Robust Multiagent Combinatorial Path Finding

Yehonatan Kidushim, Avraham Natan, Roni Stern, Meir Kalech

The Faculty of Computer Science and Information
Ben-Gurion University of the Negev
kidushim@post.bgu.ac.il, avinat123@gmail.com, roni.stern@gmail.com, kalech@bgu.ac.il

Abstract

Consider a system of multiple physical agents tasked with collaboratively collecting a set of spatially distributed goals as quickly as possible while avoiding collisions with the environment and with each other. This type of problem, which involves Multi-Agent Path Finding (MAPF) and task allocation, is called Multi-Agent Combinatorial Path Finding (*MCPF*). Prior work on *MCPF* assumed each agent has a final goal it must reach, there are no orientation constraints on the agents' movements, and the agents will follow their planned actions as intended. These assumptions rarely hold in real physical robots, which limits the applicability of existing *MCPF* algorithms in practical applications. We propose the Robust CBSS framework, a robust planning approach that solves *MCPF* without the aforementioned simplifying assumptions, and provide two implementations: a baseline version (*RCbssBase*) and an efficient version (*RCbssEff*). *RCbssEff* generalizes the Conflict-Based Steiner Search (CBSS) algorithm, building on ideas from the *p*-Robust CBS algorithm and algorithms for solving the Equality Generalized Traveling Salesman Problem. We prove that *RCbssEff* is complete and can be configured to return optimal solutions. Experimental results on benchmark *MCPF* problems show that *RCbssEff* balances planning time, solution cost, and collision reduction compared to baselines.

Code, Datasets —
https://github.com/yehonatan280198/RobustMCPF

Introduction

Multi-Agent Path Finding (MAPF) aims to compute collision-free paths for multiple agents from their start locations to their goal destinations within a shared environment (Stern et al. 2019; Silver 2005). It is significant for applications such as robotics, warehouse automation, and traffic coordination, where efficient and safe movement of many agents is critical (Ma et al. 2017; Stern 2019; Boyarski et al. 2015). When multiple agents collectively cover a set of goals, the problem becomes not only one of finding collision-free paths but also of optimally assigning the goals to agents. This dual challenge is formalized as Multi-Agent Combinatorial Path Finding (*MCPF*) (Ren, Rathinam, and Choset 2023).

The Conflict-Based Steiner Search (CBSS) algorithm (Ren, Rathinam, and Choset 2023) extends the Conflict-Based Search (CBS) algorithm (Sharon et al. 2015) to solve *MCPF* by jointly exploring path planning and goal allocation. It builds a forest of constraint trees, each representing a distinct goal allocation, and incrementally resolves conflicts by adding constraints. However, CBSS ignores orientation constraints on agents' movement, presumes fixed destination goals for each agent, and assumes a deterministic environment without uncertainty, limiting its applicability in scenarios that violate these assumptions.

We propose the Robust Multi-Agent Combinatorial Path Finding framework (Robust CBSS), which addresses key limitations of CBSS, namely, by incorporating orientation-aware movement models, supporting dynamic goal allocations without pre-assigned destinations, and integrating *p*-Robust CBS (Atzmon et al. 2020) to handle probabilistic execution delays. Robust CBSS constructs a search forest over goal sequence allocations using best-first search with conflict resolution. To promote optimality, it iteratively refines goal allocations and agent paths while also leveraging TSP-based solvers, with overall optimality bounded by the solver's optimality. On top of this framework, we present two algorithms that implement it. The first, *RCbssBase*, serves as a baseline, ranking allocations by shortest-path costs while ignoring orientation, using a TSP solver (Helsgaun 2017). The second, *RCbssEff*, improves the efficiency of *RCbssBase*, by incorporating orientation costs into the allocation process, using an E-GTSP solver (Helsgaun 2015).

We evaluate *RCbssBase* and *RCbssEff* on benchmark maps commonly used in the MAPF literature (Stern et al. 2019). Results show that *RCbssEff* scales well with the number of agents and goals and outperforms *RCbssBase*. We also assess the contribution of two core components: (a) *delay-aware planning* and (b) *orientation-aware planning*. Incorporating orientation constraints significantly reduces plan costs at runtime, while accounting for probabilistic delays decreases the number of replanning operations in complex environments.

Related Work

The field of MAPF aims to compute cost-minimizing paths for multiple agents from their start to goal locations. Despite significant progress, key challenges remain. In this work,

we focus on three: *joint goal allocation and path planning*, *planning under orientation constraints*, and a restricted form of *planning under uncertainty*.

CBSS (Ren, Rathinam, and Choset 2023) and MS* (Ren, Rathinam, and Choset 2021) jointly address goal allocation and path planning, allowing agents to visit multiple goals. CBSS builds on CBS (Sharon et al. 2015), while MS* builds on M* (Wagner and Choset 2011). Both assume deterministic environments and ignore orientation constraints, limiting their applicability in realistic settings.

Planning under orientation constraints has been studied for MAPF. Zhang et al. (2023) investigate MAPF_T, where agents have orientation and turn costs, adapting CBS to this setting. MAPF-POST is a post-processing algorithm that converts kinematics-agnostic plans (e.g., ignoring orientation) into ones that respect motion constraints (Hönig et al. 2016). While both improve realism, they support only single-goal allocations, and MAPF_T also assumes a deterministic environment.

Different forms of uncertainty have been studied in MAPF. UM* handles state uncertainty via belief-space planning but does not guarantee bounded collision probabilities (Wagner and Choset 2017). pR-CBS extends CBS with probabilistic robustness, ensuring collisions remain below a predefined threshold even with delays (Atzmon et al. 2020). MAPF under Obstacle Uncertainty (MAPFOU) addresses incomplete knowledge of traversability, assuming deterministic actions once the environment is known (Shofer, Shani, and Stern 2023). MAPF with Time Uncertainty deals with bounded edge traversal times (Shahar et al. 2021) . However, none of these approaches address the combinatorial challenge of allocating multiple goals to agents.

Problem Definition

MAPF involves a set of agents, each with a start and a goal location. The task is to find non-conflicting paths that move all agents from their starts to their goals. *MCPF* generalizes MAPF by allowing different numbers of agents and goals and allocations are unknown. Let $A = \{a_1, \ldots, a_n\}$ denote n agents operating in an environment modeled as a directed graph $G = (V, E)$, where each vertex $v \in V$ represents a unique location. Time is discretized, and at each time step an agent is in a *configuration* c, which includes its location $\text{loc}(c)$ and orientation.

At each time step, an agent may perform one of three actions: *Wait*, *Move*, or *Rotate*. *Wait* leaves the configuration unchanged, *Rotate* changes the agent's orientation without changing its location, and *Move* changes the location to an adjacent vertex in G, possibly altering other configuration properties. The available actions at each time step depend on the previous configuration. For example, if the configuration includes an *orientation*, an agent may only move to certain adjacent vertices based on its facing direction, requiring a prior *Rotate*. Each agent $a_i \in A$ starts in an initial configuration s_i, and $V_g \subset V$ denotes m distinct *goal* locations.

A single-agent plan π^i for agent a_i is a sequence of actions it can perform when starting from its initial configuration (s_i). We denote by $\pi^i(t)$ as the configuration reached after performing the t^{th} action in π^i. The cost of a single-agent plan π^i, denoted $cost(\pi^i)$ is defined as the number of actions in π^i. A conflict $\langle a_i, a_j, x, t\rangle$ between a pair of single-agent plans π^i and π^j occurs if agents a_i and a_j ($a_i \neq a_j$) occupy the same location x at time step t, i.e., when $loc(\pi^i(t)) = loc(\pi^j(t)) = x$, or when they traverse the same edge x in opposite directions from time step $t-1$ to time step t, i.e., when $(loc(\pi^i(t-1)) = loc(\pi^j(t))) \wedge (loc(\pi^i(t)) = loc(\pi^j(t-1)))$, where $(loc(\pi^i(t-1)), loc(\pi^i(t))) = x$. The former is referred to as a vertex conflict and the latter as a swapping conflict.

A solution π is a set of single-agent plans, one for each agent. The cost of a solution is the sum of costs of its constituent single-agent plans: $cost(\pi) = \Sigma_{i\in\{1,\ldots,n\}} cost(\pi^i)$. A solution π is considered conflict-free if it does not include a pair of single-agent plans that have a conflict.

In this work, we assume an agent remains at its current vertex instead of advancing to the next vertex in its planned path with some fixed probability p_{delay}, referred to as the *delay probability*. Consequently, conflicts may still arise during execution, even if the planned path π is conflict-free by design. Guaranteeing complete safety when delays may occur is often not feasible, as it requires finding vertex disjoint paths for the agents or online communication protocols. Instead, we follow prior work (Atzmon et al. 2020) and aim for a p-robust solution.

Definition 1 (**p-Robust Solution**). *A solution π is said to be p-robust if the probability that it is executed without any conflict is at least p_{safe}, where $p_{safe} \in [0, 1]$, with statistical confidence level $1 - \alpha$.*

The robust *MCPF* problem is defined as follows:

Definition 2 (**p-robust multi-agent combinatorial pathfinding problem**). *Given a set of n agents, each with an initial configuration s_i and a delay probability p^i_{delay}, a required safety threshold p_{safe}, a significance level α, and a set of goal locations $V_g = \{v_{g_1}, \ldots, v_{g_m}\}$. The p-Robust combinatorial pathfinding problem is to find a solution π such that: (1) each goal v_{g_j} is visited by at least one agent, and (2) the solution π is p-robust. An optimal solution for this problem is one that minimizes the solution cost.*

Methodology

This section introduces our approach in stages: **(1)** we review CBS and its extension CBSS, **(2)** present our enhanced Robust CBSS framework, and **(3)** describe its two implementations: the baseline *RCbssBase* and a more efficient algorithm *RCbssEff*. We focus on addressing two key limitations of CBSS: (i) its assumption that each agent is assigned a fixed destination goal, and (ii) its inability to handle orientation-dependent movement. Finally, **(4)** we show how these enhancements are integrated with probabilistic robustness to handle execution-time uncertainty, yielding the full p-robust multi-agent combinatorial pathfinding solution.

To illustrate key ideas, we use a running example with three agents and two goals in a grid environment, where agents are subject to orientation constraints. For simplicity, In this example, we assume each agent's configuration in-

cludes its location and an orientation, which can be north, south, east, or west.

Background: CBS and CBSS

CBS (Sharon et al. 2015) is a two-level search algorithm. At the **high level**, it explores a Constraint Tree (CT), where each node $Z = (\pi, cost(\pi), \mathcal{C})$ consists of a solution $\pi = (\pi^1, \dots, \pi^n)$, its total cost, and a set of constraints $\mathcal{C}$. Each constraint $(a_i, x, t) \in \mathcal{C}$ prohibits agent a_i from occupying vertex or edge x at time t. The root node Z_{root} uses empty constraints and paths planned independently by a **low-level** single-agent planner. Nodes are stored in OPEN, a priority queue ordered by cost. In each iteration, CBS pops the lowest-cost node, validates its solution, and if no conflict is found, returns it as optimal. Otherwise, for a detected conflict (a_i, a_j, x, t), CBS generates two child nodes by adding to $\mathcal{C}$ either (a_i, x, t) or (a_j, x, t). For each child, the low-level planner replans the path of the affected agent under the updated constraints, and the new nodes are inserted into OPEN for further expansion.

CBSS (Ren, Rathinam, and Choset 2023) extends CBS to solve *MCPF* problems by jointly handling goal allocation and path planning. It does so by modifying the high-level search to construct a forest of constraint trees (CTs), each corresponding to a fixed goal sequence allocation γ.

Definition 3 (Goal sequence allocation (γ)). *A goal sequence allocation $\gamma = \{\gamma^i\}_{i=1}^n$ assigns each agent a_i an ordered sequence of goals $\gamma^i = (g_1, \dots, g_h)$. Let $d(u, v)$ denote the shortest-path cost between vertices u and v on graph G. The cost of γ^i is defined as $cost(\gamma^i) = d(s_i, g_1) + \sum_{p=1}^{h-1} d(g_p, g_{p+1})$, and $cost(\gamma) = \sum_{i \in A} cost(\gamma^i)$.*

The problem of finding γ is equivalent to solving a Multiple Traveling Salesman Problem (mTSP). CBSS enumerates $\langle \gamma_1, \gamma_2, \dots \rangle$ in ascending cost. For each γ_K, it builds a constraint tree CT_K whose nodes share this allocation. To obtain each γ_K, CBSS employs the *K-Best-Sequencing* method, which generates the next allocation through a three-step process: **1)** Reduce the problem of finding γ (an mTSP) to an equivalent single-agent TSP. **2)** Apply the *K-Best-TSP* method to enumerate the K lowest-cost tours, starting from the lowest-cost and progressively generating the next by partitioning the search: for each selected tour, we traverse its edges in order, and for each edge create a subproblem that forbids including this edge while enforcing all preceding edges. From all subproblems, the next lowest-cost tour is selected, and the process repeats until K tours are obtained. **3)** Convert the K^{th} lowest-cost tour into a goal sequence allocation γ_K. Then explore its CT using CBS: resolve conflicts via constraint splitting and use the low-level planner to compute for each agent a_i a tour satisfying all constraints and visiting its goals γ_K^i.

The search begins by generating a node based on γ_1, which becomes the root of CT_1 and is added to OPEN. During the search, a node Z is popped from OPEN. Let K denote the number of root nodes generated so far. If $cost(Z.\pi) \leq cost(\gamma_K)$, the algorithm expands Z; otherwise, it creates a new tree CT_{k+1} by computing γ_{k+1} via the *K-Best-Sequencing* method. The low-level planner then computes its solution π_{newRoot}, and initializes the root node Z_{newRoot} for CT_{k+1}. Between Z and Z_{newRoot}, the node with lower cost is selected for expansion, while the other is returned to OPEN. If the selected node's solution π is conflict-free, the search terminates and returns π; otherwise, two child nodes via constraint splitting are added to OPEN.

Robust CBSS Framework

To overcome the main limitations of CBSS, we introduce the *Robust CBSS* framework (Algorithm 1). We first address its handling of *orientation-dependent movement*, refining how allocations are ranked and expanded to ensure correctness when rotation costs associated with agent orientation are considered.

Algorithm 1: Robust CBSS framework

Input: G, p_{delay}, P_{safe}, α

1 $K \leftarrow 1$
2 $\gamma_1 \leftarrow$ K-Best-Sequencing(G, K)
3 $\mathcal{C} \leftarrow \emptyset$
4 $\pi \leftarrow$ LowLevelPlan$(\gamma_1, \mathcal{C})$
5 Add $Z_{root} = (\pi, cost(\pi), \mathcal{C})$ to OPEN
6 **while** *OPEN not empty* **do**
7 $Z \leftarrow$ OPEN.pop()
8 **if** $Z.cost(\pi) > cost(\gamma_K)$ **then**
9 $K = K + 1$
10 $\gamma_K \leftarrow$ K-best-Sequencing(G, K)
11 $\pi \leftarrow$ LowLevelPlan$(\gamma_K, \emptyset)$
12 $Z_{\text{newRoot}} = (\pi, \text{cost}(\pi), \emptyset)$
13 Add Z_{newRoot} and Z to OPEN
14 Continue
15 **if** $Z.\pi$ *has no conflicts* **then**
16 **return** $Z.\pi$
17 **else**
18 ResolveConflicts$(Z, OPEN)$
19 **return** *failure*

In standard CBSS, *K-Best-Sequencing* ranks allocations based solely on spatial distances, ignoring orientation and turning costs. This worked because the estimated allocation cost $cost(\gamma)$ (derived from the TSP) was identical to the actual solution cost $cost(\pi)$ produced by the low-level planner without constraints. When orientation is considered, this correspondence breaks: the true cost $cost(\pi)$ can be significantly higher than $cost(\gamma)$, misleading the search and potentially causing it to overlook better allocations.

To address this, we treat $cost(\gamma)$ as a lower bound rather than a direct estimate of the true cost. To preserve correctness under this weaker heuristic, the algorithm takes a conservative strategy: instead of expanding only the lower-cost node between Z and Z_{newRoot}, both are reinserted into OPEN and the search restarts (Lines 13–14 in Alg. 1). A node is checked for conflicts only when $cost(\pi) \leq cost(\gamma_k)$, meaning that the cost of the current solution π is lower than or

equal to the lower bound of any unexplored allocation. Thus, no other allocation can be expected to yield a cheaper solution, making π the most promising candidate at this stage.

Next, to address CBSS's second limitation — assuming each agent has a fixed destination goal. We introduce a new transformation integrated into *K-Best-Sequencing*'s initialization (Lines 2 and 10 in Alg. 1). This transformation is implemented in two variants: *RCbssBase* and *RCbssEff*, both of which retain the enhancement of Robust CBSS for handling orientation-dependent movement.

Robust CBSS Baseline (*RCbssBase*)

To address CBSS's limitation in handling *unassigned destination goals*, we introduce *RCbssBase*. It extends *K-Best-Sequencing* (Lines 2 and 10 in Algorithm 1) with a transformation that reduces the problem of finding γ (an mTSP) to an equivalent single-agent TSP, enabling flexible allocations, including cases where some agents stay in place.

Step 1: We construct a directed graph $G' = (V', E')$ with vertices $V' = V_0 \cup V_g$, where $V_0 = \{s_i \mid i = 1, \ldots, n\}$ are the agents' initial configurations and $V_g = \{v_{g_j} \mid j = 1, \ldots, m\}$ are the goals. Edges E' include two types: **Type-1** non-zero cost edges, representing the shortest-path travel cost in G between s_i and v_{g_j} and between pairs $(v_{g_j}, v_{g_{j'}})$, ignoring rotation costs. **Type-2** zero cost auxiliary edges, added only to ensure connectivity, linking pairs $(s_i, s_{i'})$ and connecting each v_{g_j} back to every s_i. These Type-2 edges are crucial for concatenating all agents' tours into a single continuous TSP tour, enabling the use of an off-the-shelf single-agent TSP solver. Without them, many valid allocations could not be represented in this formulation. The resulting graph after this step is illustrated in Figure 1 (Left).

Step 2: We apply *K-Best-TSP* to compute the K-lowest-cost tours on G'. As described in Section *Background: CBS and CBSS*, this procedure enumerates the K-lowest-cost tours by partitioning the search space: for each edge e_p in a selected tour, it defines a subproblem that forbids including e_p while enforcing all preceding edges $\{e_1, \ldots, e_{p-1}\}$, from which the next lowest-cost tour is selected. In our formulation, we restrict this partitioning process to Type-1 edges, as they correspond to meaningful agent-to-goal assignments. Importantly, Type-2 edges are not removed from the tours, they remain part of every computed tour because they maintain the structural continuity that allows concatenating all agents' tours into a single TSP path. In other words, the partitioning considers only edges e_p of Type-1, since branching on Type-2 edges would not produce new allocations but only duplicate representations of the same allocation. This refinement prevents generating multiple tours that correspond to the same allocation. An example of a K^{th} lowest-cost tour resulting from this step is shown in Figure 1 (Middle).

Step 3: Finally, transform the selected K^{th} lowest-cost tour in G' into a goal sequence allocation γ_K by removing all Type-2 edges. These edges, used to maintain tour connectivity in the TSP formulation, are no longer needed; their removal leaves the meaningful agent-to-goal sequences that define the final allocation. An example of the resulting allocation is shown in Figure 1 (Right).

While *RCbssBase* enables flexible goal allocations and orientation-aware movement, it still inherits a key limitation: the lower bound $cost(\gamma)$ from *K-Best-Sequencing* remains purely distance-based and fails to reflect true costs under orientation constraints. Consequently, allocations that seem promising under this bound may lead to inefficient solutions, causing unnecessary expansions and slowing the search. This motivates a more informed allocation strategy that better approximates true solution costs, tightening the bound and accelerating the search.

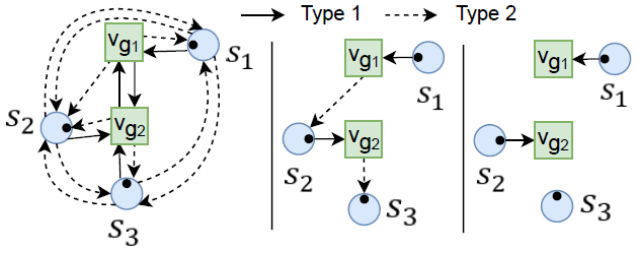

Figure 1: (Left) G' after Step 1; (Middle) K^{th} lowest-cost tour after Step 2; (Right) corresponding γ_K after Step 3.

Robust CBSS Efficient (*RCbssEff*)

We propose *RCbssEff*, which tightens the allocation bound by incorporating rotation costs associated with agent orientation directly at the allocation stage. We reformulate the problem as an instance of the *Equality Generalized Traveling Salesman Problem* (E-GTSP) (Helsgaun 2015), where vertices are partitioned into clusters and the objective is to find a minimum-cost tour visiting exactly one vertex per cluster. Here, each cluster corresponds to a goal and its vertices represent orientation-specific variants of reaching it, ensuring that computed tours capture orientation-dependent travel costs. To implement this, we update the *K-Best-Sequencing* within the *Robust CBSS* framework (Alg. 1) to solve an E-GTSP instance instead of a standard TSP.

Step 1: We construct a directed graph $G' = (V', E', C)$ from the environment graph G. The vertex set is $V' = V_0 \cup V'_g$, where $V_0 = \{s_i \mid i = 1, \ldots, n\}$ are the agents' initial configurations. The second set V'_g contains orientation-specific vertices, each combining a goal with an achievable orientation. Formally, let $X(g_j)$ be the set of orientations in which goal g_j is reached. For each $v_{g_j} \in V_g$ and $x \in X(g_j)$, we add a vertex $v_{g_{j,x}} \in V'_g$. As in our running example, with four orientations (north, south, east, west), each goal g_j yields four vertices $v_{g_{j,N}}, v_{g_{j,S}}, v_{g_{j,E}}, v_{g_{j,W}}$, each representing the goal reached with an orientation.

The cluster set $C = C_{V_0} \cup C_{V_g}$ defines the grouping for the E-GTSP. C_{V_0} contains singleton clusters $C_0^i = \{s_i\}$ for each initial configuration, and C_{V_g} groups the orientation-specific vertices of each goal into clusters $C_{g_j} = \{v_{g_{j,x}} : x \in X(g_j)\}$. These clusters enforce that each goal is visited in exactly one orientation.

Edges E' include two types: **Type-1** non-zero cost edges, representing the shortest-path travel cost in G between $s_i \in V_0$ and $v_{g_{j,x}} \in V'_g$, or between $v_{g_{j,x}} \in C_{g_j}$ and $v_{g_{j',x'}} \in$

$C_{g_{j'}}$ with $j \neq j'$. **Type-2** zero cost auxiliary edges, added only to ensure connectivity, linking vertices within the same cluster, connecting all initial configurations, and linking goal configurations back to initial configurations. As in *RCbss-Base*, these Type-2 edges serve a structural role: they allow concatenating all agents' tours into a single continuous tour, enabling its formulation as an E-GTSP instance. The resulting graph after this step is illustrated in Figure 2 (Top).

Step 2: We replace the standard *K-Best-TSP* with *K-Best-EGTSP*, a new procedure that enumerates the K lowest-cost tours by solving successive E-GTSP instances. This extension incorporates the clustering constraints of the E-GTSP and embeds rotation costs associated with agent orientation directly into the allocation stage, ensuring that the generated tours more accurately reflect the true traversal costs.

To enumerate the K lowest-cost tours, we extend the restricted TSP (rTSP) procedure (Ren, Rathinam, and Choset 2023) to E-GTSP, introducing the *restricted E-GTSP (rEGTSP)*, which applies inclusion and exclusion constraints over clusters and orientation-specific vertices.

Definition 4 (**Restricted E-GTSP (rEGTSP)**). *Given a directed graph $G' = (V', E', C)$ and two disjoint sets of ordered cluster pairs I_c and O_c, the rEGTSP finds a minimum-cost tour visiting exactly one vertex from each cluster, subject to: 1. For every $(C_i, C_j) \in I_c$: the tour must include an edge (v_i, v_j) with $v_i \in C_i$, $v_j \in C_j$. 2. For every $(C_i, C_j) \in O_c$: the tour must exclude all such edges.*

To solve an rEGTSP, we construct a modified graph G'' from G' by adjusting edge costs as follows: **If** $(C_i, C_j) \in O_c$: set $cost(v_i, v_j) = M$ (a large penalty), effectively prohibiting the edge. **If** $(C_i, C_j) \in I_c$: set $cost(v_i, v_j) = -\big(M - cost_{G'}(v_i, v_j)\big)$, encouraging its inclusion while preserving the relative traversal cost. **Otherwise**: retain the original $cost_{G'}(v_i, v_j)$. This cost adjustment incentivizes including edges between required cluster pairs without fixing which edge is chosen. The solver can still select the most efficient option based on the true traversal costs $cost_{G'}(v_i, v_j)$, which is crucial when costs depend on orientation and may include rotation penalties. Embedding $cost_{G'}(v_i, v_j)$ in this way ensures that the transitions are ranked by actual efficiency rather than treating all valid edges equally.

After building G'', the rEGTSP procedure runs an E-GTSP solver on the modified graph. Once a tour is obtained, its cost is re-evaluated using the original costs from G', discarding the artificial penalties for inclusion edges. This guarantees that the final cost reflects the true traversal cost while respecting all inclusion and exclusion constraints.

Algorithm 2 presents the above process. The *K-Best-EGTSP* method receives K and G' as inputs, where K (provided by *K-Best-Sequencing* within the Robust CBSS framework) defines the number of tours to compute. It first solves an initial E-GTSP with empty inclusion and exclusion sets, adds the resulting tour $tour^0$ with its constraints to $OPEN_{kBEST}$, initializes $\mathcal{S}$ as the solution set, and sets $k = 1$ to track the number of tours (Lines 1–5). The main loop runs while $OPEN_{kBEST}$ is nonempty (Line 6). At each iteration of the main loop, the next lowest-cost tour and its constraints are extracted from $OPEN_{kBEST}$ and added to $\mathcal{S}$. If $k = K$, the algorithm returns $\mathcal{S}$ (Lines 7–10); otherwise, it extracts all type-1 edges from $tour^k$. For each type-1 edge (v_i, v_j) in $tour^k$, we identify its incident clusters C_i and C_j (with $v_i \in C_i$ and $v_j \in C_j$) and add the ordered pair (C_i, C_j) to C^k_{alloc}. This set captures the sequence of cluster-to-cluster transitions that define the current allocation structure (Line 11). The algorithm iterates over the indices p of C^k_{alloc}, constructing for each index new constraints: $I^{k+1}_{c_p}$ extends I^k_c with all pairs before p, and $O^{k+1}_{c_p}$ extends O^k_c with the pair at p, thereby enforcing earlier allocations while excluding the p-th allocation (Lines 12–14). An *rEGTSP* instance is then solved with these constraints (Line 15). If feasible - i.e., satisfying all constraints - the tour is added to $OPEN_{kBEST}$ (Lines 16–17). Finally, k is incremented at the end of each main loop iteration (Line 18). If $OPEN_{kBEST}$ is exhausted before finding K solutions, the algorithm reports failure, indicating fewer than K distinct E-GTSP tours exist (Line 19). An example of a tour resulting from this step is shown in Figure 2 (Bottom-left).

Algorithm 2: K-Best-EGTSP

Input: K,G'

1 $I^0_c \leftarrow \emptyset, O^0_c \leftarrow \emptyset$
2 $tour^0 \leftarrow$ rEGTSP(G', I^0_c, O^0_c)
3 Add $\langle I^0_c, O^0_c, tour^0\rangle$ into $OPEN_{kBEST}$
4 $\mathcal{S} \leftarrow \emptyset$
5 $k = 1$
6 **while** $OPEN_{kBEST}$ *not empty* **do**
7 $\quad (I^k_c, O^k_c, tour^k) \leftarrow OPEN_{kBEST}$.pop()
8 $\quad$ Add $tour^k$ into $\mathcal{S}$
9 $\quad$ **if** $k = K$ **then**
10 $\quad\quad$ **return** $\mathcal{S}$
11 $\quad C^k_{\text{alloc}} \leftarrow \langle (C_i, C_j) \mid (v_i, v_j) \in tour^k,\ v_i \in C_i,\ v_j \in C_j,\ \text{type}(v_i, v_j) = 1\rangle$
12 $\quad$ **for** $p \in \{1, \ldots, |C^k_{alloc}|\}$ **do**
13 $\quad\quad I^{k+1}_{c_p} \leftarrow I^k_c \cup \{C^k_{\text{alloc}}[q] \mid 1 \leq q < p\}$
14 $\quad\quad O^{k+1}_{c_p} \leftarrow O^k_c \cup \{C^k_{\text{alloc}}[p]\}$
15 $\quad\quad tour^{k+1}_p \leftarrow$ rEGTSP$(G', I^{k+1}_{c_p}, O^{k+1}_{c_p})$
16 $\quad\quad$ **if** $tour^{k+1}_p$ *is feasible* **then**
17 $\quad\quad\quad$ Add $\langle I^{k+1}_{c_p}, O^{k+1}_{c_p}, tour^{k+1}_p\rangle$ to $OPEN_{kBEST}$
18 $\quad k = k + 1$
19 **return** failure

Step 3: The K^{th} lowest-cost tour in G' is converted into a goal sequence allocation γ_K. Removing Type-2 edges splits the tour into n segments, each forming an agent's goal sequence. Orientation-specific vertices $v_{g_{j,x}}$ are mapped back to their original goals v_{g_j}, yielding n sequences over V_g (one per agent) while preserving the original solution cost. The resulting allocation is shown in Figure 2 (Bottom-right).

On our GitHub, we prove the correctness and completeness of the K-Best-Sequencing used in *RCbssEff* and based on this, prove that *RCbssEff*, based on the Robust

CBSS framework (Alg. 1), is both complete and optimal.

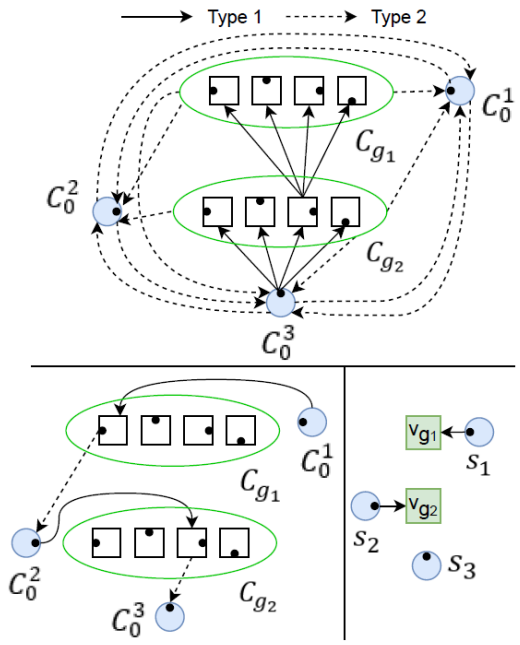

Figure 2: (Top) G' after Step 1 with goal clusters C_{g_j} and singleton clusters C_0^i; (Bottom-left) K^{th} lowest-cost tour after Step 2; (Bottom-right) corresponding γ_K after Step 3.

Robustness to Delays

Until now, our framework has assumed an idealized environment without delays, simplifying the core mechanics of the method but neglecting real-world complexities where delays from communication, computation, or physical dynamics are common and can critically affect performance.

This challenge was addressed by *p-Robust CBS* (Atzmon et al. 2020), which extends CBS by introducing *delay conflicts* - cases where agents deviate from planned timings due to stochastic delays. To resolve them, it generates two child CT nodes forbidding each agent from occupying x at time t, and a third allowing joint occupancy when the probability of simultaneous arrival is sufficiently low for a p-robust solution. Deterministic and Monte Carlo verification keep collision probabilities below a predefined safety threshold.

We integrate this probabilistic robustness into Robust CBSS with minimal modifications: the standard conflict check in Algorithm 1 Line 15 is replaced with the probabilistic delay verifier of (Atzmon et al. 2020), and the conflict resolution in Line 18 adopts its three-way splitting strategy. These adaptations elevate our method from a deterministic planner to one capable of producing p-robust solutions, broadening CBSS's applicability in stochastic environments.

Experimental Results

We evaluate *RCbssEff* on grid-based maps of varying sizes and structures from the benchmark suite of (Stern et al. 2019) (Figure 3). Our evaluation has three parts: (1) scalability analysis of *RCbssEff*; (2) performance comparison with *RCbssBase*; and (3) ablation studies on the impact of delay probabilities and orientation awareness in our framework.

Figure 3: Evaluation maps with varying sizes and densities.

(1) ***RCbssEff* Scalability**: We assess scalability by varying the number of agents ($n \in \{10, 15, 20, 25, 30, 40, 50\}$) and goals ($m \in \{20, 30, 40, 50, 60, 80, 100\}$), with $p_{\text{delay}} \in \{0.1, 0.3\}$ and safety thresholds $p_{\text{safe}} \in \{0.8, 0.9, 0.95\}$. The confidence level is fixed at $1 - \alpha = 0.95$ (Atzmon et al. 2020). For each parameter configuration, 20 instances are generated by randomly sampling agent and goal locations (with orientations) and solved within a 60-second limit.

(2) ***RCbssEff* vs. *RCbssBase***: We compare *RCbssEff* with *RCbssBase* across diverse settings, focusing on plan generation only (no execution simulation). The setup mirrors the scalability experiment, varying $n \in \{3, 5, 8, 10, 15, 20, 25\}$ and $m \in \{6, 10, 16, 20, 30, 40, 50\}$, with the same p_{delay}, p_{safe}, and α, number of instances per parameter configuration, and the time limit settings as above.

(3) **Ablation Study: Delay Awareness and Orientation Awareness**: We isolate the effects of (a) *delay-aware planning*, which incorporates probabilistic delays into conflict detection and resolution, and (b) *orientation-aware planning*, which accounts for rotation costs associated with agent orientation in planning. We simulate full plan execution with realistic delay and rotation modeling, triggering replanning whenever predicted collisions occur. Two sweeps are performed: (1) varying $n \in \{10, 25\}$ with $m \in \{10, 20, 30, 40, 50\}$, and (2) varying $n \in \{5, 10, 15, 20, 25\}$ with $m \in \{20, 50\}$, using the same p_{delay}, p_{safe}, α, number of instances per parameter configuration and time limit (planning + execution + replanning) settings as above.

Implementation. Python 3.10 on Ubuntu 24.04. Single-agent asymmetric E-GTSP instances were solved using GLKH-1.1 (Helsgaun 2015), while the *RCbssBase* baseline used LKH-3.0.11 (Helsgaun 2017). Experiments were run on a 16-core virtual machine (AMD EPYC 7702P). Statistical significance was assessed using a two-tailed independent samples t-test ($\alpha = 0.05$). Representative results are reported; full results and code are on GitHub.

Results. Table 1 shows *RCbssEff*'s performance as the number of goals increases. Success rates decrease with problem complexity, Maze degrading faster than the larger, sparser ht_chantry map due to its density and conflict poten-

tial. Interestingly, Maze maintains moderate success even at 100 goals, whereas ht_chantry map fails, likely due to timeouts rather than planning errors. These results show how map density and computation limits affect scalability.

Map	Number of goals	20	30	40	50	60	80	100
Maze	Success rate	0.94	0.85	0.75	0.68	0.67	0.65	0.61
	Avg runtime (s)	5.35	5.32	3.93	5.31	7.31	12.56	19.48
ht_chantry	Success rate	0.99	0.99	0.99	0.96	0.98	0.78	0.00
	Avg runtime (s)	7.82	12.82	19.14	26.52	35.33	55.68	–

Table 1: Success rate and runtime for varying numbers of goals (aggregated over all tested agent counts $n \in \{10, 15, 20, 25, 30, 40, 50\}$).

Figure 4 compares *RCbssEff* and *RCbssBase* on the **lak303d** map. *RCbssEff* achieves near-perfect success rates across all agent and goal counts, showing strong robustness to scaling. *RCbssBase* starts lower and drops sharply as agents or goals increase, failing entirely beyond 20 goals. Runtime grows for both methods with problem size, but *RCbssEff* remains stable with only mild increases. In contrast, *RCbssBase* 's runtime rises steeply, particularly as goals grow, reflecting higher computational overhead. Runtimes include only solved instances, likely underestimating *RCbssBase* 's true cost due to frequent failures. These results further emphasize the benefit of incorporating orientation-aware constraints at the allocation stage, enabling *RCbssEff* to maintain efficiency and success under more complex scenarios. All differences are statistically significant.

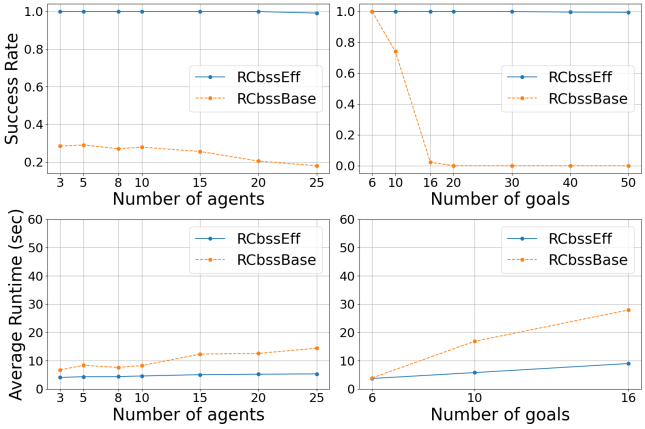

Figure 4: Success rate and runtime of *RCbssEff* vs. *RCbssBase* on lak303d map.

Ablation Study: Delay Awareness and Orientation Awareness We denote the ablation **i**gnoring **d**elay **p**robabilities as *IDP* and the one **i**gnoring **r**otation **c**osts as *IRC*. Figure 5 shows the execution Sum Of Cost (SOC) versus success rate for *RCbssEff*, *IDP*, and *IRC* on **ht_chantry** map. *IRC* consistently produces higher SOC than *RCbssEff*, confirming that ignoring rotation costs leads to underestimated plan costs and less efficient execution; the difference is statistically significant. *IDP* closely matches *RCbssEff* across the range, with no significant SOC difference.

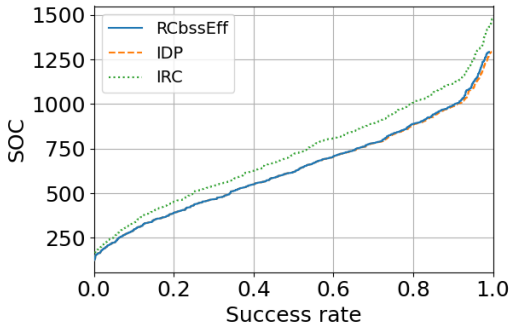

Figure 5: Execution SOC vs. success rate for *RCbssEff* and ablation variants on ht_chantry map.

Table 2 summarizes instances with more than one replanning call. *RCbssEff* triggers fewer replanning calls than ablations, indicating execution efficiency. Differences with *IDP* and *IRC* are minor, suggesting that ignoring delay probabilities or rotation costs has little effect on planning outcomes. Statistically significant differences appear between *RCbssEff* and *IDP* on **empty** and between *RCbssEff* and *IRC* on **random**; no other differences were significant.

Map	*IDP*	*IRC*	*RCbssEff*
den312d	1	1	1
den520d	0	1	1
empty-32-32	29	27	10
ht_chantry	0	8	0
lak303d	6	0	1
maze-32-32-2	27	3	12
random-32-32-20	6	6	3
room-32-32-4	9	3	3

Table 2: Cases with more than one replanning call per algorithm and map (instances where all algorithms succeeded).

Conclusion

We introduced *RCbssEff*, a robust algorithm for Multi-Agent Combinatorial Path Finding (*MCPF*). *RCbssEff* supports orientation-dependent movement, fully dynamic goal allocations, and robustness to stochastic delays. Our empirical evaluation demonstrates that *RCbssEff* not only scales effectively with the number of agents and goals but also achieves improved solution quality and robustness compared to the baselines. In particular, accounting for orientation during planning significantly reduces the cost of agents' paths, while incorporating delay probabilities during planning reduces the number of replanning operations at runtime only in complex environments. Future work will explore enhancing robust execution of *MCPF* by combining multi-agent diagnosis algorithms with safe execution mechanisms such as Action Dependency Graphs (ADG), aiming to reduce or even avoid replanning interruptions.

Acknowledgments

This research was funded by ISF grant No. 1238/23 to Roni Stern, and by the ministry of science grant No. 0006908/1001802443

References

Atzmon, D.; Stern, R.; Felner, A.; Sturtevant, N. R.; and Koenig, S. 2020. Probabilistic Robust Multi-Agent Path Finding. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1): 29–37.

Boyarski, E.; Felner, A.; Sharon, G.; Stern, R.; Wagner, G.; and Nathan, S. 2015. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 740–746.

Helsgaun, K. 2015. Solving the equality generalized traveling salesman problem using the Lin–Kernighan–Helsgaun Algorithm. *Mathematical Programming Computation*, 7(3): 269–287.

Helsgaun, K. 2017. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 12: 966–980.

Hönig, W.; Kumar, T.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Multi-agent path finding with kinematic constraints. In *International Conference on Automated Planning and Scheduling*, volume 26, 477–485.

Ma, H.; Li, J.; Kumar, T. S.; and Koenig, S. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, 837–845. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Ren, Z.; Rathinam, S.; and Choset, H. 2021. MS*: A New Exact Algorithm for Multi-agent Simultaneous Multi-goal Sequencing and Path Finding.

Ren, Z.; Rathinam, S.; and Choset, H. 2023. CBSS: A New Approach for Multiagent Combinatorial Path Finding. *IEEE Transactions on Robotics*, 39(4): 2669–2683.

Shahar, T.; Shekhar, S.; Atzmon, D.; Saffidine, A.; Juba, B.; and Stern, R. 2021. Safe Multi-Agent Pathfinding with Time Uncertainty. *Journal of Artificial Intelligence Research*, 70: 923–954.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219: 40–66.

Shofer, B.; Shani, G.; and Stern, R. 2023. Multi Agent Path Finding under Obstacle Uncertainty. *Proceedings of the International Conference on Automated Planning and Scheduling*, 33(1): 402–410.

Silver, D. 2005. Cooperative pathfinding. *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 117–122.

Stern, R. 2019. Multi-agent path finding–an overview. *Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures*, 96–115.

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T.; et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 151–158.

Wagner, G.; and Choset, H. 2011. M*: A complete multirobot path planning algorithm with performance bounds. In *IEEE/RSJ international conference on intelligent robots and systems*, 3260–3267.

Wagner, G.; and Choset, H. 2017. Path Planning for Multiple Agents under Uncertainty. *Proceedings of the International Conference on Automated Planning and Scheduling*, 27(1): 577–585.

Zhang, Y.; Harabor, D.; Bodic, P. L.; and Stuckey, P. J. 2023. Efficient Multi-Agent Path Finding with Turn Actions. In *Proceedings of the Sixteenth International Symposium on Combinatorial Search (SoCS)*.