Balancing Robustness and Efficiency in Multi-Agent Combinatorial Path Finding with Sum of Service Time

Anonymous Author(s)
Submission Id: 689

ABSTRACT

Consider a system of multiple physical agents tasked with collaboratively collecting a set of spatially distributed goals while avoiding collisions with the environment and with each other. This type of problem, which combines Multi-Agent Path Finding (MAPF) with task allocation, is known as Multi-Agent Combinatorial Path Finding (*MACPF*). Conflict-Based Steiner Search (CBSS) is an optimal algorithm for *MACPF*, which assumes that each agent has a fixed goal destination. It selects allocations that yield a solution minimizing the sum of costs (SOC), which we denote as CBSS$_{SOC}$. However, this objective is problematic in domains such as search and rescue, where timely service of all goals is more critical than minimizing SOC. We therefore propose CBSS$_{SST}$, which minimizes the Sum of Service Times (SST) across all goals using a novel mixed-integer linear programming allocation, thereby generalizing *MACPF* to settings without requiring fixed goal destinations. Since CBSS assumes perfect execution, we extend it with robust planning to handle stochastic execution delays. We propose two variants of CBSS$_{SST}$: Robust CBSS$_{SST}$ with Strict Verifier ($RCbss_T Strict$), which guarantees the desired robustness, and Robust CBSS$_{SST}$ with Anytime Verifier ($RCbss_T Anytime$), which addresses planning-time constraints by returning the most robust solution verified within the available time. Our experiments on *MACPF* benchmarks show that $RCbss_T Anytime$ solves substantially more instances than $RCbss_T Strict$ within the time limit, while reducing replanning effort and preserving robustness. These results demonstrate that $RCbss_T Anytime$ provides an effective and practical approach to *MACPF* under uncertainty.

KEYWORDS

Multi-Agent Path Finding, Multi-Agent Combinatorial Path Finding, Planning Under Uncertainty

ACM Reference Format:
Anonymous Author(s). 2026. Balancing Robustness and Efficiency in Multi-Agent Combinatorial Path Finding with Sum of Service Time. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 9 pages.

1 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 [20, 22]. It has practical applications in robotics, warehouse automation, and traffic coordination, which require efficient and safe movement of mobile physical agents [3, 10]. When multiple agents are required to visit a set of spatially distributed goals, for example autonomous robots that must rescue injured people located at different locations, the problem becomes not only one of finding collision-free paths but also of optimally assigning the goals to agents. The problem of solving this dual challenge is called Multi-Agent Combinatorial Path Finding (MACPF), where prior work typically assumes that each agent is pre-assigned a fixed goal destination, required to collect its assigned goals and finish there [14–16]. In contrast, we consider a more general setting in which all goals must be visited without fixing destinations for agents.

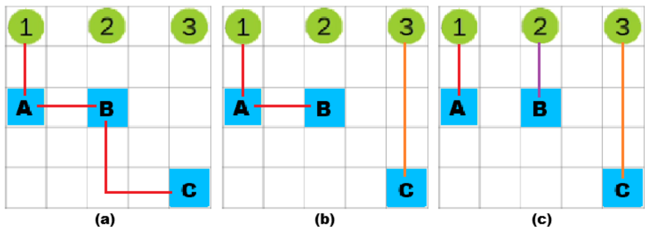

Figure 1: (a) minimizes SOC, (b) minimizes makespan, and (c) minimizes SST. Circles - agents and squares - goals.

A key difficulty in *MACPF* lies in selecting an appropriate optimization objective. Classical measures such as the Sum of Costs (SOC) or makespan capture the overall travel effort or the time of the last completed goal, but they overlook when individual goals are reached by an agent. As a result, many goals may remain unreached until the very end, even when SOC or makespan are small. The *Sum of Service Time* (SST) objective [12] addresses this by summing the completion times of all goals, thereby promoting earlier and more balanced service. This is particularly relevant in domains where responsiveness and fairness across goals are more critical. For example, in a search and rescue scenario (Figure 1), the choice of objective critically affects when each injured person is rescued. In panel (a), minimizing SOC causes agent 1 to rescue A before B, delaying B by two steps. In panel (b), minimizing makespan still leaves B waiting until after A. In contrast, panel (c) minimizes SST: agent 2 is immediately dispatched to B, ensuring all goals are served earlier and more evenly. This illustrates that SOC and makespan may delay critical goals, while SST promotes timely, balanced rescue.

The first contribution of this work is a *MACPF* algorithm that minimizes SST in the generalized setting where agents are not required to end at fixed goal destinations. In contrast, Conflict-Based Steiner Search (CBSS) [16] extends Conflict-Based Search (CBS) [18] to solve *MACPF* under the assumption that each agent has a fixed destination and minimizes the Sum of Costs (SOC), denoted CBSS$_{SOC}$. The assignment of goals in CBSS$_{SOC}$ is obtained via a reduction to the Traveling Salesman Problem (TSP) and a dedicated procedure [23], which enumerates TSP tours in increasing order of cost, each corresponding to a specific goal allocation. To adapt CBSS to minimize SST (CBSS_{SST}), we reduce the allocation problem to a Traveling Repairman Problem (TRP) [2] and design a Mixed-Integer Linear Programming (MILP) formulation that enumerates allocations in increasing order of SST. A related MILP was proposed in [4], but it computes only a single optimal tour, whereas our formulation efficiently generates multiple tours (allocations).

A further limitation of CBSS is the assumption of perfect execution, which is unrealistic in real-world multi-agent systems. In practice, agents may be delayed stochastically, leading to deviations from the planned schedule and potential conflicts. **The second contribution of this work** is an extension of CBSS_{SST} that accounts for such delays. One approach to address these delays is to replan during execution when a conflict is about to occur, but this is often costly and yields less effective solutions. Inspired by prior work on MAPF [1], we instead consider a verification-based approach in which CBSS estimates the *robustness* of a plan to delays, i.e., the probability that agents following it will avoid conflicts. This extension, termed $RCbss_T Strict$ (Robust CBSS_{SST} with a Strict Verifier), accepts a desired robustness p and uses Monte Carlo simulations to return a plan that satisfies it.

$RCbss_T Strict$ enables reasoning about execution uncertainty but creates a trade-off: higher robustness reduces online replanning but requires more simulations and deeper search, increasing planning time. **The third contribution of this work** is to introduce $RCbss_T Anytime$ (Robust CBSS_{SST} with an Anytime Verifier), which balances this trade-off. Rather than requiring the desired robustness (p) regardless of time constraints, $RCbss_T Anytime$ maintains and updates the highest robustness confirmed so far. If the desired robustness is verified within the time limit, $RCbss_T Anytime$ terminates successfully; otherwise, it returns the solution with the highest robustness validated up to that point. This approach guarantees that a solution is always produced within the runtime, balancing the benefits of robustness with the constraints of planning time.

We evaluate $RCbss_T Strict$ and $RCbss_T Anytime$ on the standard grid-based MAPF benchmark maps [22]. The results show that $RCbss_T Anytime$ solves a larger number of instances within the time limit while maintaining strong robustness guarantees. In particular, it achieves a better balance between reducing the need for replanning during execution and keeping planning times tractable, highlighting the practical advantages of the anytime approach.

2 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. MACPF 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 occupies a single vertex in G. At each time step, an agent may perform one of two actions: *Wait*, which keeps the agent at its current vertex, or *Move*, which changes the agent's location to an adjacent vertex. Each agent $a_i \in A$ starts at an initial vertex v_{0_i}, and $V_0 = \{v_{0_1}, \ldots, v_{0_n}\} \subset V$ denotes the set of all initial vertices. Similarly, let $V_g = \{v_{g_1}, \ldots, v_{g_m}\} \subset V$ denote the set of m distinct goal vertices.

A single-agent plan π^i for agent a_i is a sequence of actions starting from v_{0_i}. We denote by $\pi^i(t)$ the vertex occupied after performing the t^{th} action in π^i. A plan π^i specifies how agent a_i moves in the graph in order to collect a (possibly empty) subset of goals. Since agents operate simultaneously, conflicts may arise when their paths intersect. 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 $\pi^i(t) = \pi^j(t) = x$, or when they traverse the same edge x in opposite directions from time step $t-1$ to time step t, i.e., when $\pi^i(t-1) = \pi^j(t) \wedge \pi^i(t) = \pi^j(t-1)$, where $(\pi^i(t-1), \pi^i(t)) = x$. The former is referred to as a vertex conflict and the latter as a swapping conflict [22].

A solution π is a set of single-agent plans $\{\pi^1, \ldots, \pi^n\}$, one for each agent. A solution is *conflict-free* if it does not contain a conflict as defined above.

*Definition 2.1 (**Sum of Service Times (SST)**).* Given a solution $\pi = \{\pi^1, \ldots, \pi^n\}$, the *service time* of a goal $v_{g_j} \in V_g$ is the earliest time step at which some agent reaches it in π. Formally [12],

$$SST(\pi) := \sum_{v_{g_j} \in V_g} \min\{\, t \mid \pi^i(t) = v_{g_j} \text{ for some } a_i \in A \,\}.$$

Equipped with these definitions, we can now formally define the MACPF problem.

*Definition 2.2 (**Multi-Agent Combinatorial Path Finding Problem**).* Given a set of n agents with initial vertices V_0 and a set of goal vertices V_g, the *multi-agent combinatorial path finding problem* (MACPF) is to find a conflict-free solution π such that every goal in V_g is visited by at least one agent.

In this work, we aim to optimize SST and define an optimal solution accordingly, Unlike prior formulations of MACPF, which assumed each agent is pre-assigned a goal destination, our definition only requires that all goals be visited without fixing destinations for individual agents.

3 BACKGROUND AND RELATED WORK

We next provide background on CBS and its extension CBSS, which form the foundation for our work.

CBS [18] is a two-level MAPF algorithm that optimizes SOC. At the **high level**, it explores a Constraint Tree (CT), where each node $Z = (\pi, cost(\pi), C)$ consists of a solution $\pi = (\pi^1, \ldots, \pi^n)$, its total cost, and a set of constraints C. Each constraint $(a_i, x, t) \in C$ 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 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 [16] extends CBS to solve *MACPF* problems by jointly handling goal allocation and path planning, assuming that each agent is pre-assigned a fixed goal destination. It modifies the high-level search to construct a forest of constraint trees (CTs), each corresponding to a fixed *goal sequence allocation.* It selects allocations that yield solutions minimizing SOC, which we denote as CBSS$_{SOC}$.

*Definition 3.1 (**Goal sequence allocation**).* A goal sequence allocation $\gamma = \{\gamma^i\}_{i=1}^n$ assigns each agent a_i an ordered sequence of goal vertices $\gamma^i = (v_{g_1}, \ldots, v_{g_h})$. Let $d(u, v)$ denote the shortest-path cost between vertices u and v on graph G, and let v_{0_i} be the initial vertex of agent a_i. The cost of γ^i, denoted $cost(\gamma^i)$, is defined as $d(v_{0_i}, v_{g_1}) + \sum_{p=1}^{h-1} d(v_{g_p}, v_{g_{p+1}})$, and the cost of the full allocation is $cost(\gamma) = \sum_{i \in A} cost(\gamma^i)$.

Algorithm 1: CBSS (SOC or SST objectives)

```
Input: $G$
1  $K \leftarrow 1$
2  $\gamma_1 \leftarrow$ K-Best-Sequencing($G, K$)
3  $C \leftarrow \emptyset$
4  $\pi \leftarrow$ LowLevelPlan($\gamma_1, C$)
5  Add $Z_{root} = (\pi, cost(\pi), C)$ 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 $Z_{\text{newRoot}}$ 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
```

CBSS enumerates goal sequence allocations $\gamma_1, \gamma_2, \ldots$ 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. This method accepts a parameter K and returns the K^{th} cheapest allocation. In CBSS$_{SOC}$ this is achieved via a reduction to a Traveling Salesman Problem (TSP). The *K-Best-Sequencing* method maintains a priority queue of tours ordered by cost. It first solves the TSP to obtain the cheapest tour. At each step, the lowest-cost tour is extracted; if it is the K^{th}, it is returned, otherwise new tours are generated by forbidding each edge of the extracted tour, solving the resulting TSPs, and inserting them back into the priority queue. In this way, CBSS$_{SOC}$ computes a sequence of allocations ordered by increasing SOC. Once γ_K is determined, CBSS explores its corresponding CT_K using CBS: it resolves conflicts via constraint splitting and uses the low-level planner to compute for each agent a_i a plan satisfying all constraints and visiting its goals γ_K^i.

CBSS is formally described in Algorithm 1. 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. CBSS then computes its solution π_{newRoot}, and initializes the root node Z_{newRoot} for CT_{k+1}. Both Z and Z_{newRoot} are returned to OPEN, and the node with the lower cost will be selected for expansion in the next iteration. 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.

A key limitation of CBSS is that it assumes each agent is assigned a fixed destination goal in advance, which restricts the ability to dynamically assign goals when such pre-assignments are suboptimal or undesirable. In addition, CBSS optimizes only SOC (CBSS$_{SOC}$). While SOC is appropriate in many MACPF settings, it fails to capture scenarios where the relevant performance measure is SST. These limitations motivate the development of an alternative framework that supports flexible goal assignments and alternative optimization objectives such as SST.

3.1 Related Work

The field of Multi-Agent Path Finding (MAPF) studies how to compute conflict-free paths for multiple agents from given start to goal locations. In the classical MAPF setting, each agent is assigned a single fixed goal destination, and the task is to plan paths that bring all agents to their goals without conflicts. Beyond the classical setting, two lines of research are particularly relevant to this work. The first extends MAPF by integrating goal allocation with path planning, addressing scenarios where agents must serve multiple or flexible goals [6, 8, 9, 14–16, 21]. The second introduces uncertainty into planning, modeling imperfect execution or stochastic delays [1, 11, 17, 19, 25].

3.1.1 ***Related work on integrating MAPF and goal allocation***. Prior work on MAPF considered settings with an equal number of agents and goals, where no goal was pre-assigned to a specific agent. Instead, a one-to-one allocation had to be determined, assigning each agent exactly one goal [8, 9, 21]. In contrast, our work removes the restriction that the number of agents must equal the number of goals. Agents may be assigned multiple goals (or none), and the solution must ensure that all goals are eventually visited. Another extension of MAPF considers settings where each agent is associated with a set of potential goals and must ultimately be assigned to exactly one of them [6]. While in this setting each agent is assigned from a set of possible goals, it is still limited to exactly one goal per agent. In contrast, our formulation allows agents to be assigned multiple goals (or none). Closer to our setting, CBSS [16], MS* [15], and DMS* [14] directly address the MACPF problem under the standard assumption that each agent has a fixed destination goal. CBSS builds on CBS [18] and minimizes the SOC objective, MS* builds on M* [24] and also minimizes SOC, while DMS* focuses on optimizing makespan. These works assume fixed destination goals and optimize SOC or makespan, whereas we remove the requirement of fixed destinations and optimize SST.

3.1.2 ***Related work on MAPF under uncertainty***. Another important direction extends MAPF to account for uncertainty in execution and the environment. UM* handles state uncertainty via belief-space planning but does not guarantee bounded collision probabilities [25]. pR-CBS [1] extends CBS with probabilistic robustness, ensuring that collisions remain below a predefined threshold even under stochastic delays. MAPF under Obstacle Uncertainty (MAPFOU) addresses incomplete knowledge of traversability, assuming deterministic actions once the environment is revealed [11, 19]. MAPF with Time Uncertainty models bounded edge traversal times [17]. However, none of these approaches consider the combinatorial challenge of allocating multiple goals to agents.

4 OPTIMIZING SST WITH CBSS

CBSS solves MACPF under the assumption that each agent is assigned its own fixed destination goal, i.e., a specific goal vertex at which the agent must end its plan after collecting all goals assigned to it. To solve MACPF problems without fixed destinations and to minimize SST, we adapt CBSS as follows. First, the cost associated with each CT node in the high-level CBSS search is the SST of the set of single-agent plans it represents, instead of its SOC. Second, we modified the cost associated with a goal sequence allocation γ to be aligned with optimizing SST. This change also requires modifying the *K-Best-Sequencing* method used in CBSS. The change to the goal sequence allocation and corresponding *K-Best-Sequencing* method are not trivial, and we discuss them in details below.

Changing the cost of a goal sequence allocation. We define the *service time* of a goal vertex v_g in a goal allocation γ, denoted $sst(v_g, \gamma)$, as the sum of distances the agent allocated to v_g must traverse according to γ before reaching v_g. More formally, if a goal vertex v_g is assigned to agent a_i in γ, i.e., $v_g \in \gamma^i$, and v_g is the h^{th} goal vertex in the sequence $\gamma^i = (v_{g_1}, \dots, v_{g_{|\gamma^i|}})$ with $v_{g_h} = v_g$, then $sst(v_g, \gamma) = d(v_{0_i}, v_{g_1}) + \sum_{j=1}^{h-1} d(v_{g_j}, v_{g_{j+1}})$. The SST of goal sequence allocation γ, denoted $sst(\gamma)$, is the sum of the service times of all goals across agents, i.e., $\sum_{v_g \in \gamma} sst(v_g, \gamma)$. Observe that $sst(\gamma)$ is a lower bound on the SST of any solution to the respective MACPF problem that is consistent with γ.

Next, we define a Mixed Integer Linear Program (MILP) whose solution corresponds to a goal sequence allocation that minimizes its SST. We assign each vertex $v \in V_0 \cup V_g \subseteq V$ of the underlying graph $G = (V, E)$ a unique index from 1 to $n + m$: indices $1, \dots, n$ correspond to the n agents' initial vertices in V_0, and indices $n + 1, \dots, n + m$ correspond to the m goal vertices in V_g. This indexing enables a uniform definition of the decision variables.

Decision variables. To encode the structure of a goal sequence allocation, we introduce binary variables $x_{ij} \in \{0, 1\}$, where $i \in \{1, \dots, n + m\}$ denotes either an initial vertex in V_0 or a goal vertex in V_g, and $j \in \{n + 1, \dots, n + m\}$ denotes a goal vertex in V_g. Setting $x_{ij} = 1$ indicates that j is the immediate successor of i in the allocation sequence (visited directly after i), and $x_{ij} = 0$ otherwise.

We define an integer variable $t_j \in \mathbb{Z}_{\geq 0}$ representing the service time for each $j \in \{1, \dots, m\}$, where each j denotes a goal vertex in V_g. Since the service times quantify the quality of a plan, the objective is to minimize the sum of service time across all goals:

$$\min \sum_{j=1}^{m} t_j.$$

Constraints. The formulation includes the following feasibility constraints:

(1) Each goal must have exactly one incoming edge, ensuring that it is collected exactly once:

$$\sum_{\substack{i=1 \\ i \neq j}}^{n+m} x_{ij} = 1, \quad \forall j \in \{n + 1, \dots, n + m\}.$$

(2) Each goal can have at most one outgoing edge, ensuring that it leads to at most one successor:

$$\sum_{\substack{j=n+1 \\ j \neq i}}^{n+m} x_{ij} \leq 1, \quad \forall i \in \{n + 1, \dots, n + m\}.$$

(3) Each agent can connect to at most one goal, ensuring that every agent can start at most one sequence of assigned goals:

$$\sum_{j=n+1}^{n+m} x_{ij} \leq 1, \quad \forall i \in \{1, \dots, n\}.$$

(4) Service times must be consistent with the paths:

(4.1) If a goal j is the first to be collected by agent i, its service time equals the travel cost c_{ij} from the agent's start to j:

$$\begin{aligned} t_{j-n} &\geq c_{ij} - (1 - x_{ij}) \cdot M, \\ t_{j-n} &\leq c_{ij} + (1 - x_{ij}) \cdot M, \end{aligned} \quad i \in \{1, \dots, n\},\ j \in \{n + 1, \dots, n + m\}.$$

(4.2) If a goal j is collected after goal i, its service time equals the service time of i plus the travel cost c_{ij}:

$$\begin{aligned} t_{j-n} &\geq t_{i-n} + c_{ij} - (1 - x_{ij}) \cdot M, \\ t_{j-n} &\leq t_{i-n} + c_{ij} + (1 - x_{ij}) \cdot M, \end{aligned} \quad i, j \in \{n + 1, \dots, n + m\},\ i \neq j.$$

Here, c_{ij} denotes the shortest travel cost between the vertices whose MILP indices are i and j in G, and M is a sufficiently large constant. These inequalities are enforced only when $x_{ij} = 1$, i.e., when the corresponding edge is part of the allocation. If $x_{ij} = 0$, the edge is inactive and the large constant M relaxes the inequalities, making them trivially satisfied. In other words, M serves purely as a switch that activates or deactivates constraints depending on x_{ij}. This is the standard Big-M technique for encoding conditional constraints in MILP [13].

The K^{th} cheapest allocation. To obtain the K^{th} cheapest allocation, we run an iterative procedure for $p = 1, \dots, K$, where at each iteration solving the MILP yields Sol_p components (Eq. a), and we then add the exclusion constraint (Eq. b) to prevent Sol_p from reappearing in later iterations:

$$Sol_p = \{(i, j) \mid x_{ij} = 1\}, \quad Cost(Sol_p) = \sum_{k=1}^{m} t_k, \tag{a}$$

$$\sum_{(i,j) \in Sol_p} x_{ij} \leq |Sol_p| - 1. \tag{b}$$

At $p = K$, Sol_K defines γ_K with $sst(\gamma_K) = Cost(Sol_K)$. To reconstruct γ_K, we interpret each index $i, j \in \{1, \dots, n + m\}$ according to the mapping defined earlier, where $1, \dots, n$ correspond to initial vertices in V_0 and $n + 1, \dots, n + m$ to goal vertices in V_g of the underlying graph $G = (V, E)$. For each $k \in \{1, \dots, n\}$, we initialize γ^k. We then follow the unique active pair $(i, j) \in Sol_K$ with $i = k$, append the goal vertex represented by j to γ^k, and continue iteratively from j with the next pair $(j, \ell) \in Sol_K$, until no outgoing pair is found. Applying this procedure for all $k = 1, \ldots, n$ yields $\gamma_K = \{\gamma^1, \ldots, \gamma^n\}$ as the collection of ordered goal sequences.

In summary, the MILP formulation replaces the K-best sequencing step in CBSS, enabling optimization under the SST objective without requiring fixed destinations, and returns the K^{th} cheapest allocation. We call the resulting algorithm CBSS_{SST}.

5 ROBUSTNESS TO DELAYS

In real-world multi-agent systems, agents rarely execute their plans in perfectly synchronized steps. We consider next the case where each agent may experience a delay with some fixed probability before each step it executes. Let $p_{\text{delay}} = (p^1_{\text{delay}}, \ldots, p^n_{\text{delay}})$ denote the vector of delay probabilities per agent, where at each time step agent i may fail to execute a move action and remain at its current location with probability p^i_{delay} (and otherwise proceeds along its planned path). Delays are sampled independently across agents and time steps. This simple model captures a wide range of realistic sources of uncertainty, including communication lags, computational overhead, and physical dynamics such as actuator or sensor latency. Importantly, even if a solution is conflict-free under ideal execution, such delays may lead to new conflicts during execution, which motivates the need for robustness guarantees.

To address this uncertainty, we adopt the notion of p-robustness introduced in p-Robust CBS [1]. This concept reflects the intuition that absolute safety under stochastic delays is rarely attainable (unless agents are forced onto vertex-disjoint paths, which is often impractical). Instead, p-robustness provides a principled trade-off: it ensures that the probability of a conflict-free execution is at least a specified threshold, while still permitting efficient plans where agents may traverse partially overlapping paths.

*Definition 5.1 (p-**Robust Solution**).* The robustness of a solution π with respect to a given vector of delays p_{delay} is the probability that agents executing π will not conflict. A solution π with a robustness p is called a p-robust solution.

Computing the exact robustness of a solution is challenging. Instead, we aim to find a solution that is p-robust with statistical confidence $1 - \alpha$ for a given value of $\alpha \in (0, 1)$. Next, we describe how to extend our CBSS algorithm such that it optimizes SST and returns p-robust solutions with confidence $1 - \alpha$.

5.1 Robust CBSS - Strict Verifier ($RCbss_TStrict$)

One way to handle stochastic delays is to replan online during execution, shortly before an imminent conflict. However, such online replanning is inherently myopic, as it reacts only to local, near-term conflicts, and may therefore yield high SST solutions. It also incurs communication and computational overhead, since agents must synchronize with the planner during execution. As an alternative, we aim to produce *robust* plans offline that anticipate delays and reduce the need for online replanning.

To extend CBSS (Algorithm 1) to stochastic environments, we integrate the robustness mechanisms used by p-Robust CBS [1] for MAPF. A solution π is returned only if it can be statistically verified to satisfy the required robustness threshold p; otherwise, it is rejected. This requires two modifications to Algorithm 1: (i) replacing the deterministic conflict check on line 15 with a probabilistic verification step, and (ii) extending the conflict resolution step in line 18.

Monte Carlo verification in CBSS. CBSS deterministically checks whether the candidate solution contains conflicts (Algorithm 1, line 15). In our robust extension, this check is replaced by the Monte Carlo verifier (Algorithm 2) used in p-Robust CBS [1], which estimates a solution's robustness by repeatedly simulating randomized executions of π with the per-agent delay probabilities $\boldsymbol{p}_{\text{delay}}$ defined above. Then, it applies a standard statistical test to determine if the given solution indeed reached a robustness of p with confidence level $1 - \alpha$. The procedure is described in Algorithm 2.

Algorithm 2: Monte Carlo Verifier

```
   Input: $\pi$, $p$, $\alpha$, $p_{\text{delay}}$
1  Initialize $s_0$ // Equation (1);
2  Run $s_0$ simulations under $\boldsymbol{p}_{\text{delay}}$;
3  while TRUE do
4      $P_0 \leftarrow$ ratio of simulations without a conflict;
5      Calculate $c_1$ // Equation (2);
6      Calculate $c_2$ // Equation (3);
7      if $P_0 \geq c_1$ then
8          return TRUE;
9      if $P_0 < c_2$ then
10         return FALSE;
11     $s_0 \leftarrow s_0 + 1$;
12     Run one more simulation;
```

Algorithm 2 begins by determining the initial number of simulations s_0 (line 1). This value must be sufficiently large to ensure the validity of the statistical test. The initial number of simulations s_0 is given by

$$s_0 = \max\left(30, \left\lceil \frac{z^2_{1-\alpha} \cdot p}{1 - p} \right\rceil\right), \tag{1}$$

where $z_{1-\alpha}$ denotes the critical value of the standard normal distribution for confidence level $1 - \alpha$. The constant 30 in Eq. (1) ensures that the binomial distribution of s_0 Bernoulli trials (each simulation) is well approximated by a normal distribution, as guaranteed by the Central Limit Theorem. The second term in Eq. (1) follows from the requirement that line 7 in Algorithm 2 can be satisfied. This holds only if c_1 in Eq. (2) is smaller than 1, which implies

$$s_0 > z^2_{1-\alpha} \cdot \frac{p}{1 - p}.$$

Next, s_0 randomized simulations of the solution π are performed (line 2). The fraction of conflict-free simulations, referred to as the *empirical robustness*, is denoted by P_0 (line 4). This value is then evaluated against two statistical thresholds. The first threshold c_1 (line 5) is the minimal value of P_0 required to *accept* the hypothesis that the solution π is p-robust with confidence level $1 - \alpha$:

$$c_1 = p + z_{1-\alpha} \cdot \sqrt{\frac{p(1 - p)}{s_0}}. \tag{2}$$

The second threshold c_2 (line 6) is the maximal value of P_0 required to *reject* the hypothesis that π is p-robust with confidence level $1-\alpha$:

$$c_2 = p - z_{1-\alpha} \cdot \sqrt{\frac{p(1-p)}{s_0}}. \tag{3}$$

If the observed probability P_0 is greater than or equal to c_1, the verifier concludes that the solution is p-robust and returns TRUE (line 8). Conversely, if $P_0 < c_2$, it concludes that the solution does not meet the robustness requirement and returns FALSE (line 10). In cases where P_0 lies between these thresholds, the algorithm increases s_0 by one (line 11), performs an additional simulation (line 12), and repeats until a decision is reached.

Three-way split conflict resolution in CBSS. In Algorithm 1, in line 18, CBSS applies the CBS two-way split: given a conflict $\langle a_i, a_j, x, t\rangle$, it generates two children, each forbidding one agent from occupying vertex x or traversing edge x at time t. In the robust extension, this is insufficient: in CBS, the case where both agents are planned to occupy x at time t is treated as illegal and discarded. However, under stochastic delays this potential conflict may occur only with a small probability, and discarding it could prune solutions that still satisfy the requirement p. Therefore, following p-Robust CBS, we adopt a three-way split: in addition to the two standard children, we generate a third child with a positive constraint that enforces the occurrence of the conflict. This allows the search to retain and evaluate solutions where the conflict is tolerated probabilistically, preserving potentially optimal p-robust solutions that would otherwise be pruned.

We refer to the robust extension of CBSS$_{SST}$, which uses the Monte Carlo verifier to identify conflicts and the three-way split described above to resolve them, as *RCbss$_T$Strict*.

5.2 Balancing Robustness and Efficiency

Although *RCbss$_T$Strict* provides principled robustness guarantees under stochastic delays, it also raises important questions about the balance between robustness and efficiency. *RCbss$_T$Strict* reduces the likelihood of execution-time conflicts as higher values of p lower the need for online replanning. However, achieving higher robustness comes at a significant cost: the verifier must run more simulations and the search must explore larger parts of the constraint tree, which increases runtime and reduces scalability. This creates a clear trade-off: increasing p reduces the amount of replanning required during execution but makes planning considerably slower and may yield longer plans (increasing SST).

Rather than committing solely to robust planning or to online replanning, a promising direction is to combine both: designing solutions that incorporate a sufficient degree of robustness while retaining the flexibility to resolve conflicts dynamically during execution. Next, we propose *RCbss$_T$Anytime*, an anytime variant of *RCbss$_T$Strict* that searches for a solution that satisfies the desired robustness p, but if no such solution is confirmed within the available runtime, it returns the solution with the highest *empirically verified robustness* found thus far.

5.2.1 ***Robust CBSS - Anytime Verifier (*RCbss$_T$Anytime*).***

In p-Robust CBS [1] and *RCbss$_T$Strict*, the Monte Carlo verifier checks if we can accept or reject the hypothesis that the given solution is p-robust (Algorithm 2, lines 5 and 6). In *RCbss$_T$Anytime* we invert the perspective and asks *for which values of p would we have been able to accept the solution at hand given the empirical robustness of the performed simulations?* Concretely, when *RCbss$_T$Anytime* computes two values p_{c1} and p_{c2} for a given solution π and s_0 simulation with an empirical robustness P_0. p_{c1} is a lower bound on the robustness we can verify and p_{c2} is an upper bound on it. We refer to p_{c1} as the empirically verified robustness. If p_{c1} is equal to or greater than p, the solution π is accepted as p-robust. If p_{c2} is smaller than p, the solution is declared to not be p-robust. If neither condition holds, additional simulations are performed. Most importantly, during execution *RCbss$_T$Anytime* keeps track of the highest empirically verified robustness value observed so far and the corresponding solution, which is returned if *RCbss$_T$Anytime* is halted before finding a solution that is empirically verified to be p-robust.

Computing p_{c1} and p_{c2}. Starting from the boundary equations $P_0 = c_1$ and $P_0 = c_2$, we substitute the definitions of c_1 and c_2 (Equations 2 and 3):

$$P_0 = p \pm z_{1-\alpha}\sqrt{\frac{p(1-p)}{s_0}}.$$

Each of these equations expresses a condition on p under which the empirical robustness P_0 lies exactly on the acceptance or rejection boundary. By standard algebraic manipulation, this leads to the quadratic equation

$$\underbrace{\left(s_0 + z_{1-\alpha}^2\right)}_{A} p^2 + \underbrace{\left(-(2s_0P_0 + z_{1-\alpha}^2)\right)}_{B} p + \underbrace{s_0P_0^2}_{C} = 0, \tag{4}$$

where A, B, and C are the coefficients defined above. Since the equation is quadratic in p, its solutions can be obtained using the quadratic formula, yielding the lower and upper bounds on the robustness, p_{c1} and p_{c2}.

Algorithm 3: Anytime Verifier

```
   Input: $\pi, p, \alpha, p_{\text{delay}}$
1  Initialize $s_0$ // Equation (1);
2  Run $s_0$ simulations under $p_{\text{delay}}$;
3  while TRUE do
4      $P_0 \leftarrow$ ratio of simulations without a conflict;
5      Calculate $p_{c1}, p_{c2}$ // based on Equation (4) ;
6      if $p_{c1} > p_A$ then
7          $\pi_{\text{best}} \leftarrow \pi; p_A \leftarrow p_{c1}$;
8          if $p_{c1} \geq p$ then
9              return TRUE;
10     if $p_{c2} < p$ then
11         return FALSE;
12     $s_0 \leftarrow s_0 + 1$;
13     Run one more simulation;
```

***Using* p_{c1} *and* p_{c2} *in* $RCbss_TAnytime$.** The main difference between $RCbss_TStrict$ and $RCbss_TAnytime$ is in the Monte Carlo verifier they use. Algorithm 3 lists the pseudo code for the verifier used by $RCbss_TAnytime$. It follows the same initialization and simulation steps as the plan verifier of $RCbss_TStrict$ (Algorithm 2, lines 1–4). The difference arises after computing the boundary values p_{c1} and p_{c2} (line 5). If the computed p_{c1} exceeds the highest empirically verified robustness level found so far (line 6), denoted by p_A, then it updates it (line 7) and records the corresponding solution as the incumbent solution, denoted π_{best}. If at this point $p_{c1} \geq p$ (line 8), the verifier terminates successfully (line 9). Conversely, if $p_{c2} < p$ (line 10), the verifier concludes that the solution cannot achieve robustness p and terminates with `FALSE` (line 11). If neither condition holds, the verifier continues as in Algorithm 2, incrementing s_0 (line 12), running one more simulation (line 13), and repeating the loop (line 3).

$RCbss_TAnytime$ guarantees that a solution is always produced within the time budget, returning the incumbent solution π_{best} whether the desired robustness p has been verified or not.

6 EXPERIMENTAL RESULTS

We compare experimentally two proposed variants of Robust CBSS — $RCbss_TStrict$ and $RCbss_TAnytime$— with several natural baselines on four diverse grids of varying sizes and structures from the standard grid-based MAPF benchmark [22] (illustrated in Figure 2).

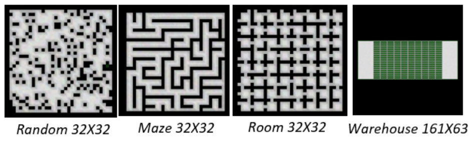

Figure 2: Evaluation maps: Random, Maze, Room, and Warehouse, with different obstacle densities and structures.

Planner Configurations and Baselines. We use the term *planner configuration* to denote a specific planner variant together with a desired safety level, when applicable. We consider four planner configurations, where (1) and (2) are the baselines: (1) $CBSS_{SST}$ - deterministic CBSS that ignores delays during planning. (2) $RCbss_TStrict$ with $p = 0$ - delays are modeled, but robustness is not enforced; all plans pass verification and delay-induced conflicts are tolerated during planning. (3) $RCbss_TStrict$ with a desired p. (4) $RCbss_TAnytime$ with a desired p. The desired levels p are $p_{\text{safe}} \in \{0.05, 0.25, 0.5, 0.8, 0.95, 0.99, 0.999, 0.9999\}$.

Experimental Design. We varied the number of agents, setting $n \in \{15, 20, 25, 30, 35, 40, 45, 50\}$, and fixed the number of goals to $m = 50$. The delay vectors p_{delay} used in our experiments are uniform, setting entries to be equal to the same value, chosen from $\{0.1, 0.3, 0.5\}$. For each planner configuration, for every combination of agent count n and delay level p_{delay}, we generate 300 random instances by uniformly sampling agent and goal locations. The confidence level used in our experiments was $1 - \alpha = 0.95$, following p-Robust CBS [1]. Each instance is solved under a total time budget of 600 seconds. Within this budget, each planning iteration is capped at 60 seconds.

We considered the following metrics in our experiments.

- **Failures.** Percentage of instances that were not be solved.
- **≥1 Replan.** Percentage of instances that required at least one online replanning.
- **Avg Online Runtime.** Average online (replanning) time per instance, in seconds.
- **Avg Online SST.** Average sum of service time during execution per instance.
- **Avg Nodes Expanded.** Average number of nodes expanded per instance during the high-level search.

Online Execution and Replanning Triggers. Given a solution π produced by a chosen planner configuration, execution proceeds in an online environment where stochastic delays may occur at every step. At each time step, *before* applying the next planned moves, we perform a one-step look-ahead to detect if there is any chance that some pair of agents will collide at the next step, either because they are planned to occupy the same location at that step or because a one-step delay of one agent could place them at the same location. If this occurs, we immediately invoke the planner to perform *replanning* from the agents' current locations using the same planner configuration. To avoid a deadlock in which execution would repeatedly trigger replanning on the first step for the same potential conflict, we augment the solution-verifier of every planner configuration to reject any solution where there is a chance to have conflicts in the first step.

Implementation. Planner configurations and the online execution environment were implemented in Python 3.10 on Ubuntu 24.04. The MILP-based allocation subproblems were solved with the Gurobi Optimizer (v11.0.3) via `gurobipy` [5]. Experiments were run on a 16 -core virtual machine (AMD EPYC 7702P). Only representative results are reported here; full results and code are available at https://github.com/anonymousDD112233/RCbssTAnytime.

6.1 Results

Table 1 reports the performance of $RCbss_TStrict$, $RCbss_TAnytime$, and the baselines on the Maze map across different desired robustness levels p. The values in the columns **Failures** and **Avg Expanded Nodes** are computed over all instances. For the other metrics (**≥1 Replan**, **Avg Online Runtime**, and **Avg Online SST**), averages are taken separately: in the top and middle parts of the table over the common subset of instances solved by all baselines and $RCbss_TStrict$ configurations, and in the bottom part over the subset of instances solved by all $RCbss_TAnytime$ configurations.

Consider first the results for $RCbss_TStrict$. These results highlight the trade-off between robustness and efficiency that occurs when using $RCbss_TStrict$. As p increases, more robust plans reduce the proportion of instances requiring at least one replan, resulting in fewer replans and shorter average online runtimes. At the same time, stricter robustness requirements raise the failure rate in two ways: first, by increasing the rejection of candidate plans, which forces the search to expand more nodes in the tree; and second, by increasing the minimum number of simulations per verification, which slows down the verification process itself. This is evident from the sharp growth in required minimum simulations (Eq. 1): about 52 at $p = 0.95$, 268 at $p = 0.99$, 2,704 at $p = 0.999$ and over

27,058 at $p = 0.9999$. This explains the observed trend: as p grows, the number of expanded nodes generally increases due to more frequent rejections, but at $p = 0.99$ and above, the high minimum simulation requirement dominates, slowing verification and resulting in fewer expansions overall. The average online SST remains essentially unchanged across all configurations, indicating that the trade-off, at least in our experiments, primarily concerns failure rates and replanning effort rather than SST. Under our time budget, $RCbss_T Strict$ with $p = 0$ achieves the lowest failure rate, but at the cost of more instances requiring replanning during execution and a longer average online runtime compared to more robust settings. Conversely, as p increases, solutions involve fewer replanning events and shorter average online runtimes, but this comes at the expense of higher failure rates.

Planner Configurations	Failures	≥1 Replan	Avg Online Runtime (sec)	Avg Online SST	Avg Expanded Nodes
CBSS_{SST}	32.31%	4.33%	0.97	487.39	9.41
$RCbss_T Strict$- $p = 0$	29.01%	6.81%	1.48	487.41	1.54
$RCbss_T Strict$- $p = 0.05$	29.38%	6.07%	1.37	487.11	22.02
$RCbss_T Strict$- $p = 0.25$	30.12%	5.40%	1.24	487.12	145.65
$RCbss_T Strict$- $p = 0.5$	31.99%	3.88%	0.92	487.14	393.63
$RCbss_T Strict$- $p = 0.8$	39.33%	1.69%	0.44	487.16	785.21
$RCbss_T Strict$- $p = 0.95$	47.62%	0.62%	0.14	487.22	806.65
$RCbss_T Strict$- $p = 0.99$	55.58%	0.34%	0.08	487.22	344.52
$RCbss_T Strict$- $p = 0.999$	63.57%	0.00%	0.00	487.15	61.70
$RCbss_T Strict$- $p = 0.9999$	74.43%	0.00%	0.00	487.16	9.08
$RCbss_T Anytime$- $p = 0.05$	28.71%	27.77%	7.74	588.83	12.48
$RCbss_T Anytime$- $p = 0.25$	28.39%	26.74%	7.41	588.95	76.17
$RCbss_T Anytime$- $p = 0.5$	26.38%	23.42%	6.86	589.09	220.92
$RCbss_T Anytime$- $p = 0.8$	24.01%	17.18%	5.85	589.27	501.23
$RCbss_T Anytime$- $p = 0.95$	22.71%	14.22%	5.83	589.16	567.57
$RCbss_T Anytime$- $p = 0.99$	22.26%	13.44%	5.64	589.14	255.53
$RCbss_T Anytime$- $p = 0.999$	22.90%	14.67%	6.61	589.10	47.13
$RCbss_T Anytime$- $p = 0.9999$	25.69%	20.57%	12.49	589.17	7.51

Table 1: Results for *$RCbss_T Anytime$* (bottom), *$RCbss_T Strict$* (middle), and the baselines (top) on the Maze 32X32 map.

The bottom part of Table 1 shows a different trend compared to $RCbss_T Strict$. Across all tested values of p, $RCbss_T Anytime$ solved more instances within the same time limit, consistently achieving lower failure rates than $RCbss_T Strict$. Notably, while the average online SST remains essentially unchanged across different values of p, it is larger than in the $RCbss_T Strict$ results because $RCbss_T Anytime$ solves more and often harder instances, which in turn also raises the ≥1 Replan and Avg Online Runtime metrics. Within $RCbss_T Anytime$ itself, three insights emerge: **(1)** As p increases up to 0.99, the proportion of instances requiring more than one replan decreases, the average online runtime shortens, and the overall failure rate drops. In other words, higher robustness not only reduces the need for replanning, as expected, but also enables solving more instances within the time limit. **(2)** Beyond $p = 0.99$, performance no longer improves. At $p = 0.999$ and 0.9999, the *minimum* number of simulations required per verification (2,704 and 27,058, respectively; Eq. 1) grows sharply, slowing verification and constraining the search. This is reflected in fewer expansions; with fewer nodes explored, the planner tends to return lower-quality plans. Consequently, more replanning is triggered and each replan must first verify that the initial step is 1-robust before checking p-robustness. If no such robust initial step plan is found within the time limit, the instance fails. **(3)** An interesting effect occurs at $p = 0.99$. Although the *minimum* number of simulations per verification increases from 52 at 0.95 to 268 at 0.99 (Eq. 1), verification time is higher yet not prohibitive. Consequently, fewer nodes are expanded than at 0.95, but the plans that do pass verification are stronger, yielding more solved instances within the time limit. In contrast, at $p > 0.99$ the verification time grows steeply, dominates runtime, and constrains search, reducing the number of solved instances.

Figure 3 compares $RCbss_T Strict$ and $RCbss_T Anytime$. The x-axis shows the success rate, i.e., the percentage of problem instances solved successfully. The y-axis shows the online SST level. Each point (x, y) on a curve means that $x\%$ of the instances are solved with online SST $\leq y$. At low robustness levels ($p = 0.05, 0.5$), the two variants perform similarly, solving nearly the same fraction of instances for comparable average Online SST. As robustness increases, however, their behavior diverges: $RCbss_T Strict$ suffers from rising failure rates, while $RCbss_T Anytime$ consistently solves more instances for the same Online SST. This highlights the advantage of the anytime verifier, which sustains reliability under high robustness requirements without increasing service time.

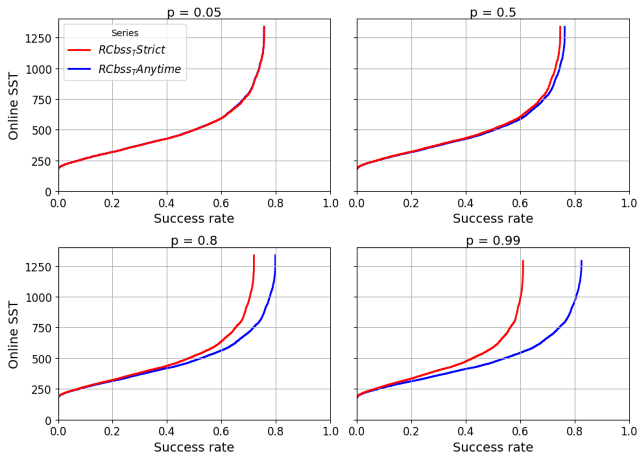

Figure 3: Online SST vs. success rate (%) for *$RCbss_T Strict$* and *$RCbss_T Anytime$* on the Random 32×32 map.

7 CONCLUSION

We studied Multi-Agent Combinatorial Path Finding (*MACPF*) and revised Conflict-Based Steiner Search (CBSS), which in its original form (CBSS_{SOC}) assumes fixed goal destinations and minimizes the sum of costs (SOC). We proposed CBSS_{SST}, which generalizes the setting by removing fixed goal destinations and optimizing the Sum of Service Times (SST) via a MILP-based allocation procedure. Building on this foundation, we introduced two robust extensions: $RCbss_T Strict$, which guarantees strict robustness, and $RCbss_T Anytime$, which provides an anytime approach that balances robustness with limited planning time. Our experiments show that $RCbss_T Anytime$ significantly improves the trade-off between robustness, replanning effort, and efficiency, making it practical for real-world scenarios. Future work can continue to explore ways to balance offline planning and online replanning, considering meta-reasoning techniques to reduce the overall Goal Achievement Time, which includes both runtime and solution cost [7].

REFERENCES

[1] Dor Atzmon, Roni Stern, Ariel Felner, Nathan R. Sturtevant, and Sven Koenig. 2020. Probabilistic Robust Multi-Agent Path Finding. *Proceedings of the International Conference on Automated Planning and Scheduling* 30, 1 (Jun. 2020), 29–37. https://doi.org/10.1609/icaps.v30i1.6642

[2] Avrim Blum, Prasad Chalasani, Don Coppersmith, Bill Pulleyblank, Prabhakar Raghavan, and Madhu Sudan. 1994. The minimum latency problem. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. 163–171.

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

[4] Maria Elena Bruni, Sara Khodaparasti, Iris Martínez-Salazar, and Samuel Nucamendi-Guillén. 2022. The multi-depot k-traveling repairman problem. *Optimization Letters* 16, 9 (2022), 2681–2709. https://doi.org/10.1007/s11590-021-01845-7

[5] Gurobi Optimization, LLC 2025. *Gurobi Optimizer Reference Manual*. Gurobi Optimization, LLC. https://www.gurobi.com

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

[7] Scott Kiesel, Ethan Burns, and Wheeler Ruml. 2015. Achieving goals quickly using real-time search: experimental results in video games. *Journal of Artificial Intelligence Research* 54 (2015), 123–158.

[8] S. Kloder and S. Hutchinson. 2006. Path planning for permutation-invariant multirobot formations. *IEEE Transactions on Robotics* 22, 4 (2006), 650–665. https://doi.org/10.1109/TRO.2006.878952

[9] Hang Ma and Sven Koenig. 2016. Optimal Target Assignment and Path Finding for Teams of Agents. *CoRR* abs/1612.05693 (2016). arXiv:1612.05693 http://arxiv.org/abs/1612.05693

[10] Hang Ma, Jiaoyang Li, T.K. Satish Kumar, and Sven Koenig. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems* (São Paulo, Brazil) *(AAMAS '17)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 837–845.

[11] Nir Malka, Guy Shani, and Roni Stern. 2024. Online Planning for Multi Agent Path Finding in Inaccurate Maps. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 10214–10221.

[12] Jonathan Morag, Noy Gabay, Daniel Koyfman, and Roni Stern. 2025. Should Multi-Agent Path Finding Algorithms Coordinate Target Arrival Times? *Proceedings of the International Symposium on Combinatorial Search* 18, 1 (Jul. 2025), 231–235. https://doi.org/10.1609/socs.v18i1.35998

[13] Chiara Piacentini, Margarita Castro, Andre Cire, and J. Christopher Beck. 2018. Compiling Optimal Numeric Planning to Mixed Integer Linear Programming. *Proceedings of the International Conference on Automated Planning and Scheduling* 28, 1 (Jun. 2018), 383–387. https://doi.org/10.1609/icaps.v28i1.13919

[14] Zhongqiang Ren, Anushtup Nandy, Sivakumar Rathinam, and Howie Choset. 2024. DMS*: Towards Minimizing Makespan for Multi-Agent Combinatorial Path Finding. *IEEE Robotics and Automation Letters* 9, 9 (Sept. 2024), 7987–7994. https://doi.org/10.1109/lra.2024.3436333

[15] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. 2021. MS*: A new exact algorithm for multi-agent simultaneous multi-goal sequencing and path finding. In *IEEE international conference on robotics and automation (ICRA)*. 11560–11565.

[16] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. 2023. CBSS: A New Approach for Multiagent Combinatorial Path Finding. *IEEE Transactions on Robotics* 39, 4 (2023), 2669–2683. https://doi.org/10.1109/TRO.2023.3266993

[17] Tomer Shahar, Shashank Shekhar, Dor Atzmon, Abdallah Saffidine, Brendan Juba, and Roni Stern. 2021. Safe Multi-Agent Pathfinding with Time Uncertainty. *Journal of Artificial Intelligence Research* 70 (2021), 923–954. https://doi.org/10.1613/jair.1.12397

[18] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219 (2015), 40–66. https://doi.org/10.1016/j.artint.2014.11.006

[19] Bar Shofer, Guy Shani, and Roni Stern. 2023. Multi agent path finding under obstacle uncertainty. In *International Conference on Automated Planning and Scheduling*. 402–410.

[20] David Silver. 2005. Cooperative pathfinding. In *AAAI conference on artificial intelligence and interactive digital entertainment*. 117–122.

[21] Kiril Solovey and Dan Halperin. 2016. On the hardness of unlabeled multi-robot motion planning. *The International Journal of Robotics Research* 35, 14 (2016), 1750–1759. https://doi.org/10.1177/0278364916672311 arXiv:https://doi.org/10.1177/0278364916672311

[22] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *International Symposium on Combinatorial Search*. 151–158.

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

[24] Glenn Wagner and Howie Choset. 2011. M*: A complete multirobot path planning algorithm with performance bounds. In *IEEE/RSJ international conference on intelligent robots and systems*. 3260–3267.

[25] Glenn Wagner and Howie Choset. 2017. Path planning for multiple agents under uncertainty. In *International Conference on Automated Planning and Scheduling*. 577–585.