approximation or heuristic target sequencing methods, or by incorporating CBS-related techniques to speed up the multi-agent path planning in CBSS.

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

[4] 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.

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

[6] 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.

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

[8] 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.

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

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

[11] 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.

[12] Z. Ren, S. Rathinam, and H. Choset, "Conflict-Based Steiner Search for Multi-Agent Combinatorial Path Finding," in *Proceedings of Robotics: Science and Systems*, New York City, NY, USA, June 2022.

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

[14] D. Silver, "Cooperative pathfinding." 01 2005, pp. 117–122.

[15] 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.

[16] L. Cohen, T. Uras, T. S. Kumar, and S. Koenig, "Optimal and bounded-suboptimal multi-agent motion planning," in *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[17] 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.

[18] ——, "A conflict-based search framework for multiobjective multiagent path finding," *IEEE Transactions on Automation Science and Engineering*, pp. 1–13, 2022.

[19] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, "The traveling salesman problem," in *The Traveling Salesman Problem*. Princeton university press, 2011.

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

[21] 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.

[22] M. Brazil, R. L. Graham, D. A. Thomas, and M. Zachariasen, "On the history of the euclidean steiner tree problem," *Archive for History of Exact Sciences*, vol. 68, no. 3, pp. 327–354, 2014.

[23] J. Rodríguez-Pereira, E. Fernández, G. Laporte, E. Benavent, and A. Martínez-Sykora, "The steiner traveling salesman problem and its extensions," *European Journal of Operational Research*, vol. 278, no. 2, pp. 615–628, 2019.

[24] 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, no. 6, pp. 747–753, 2007.

[25] 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.

[26] 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.

[27] H. Ma, S. Koenig, N. Ayanian, L. Cohen, W. Hönig, T. S. Kumar, T. Uras, H. Xu, C. Tovey, and G. Sharon, "Overview: Generalizations of multi-agent path finding to real-world scenarios," *arXiv preprint arXiv:1702.05515*, 2017.

[28] 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.

[29] 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.

[30] L. A. Wolsey, *Integer programming*. John Wiley & Sons, 2020.

[31] 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.

[32] 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.

[33] H. Ma, J. Li, T. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *Conference on Autonomous Agents & Multiagent Systems*, 2017.

[34] 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.

[35] Q. Xu, J. Li, S. Koenig, and H. Ma, "Multi-goal multi-agent pickup and delivery," *arXiv preprint arXiv:2208.01223*, 2022.

[36] 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.

[37] Z. Chen, J. Alonso-Mora, X. Bai, D. D. Harabor, and P. J. Stuckey, "Integrated task assignment and path planning for capacitated multi-agent pickup and delivery," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5816–5823, 2021.

[38] 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.

[39] Z. Ren, S. Rathinam, and H. Choset, "Multi-objective conflict-based search for multi-agent path finding," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021.

[40] P. Oberlin, S. Rathinam, and S. Darbha, "A transformation for a heterogeneous, multiple depot, multiple traveling salesman problem," in *2009 American Control Conference*. IEEE, 2009, pp. 1292–1297.

[41] 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.

[42] M. Phillips and M. Likhachev, "Sipp: Safe interval path planning for dynamic environments," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 5628–5635.

[43] Z. Ren, S. Rathinam, and H. Choset, "Multi-objective conflict-based search using safe-interval path planning," *arXiv preprint arXiv:2108.00745*, 2021.

[44] S. Wilson, P. Glotfelter, L. Wang, S. Mayya, G. Notomista, M. Mote, and M. Egerstedt, "The robotarium: Globally impactful opportunities, challenges, and lessons learned in remote-access, distributed control of multirobot systems," *IEEE Control Systems Magazine*, vol. 40, no. 1, pp. 26–44, 2020.

[45] 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 (Student Member, IEEE) received the Ph.D. and M.S. degree from Carnegie Mellon University, Pittsburgh, PA, USA, and the dual B.E. degree from Tongji University, Shanghai, China, and FH Aachen University of Applied Sciences, Aachen, Germany. He is currently a Postdoctoral Fellow at Carnegie Mellon University.

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.