arXiv:2608.08175v1 [cs.MA] 8 Aug 2026

AOC-CBS: Anytime-Optimal Continuous-time Conflict-Based Search for Generalised Multi-Agent Path Finding

Alvin Combrink[1], Sabino Francesco Roselli[1], Martin Fabian[1]

Abstract

Many research fields share a common structure: a set of agents, each pursuing its own goal, whose actions must be coordinated so that no two of them conflict. Multi-Agent Path Finding (MAPF) is a concrete instance of this structure, with applications from warehouses to road traffic and airports. Much of MAPF research assumes discrete time, circular agents sharing one spatial graph, a single goal per agent, and that an agent must remain at its goal once reached, precluding heterogeneous fleets, non-geometric conflicts, task sequences, and agents that move on after completing them. We generalise the MAPF formulation to lift these assumptions, and present Anytime-Optimal Continuous-time Conflict-Based Search (AOC-CBS), an exact and solution-complete solver for it. AOC-CBS guarantees the eventual return of an optimal solution, while reporting an incumbent with a known optimality gap upper bound throughout its runtime; it is configurable with a portfolio of repair functions, one of which we introduce (Tier-Prioritized Safe Interval Path Planning), and can exploit multiple processor cores. We demonstrate AOC-CBS on a mixed fleet of non-convex agents moving along smooth, kinodynamically feasible trajectories. Preliminary experiments against the exact solver OC-CBS, on well-known benchmarks and roadmaps we sample from them, show AOC-CBS is comparable at finding optimal solutions while extending scalability from the tens to the hundreds of agents when a bounded optimality gap is accepted.

1 Introduction

Many research fields face a version of the same underlying problem: a set of agents, each trying to accomplish its own goal, whose actions must be coordinated so that no two of them conflict. The agents may be mobile robots navigating a shared space [1], jobs contending for machines in a production schedule [2], patients moving through a hospital ward [3], or robotic arms moving through a shared configuration space [4]. What unites these problems is not the presence of robots or physical space, but a common structure: each agent moves through its own space of possible states, and two agents are said to conflict when they simultaneously occupy states that are mutually disallowed.

This work is grounded in the setting of mobile robots, *Multi-Agent Path Finding* (MAPF). We use it as a concrete starting point, since its formal structure is easy to state precisely:

[1]Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden. {combrink, rsabino, fabian}@chalmers.se

given a graph and a start and goal vertex for each agent, MAPF asks for a set of plans, one per agent, such that every agent gets to its goal and no two agents conflict [1]. MAPF finds its application in warehouses and factories [5, 6, 7, 8, 9], video games [10, 11, 12], road traffic and airports [13, 14, 15, 16], lunar operations [17], and many more. In the overwhelming majority of MAPF research, a conflict is understood to be two agents occupying the same vertex or traversing the same edge at the same time, typically under a discretization of time that makes this condition well defined [1]. This is a simplification of physical collision, and narrows the range of real-world problems the model can represent to those where two agents in different states never collide. Here, we take a broader view still: not only do we drop this simplification and reason about collision directly, we treat collision itself as just one instance of a more general *pairwise conflict*.

Furthermore, there are several aspects that are motivated by real-world problems which, to our knowledge, are seldom considered together in the MAPF literature.

First, real-world fleets can be heterogeneous not only in how the agents are shaped, but also in how they move. A fixed-wing aircraft, a ground vehicle, and a quad-rotor do not share a motion model, so requiring every agent to follow trajectories on one common graph is itself a limiting assumption, independent of any assumption about waiting or goal structure. We instead follow the same modelling approach as in [18] and let each agent move on its own graph over its own state space, so that an agent's kinodynamic constraints are captured entirely by the states and trajectories that are available to it, and heterogeneous fleets can be coordinated without forcing a shared representation. This formulation allows for completely different types of agents, such as mobile robots and robotic arms, in the same problem.

Second, not every real-world conflict is a geometric collision. Two airborne drones may be in an unacceptable configuration—one caught in the downwash of the other—without their physical volumes ever intersecting. Treating collision as the only source of conflict, and volume as the quantity that determines it, therefore excludes by construction conflicts of this kind. We instead treat volume-based collision as one instance of a conflict definition that is left abstract: for any two agents, each in some state, whether or not the pair conflicts is specified by an external function. Conflicts stemming from causes other than physical overlap can therefore be represented directly.

Third, several assumptions about how agents use their goals and vertices limit real-world applicability. Some tasks involve traversing an edge rather than being located at a specific position: aerial survey missions require aircraft to fly along specific trajectories, and fire-fighting missions may require water-drops along specific boundaries. We therefore allow tasks to be either edge traversals or vertex occupations. Real-world tasks also typically involve a service time at a location, modelling an agent doing something at its goal (e.g., loading or unloading) before departing, rather than simply arriving. Additionally, the standard requirement that every agent must remain at its goal indefinitely once reached forces a solution to transition the entire system from its initial collective starting state to a final state where all agents occupy their goals simultaneously. Consequently, an agent cannot reach its goal early and then step out of the way to let others pass without returning to it again. Relaxing this requirement has recently been shown to substantially improve solver performance [19], and we correspondingly allow agents to move after completing their tasks.

Finally, it is commonly assumed that every vertex is *waitable*, i.e., that an agent may remain there indefinitely. This precludes vertices representing states with non-zero higher-order derivatives: a fixed-wing aircraft, for instance, cannot remain stationary while airborne, even though specific airborne states must still be reachable.

With these practical motivations, we make a number of extensions to the common MAPF setting:

1. Each agent moves within its own state space according to its own graph within that state space, allowing heterogeneous agents to coexist within a single problem.
2. Time is continuous: edges can be traversed at any real-valued time, and edge traversals and waits at vertices can take any real-valued duration.
3. Conflicts are defined abstractly rather than geometrically, with volume-based collision as one possibility, allowing conflicts such as drone downwash that do not correspond to any physical overlap.
4. Tasks form a sequence of edge traversals or vertex occupations, each vertex with an associated service time; agents without any tasks (idle agents) are also considered.
5. Agents may continue to move after completing all of their tasks, so as to avoid other agents or return to a rest-position, rather than remaining fixed at a single goal.
6. Vertices do not need waitable, allowing for states that cannot be held indefinitely.

Formally, we consider the problem that follows from these extensions. There is a set of agents, each moving on its own graph over its own state space. Agents may traverse edges starting at any real-valued time, and edges may require any real-valued duration to traverse; agents can also wait at vertices for any real-valued duration. Each agent is assigned an arbitrarily long, possibly empty, sequence of tasks, where each task is either to occupy a specific vertex for at least some service duration (possibly forever) or to traverse a specific edge. Conflicts are defined abstractly: for any two agents, each in some state, an external conflict definition specifies whether the pair conflicts. We make two assumptions on conflicts. First, they are pairwise. That is, we consider only conflicts that are defined for exactly two agents. We do not model conflicts with more than two agents, such as, for instance, a conflict where the combined downwash of two drones disturbs a third below. Second, we only consider conflicts that last for at least a few orders of magnitude longer than the smallest time precision of the machine on which a solution algorithm runs; conflicts lasting a shorter time than that could be impossible to detect. A solution is a plan for each agent that takes it from its starting vertex through all of its tasks, in order, and finally to a vertex where it can remain indefinitely without conflict—only this terminal state is required to be waitable, so that a solution never leaves an agent obligated to depart again. Each agent's plan incurs a cost, and the global cost is a function of these individual plan costs. The common *sum-of-costs* (SOC) and *makespan* are two examples. An optimal solution minimises the global cost.

We do not aim to provide a full accounting of all of the problems that the above formulation may encompass. However, we postulate that this formulation generalises *classical MAPF* [1], continuous-time MAPF [20], *MAPF with large agents* [21], *Multi-Agent Motion*

Planning [18], *Multi-Goal MAPF* [22], and can be adapted to various problems from other fields, such as job-shop scheduling [23], quantum qubit routing [24], railway traffic management [25], and computer chip design [26]. The generality of the problem formulation contributes to the value of a method for obtaining practical, or even optimal, solutions.

This work introduces *Anytime-Optimal Continuous-time Conflict-Based Search* (AOC-CBS), a solver for the above problem that converges on an optimal solution in finite time, while providing an improving incumbent solution with a known upper bound on its optimality gap throughout runtime. AOC-CBS builds on the exact solver OC-CBS [27] and addresses its most significant limitation: no solution is returned until the optimal has been found, which may take an unpractical amount of time despite guarantees of a solution in finite time. AOC-CBS retains OC-CBS's exactness and solution completeness on this more general problem formulation, while at the same time managing a configurable portfolio of repair functions that constantly searches for suboptimal solutions. As one such repair method, we introduce *Tier-Prioritized Safe Interval Path Planning* (TP-SIPP), a prioritized planner that resolves a limitation of prior continuous-time prioritized methods [28] by allowing agents to vacate a location after completing a task there, rather than assuming each agent's final position is fixed in advance. The AOC-CBS architecture opens the possibility for multi-core processing, which our experiments show to greatly improve performance on harder problems.

The value of AOC-CBS is not necessarily in finding optimal solutions, although our experiments show equivalent performance with OC-CBS. Instead, AOC-CBS's value lies in the anytime property: an incumbent solution with a known optimality gap upper bound is available at any time once a first solution has been found. A purely heuristic method could offer something similar, but without any guarantee of ever finding a solution, let alone an optimal one, and without an updating optimality gap upper bound; AOC-CBS retains both the guarantee and the anytime availability.

The generality of the problem formulation is handled by AOC-CBS not itself encoding any domain-specific detail, these are supplied separately as a pre-processing step. Specifically, domain knowledge about agents, their state spaces and graphs, and the conflict definitions lies outside of AOC-CBS. We provide implementations of these preprocessing steps for MAPF with 2D circular and arbitrary (including non-convex) polygonal agents, with straight-line or arbitrary connected and continuous trajectories. Work is on-going to extend these 2D models to 3D space. Figure 1 showcases an example of a problem solved by AOC-CBS, including a mixed fleet of non-convex agent shapes, moving on a graph with smooth trajectories under acceleration limits. The pre-processing step is a one-time cost that is reusable for any instance on the same graph using any number of the same types of agents. This complex system of non-convex agents on smooth trajectories took around 11 minutes to pre-process; circular agents, and straight-line trajectories, are significantly shorter (see Section 6). AOC-CBS took 5.50 milliseconds to find the optimal solution. More details can be found in Section 6.4. We compare AOC-CBS to OC-CBS on the well-known MovingAI [29] benchmark problems using 2D circular agents and straight-line traversals, showing that AOC-CBS finds optimal solutions equivalently while extending scalability from the tens to the hundreds of agents.

The remainder of this article is structured as follows: Section 2 presents existing problem formulations and solution methods, putting this work in the context of the state of the

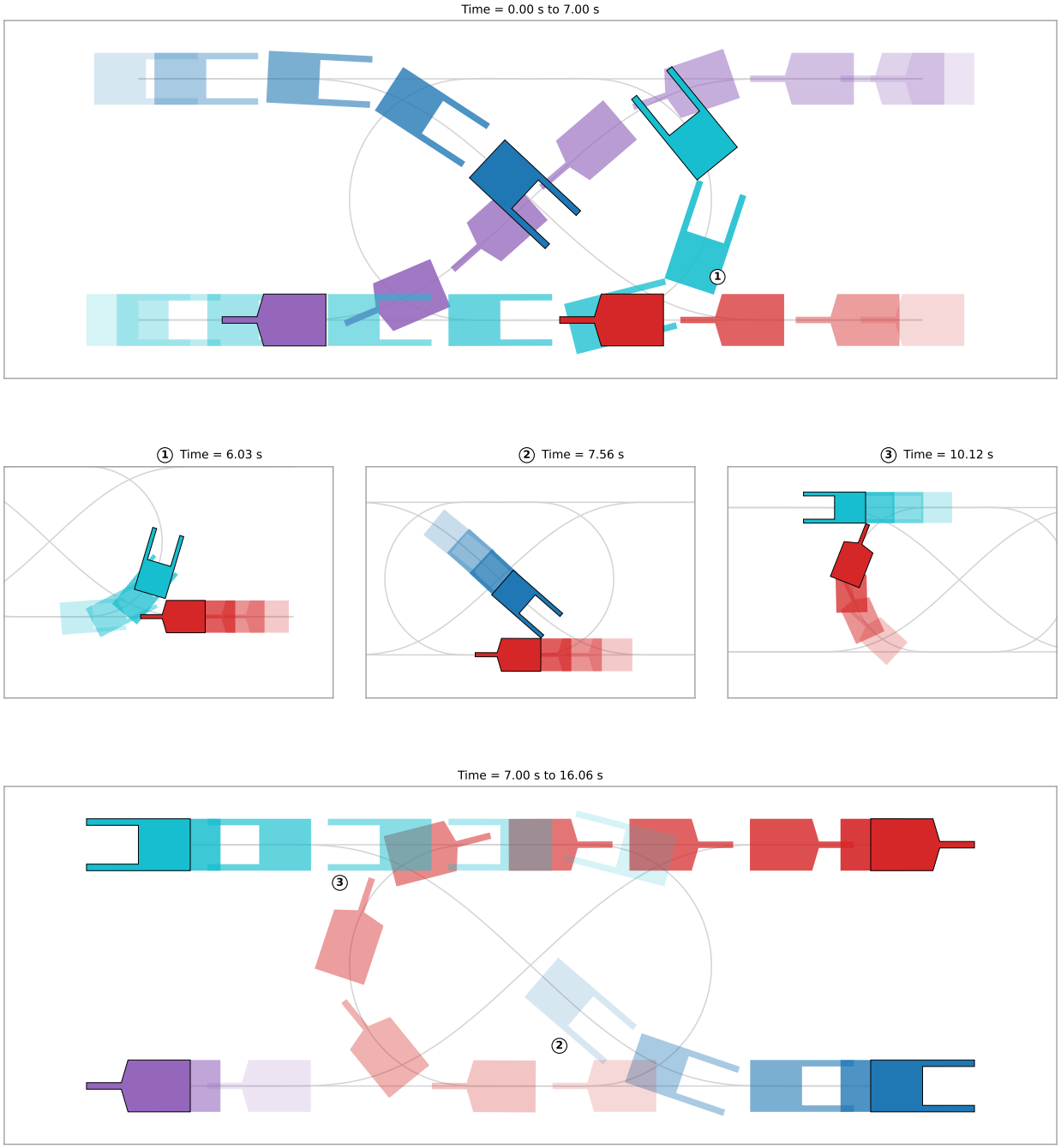


Figure 1: An example problem with two forklifts (starting on the left) and two mobile cranes (starting on the right), solved to optimality by AOC-CBS. The top figure shows the first half of the solution, with each agent's trace showing where it started; the centre figures show each of the three closest encounters between agents (enumerated) in the top and bottom figures; and the bottom shows the second half of the solution and each agent at its goal.

art. The problem formulation addressed here is formally defined in Section 3, followed by Section 4's introduction of AOC-CBS and TP-SIPP. AOC-CBS's theoretical guarantees are derived in Section 5, and Section 6 presents an experimental ablation study, comparisons with OC-CBS, and more details on the specific demonstration problem in Figure 1. Finally, discussions surrounding the experimental results and future work are found in Section 7, followed by conclusions is Section 8. For additional details regarding the specific MAPF implementation and benchmark map sampling, see Appendices A and B, respectively.

2 Background and Related Work

2.1 Existing Formulations

This section details a few relevant formulations of the many that exist for MAPF, together with some discussion regarding the implications of their assumptions.

2.1.1 Classical MAPF

The classical MAPF problem [1] is posed on an undirected and unweighted graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ with vertices $\mathcal{V}$ and edges $\mathcal{E}$. Agents in $\mathcal{A}$ exist on the graph, starting at time $t = 0$ at a unique vertex and tasked with reaching a unique goal vertex. Time is discrete, such that at every timestep an agent can either wait at its current vertex or traverse an edge to a neighbouring vertex. Two agents conflict if they are located at the same vertex or traverse the same edge in opposite directions at the same timestep. A plan π^a is a sequence of actions, one for each timestep, or agent a to execute. The plan π^a is *valid* if it respectively starts and ends at a's start and goal vertex. A joint plan $\Pi = \{\pi^a \mid a \in \mathcal{A}\}$ contains one plan for each agent, and is valid if every one of its plans is valid. A solution is a valid joint plan where no two agents conflict while following their plans. Each plan π has a *plan cost* $\sigma^\pi(\pi) \in \mathbb{R}$, which is commonly the length of the plan. The *global cost* $\sigma(\Pi)$ is a function determining the quality of a joint plan Π; an optimal solution minimises σ. Two common cost functions are

$$\text{Sum-of-Costs (SOC):} \quad \sigma(\Pi) = \sum_{\pi \in \Pi} \sigma^\pi(\pi),$$

$$\text{Makespan:} \quad \sigma(\Pi) = \max_{\pi \in \Pi} \sigma^\pi(\pi).$$

Under the common setting that MAPF is applied to mobile robots in a shared space, the assumptions in this classical MAPF formulation bring with them a number of implications. First, the physical shape and size of an agent—referred to as the agent's *volume*—is not considered; conflicts depend only on the location of agents on the graph. A real-world deployment would require $\mathcal{G}$ to be constructed so that no two agents conflict when located at different vertices or traversing different edges, thus setting a lower bound on the distance between two graph elements (assuming the agents represent non-zero sized entities) and therefore also the resolution at which the real environment can be modelled. Concretely, if $\mathcal{G}$ represents the environment with a higher resolution than the physical size of the agents,

then it would be possible for two agents to be located on different vertices in the graph while still colliding in the real system.

Second, since time is discrete, all edges take exactly one timestep to traverse. Real-world environments do not always follow such a grid-like structure that this assumption implies. Consequently, long corridors must be modeled with multiple edges when otherwise a single longer edge would suffice; if not all edges are unit-length, a real deployment would require agents to wait idly until every agent has reached its destination vertex, else the agents de-synchronize.

2.1.2 MAPF in Continuous Time

Multi-Agent Path Finding in Continuous-time (MAPF_R) [20] drops several of the assumptions made in classical MAPF, most prominently, that time is discrete. The MAPF_R is posed over a directed graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ where each vertex in $\mathcal{V}$ corresponds to a position in a metric space $\mathcal{M}$, and each edge in $\mathcal{E}$ corresponds to at least one (but possibly several) trajectories in $\mathcal{M}$, starting and ending at the source and target vertex's position, respectively. Just as in classical MAPF, agents start at a unique vertex and are assigned a unique goal vertex. However, agents can now traverse edge trajectories at any real-valued time and wait at vertices for any real-valued duration. Time being continuous means that agents can be moved arbitrarily close to each other on the graph; for instance, an agent's traversal can be timed such that it arrives at the target vertex at some miniscule ϵ time after another agent arrives there. Thus, for a MAPF_R solution to be practically relevant in a setting where the agents represent physical robots and conflicts are collisions, it is necessary to consider the agents physical volumes. However, [27] highlights that conflicts need not be strictly geometric-based; instead, conflicts can be defined by an abstract conflict-function, which is then definable for each specific application. In this work, we adopt this abstract definition of conflicts.

The MAPF_R formulation is more expressive than classical MAPF, however, there still remains several limiting assumptions. Since all agents exist on the same graph, it is assumed that every agent is able to follow every edge trajectory; the edge trajectories are therefore bounded by the most limiting motion model among the agents. Furthermore, it is assumed that every agent is able to remain indefinitely at every vertex in the graph. These assumptions on edge trajectories and vertices together practically limit the types of agents that can be modeled. Consider fixed-wing aircraft that cannot remain still while airborne. First, these cannot be modeled together with ground vehicles since ground vehicles cannot follow flight trajectories. Second, even in a setting with only fixed-wing aircraft, the only vertices that can be included are those along the ground; fix-wing aircraft generally cannot remain for an arbitrary amount of time at a position in the air. Instead, every possible trajectory from one ground-based vertex to another would have to be included as an edge-trajectory—from takeoff to landing—which is likely infeasible in real-world applications.

2.1.3 Multi-Agent Motion Planning

The *Multi-Agent Motion Planning* (MAMP) problem [18] bares many similarities with the MAPF_R problem. Time is continuous, such that edges (representing trajectories) can be traversed at any real-valued time and vertices can be occupied for any real-valued duration. However, each agent $a \in \mathcal{A}$ exists on it's *own* graph $\mathcal{G}^a = \langle \mathcal{V}^a, \mathcal{E}^a \rangle$ defined on a state space $\mathcal{S}$. Therefore, edge trajectories and vertex positions are defined in $\mathcal{S}$. This allows for an agent's graph to be tailored specifically to the agent's motion model, thereby avoiding many of the issues with all agents existing on the same graph that are present in the MAPF_R formulation. In the MAMP formulation from [18], conflicts are inherently collision-based: space is discretized into cells; each vertex and edge is associated with the set of cells that are swept by the agent's volume as they are used. A collision occurs when two agents occupy the same cell at the same time. Thus, this formulation also supports arbitrary agent volumes by the set of cells that are occupied when an agent uses a specific graph element.

Under finite-precision arithmetic, which we assume all practical implementations operate under, some level of discretization will always be present. However, the discretization of space into cells directly couple precision with computational overhead. Collision detection will always be conservative. For instance, at any level of discretization, it will always be possible for two agents to occupy the same cell without their volumes actually overlapping; this would still register as a collision. Using course cells would exacerbate the conservativeness, while using granular cells increases the computational overhead associated with tracking the usage of each of those cells. This is one motivation behind the way conflicts are handled in [27] and in this work: instead of discretizing space into cells, we instead store the incompatibility of graph elements directly. Conceptually, this is like storing a single cell for each pair of graph elements where agents on each respective element could conflict.

2.1.4 Task/goal-structure and hybrid approaches

Classical MAPF, continuous-time MAPF, and MAMP all share a further restriction, independent of how they model time, space, or graphs: every agent is assigned exactly one goal vertex, and a valid plan must end there. This single requirement carries several consequences that limit real-world applicability.

First, since every agent's plan must end at its goal, an agent cannot reach its goal and then move out of the way for other agents without returning to it afterward. Consequently, the solver is not merely tasked with planning each agent to visit its goal vertex, but with finding a mapping from the start configuration, in which every agent is simultaneously at its start, to the goal configuration, in which every agent is simultaneously at its goal. It is conceivable that some applications do not require such tight coordination; it may be sufficient for each agent to have reached its goal at some point during the plan, after which it is free to continue moving. Second, requiring every plan to end at a goal vertex also requires goals to be unique, which is incompatible with applications where several agents must, for instance, deliver goods to the same location. Third, this requirement leaves no way to represent idle agents—those that have either completed their task or had no task to begin with—without assigning them a goal. These idle agents are sometimes handled by assigning *dummy goals* [19, 30], which

introduces the further problem of deciding where a dummy goal should be placed.

Several works relax one or more of these consequences directly. The pickup-and-delivery problem [31] allows each agent two goal locations rather than one, and Multi-Goal MAPF [22] generalizes this to an arbitrary sequence of goals per agent, addressing goal uniqueness and multiplicity. Separately, [32] critiques the requirement that a solution end with all agents simultaneously at rest at their goals, arguing that many real-world problems only require each agent to visit its goal at some point, not to remain there in synchrony with every other agent. Closest to our own treatment of this issue, [19] proposes a reachability objective in which an agent's plan is permitted to continue moving after its goal has been visited, rather than assuming its final position is fixed at the goal in advance.

2.1.5 Summary of the Remaining Gaps

Throughout Sections 2.1.1–2.1.4 we have discussed the various assumptions made on a select set of MAPF variants regarding time, space, conflicts, and goal structure. We now provide a summary in Table 1 of these aspects and how this work departs from each.

Table 1: Comparison of existing MAPF formulations along the discussed axes, and where this work departs from each.

	Classical MAPF	**Continuous-time MAPF**	**MAMP**	**This work**
Time	Discrete	Continuous	Continuous	Continuous
Graph	Shared across agents	Shared across agents	Per-agent	Per-agent
Vertex waitability	All waitable	All waitable	All waitable	May be non-waitable
Conflict basis	Shared vertex/edge only	Geometric (abstractable)	Geometric (cell-based)	Abstract
Goals per agent	One, unique	One, unique	One, unique	Sequence
Post-goal behavior	Must remain at goal	Must remain at goal	Must remain at goal	May continue moving
Idle agents	Not handled	Not handled	Not handled	Handled
Task type	Vertex only	Vertex only	Vertex only	Vertex (with service time) or edge

2.2 Conflict-Based Search Solvers

This section introduces the lineage of discrete-time Conflict-Based Search, leading up to Optimal Continuous-time Conflict-Based Search.

2.2.1 Discrete-time Conflict-Based Search

The work in [33] introduces the seminal *Conflict-Based Search* (CBS), a solver for the classical MAPF problem that is *exact* (returns only optimal solutions) and *solution complete* (returns a solution if one exists, but does not necessarily report a problem to be unsolvable). CBS consist of two levels; a high-level search over a binary constraint tree (CT), and a low-level path planner. Informally, the search starts at the CT root node, representing the problem without any conflict-avoidance constraints. Every node in the CT contains a lower bound on the global cost that any solution in its sub-tree can achieve; the leaf node minimising this lower bound is selected for expansion, at which point either it contains a solution (which is shown to be the optimal solution) or new conflict-avoidance constraints are added to its

two newly spawned children. The low-level path planner is invoked at every child node to compute new plans satisfying the newly added constraints.

Specifically, every CT node N contains a set of constraints N_C and a valid joint plan N_Π, where every plan $\pi \in N_\Pi$ minimises the plan cost $\sigma^\pi(\pi)$ over all valid plans satisfying N_C. The search starts at the CT root R with $R_C = \varnothing$; every plan $\pi^a \in N_\Pi$ is the lowest-cost plan for a without considering conflicts with other agents. At every iteration of the search, the leaf node N minimizing $\sigma(N_\Pi)$ (initially R) is selected for expansion. During expansion, an arbitrary conflict in N_Π is selected, say involving agents i and j, and a *branching rule* is applied to the conflict to create constraints c_i and c_j for each respective agent is created. These constraints ensure that their corresponding agent avoids this specific conflict. For instance, c_i and c_j could respectively forbid i and j from occupying the vertex where and when the collision occurred, if the collision involved using the same vertex at the same timestep. Two child nodes to N, N^i and N^j, are subsequently spawned with $N^i_C = N_C \cup \{c_i\}$ and $N^j_C = N_C \cup \{c_j\}$. That is, i is constrained in N^i to give way to j, while j is constrained in N^j to give way to i. The joint plans N^i_Π and N^j_Π are recomputed by the low-level path planner to satisfy N^i_C and N^j_C, respectively. In practice, only i and j's plans need recomputing, since they are the only two agents with new constraints; the remaining agents' plans in N_Π already satisfy the child nodes' constraints.

Let $\mathbb{S}(N)$ be the set of all solutions consistent with the constraints in N_C, and

$$LB\left(N\right) \leq \min_{\Pi \in \mathbb{S}(N)} \sigma(\Pi) \tag{1}$$

denote a lower bound on all such solutions. The two properties underpinning CBS's exactness and solution completeness are

$$\mathbb{S}(N) \;=\; \mathbb{S}\left(N^i\right) \cup \mathbb{S}\left(N^j\right) \tag{2}$$

$$LB\left(N\right) \;\leq\; \min\left(LB\left(N^i\right), LB\left(N^j\right)\right). \tag{3}$$

Property (2) ensures soundness in the branching, such that no solutions are removed during the search. This property being satisfied depends entirely on the branching rule used to create the constraints that are added to the child nodes. For a branching to be sound, it must hold that any solution must satisfy at least one of the constraints that the branching rule creates. For instance, if the constraint c^i forbids i from occupying vertex v at timestep t, and c^j forbids j from occupying v at t also, then any solution must satisfy at least one of these constraints since otherwise i and j conflict.

Property (3) ensures admissibility in the global cost lower bound, such that the first solution encountered during the best-first CT search is indeed the optimal. In CBS, the joint plan of a CT node, N_Π, is used as the global cost lower bound: $LB\left(N\right) = \sigma(N_\Pi)$. Whether this holds or not depends on two things. First, the global cost function $\sigma(\Pi)$ must be decomposable into a function over, and non-decreasing in, individual plan costs $\sigma^\pi(\pi), \pi \in \Pi$ [27]. With these properties, ensuring that every individual plan cost is optimal ensures that the global cost is optimal. Second, the low-level planner must be able to find the optimal plan for a single agent given a set of constraints. If the low-level planner is able to do so, then at every CT node N we know that every plan $\pi \in N_\Pi$ is optimal under N_C, implying N_Π is optimal

under N_C. Finally, note that for every node N' in N's sub-tree it holds that $N_C \subseteq N'_C$, since constraints are only added and never removed during branchings. Adding constraints can never lead to lower-cost solutions, hence $\sigma(N_\Pi) \leq \sigma(N^i_\Pi)$ and $\sigma(N_\Pi) \leq \sigma(N^i_\Pi)$. Property (3) is satisfied.

2.2.2 Continuous-time Conflict-Based Search

Continuous-time Conflict Based Search (CCBS) [20] is an inexact solver [34, 27] for the MAPF_R problem, extending CBS to allow for edge traversal times, edges durations, and vertex wait durations, in real-valued time.

Besides allowing for arbitrary agent volumes and edge trajectories (theoretically), the overall structure of the CCBS solver matches closely with CBS: a two-levelled search consisting of a high-level search in the binary CT and a low-level search for finding optimal single agent plans. The main difference between CCBS and CBS can be found in the branching rule. Consider two agents i and j that conflict when respectively performing an action, an action being to either wait at a vertex or traverse an edge. Agent i executes action α^i starting at time t^i, and j executes α^j starting t^j. As described theoretically, CCBS's branching rule then finds for agent i the earliest time $t^i_u > t^i$ when α^i can be executed without conflicting with j performing α^j at t^j. Likewise, $t^j_u > t^j$ is found for j. The two constraints $c^i = \langle i, \alpha^i, [t^i, t^i_u)\rangle$ forbidding i from executing α^i starting at any time in $[t^i, t^i_u)$, and $c^j = \langle j, \alpha^j, [t^j, t^j_u)\rangle$ forbidding j from executing α^j starting at any time in $[t^j, t^j_u)$, are created and added to the constraint set of each respective child node.

The branching rule that is actually used in the publicly available implementation applies the same procedure to move actions, actions involving traversing an edge, as theoretically described. However, wait actions, involving waiting at a vertex, are treated in a different way. Consider an agent j waiting at a vertex and while doing so conflicting with agent i traversing an edge. Agent j waits at the vertex during a specific time interval, however, if j waited at the vertex for all time instead, then the interval during which i and j conflict can be recorded as the *intersection interval*. In the implemented branching rule, the constraint applied to j forbids it from occupying the vertex at any time in the intersection interval.

The low-level planner, CSIPP [20], used in CCBS is based on *Safe Interval Path Planning* (SIPP) [35], which splits the times when a vertex can be occupied or an edge can be traversed by an agent into safe and unsafe intervals. The unsafe intervals are defined by the constraints in a given constraint set, and the safe intervals are the unsafe intervals' complement. An A* [36] search is then done in the vertex-safe interval state space, collapsing the space of dense times into discrete states, and thereby returning an optimal single agent plan in continuous time.

The work in [34] finds that the theoretically described branching rule can lead to an infinite expansion of CT nodes between the root and a node representing an optimal solution, thus potentially leading to non-termination. They also find that the implemented branching rule in some cases removes solutions from the search, thus violating property (2). Counter-examples are provided in [34, 27] showing that CCBS returns a sub-optimal solutions. Despite the lack of formal guarantees, comparisons with the exact OC-CBS (described in the

next section) show that CCBS is often able to find the optimal solution faster due to its more aggressive search space pruning [27].

2.2.3 Optimal Continuous-time Conflict-Based Search

Optimal Continuous-time Conflict-Based Search (OC-CBS) [27] restores exactness and solution completeness to CCBS by introducing a new branching rule, δ-BR, that changes how conflicting wait actions are handled.

δ-BR is based on shifting constraints from [34] and applies the same rule as CCBS to conflicts involving two move actions. However, when applied to a conflict involving a move and a wait action, δ-BR finds a specific value δ based on the intersection interval $I = [t_1^I, t_2^I)$ and the time interval during which the waiting agent occupies its vertex. Then, for the moving agent i executing the move action α^i at time t^i, a constraint c^i is created forbidding i from executing α^i starting at any in $[t^i, t^i + \delta)$. For the other agent j waiting at v, a constraint c^j is created forbidding j from occupying v at any time in $[t_1^I + \delta, t_2^I)$ and from executing any move action from v starting at any time in $[t_1^I + \delta, t_2^I)$. It is shown in [27] that OC-CBS is exact and guaranteed to return a solution in finite time, under the assumption that a solution exists.

Essentially, the work in [34] identifies that properties (2) and (3) are not sufficient in continuous time to guarantee that a solution will be returned. That is, the theoretically described CCBS branching rule satisfies both, yet allows for the possibility of an infinite number of nodes between the CT root and one representing an optimal solution. This problem is not encountered in CBS since time is discretised, thereby allowing for only a finite number of steps between the root and an optimal solution. Consequently, an additional property is introduced in [27] to ensure that an optimal solution is reachable in a finite number of steps in the CT. That is, the following three properties must be satisfied for OC-CBS to return an optimal solution in finite many iterations (restating properties (2) and (3) for ease of reference):

CT-I No solutions are removed from the search:

$$\mathbb{S}(N) \;=\; \mathbb{S}\left(N^i\right) \cup \mathbb{S}\left(N^j\right).$$

CT-II The global cost lower bound monotonically increases when descending the CT:

$$LB\left(N\right) \;\leq\; \min\left(LB\left(N^i\right), LB\left(N^j\right)\right).$$

CT-III For every infinite path of nodes $N_1, N_2, N_3, \ldots$ in the CT, where N_i is the parent of N_{i+1}, and any $c \in \mathbb{R}$,

$$\exists k \in \mathbb{N} : \forall i \geq k : c < LB\left(N_i\right).$$

That is, for **CT-III**, constraints are accumulated along the path $N_1, N_2, N_3, \ldots$; it must hold that the constraints that accumulate must eventually push the global cost lower bound

beyond some threshold c, wherever that threshold may be. Intuitively, this means that as the search continues, either such a path will lead to the optimal solution, or it will be ignored in the best-first search in favour of other paths with lower global cost lower bounds in which the optimal solution lies.

2.3 Suboptimal, Bounded-Suboptimal, and Anytime Methods

We introduce methods that find sub-optimal solutions, with additionally bounds on sub-optimality that sub-optimality, an *anytime* property of providing improving solutions over runtime, and possibly guarantees of eventual convergence to an optimal solution.

2.3.1 Prioritized SIPP

Prioritized SIPP (PSIPP) [28] applies prioritized planning to a continuous-time MAPF formulation with circular agents and straight line constant-speed traversals. Agents can traverse edges at any real-valued time, and edge traversals can have any positive real-valued duration. Each agent starts at a unique vertex and is assigned a unique goal vertex, with no two agents at their goal vertices conflicting with each other. Given some ordering of the agents, PSIPP plans the movement of one agent at a time, using SIPP, to move from its starting vertex to its goal vertex while avoiding collisions with other previously planned agents.

PSIPP outperforms CCBS for fast suboptimal solutions, on some maps planning more than 500 agents in less than a second. However, PSIPP is inherently a heuristic method that is subject to the initial agent ordering, meaning that the returned solutions have no guarantees of optimality or of even finding a solution.

2.3.2 Enhanced Conflict-Based Search in Continuous Time

Enhanced Conflict-Based Search in Continuous Time (ECBS-CT) [18] is a bounded suboptimal solver for the MAMP problem, building on the *Enhanced CBS* solver [37].

Given a suboptimality bound $w \geq 1$, ECBS-CT searches for a solution Π with $\frac{\sigma(\Pi)}{\sigma^*} \leq w$, where σ^* is the optimal global cost, such that Π's suboptimality is at most w. Setting $w = 1$ collapses the method into searching for an optimal solution. It does this by applying the general CBS framework, with a high-level search in a binary constraint tree and a low-level single-agent path planner, *Soft Conflict Interval Path Planning* (SCIPP), based on SIPP. However, instead of always selecting the CT node minimising the global cost, ECBS-CT applies a focal search: with σ_{min} being the minimum $LB(N)$ over all CT leaf nodes N, the search selects from the set of nodes satisfying $LB(N) \leq w\sigma_{min}$. Among these satisfying leaf nodes, a secondary heuristic minimising the total duration of time intervals in which two or more agents collide is applied. That is, ECBS-CT focuses the search not on the nodes with highest potential solution quality, but instead on the nodes with satisfactory cost bounds and which are nearest to representing a solution.

Like SIPP, SCIPP performs an A* search in the vertex-safe interval state space, with safe intervals defined by the set of *hard* constraints given by the high-level search. However, SCIPP applies a similar focal search as in the high-level search: instead of always selecting

the state with minium total estimated path cost f_{min}, any state with $f \leq w \cdot f_{min}$ is eligable for selection. Among these, the secondary heuristic selects the state minimising the number of *soft conflicts*, that is, conflicts with other agents in the high-level node's joint plan.

On these two search levels, ECBS-CT trades some of the slack afforded by w for joint plans that are closer to being solutions, and plans that pre-emptively avoid conflicts. The results show that by increasing from $w = 1$ to $w = 1.2$ (20% optimality gap) and beyond, runtime decreases and success rate increases, such that plans can be found in some cases for over 10 times as many agents than otherwise.

Although this focal search mechanism allows ECBS-CT to trade optimality for a substantial gain in scalability, the suboptimality bound w must be fixed before the search begins, and lowering it to tighten the bound requires restarting the search from scratch. Anytime or incremental variants that retain this work across values of w are not part of the ECBS-CT framework.

2.3.3 Anytime Methods in Discrete Time

An anytime solver makes a solution available early and improves it as time allows. Where a bounded-suboptimal solver fixes its quality target before the search begins, an anytime solver maintains an incumbent solution whose quality improves monotonically and, ideally, converges to optimal. Work on anytime MAPF in discrete time follows three broad approaches.

The first keeps the CBS structure and makes the bounded-suboptimal search itself anytime. Anytime Focal Search (AFS) [38] iteratively tightens the suboptimality factor w while reusing the search effort already expended under looser factors. Applied by its authors to the high level of BCBS [37], it gives what later work [39] calls anytime BCBS, and what [38] describes as the first anytime MAPF solver. Notably, AFS reports each incumbent together with the ratio of its cost to the smallest f-value remaining in the open list, and prunes states whose cost exceeds the current incumbent—so the incumbent, the bound certifying it, and the pruning rule all derive from the single focal search over the constraint tree.

The second approach replaces systematic search with local search over complete solutions. MAPF-LNS [39] computes an initial solution with a fast suboptimal method, then repeatedly removes the plans of a subset of agents, selected by a randomized removal heuristic, and repairs them by prioritized planning, keeping the result whenever it lowers the cost. Subsequent work largely differs in how the subset is chosen or how the repair is performed: MAPF-LNS2 [40] repairs plans that still contain collisions rather than requiring a feasible starting solution, while [41] and [42] learn, respectively via supervised learning and multi-armed bandits, which neighbourhoods are worth replanning. These methods scale to hundreds of agents, but the incumbent is never accompanied by a lower bound on the optimal cost, so nothing is known about how far from optimal a returned solution is, nor is optimality reached in the limit.

The third performs an anytime search directly over the joint configuration space. LaCAM* [43] uses lazy successor generation to return an initial solution very quickly, and then continues searching while revising parent relations among search nodes, converging to an optimal solu-

tion for costs accumulated over transitions. It is thus both scalable and eventually optimal, though—as with the first approach—the guarantee is stated over the enormous discrete-time configuration space.

Two observations carry over to this work. First, all of the above assume the classical MAPF setting of Section 2.1.1: discrete time, a shared graph, and one goal per agent at which the plan ends. Second, the destroy-and-repair idea underlying LNS is close in spirit to the repair methods in AOC-CBS's portfolio (Section 4.3): both take an existing joint plan and replan a subset of agents around the remainder. The difference lies in where the plan comes from and what is known about the result. In LNS the repaired plan is the incumbent itself and the repair is the only source of information about solution quality; in AOC-CBS the plan being repaired is a CT node's joint plan, and the search that produced it simultaneously supplies a lower bound, so any repaired solution can be reported together with a certified optimality gap.

2.3.4 Anytime Methods in Continuous Time

To our knowledge, no anytime solver has been proposed for the continuous-time formulations of Section 2.1. That literature is instead split between exact solvers that return nothing at all before the optimum is found (CCBS [20], OC-CBS [27]), bounded-suboptimal solvers whose bound is fixed before the search (ECBS-CT [18]), and unbounded-suboptimal prioritized planners that return a single solution and stop (PSIPP [28], CPLP [44]).

The reason is largely one of timing. Any search-based anytime method reports its incumbent against a lower bound, and in continuous time that bound is trustworthy only under **CT-I**–**CT-III**. CCBS was believed to satisfy these until recently, when [34] showed its branching rule violates the first and third; only with OC-CBS's branching rule were the properties established on a continuous-time constraint tree at all. AOC-CBS is, to our knowledge, the first anytime method built on this foundation.

LNS-style repair faces no comparable obstacle when agents are required to remain at their goal once reached: a continuous-time prioritized planner such as PSIPP [28] is a perfectly serviceable repair primitive under that restriction, and nothing here rules out a continuous-time analogue of MAPF-LNS. To our knowledge none has been published. PSIPP's actual limitation is orthogonal to LNS suitability: it assumes each agent's final position is fixed at its goal, so it cannot vacate a completed task to let another agent through (Section 4.3). This is what our modified version of PSIPP, *Tier-Prioritized SIPP*, is built to resolve. The limitation is not specific to LNS or to PSIPP: it would affect any repair method, prioritized or otherwise, once agents are allowed to move on after completing their tasks. Configuration-space search of the LaCAM* kind relies on a finite successor set per configuration, which is not immediate when action durations are real-valued.

A focal-search adaptation in the manner of [38] is now equally possible, and would in fact be expressible within AOC-CBS as one selection policy ψ_{sel} among others, retaining correctness through Theorems 5.4 and 5.5. What distinguishes AOC-CBS is not the constraint tree but the decoupling: focal search derives incumbent and bound from a single node ordering, whereas here the bound comes from the CT search and the incumbent from repair, so

heuristics may be added freely without bearing on correctness.

2.3.5 Kinodynamic Feasibility: Repairing After Search versus Searching Within the Feasible Space

A separate line of work attains continuous, kinodynamically feasible motion by planning in a simpler space and recovering feasibility afterwards. *Concrete Multi-Agent Path Planning* [45] plans discrete paths with Tree-LaCAM, an anytime and complete solver built on LaCAM/LaCAM* [43], converts them into trajectories using learned robot dynamics, and executes them under optimal control that repairs residual collisions; a heterogeneous fleet of 40 physical ground and aerial robots is coordinated this way, concurrently and onboard, at a hardware scale no continuous-time planner operating directly on trajectories has been deployed at. In multi-robot kinodynamic motion planning, db-CBS [46] takes a related route, searching over precomputed motion primitives with a bounded discontinuity between them, optimizing the resulting trajectory in the joint space, and repeating with a reduced discontinuity bound; the method is anytime and asymptotically optimal.

AOC-CBS makes the opposite trade. Kinodynamic feasibility is a property of graph construction (Section 3.1) rather than of planning, so the search runs directly in the space of motions the agents can realize. Any incumbent is executable by construction, with no trajectory-generation or repair stage after the fact, and the reported optimality gap is with respect to the agents' graphs. What is given up is the ability to improve upon that fixed discrete abstraction: solution quality is bounded by the trajectories present in the graphs, and enriching them moves cost into the precomputation stage (Section A). Such a formulation is appealing in tightly constrained or safety-critical settings, where stronger end-to-end feasibility guarantees may justify the additional computational burden.

Taken together, anytime methods are well developed for discrete-time MAPF but absent from its continuous-time counterparts, while the methods that do plan kinodynamically feasible motion obtain their anytime behaviour from a discrete abstraction and restore feasibility only afterwards. AOC-CBS sits at the intersection: anytime, with a certified optimality gap, over a formulation that is continuous in time and feasible by construction.

3 Problem Formulation

This problem formulation combines and extends those introduced in Section 2.1 to allow for continuous-time movement of a fleet of heterogeneous agents along a discrete set of trajectories in state spaces, with arbitrary definitions of conflicts, sequences of tasks at vertices and edges, and several other practically motivated generalizations.

A graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ is a directed multi-graph over a state space $\mathcal{S}$ which may be unique to $\mathcal{G}$ (although we do not denote this for brevity). Every vertex $v \in \mathcal{V}$ corresponds to a unique state $v_s \in \mathcal{S}$, and $\mathit{waitable}(v) \in \{\mathit{true}, \mathit{false}\}$ denotes if v_s is a state that can be remained at indefinitely or not. For instance, it is not possible to remain in a state that contains both a measure and its non-zero derivative. Each edge $e \in \mathcal{E}$ has a source vertex $\cdot\overset{v}{\rightarrow}(e) \in \mathcal{V}$ and a target vertex $\overset{v}{\rightarrow}\cdot(e)$; since $\mathcal{G}$ is a multi-graph, there may exist multiple edges with the same

source and target vertices. Edge e corresponds to a trajectory $e_\tau : [0, e_D) \mapsto \mathcal{S}$ mapping time to the state space, with duration $e_D \in \mathbb{R}_{>0}$, starting at state $e_\tau(0) = \overset{v}{\cdot\!\rightarrow}(e)_s$ and ending at state $e_\tau(e_D) = \overset{v}{\rightarrow\!\cdot}(e)_s$.

An agent a from the set of all agents $\mathcal{A}$ exists on a possibly unique graph $\mathcal{G}^a = \langle \mathcal{V}^a, \mathcal{E}^a \rangle$ defined over a state space $\mathcal{S}^a$. The agent a being located at vertex $v \in \mathcal{V}^a$ means that it is in the state v_s; traversing edge $e \in \mathcal{E}^a$ means that it follows the state trajectory e_τ. Initially, agent a is located at $v^a_{start} \in \mathcal{V}^a$ at time $t = 0$; we do not assume *waitable*(v).

An action α is executed by an agent and can mean to either traverse an edge at a specific time or wait at a vertex during some time interval. A *move action* $m = \langle e, t \rangle \in \mathcal{E} \times \mathbb{R}$ means to begin traversing e at time t. That is, m's trajectory is the same as e's trajectory but shifted in time: $m_\tau(t') = e_\tau(t' - t)$ defined over $t' \in [t, t + e_D)$. We say that m's source and target vertices are the same as e's: $\overset{v}{\cdot\!\rightarrow}(m) = \overset{v}{\cdot\!\rightarrow}(e)$ and $\overset{v}{\rightarrow\!\cdot}(m) = \overset{v}{\rightarrow\!\cdot}(e)$. We also denote m's start and end time with $\overset{t}{\cdot\!\rightarrow}(m) = t$ and $\overset{t}{\rightarrow\!\cdot}(m) = t + e_D$, and its duration matching e's duration, $m_D = e_D$.

A *wait action* $w = \langle v, t_1, t_2 \rangle \in \mathcal{V} \times \mathbb{R} \times (\mathbb{R} \cup \{\infty\})$ means to wait at v during the time interval $[t_1, t_2)$. Only wait actions with a positive duration ($t_1 < t_2$) at a vertex that can be waited at (*waitable*(v)) are considered. A wait action with $t_2 = \infty$ is referred to as an *infinite wait action*. Matching the notation of move actions, w's trajectory maps to the state of v over the full waiting interval: $w_\tau(t') = v_s$ defined over $t' \in [t_1, t_2)$. We also denote $\overset{v}{\cdot\!\rightarrow}(w) = \overset{v}{\rightarrow\!\cdot}(w) = v$, the start time $\overset{t}{\cdot\!\rightarrow}(w) = t_1$, the end time $\overset{t}{\rightarrow\!\cdot}(w) = t_2$, and the duration $w_D = t_2 - t_1$.

Every action α has a *cost* α_σ. In the majority of the MAPF literature, this cost equals the action's duration, meaning $m_\sigma = m_D$ for a move action and $w_\sigma = w_D$ for a wait action. However, in some settings the cost is not measured in time but energy, such that waiting does not cost while moving does. We leave action cost as a domain-specific attribute.

With superscript α^a we mean that the action is executed specifically by agent $a \in \mathcal{A}$, such that $e \in \mathcal{E}^a$ (for a move action) and $v \in \mathcal{V}^a$ (for a wait action). However, we omit the superscript for notational simplicity whenever it is otherwise clear.

A task sequence $\Lambda = \langle \lambda_1, \lambda_2, \ldots, \lambda_N \rangle \in ((\mathcal{V} \cup \mathcal{E}) \times \mathbb{R}_{\geq 0})^N$ is a sequence of N *tasks*, with a task meaning to either traverse an edge or to wait at a vertex for at least a specific amount of time. Tasks in a task sequence must be completed in order and one-at-a-time; only once task λ_i is completed can the traversing of an edge or waiting at a vertex to complete λ_{i+1} begin. For instance, if both λ_i and λ_{i+1} involve waiting at the same vertex, only once λ_i is complete does the time start counting toward λ_{i+1}.

An *edge task* $\lambda = \langle e, e_D \rangle$ with $e \in \mathcal{E}$ is completed by an agent once the agent traverses e. Continuing with the same notation as for actions, we say that λ's source and target vertices match the edge e: $\overset{v}{\cdot\!\rightarrow}(\lambda) = \overset{v}{\cdot\!\rightarrow}(e)$ and $\overset{v}{\rightarrow\!\cdot}(\lambda) = \overset{v}{\rightarrow\!\cdot}(e)$. Such an edge task's duration is denoted $\lambda_D = e_D$.

A *vertex task* $\lambda = \langle v, D \rangle$ with $v \in \mathcal{V}$ is completed by an agent once the agent has been located at v for at least D time. For such a vertex task we denote $\overset{v}{\cdot\!\rightarrow}(\lambda) = \overset{v}{\rightarrow\!\cdot}(\lambda) = v$ and its duration $\lambda_D = D$.

Just as with actions, the superscript $\Lambda^a = \langle \lambda_1^a, \lambda_2^a, \ldots, \lambda_N^a \rangle$ denotes that agent $a \in \mathcal{A}$ is assigned to Λ^a. This means that $e \in \mathcal{E}^a$ (for edge tasks) and $v \in \mathcal{V}^a$ (for vertex tasks). Also, number of tasks N may be unique to each agent. When otherwise clear from context, we omit the superscript for notational simplicity.

A plan $\pi^a = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$ (with n possibly unique to a) is a sequence of actions for agent $a \in \mathcal{A}$ to execute in series. The duration of π^a is the sum of its actions' durations, $\pi^a_D = \sum_{\alpha \in \pi^a} \alpha_D$, however excluding any final infinite wait action if present. We say that π^a is *valid* if the following conditions are true:

- The first action α_1 starts at a's starting vertex and at time $t = 0$:

$$\overset{v}{\cdot\!\rightarrow}(\alpha_1) = v^a_{start} \ \wedge \ \overset{t}{\cdot\!\rightarrow}(\alpha_1) = 0.$$

- Every successive action is connected to the previous in time and place:

$$\overset{t}{\rightarrow\!\cdot}(\alpha_{k-1}) = \overset{t}{\cdot\!\rightarrow}(\alpha_k) \ \wedge \ \overset{v}{\rightarrow\!\cdot}(\alpha_{k-1}) = \overset{v}{\cdot\!\rightarrow}(\alpha_k) \quad \forall k \in [2, .., n].$$

- All tasks in $\Lambda^a = \langle \lambda_1, \lambda_2, \ldots, \lambda_N \rangle$ are completed in order and one-at-a-time: there exists an ordered set of times $\langle t_1, t_2, ..., t_N \rangle \in \mathbb{R}^N_{\geq 0}$ (with t_i being the start time of λ_i) where

$$t_1 \leq t_1 + \lambda_{1,D} \leq t_2 \leq t_2 + \lambda_{2.D} \leq \cdots \leq t_N$$

  for which the following holds:

  - For every edge task $\lambda_i = \langle e, e_D \rangle \in \Lambda^a$ there exists a move action $m = \langle e, t_i \rangle \in \pi^a$.
  - For every vertex task $\lambda_i = \langle v, D \rangle \in \Lambda^a$, it holds that if $D = 0$ then the agent either starts in v or at some time moves to it,

$$v^a_{start} = v \ \vee \ \exists m \in \pi^a : \overset{v}{\rightarrow\!\cdot}(m) = v,$$

    otherwise if $D > 0$ then the agent waits at v for at least D,

$$\exists w \in \pi^a : \ \overset{v}{\cdot\!\rightarrow}(w) = v \ \wedge \ \overset{t}{\cdot\!\rightarrow}(w) \leq t_i < t_i + \lambda_{i,D} \leq \overset{t}{\rightarrow\!\cdot}(w)$$

- The final action α_n is an infinite wait action ($\alpha_{n,D} = \infty$). This is to model that agents are assumed to remain idle once completing all of their given instructions.

Without loss of generality, we assume that no two subsequent actions $\alpha_i, \alpha_{i+1} \in \pi^a$ are both wait actions, since they can equivalently be merged into one.

Conflicts are fundamentally defined by a function $\mathit{InConflict}\,(i, s_i, j, s_j) \in \{\mathit{true}, \mathit{false}\}$, denoting if agent $i \in \mathcal{A}$ in state $s_i \in \mathcal{S}^i$ conflicts with agent j in state $s_j \in \mathcal{S}^j$ or not. We leave it implicit that no agent can conflict with itself. This problem formulation makes no assumptions on the reason for such a joint state to be a conflict. A conflict in the typical MAPF setting would mean that the agents' physical volumes intersect each other—a *geometric collision*. However, this formulation also allows for other types of conflicts. It may for instance be that i and j are both airborne drones, and this configuration unacceptably

places one agent in the downwash of the other. All application-specific reasons for joint states being a conflict or not are abstracted into *InConflict*. Thus, this formulation generalizes beyond agent volumes.

A *conflict* $\langle \alpha^i, \alpha^j \rangle$ is defined to mean that the states of i and j conflict while respectively executing α^i and α^j. That is, for some time t when both α^i_τ and α^j_τ are defined, $\mathit{InConflict}\,(i, \alpha^i_\tau(t), j, \alpha^j_\tau(t))$ holds. We use the mapping $\mathit{IsConflict}\,(\alpha^i, \alpha^j) \in \{\mathit{true}, \mathit{false}\}$ (*is* conflict instead of *in* conflict) to denote if $\langle \alpha^i, \alpha^j \rangle$ is a conflict or not. Similarly, we abuse the notion to let $\mathit{IsConflict}\,(\pi^i, \pi^j)$ denote if $\exists \langle \alpha^i, \alpha^j \rangle \in \pi^i \times \pi^j$ such that $\mathit{IsConflict}\,(\alpha^i, \alpha^j)$.

A joint plan $\Pi = \{\pi^a \mid a \in \mathcal{A}\}$ contains one plan for each agent. If all plans $\pi^a \in \Pi$ are valid, then Π is valid.

A solution is a joint plan Π for which $\neg \mathit{IsConflict}\,(\pi^i, \pi^j)$ for all $\pi^i, \pi^j \in \Pi$. An *optimal solution* Π^* is a solution minimising a *global cost* function σ over all solutions.

As discussed in Section 2.2.1, the correctness guarantees of OC-CBS rely on properties **CT-I–CT-III** being satisfied. This requires $LB\,(N) = \sigma(N_\Pi)$ for every CT node N. That is, no solution consistent with N_C can have a lower cost than $\sigma(N_\Pi)$. For this to be satisfied, two things must hold. First, we assume that $\sigma(\Pi)$ is decomposable into a function over, and non-decreasing in, individual plan costs $\sigma^\pi(\pi), \pi \in \Pi$. Second, we assume that a low-level planner is available to compute a single-agent plan π minimising $\sigma^\pi(\pi)$ over all plans consistent with a given set of constraints N_C. In our low-level planner, TP-SIPP (Section 4.3), our assumptions are consistent with the majority of the MAPF literature: action costs equal their duration, $\alpha_\sigma = \alpha_D$, and a plan's cost is equal to the sum of its actions' costs, $\sigma^\pi(\pi) = \sum_{\alpha \in \pi} \alpha_\sigma$. However, any other cost definitions may require adapting the low-level planner to guarantee that it returns an optimal single-agent plan.

3.1 Model Generality

Based on the problem formulation above, each agent moves on a possibly unique graph $\mathcal{G}^a$ defined over a possibly unique state space $\mathcal{S}^a$. Nothing in this definition ties $\mathcal{S}^a$ to a spatial pose, a rigid body, or even a physical robot: a state space is an arbitrary set, and a graph is nothing more than states and inter-state trajectories the agent can realize. The formulation is therefore agnostic to what an agent *is*—it may be a vehicle, a robotic arm operating in joint-configuration space, an aircraft, or an entity with no physical embodiment at all, such as a scheduled job for which *InConflict* encodes resource contention rather than spatial overlap.

All differential and kinematic constraints on an agent's motion—nonholonomy, actuation limits, dynamics, whatever they may be—are captured entirely by which vertices and edges are present in $\mathcal{G}^a$, not by any separate mechanism in the plan or solution definitions. Kinodynamic feasibility is thus a property of graph construction, not of planning: once $\mathcal{G}^a$ is fixed, every edge in it is by definition traversable, and the rest of the formulation (actions, tasks, plans) reasons only over this discrete abstraction. This separates the agent-specific, possibly continuous problem of generating feasible motion primitives from the agent-agnostic, discrete problem of coordinating them.

Conflicts are likewise defined without reference to what an agent is, only to the states two agents occupy: $\mathit{InConflict}\,(i, s_i, j, s_j)$ takes just a pair of states as argument, internalizing whatever embodiment-specific test (e.g., geometric intersection of swept volumes, a downwash cone, an interference footprint) determines whether that pairing is acceptable. Agent volumes are thus not primitive to the formulation, only one possible implementation of *InConflict* for a particular class of agents. This is what admits heterogeneous agents in the first place: two agents need not share a state space, a motion model, or a notion of physicality, only a well-defined *InConflict* function relating their respective states.

We restrict conflicts to be pairwise, and take *InConflict* to be symmetric: $\mathit{InConflict}\,(i, s_i, j, s_j) = \mathit{InConflict}\,(j, s_j, i, s_i)$ for all arguments. This is not a claim that both agents are equally *affected*, only that the joint configuration itself is unacceptable, regardless of which agent, if either, bears the consequence. A drone flying through another's downwash may disturb only the former; the joint behavior is still what is considered unacceptable. Symmetry follows because a conflict records only whether unacceptable behavior occurred, not who caused it or who suffered it.

Because each agent's state space may be unique, defining $\mathit{InConflict}\,(i, s_i, j, s_j)$ requires being able to relate s_i and s_j to each other: it must be stated concretely what about their joint occupation of these states is unacceptable. We take this as a modeling obligation rather than a limitation of the formulation: a conflict that cannot be described in terms of the two agents' states is not a well-defined conflict. This obligation grows with the number of distinct agent-type pairings in $\mathcal{A}$, but that cost is intrinsic to any formulation admitting heterogeneous agents and non-geometric conflicts, not an artifact of how this formulation is posed.

The final action of a valid plan is required to be an infinite wait action at a waitable vertex. This requirement may seem like an artificial restriction; after all, an agent having completed all of its tasks is free to do what it wants thereafter. We impose it deliberately as a hard stand-in for a softer requirement that a solution must not leave the system in a configuration from which a conflict could still arise later. Terminating every plan in an indefinitely-waitable state (i.e., leaving the system at rest) closes off that possibility without requiring the solver to reason explicitly about an unbounded horizon past each agent's last action.

4 Anytime-Optimal Continuous-Time Conflict-Based Search

AOC-CBS extends OC-CBS to a richer problem formulation, with the main difference lying in the high-level search.

OC-CBS targets only optimal solutions. At each iteration, it selects the CT node minimizing the objective function for expansion. If that node is a solution, it is returned as optimal; otherwise, it spawns two children and the iteration ends. This continues until an expanded node represents a solution.

OC-CBS's design leaves much of the information already present in the CT unused. A node

N is selected because $LB(N)$ is currently known solution lower bound. During expansion, two children are spawned, OC-CBS does not check whether any of these children represent a solution; it a child does represent a solution Π, and $\sigma(\Pi) = LB(N)$, then already here we know that Π is optimal. Instead of halting with this optimal solution, OC-CBS will continue iterating until the child node is selected. Even if $\sigma(\Pi) > LB(N)$, Π may still be optimal, only that its optimality has not yet been confirmed. Regardless, Π can be made available as an incumbent with an optimality gap upper bound $\sigma(\Pi)/LB(N)$. Furthermore, OC-CBS is also inherently sequential, expanding one CT node at a time, which does not allow for the use of parallel computations.

AOC-CBS addresses both limitations. First, the child nodes at every expansion are checked for a solution; if one represents a solution that is better than the incumbent, then it replaces the incumbent. Second, AOC-CBS allows for the expansion of multiple CT nodes simultaneously on separate processors, with the results of these expansions aggregated to maintain correctness guarantees (Section 5 provides correctness proofs). Informally, the nodes are selected and expanded according to a configurable *portfolio*. This portfolio consists of an arbitrary number of *policies*, with each policy managing the expansion of one node. The policy operations are independent of each other, thus, opening the possibility for parallel computations. A policy specifies a *selection policy* that determines which of the CT leaf nodes are selected, and a *set of repair methods* that are applied to the CT node's joint plan, in addition to the regular branching using δ-BR. AOC-CBS is agnostic to the selection policy and repair methods, hence it remains open to future repair methods. These kinds of configurable methods have been widely used in the literature, such as RHCR [47] and LNS [39], while the ability to incorporate other CT node selection policies than by minimum cost lower bound has been useful when searching for sub-optimal solutions fast [38, 18].

4.1 High-Level Search

The high-level search is done over the CT using a modular search portfolio $\Psi = \langle \psi^1, \psi^2, \ldots, \psi^p \rangle$ containing a finite number of policies. Each policy $\psi^i \in \Psi$ may be unique, consisting of a selection policy ψ^i_{sel} and a set of repair methods $\psi^i_{\mathbf{R}}$:

- **Selection policy**: ψ^i_{sel} controls which unexpanded CT node is selected for expansion. In OC-CBS, the default policy is to expand a node minimizing the global cost lower bound $LB(N)$. In contrast, AOC-CBS allows for any selection policy, including, for instance, random selection, selection by fewest collisions, selection by least amount of time in conflict, and so on. Note however that AOC-CBS's correctness guarantees (Section 5) requires at least one selection policy to be by minimum $LB(N)$, that is until an incumbent solution has been found, after which any selection policy can be used.
- **Set of repair methods**: $\psi^i_{\mathbf{R}} = \{r^{i,1}, r^{i,2}, \ldots\}$ contains a finite number of repair methods. Each repair method $r^{i,j} \in \psi^i$ takes an existing joint plan as input and produces either a solution—which may or may not be optimal—or a report of failure.

All policies in Ψ are independent of each other, and all repair functions in $\psi_{\mathbf{R}}$ are also independent, allowing for parallel execution.

At iteration k of AOC-CBS, the set $\mathcal{Q}_k$ contains all CT leaf nodes, and Π_k^{UB} denotes the best solution found so far, with $UB_k = \sigma\left(\Pi_k^{UB}\right)$ if a solution has been found and $UB_k = \infty$ otherwise. Initially $\mathcal{Q}_0 = \{R\}$ contains the root node. Let $LB_k = \min_{N \in \mathcal{Q}_k} LB(N)$ denote the minimum cost lower bound over all leaf nodes if $\mathcal{Q}_k \neq \varnothing$, otherwise $LB_k = UB_k$. Each policy $\psi \in \Psi$, in turn, selects a node from $\mathcal{Q}_k$ to expand according to its selection rule ψ_{sel}; once selected, a node becomes unavailable to subsequent policies. We denote by $\mathcal{Q}^{sel}$ the set of nodes selected this way. This continues until every policy has selected a node or no selectable nodes remain in $\mathcal{Q}_k$. Each policy ψ then independently applies δ-BR to its selected node, spawning two children; the set of all such children is denoted $\mathcal{Q}^{new}$. Each policy also applies its repair methods in $\psi_{\mathbf{R}}$ to its selected CT node, and additionally checks whether each child's joint plan is a solution. Any solutions produced this way, whether through repair or from the children, are collected in the set $\mathbb{S}^{new}$. After all selected nodes have been processed, the incumbent solution for the next iteration is

$$\Pi_{k+1}^{UB} = \underset{\Pi \in \mathbb{S}^{new} \cup \{\Pi_k^{UB}\}}{\arg\min} \sigma(\Pi)$$

with the new upper bound $UB_{k+1} = \sigma\left(\Pi_{k+1}^{UB}\right)$ and the set of unexpanded nodes

$$\mathcal{Q}_{k+1} = \left\{N \in (\mathcal{Q}_k \setminus \mathcal{Q}^{sel}) \cup \mathcal{Q}^{new} \mid LB(N) < UB_{k+1}\right\}$$

where all nodes with global cost lower bound greater or equal to UB_{k+1} pruned.

Section 5 shows that at every iteration k of AOC-CBS on a solvable problem instance with optimal solution Π^* and optimal objective value σ^*, it holds that

- $LB_k \leq \sigma^* \leq UB_k$;
- $\Delta^{UB} = UB_k / LB_k - 1$ is a true optimality gap upper bound on Π_k^{UB};
- if $\mathcal{Q}_k = \varnothing$ then $\sigma(\Pi_k^{UB}) = \sigma^*$ and Π_k^{UB} is optimal; otherwise
- if $\mathcal{Q}_k \neq \varnothing$ then $\mathcal{Q}_k$ contains a CT node from which Π^* is reachable.

It is also shown that if either an incumbent solution has been found or at least one policy $\psi \in \Psi$ always selects a node $N \in \mathcal{Q}_k$ minimizing $LB(N)$, then AOC-CBS is guaranteed to return an optimal solution in a finite number of iterations.

Once an initial incumbent solution has been found, AOC-CBS provides a solution at any time with a known upper bound on its optimality gap. Given time, LB_k will progressively increase due to the expansion of CT nodes, and UB_k will progressively decrease due to better solutions being found (be that via the repair functions or due to the expansion of the CT nodes). Within finitely many iterations, these bounds will meet and an optimal solution will be found.

We make a final note on the portfolio: Ψ does not need to be static over a search; each iteration of the search may use a unique portfolio. For instance, Section 6 details experiments using

- a portfolio for quickly finding an incumbent (mixed selection policies for increased diversity in the nodes being selected, and every policy using a repair function),

- then, at an increasing rate as Δ^{UB} decreases, using a portfolio for raising the global solution cost lower bound LB as fast as possible (all policies selecting by minimum cost lower bound $LB\,(N)$, no repair function to decrease iteration time).

This provides merely one example of many possible non-static portfolios.

4.1.1 Cardinal Conflicts

A *cardinal* conflict $\langle \alpha^i, \alpha^j \rangle$ is a conflict which upon branching results in an increase in the plan costs $\sigma^\pi(\pi^i)$ and $\sigma^\pi(\pi^j)$ for each agent i and j [48]. A *semi-cardinal* conflict results in an increase for only one agent, and a *non-cardinal* conflict does not increase either. Branching on non-cardinal conflicts increases the number of leaf nodes without raising the CT objective lower bound. Cardinal conflicts, on the other hand, contribute to raising the objective lower bound and are therefore preferred. When branching on a CT node, our AOC-CBS implementation supports four modes:

- *Arbitrary*: selects an arbitrary conflict.
- *Semi-/Cardinal*: selects an arbitrary semi-cardinal or cardinal conflict. If none exist, an arbitrary conflict is selected.
- *Best-of-n*: arbitrarily selects a set of at most n conflicts and selects the best cardinal conflict, falling back to the best semi-cardinal conflict, falling back to an arbitrary conflict. We define the best conflict as one maximising over all conflicts the minimum plan cost increase over the two resulting child nodes.

Note that $n = \infty$ returns the best conflict over all conflicts at a node, and $n = 1$ reduces to arbitrary selection. A lower n makes the conflict selection faster, while a higher n increases the chance of selecting a better conflict.

4.2 Low-Level Path Planner

Our low-level path planner *Multi-Goal CSIPP* (MG-CSIPP) extends CSIPP [20], itself based on SIPP [35], however incorporating the treatment of multiple sequential tasks in the style of [47], while also allowing for the handling of idle agents.

As discussed in Section 3, for the correctness guarantees to hold, the low-level path planner must find for a given agent the lowest-cost plan that is consistent with a given set of constraints. This means that the path planner is coupled with the definition of a plan's cost. MG-CSIPP assumes that an action's cost is equal to its duration, and that a plan's cost is equal to the sum of its actions' costs. Modifications may be required if other plan cost definitions are used.

In MG-CSIPP, when planning for agent $a \in \mathcal{A}$, a state $\langle v, I_v^{safe}, \lambda \rangle \in \mathcal{V}^a \times (\mathbb{R} \times \mathbb{R}) \times (\Lambda^a \cup \{none\})$ contains a vertex v and a safe interval $I_v^{safe} = [t_v^{start}, t_v^{end})$, as in SIPP and CSIPP, but also the next task λ_i the agent must complete (*none* if no tasks remain). The A* search seeks a path from the agent's current state to a state $\langle v, [\cdot, \infty), none \rangle$ with $waitable(v)$, i.e., a state in which all tasks are completed and the agent can safely remain at v indefinitely.

From a state $\langle v, I_v^{safe}, \lambda_i \rangle$ with arrival time t^{arrive}, two kinds of transitions are possible:

1. *Move transitions* are constructed exactly as in CSIPP: for every edge e with $\overset{v}{\cdot\!\rightarrow}(e) = v$, the set of safe times to traverse e and arrive at $v' = \overset{v}{\rightarrow\!\cdot}(e)$ is

$$P_e = [t^{arrive}, t_v^{end}) \cap S_e \cap S_{v'}^{+e_D} \tag{4}$$

   where S_e is the set of safe times to traverse e and $S_{v'}^{+e_D}$ is the set of safe times to occupy v' but shifted by e_D to account for the agent arriving there only after traversing e. A transition to $\langle v', I_{v'}^{safe}, \lambda_i \rangle$ exists at the earliest time of each maximally connected interval in P_e. If λ_i is a vertex task at v, P_e is additionally restricted to $[t^{arrive}, t^{arrive} + \lambda_{i,D})$: at $t^{arrive} + \lambda_{i,D}$ the task is completed regardless, so later departures are already captured at state $\langle v, I_v^{safe}, \lambda_{i+1} \rangle$. Otherwise, if λ_i is an edge task on e, then all states reached by a transition along e lead to a state containing the next task λ_{i+1} (or *none* if λ_i was the last).

2. A *vertex task completion transition* exists if λ_i is a vertex task at v and $t^{arrive} + \lambda_{i,D} < t_v^{end}$. This is a single transition to $\langle v, I_v^{safe}, \lambda_{i+1} \rangle$ with arrival time $t^{arrive} + \lambda_{i,D}$. Move and task-completion transitions out of this new state are generated in the same way as any other state, so a task may be completed and the agent may immediately depart within the same safe interval.

Let $MinimumTime(v, v')$ be the minimum possible time to move from $v \in \mathcal{V}^a$ to $v' \in \mathcal{V}^a$ ignoring constraints and unsafe intervals. We define the heuristic at state $\langle v, I_v^{safe}, \lambda_i \rangle$ with arrival time t^{arrive} as

$$\begin{aligned} f\left(\langle v, I^{safe}, \lambda_i \rangle\right) = t^{arrive} &+ MinimumTime\left(v, \overset{v}{\cdot\!\rightarrow}(\lambda_i)\right) + \lambda_{i,D} \\ &+ \sum_{j=i+1}^{|\Lambda^a|} MinimumTime\left(\overset{v}{\rightarrow\!\cdot}(\lambda_{j-1}), \overset{v}{\cdot\!\rightarrow}(\lambda_j)\right) + \lambda_{j,D} \end{aligned}$$

which captures the shortest possible time from state $\langle v, I_v^{safe}, \lambda_i \rangle$ for the agent to complete all remaining tasks. In common A* terminology, $f = g + h$ where $g = t^{arrive}$ is the cost so far and h is the sum of the remaining terms with captures a lower bound on the remaining cost to a goal state.

MG-CSIPP retains the two properties that make SIPP optimal: (i) the safe-interval decomposition is exact, so the state space contains every arrival time at which a distinct set of future transitions becomes available, and no reachable plan is excluded; (ii) f is admissible since the estimated remaining cost to a goal state cannot be better than taking the shortest path through all tasks. Since MG-CSIPP is an A* search over this state space with an admissible heuristic, it is guaranteed to return a plan of minimum duration (i.e., the earliest possible completion time of all tasks) whenever one exists.

A state with $I_v^{safe} = [\cdot, \infty)$ and $\lambda = none$ requires no further move action and is a valid goal; an infinite-wait action is implicitly available there.

4.3 Repair Method: Tier-Prioritized Safe Interval Path Planning

Tier-Prioritized SIPP (TP-SIPP) resembles PSIPP [28] in that both apply prioritized planning. Although PSIPP does not natively support multiple vertex and edge tasks, each with durations, this is easy to address by modifying the low-level planner, as shown with MG-CSIPP in the previous section. PSIPP is nonetheless unsuitable here since it lacks a mechanism for moving an idle agent out of the way and instead plans each agent to a single, specific vertex. For instance, consider a problem where two agents are each assigned a vertex task of non-infinite duration, and the two tasks conflict with each other. Without modification, PSIPP first plans one agent to its task and leaves it there, making the other agent's task inaccessible from the first agent's arrival time onward. When PSIPP then plans the second agent, it can only succeed if the second agent can reach its task, complete it, and clear the vertex before the first agent arrives—otherwise PSIPP fails to find a solution, even though one exists: the first agent could simply move out of the way once its own task is complete. Trying a different planning order does not fix this in general, and the number of possible orders grows exponentially in the number of agents.

TP-SIPP is a repair function that takes a valid but non-solution joint plan Π as input and returns either a repaired solution Π' or failure. The pseudo-code is found in Algorithm 1, on which the following description relies. Since Π is valid, no conflict-free plan $\pi \in \Pi$ blocks another agent from completing its tasks. If it did, π would conflict with that agent's own valid plan and therefore not be conflict-free. So every conflict-free plan in Π is copied directly into Π', and only the conflicting agents are left unplanned (lines 2–4); the better Π is to begin with, the fewer repairs are needed. Each unplanned agent has a deadline: the latest time it may still occupy its starting vertex, determined by whether that vertex is waitable and, if so, whether some plan already in Π' eventually conflicts with it (lines 6–7). Unplanned agents are grouped into three tiers by this deadline: $t = 0$ (severely constrained), $t \in (0, \infty)$ (constrained), and $t = \infty$ (unconstrained) (lines 8–10). Starting with the severely constrained tier, the agents are arbitrarily ordered and an agent a is selected (line 27). A valid plan π^a is sought using MG-CSIPP that (i) avoids conflicts with plans already in Π', (ii) avoids conflicting with any still-unplanned agent at that agent's starting vertex before its deadline, and (iii) does not end at a vertex that conflicts with any still-unplanned agent's tasks (line 30). Together, these conditions guarantee that π^a leaves every remaining unplanned agent free to use its full allotted time at its start, and does not permanently block any of their tasks. If such a plan is found, it is added to Π' and a is no longer unplanned. However, unlike PSIPP, if such a plan cannot be found then instead of reporting failure, TP-SIPP requeues the agent and the tier continues, so failure only occurs once every agent in the tier has been cycled through without a single plan being found. Once the entire tier is planned, the same procedure repeats for the constrained tier (agents taken in increasing order of deadline) and then the unconstrained tier (agents taken in arbitrary order). Since plans are only ever added to Π' if they do not conflict with any existing plan in Π', we know that Π' is a solution if it contains a plan for every agent. At that point, Π' is returned (line 18).

When using TP-SIPP as a repair function in the high-level search's portfolio, it is applied to many different CT nodes and their possibly unique joint plans. Additionally, the severely

Algorithm 1 Tier-Prioritized SIPP

function TP-SIPP(Π)
  $P \leftarrow \{a \in \mathcal{A} \mid \pi^a \in \Pi \text{ is conflict-free}\}$ ▷ Planned agents, mutable below
  $\Pi' \leftarrow \{\pi^a \in \Pi \mid a \in P\}$ ▷ Decided plans, mutable below
  $U \leftarrow \mathcal{A} \setminus P$ ▷ Unplanned agents

  for $a \in U$ **do** ▷ Deadlines reflect only planned agents in P
    $\delta(a) \leftarrow \textsc{Deadline}(a, \Pi')$
  $U^0 \leftarrow \textsc{RandomOrder}(\langle a \in U \mid \delta(a) = 0 \rangle)$
  $U^{(0,\infty)} \leftarrow \textsc{IncreasingOrder}(\langle a \in U \mid 0 < \delta(a) < \infty \rangle, \delta)$
  $U^{\infty} \leftarrow \textsc{RandomOrder}(\langle a \in U \mid \delta(a) = \infty \rangle)$

  if $\textsc{PlanTier}(U^0) =$ Failed **then** ▷ Plan severely constrained agents
    return Failed
  if $\textsc{PlanTier}(U^{(0,\infty)}) =$ Failed **then** ▷ Plan constrained agents
    return Failed
  if $\textsc{PlanTier}(U^{\infty}) =$ Failed **then** ▷ Plan unconstrained agents
    return Failed
  return Π'

function $\textsc{Deadline}(a, \Pi')$
  if $\neg \mathit{waitable}(v^a_{\mathit{start}})$ **then return** 0
  return the safe interval end at v^a_{start} starting at $t = 0$, w.r.t. Π'

function $\textsc{PlanTier}(U')$
  marker= *none*
  while $U' \neq \varnothing$ **do**
    $a = U'.\textsc{PopFront}$
    if marker= a **then return** Failed ▷ U' cycled without change

    $\pi^a = \textsc{PlanAgent}(a, \Pi', P)$
    if $\pi^a \neq$ Failed **then**
      $\Pi'[a] \leftarrow \pi^a$
      $P \leftarrow P \cup \{a\}$
      marker= *none*
    else
      $U'.\textsc{AppendToBack}(a)$
      if marker= *none* **then** marker= a
  return Success

constrained and unconstrained agents are ordered arbitrarily and can therefore be ordered, e.g., randomly. This introduces substantial variation in how repairs unfold across calls and therefore provides many chances throughout solving a problem to find a solution.

5 Theoretical Guarantees

In this work, we say that an algorithm is

- **sound** if it only returns solutions,
- **exact** if it only returns optimal solutions, and
- **solution complete** if it returns a solution in finite time if one exists.

Observe that exactness implies soundness, and that a solution complete algorithm is not required to report the non-existence of a solution while a *complete* algorithm is. We show in the section that AOC-CBS is exact and solution complete.

Just as in OC-CBS, this work makes explicit that real values cannot be represented exactly under finite-precision arithmetic. Since our implementation (like all implementations) is subject to a precision ϵ_{mach}, our guarantees are given in that context. Thus, it is possible that a solver in this setting is unable to find solutions that would require a finer precision than what the underlying machine can provide, or that a returned solution actually contains conflicts on the time scale of a few orders of magnitude above the machine's finest precision. Nonetheless, we argue that this accuracy is well beyond what most real-world applications require, where, for instance, sensor and actuator noise would dominate. In Section A.1, we discuss how conflict information is practically computed and stored for use during runtime. Since this conflict information is the sole connection between AOC-CBS and the specific application that it plans over, all correctness guarantees are with respect to the accuracy of this conflict information.

The remainder of this section provides various useful properties of AOC-CBS, including the conditions required for guaranteed termination in a finite number of iterations with an optimal solution. The invariant (Theorem 5.1) shows that at every iteration, the search either still has access to an optimal solution through some unexpanded leaf node, or has already found one; termination (Corollary 5.2) shows that once the node queue empties, this invariant forces the incumbent to be optimal; and bound correctness (Corollary 5.3) confirms that the lower and upper bounds maintained throughout the search are valid, giving a meaningful optimality gap at any point of interruption. The remaining two theorems then give the conditions under which an optimal solution guaranteed in a finite number of iterations: either an incumbent solution has been found (Theorem 5.4), or at least one policy in the portfolio always expands the node minimizing the lower bound (Theorem 5.5), inheriting termination guarantees from OC-CBS.

All proofs assume that AOC-CBS is applied to a solvable problem instance, with Π^* being an optimal solution and $\sigma^* = \sigma(\Pi^*)$. Recall the following notation from Sections 2.2 and 4.1: for a CT node N, $\mathbb{S}(N)$ is the set of all solutions permitted under N_C; $LB(N) \leq \min_{\Pi \in \mathbb{S}(N)} \sigma(\Pi)$

is a cost lower bound over all solutions in $\mathbb{S}(N)$; Π_k^{UB} is the best-found incumbent solution at iteration k, with $UB_k = \sigma(\Pi_k^{UB})$; $LB_k = \min_{N \in \mathcal{Q}_k} LB(N)$ if $\mathcal{Q}_k \neq \varnothing$, otherwise $LB_k = UB_k$; and $\Delta_k^{UB} = UB_k / LB_k - 1$ is an upper bound on Π_k^{UB}'s optimality gap.

Theorem 5.1 (AOC-CBS Invariant)**.** *At the start of every iteration k of AOC-CBS, either $\exists N^* \in \mathcal{Q}_k : \Pi^* \in \mathbb{S}(N^*)$ (Reachability), $UB_k = \sigma^*$ (Optimality), or both are true.*

Proof. The invariant is proven by induction:

- *Base Case*: At $k = 0$ we have $\mathcal{Q}_0 = \{R\}$. Since the empty constraint set $R_C = \varnothing$ permits all solutions, $\Pi^* \in \mathbb{S}(R)$. Reachability holds at $k = 0$.
- *Inductive step*: Assume the invariant holds at the start of iteration k. Recall from Section 4.1 that during iteration k, a set of nodes $\mathcal{Q}^{sel} \subseteq \mathcal{Q}_k$ is selected for expansion, producing a set of child nodes nodes $\mathcal{Q}^{new}$ and a set of newly found solutions $\mathbb{S}^{new}$. At the start of iteration $k+1$, the incumbent solution is

$$\Pi_{k+1}^{UB} = \underset{\Pi \in \mathbb{S}^{new} \cup \{\Pi_k^{UB}\}}{\arg\min} \sigma(\Pi)$$

  and the set of unexpanded nodes is

$$\mathcal{Q}_{k+1} = \left\{ N \in (\mathcal{Q}_k \setminus \mathcal{Q}^{sel}) \cup \mathcal{Q}^{new} \mid LB(N) < UB_{k+1} \right\}$$

  where all nodes N with $LB(N) \geq UB_{k+1}$ are pruned. There are three cases:

  1. If $UB_k = \sigma^*$, then $UB_{k+1} = \sigma^*$ since no solution can have a cost lower than σ^*. Optimality holds at the start of iteration $k+1$.
  2. If instead $UB_k > \sigma^*$ but during the expansion a solution with cost σ^* was found, then $UB_{k+1} = \sigma^*$. Optimality holds at the start of iteration $k+1$.
  3. Otherwise, $UB_k \geq UB_{k+1} > \sigma^*$. Then optimality does not hold, and therefore by the invariant there exists $N^* \in \mathcal{Q}_k$ such that $\Pi^* \in \mathbb{S}(N^*)$. This gives two subcases:

     (i) N^* **is not expanded**: $N^* \notin \mathcal{Q}^{sel}$. Since $\Pi^* \in \mathbb{S}(N^*)$ and $\sigma^* < UB_{k+1}$,

$$LB(N^*) \leq \sigma(\Pi^*) = \sigma^* < UB_{k+1}.$$

     Hence, N^* was not pruned and therefore $N^* \in \mathcal{Q}_{k+1}$. Reachability holds at the start of iteration $k+1$.

     (ii) N^* **is expanded**: $N^* \in \mathcal{Q}^{sel}$. Since no solutions are removed during the expansion of a node, there exists a child N^c of N^* such that

$$\Pi^* \in \mathbb{S}(N^c),$$

     meaning that

$$LB(N^c) \leq \sigma(\Pi^*) = \sigma^* \leq UB_{k+1}.$$

     Thus, N^c is not pruned and therefore $N^c \in \mathcal{Q}_{k+1}$. Reachability holds at the start of iteration $k+1$.

In all cases, at the start of $k+1$, either optimality holds or reachability holds. Thus, the invariant is preserved.

By induction, the invariant holds at the start of every iteration. □

Corollary 5.2 (AOC-CBS Termination)**.** *If $\mathcal{Q}_k = \varnothing$, then $UB_k = \sigma^*$ and Π_k^{UB} is an optimal solution.*

Proof. The invariant from Theorem 5.1 holds for this iteration k. However, since reachability is impossible, optimality must hold. That is, $UB_k = \sigma^*$ and Π_k^{UB} is an optimal solution. □

Corollary 5.3 (AOC-CBS Bound Correctness)**.** *At every iteration k, the optimal solution lower and upper bounds LB_k and $UB_k = \sigma(\Pi_k^{UB})$, and the incumbent solution Π_k^{UB}'s optimality gap upper bound $\Delta_k^{UB} = UB_k/LB_k - 1$, are correct.*

Proof. If no incumbent solution Π_k^{UB} has been found then $UB_k = \infty$. The cost of any solution cannot be greater than ∞. Otherwise, $UB_k = \sigma(\Pi_k^{UB})$, and it must hold that $\sigma(\Pi^*) \leq UB_k$, else Π_k^{UB} provides a counterexample. Thus, at every iteration k, UB_k is a true upper bound on the optimal cost.

If *optimality* in the invariant of Theorem 5.1 holds, such that $UB_k = \sigma^*$, then either $\mathcal{Q}_k = \varnothing$ and $LB_k = UB_k$ which is a true lower bound on σ^*, or $\mathcal{Q}_k \neq \varnothing$ and $LB_k = \min_{N \in \mathcal{Q}_k} LB(N) < UB_k$ since $\forall N \in \mathcal{Q}_k : LB(N) < UB_k$. If instead *optimality* in the invariant of Theorem 5.1 doesn't hold, then *reachability* must hold: $\exists N^* \in \mathcal{Q}_k : \Pi^* \in \mathbb{S}(N^*)$. Then $LB_k = \min_{N \in \mathcal{Q}_k} LB(N) \leq LB(N^*) \leq \sigma^*$. Thus, at every iteration k, LB_k is a true lower bound on the optimal cost.

Finally, since $LB \leq \sigma^*$, it must hold that

$$\frac{\sigma(\Pi_k^{UB})}{LB_k} = \frac{UB_k}{LB_k} \geq \frac{UB_k}{\sigma^*}$$

Thus, at every iteration k, $\Delta_k^{UB} = UB_k/LB_k - 1$ is a true upper bound on the optimality gap of the incumbent solution Π_k^{UB}. □

Theorem 5.4 (Incumbent-driven Termination)**.** *If an incumbent solution Π_k^{UB} has been found, AOC-CBS is guaranteed to terminate in a finite number of iterations with an optimal solution.*

Proof. Termination follows from the CT in AOC-CBS satisfying property **CT-III** as a consequence of using δ-BR: for every infinite path $N_1, N_2, N_3, \ldots$ in the CT (where N_i is the parent of N_{i+1}), and any $c \in \mathbb{R}$,

$$\exists i \in \mathbb{N} : \forall j \geq i : c < LB(N_j).$$

This means that for any descending path in the CT, eventually the lower bound on the cost will be pushed beyond some threshold c. Consider here that $c = UB_k$. Since all nodes N with $LB(N) \geq UB_k$ are pruned and therefore not included in $\mathcal{Q}_k$, every node in $\mathcal{Q}_k$ will

either be pruned or expanded to eventually reach children that are pruned. Consequently, $\mathcal{Q}_K = \varnothing$ for some bounded iteration K. At that point, by the invariant in Theorem 5.1, the incumbent Π_K^{UB} will be an optimal solution. □

Theorem 5.5 (Policy-driven Termination)**.** *If there exists at least one policy $\psi \in \Psi$ that always selects $N \in \mathcal{Q}_k$ minimizing $LB(N)$ for expansion, then AOC-CBS is guaranteed to terminate in a finite number of iterations with an optimal solution.*

Proof. If at least one such policy ψ exists, then at every iteration at least one node $N \in \mathcal{Q}_k$ minimizing σ will be expanded. This is a best-first search (with possibly additional node expansions) using δ-BR, which is consistent with an OC-CBS search and therefore inherits OC-CBS's termination guarantees [27].

Specifically, since properties **CT-I**–**CT-III** are satisfied by the CT in AOC-CBS and a node $N \in \mathcal{Q}_k$ minimizing $LB(N)$ is selected for expansion at every iteration k, $\min_{N \in \mathcal{Q}_k} LB(N)$ will necessarily increase as k grows. Since no solution is removed during the search, and property **CT-III** ensures progress is made, eventually an optimal solution will be found. □

6 Experimental Evaluation

This section presents a preliminary experimental evaluation of AOC-CBS, with further testing ongoing. All experiments are run on a 2025 Mac Studio, 16-core M4 Max CPU (12 performance cores, 4 efficiency cores), 64 GB RAM, macOS Tahoe (26.5.2). AOC-CBS is implemented in Python; the source code, benchmark sets, results, and demonstrations will be made available.

We demonstrate AOC-CBS on three difference MAPF-based models. The first operates under the common assumptions of circular agents and straight-line traversals at constant speed. It is under this model that we perform an ablation study in Section 6.2 using various solver configurations, with some comparison with OC-CBS, followed by additional comparisons in Section 6.3. The second model includes arbitrary continuous and connected edge trajectories, and the third model further introduces polygonal (possibly non-convex) agent shapes. We provide a demonstration of this richer model in Section 6.4. Appendix A describes our implementation of these three models, specifically regarding their interface with AOC-CBS and how to preprocess and store information for lookup during runtime.

Figure 2 illustrates an example of the solver progress over runtime, showing how the optimal cost lower bound (derived from the CT) increases while the upper bound (the incumbent solution's cost) decreases. The incumbent solution at first comes from the repair function (TP-SIPP), however, later in the search the solutions from the child nodes during CT branchings are of higher quality. The solution found at 20.3 seconds (with an optimality gap upper bound of 0.033%) was verified optimal by 22.5 seconds, at which point the upper and lower bound on the objective value coincide.

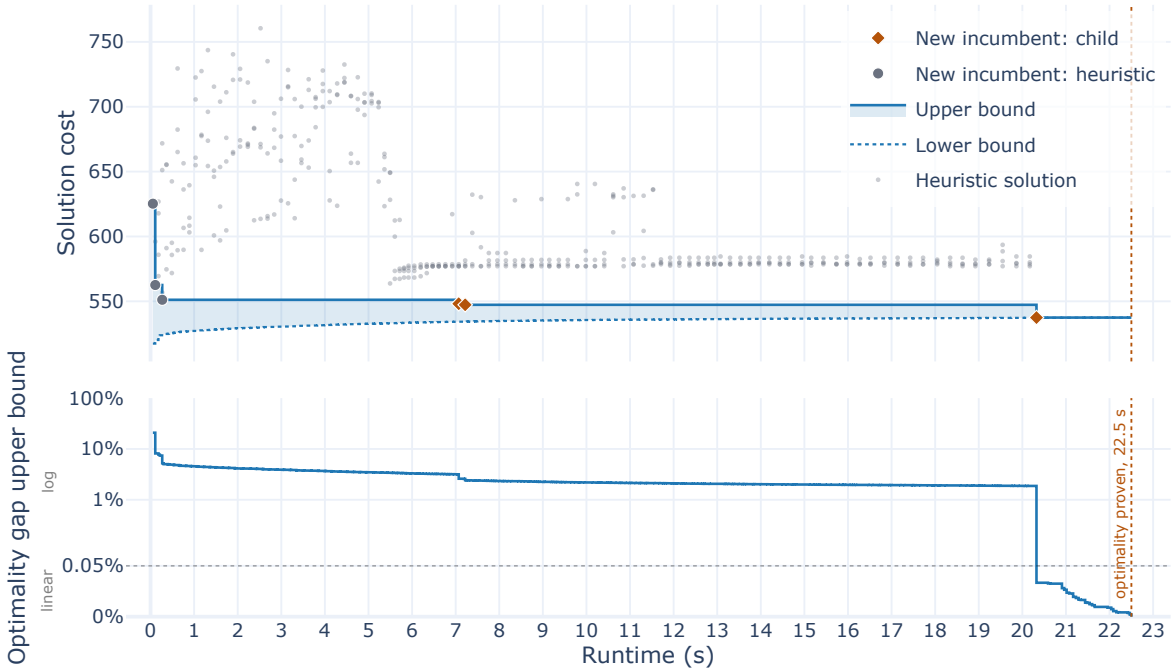
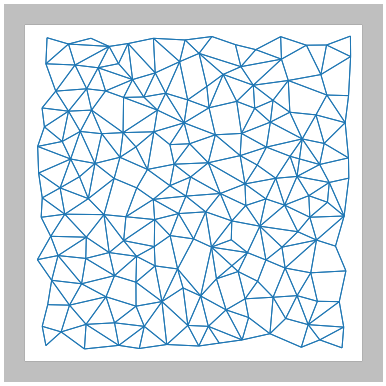


Figure 2: The solver progress over runtime. The top plot shows the lower bound (from the CT tree) and the upper bound (the incumbent solution cost) of the optimal cost, forming the shaded region that corresponds to the incumbent solution's optimality gap upper bound. The bottom plot shows this optimality gap upper bound in percent (note the mixed linear and log scale). The upper plot also shows the solutions found by the repair function (TP-SIPP), as well as the incumbent solution's origin, being either from the repair function or from a child node during a branching. Finally, the solution found at 20.3 seconds was verified optimal at 22.5 seconds.

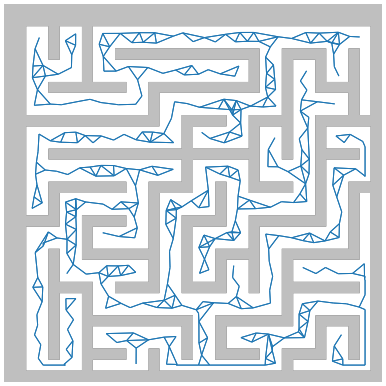
6.1 Benchmark Setup: Maps, Scenarios, and Pre-processing

The ablations and comparisons with OC-CBS are done using the standard MovingAI benchmark *gridmaps* [29], and *roadmaps* sampled from these gridmaps. Gridmaps are maps with unit-length edges, typically used in discrete-time MAPF, while roadmaps may include non-unit-length edges. We use the *Den520d Sparse* roadmap from [20], and additionally our roadmaps sampled with our method detailed in Appendix B. A full roadmap benchmarking set using this method, along with scenario files, which will be made available on our repository. Figure 3 shows three such sampled roadmaps.

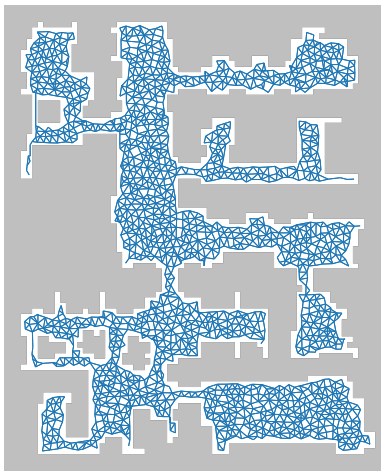
The benchmarking scheme from [1] is used here: for each map, a *scenario* defines a list of start and goal vertex pairs. A problem with n agents is created from the first n start and goal vertex pairs, starting with $n = 2$ and incrementing until no solution (including sub-optimal) can be found for a problem within 30 seconds. We used the *random* scenarios from [29] for the gridmaps, the scenarios for *Den520d Sparse* from [20], and our own generated scenarios (see Appendix B) for our sampled roadmaps. To match prior work, all experiments on the



(a) *Empty-16-16* with $|\mathcal{V}| = 178$ and $|\mathcal{E}| = 916$.



(b) *Maze-32-32-2* with $|\mathcal{V}| = 362$ and $|\mathcal{E}| = 1\,088$.



(c) *Den312d* with $|\mathcal{V}| = 1\,607$ and $|\mathcal{E}| = 7\,894$.

Figure 3: Roadmaps sampled from MovingAI gridmaps, using our method detailed in Appendix B with a density $\rho = 1.0$.

gridmaps and *Den520d Sparse* use circular agent with radius $1/(2\sqrt{2})$. The agents on our roadmaps have radius 0.5.

As described in Appendix A, pre-processing of graph and conflict information is done using knowledge that is assumed to be known beforehand. This includes information about the graphs and types of agents within the system. The pre-processed graph information contains the all-pairs shortest path lengths, and the conflict information distils all domain specifics, such as geometry and conflict definitions, and stores that information for use during solving. Information about a specific instance, such as the agent's starting positions and task sequences are not assumed to be known beforehand and therefore are not used for any preprocessing. Thus, the reported runtime in all experiments begins from the time instance-specific information is given to the solver.

Figure 4 visualise the graph and conflict information preprocessing time for each map, using parallel processing on 12 CPU cores. Although the pre-processing times remain well within a practical range, these are done specifically for the circular agent, straight-line, constant-velocity model. The time required to perform this pre-processing for, e.g., curved trajectories and non-circular agents, is the focus of ongoing work. Nonetheless, the measured pre-processing times appear to follow a predictable trend under this model, providing an estimate for other maps.

6.2 Ablations

This section presents the results of an ablation study over AOC-CBS solver configurations together with comparisons with OC-CBS. To match the problem formulation that OC-CBS is implemented for, all instances in this section use maps with straight-line trajectories, circular agents with radius $1/2\sqrt{2}$ (such that agents do not collide under the classical MAPF

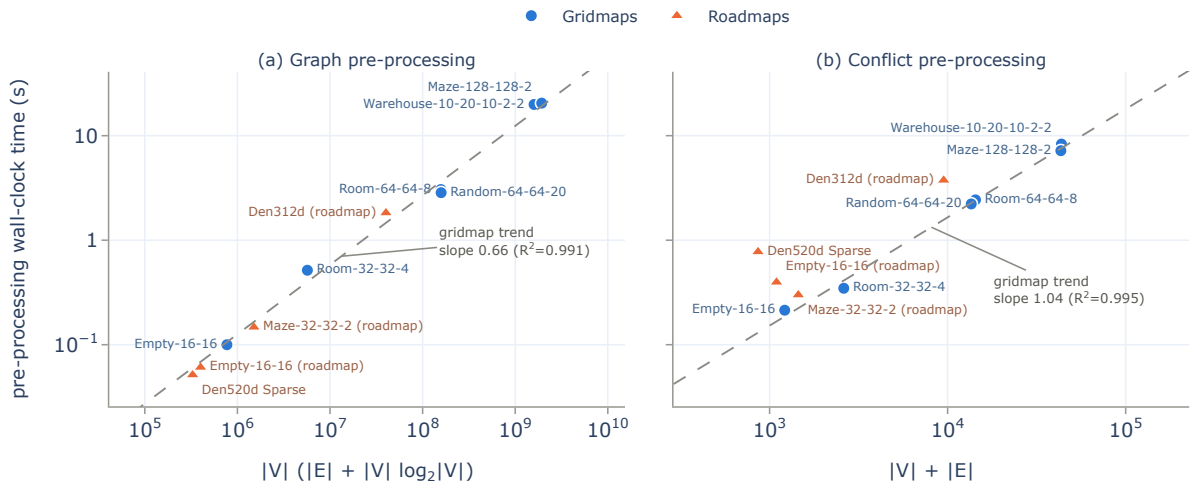


Figure 4: The graph and conflict information pre-processing runtime (see Appendix A), for each map used in the ablation and benchmark sections. The graph pre-processing computes the all-pairs shortest path lengths, which scales with $|\mathcal{V}|(|\mathcal{E}| + |\mathcal{V}| \log_2 |\mathcal{V}|)$. Conflict information pre-processing distils all domain-specific information, such as geometry and conflict definitions, and stores that information for use during solving.

formulation on a grid map with unit length edges), and only one vertex task per agent with an infinite service time.

These ablations are limited to maximum 200 agents. We use all 25 scenarios for each of the grid maps *Room* (Room-64-64-8) and *Random* (Random-64-64-20) from [29] using connectedness 2 (grid layout) and the road map *Den520d Sparse* from [20]. We compare AOC-CBS implemented in Python, making extensive use of data-structure optimizations, with the original C++ OC-CBS implementation[1]. Note that OC-CBS returns only optimal solutions; in contrast, when AOC-CBS returns a suboptimal solution, we report on its optimality gap *upper bound* Δ^{UB}, the true optimality gap may be lower.

6.2.1 Ablations on AOC-CBS Configurations

Table 2 details the six AOC-CBS configurations used in these ablations. They vary along three axes: single- versus multi-policy portfolios (each policy run on a separate CPU core), presence or absence of a repair function (TP-SIPP), and the node-selection rule. Selection is either minimum cost throughout, or a mix of three policies selecting by minimum cost, four by minimum conflict count, and four by minimum summed conflict time—where conflict time is the duration an agent's action must shift to avoid a move-move conflict, or the overlap duration for a move-wait conflict. The multi-policy configurations use 11 worker processes alongside the coordinating process, matching the 12 performance cores of our hardware. OC-CBS is single-threaded, so only the single-policy configurations use comparable hardware.

[1] `https://github.com/Adcombrink/Optimal-Continuous-CBS`

Table 2: Overview of the experimental AOC-CBS configurations.

	Nr of Policies	Repair Function	Node Selection	Portfolio Type
C1	1	—	Min. Cost	Static
C2	1	TP-SIPP	Min. Cost	Static
C3	11	—	Min. Cost	Static
C4	11	TP-SIPP	Min. Cost	Static
C5	11	TP-SIPP	Mixed	Static
C6	11	w/wo TP-SIPP	Mixed/Min. Cost	Gap Cadence

Configurations **C1**–**C5** use a static portfolio, while **C6** uses a *gap-cadence* portfolio that shifts between two sub-portfolios as the optimality gap upper bound Δ^{UB} tightens: a *solution-finding* portfolio (**C5**) that prioritizes finding a solution quickly while Δ^{UB} is high, and an *LB-raising* portfolio (**C3**) that prioritizes raising the CT solution lower bound as Δ^{UB} falls. The solution-finding portfolio runs every $\lceil 0.05/\Delta^{UB} \rceil$ iterations (every iteration when $\Delta^{UB} = \infty$), so it is used continuously while $\Delta^{UB} \geq 5\%$ and is progressively phased out in favour of the LB-raising portfolio as Δ^{UB} decreases. At $\Delta^{UB} = 1\%$ it runs only every fifth iteration.

Figure 5 visualises the success rate of solving to Δ^{UB} values on each of the maps, using AOC-CBS with configurations **C1**–**C6** and OC-CBS. Success rate is measured by the share of the 25 scenarios that could be solved to the given Δ^{UB} at each agent count. Configuration **C1** is most alike OC-CBS (both single-core, no repair, Min. Cost node selection), however, OC-CBS is implemented in C++ which is often faster than Python [49]. Despite this, **C1** outperforms OC-CBS on *Room*, is on par on *Random*, and lags behind on *Den520d Sparse*. This indicates that the various implementation optimizations in AOC-CBS are advantageous; that these same optimizations in OC-CBS could yield meaningful improvements in runtime performance; and that a C++ implementation of AOC-CBS could further increase the scale of problems that can be solved. Extending **C1** to multi-core with **C3** shows, in this figure, only slight improvements. We revisit this below. Interestingly, by comparing the performance of **C1** and **C3** for optimal solutions with their performance for any solution, it can be seen most clearly on *Den520d Sparse* that despite not using a repair function, they are still able to find suboptimal solutions by only checking the child nodes during expansions.

The repair function is what separates the two groups. At loose Δ^{UB} the deepest sweep on *Room* reaches 153 agents for the repair-using configurations against 25 for **C1**, on *Den520d Sparse* 58 against 22, and on *Random* the 200-agent limit against 56; **C2** remains slightly behind **C4**, the difference being 1 vs 11 policies. That gain is not free, however. At $\Delta^{UB} = 0$ the repair-using configurations are consistently *worse* than their non-repairing counterparts—**C2** below **C1** and **C4** below **C3** on all three maps—because time spent in TP-SIPP is time not spent expanding CT nodes to raise the lower bound. Configuration **C6** is designed to avoid exactly this trade-off, and the results show that it does: it matches **C5** at loose Δ^{UB} while recovering **C3**'s performance at $\Delta^{UB} = 0$, which no static configuration achieves on both ends.

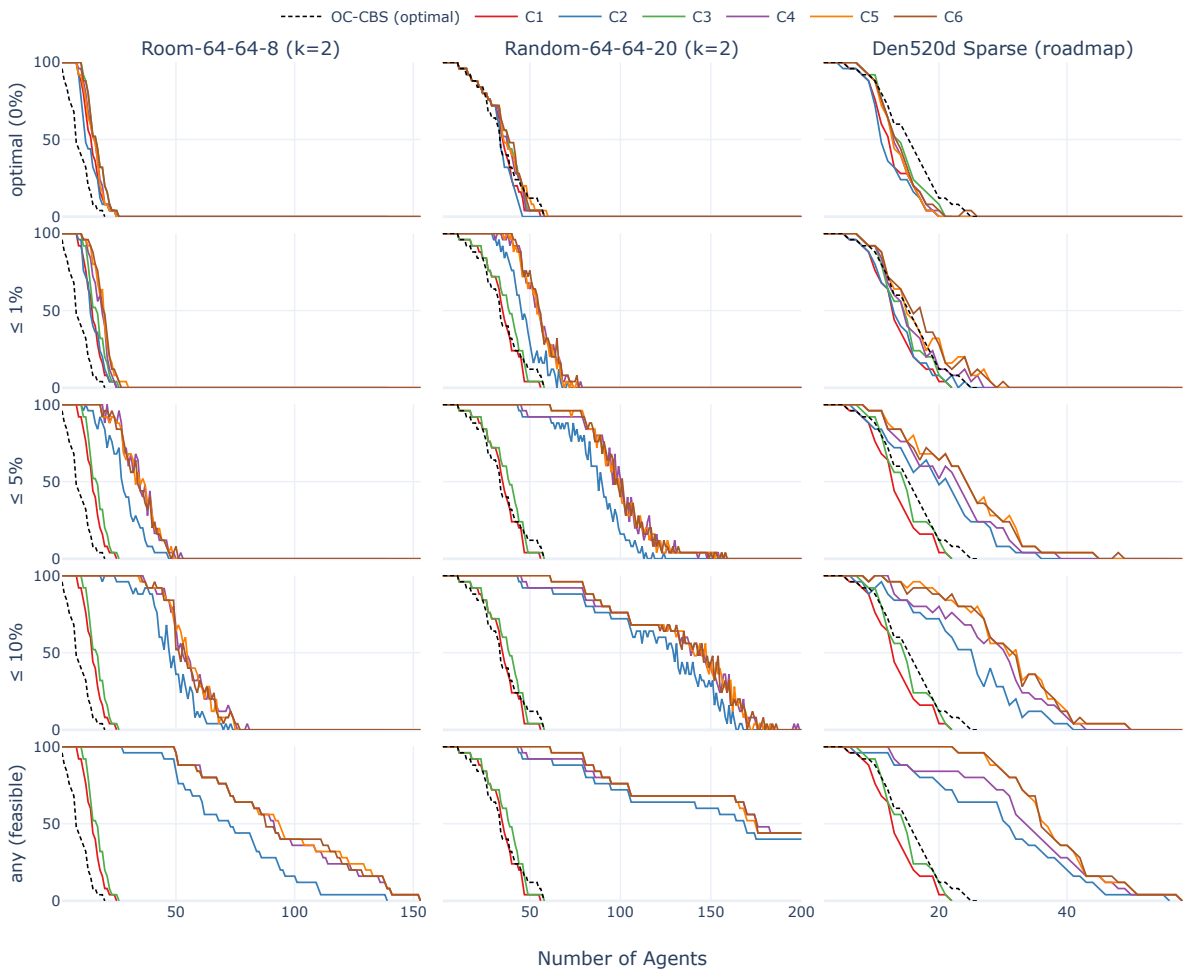


Figure 5: Success rate of AOC-CBS configurations **C1**–**C6** and OC-CBS, as the share of the 25 scenarios per map solved at each agent count. Columns are maps; rows are optimality gap upper bounds Δ^{UB}, from proven optimal (top) to any feasible solution (bottom). OC-CBS returns only optimal solutions, so its curve is identical in every row.

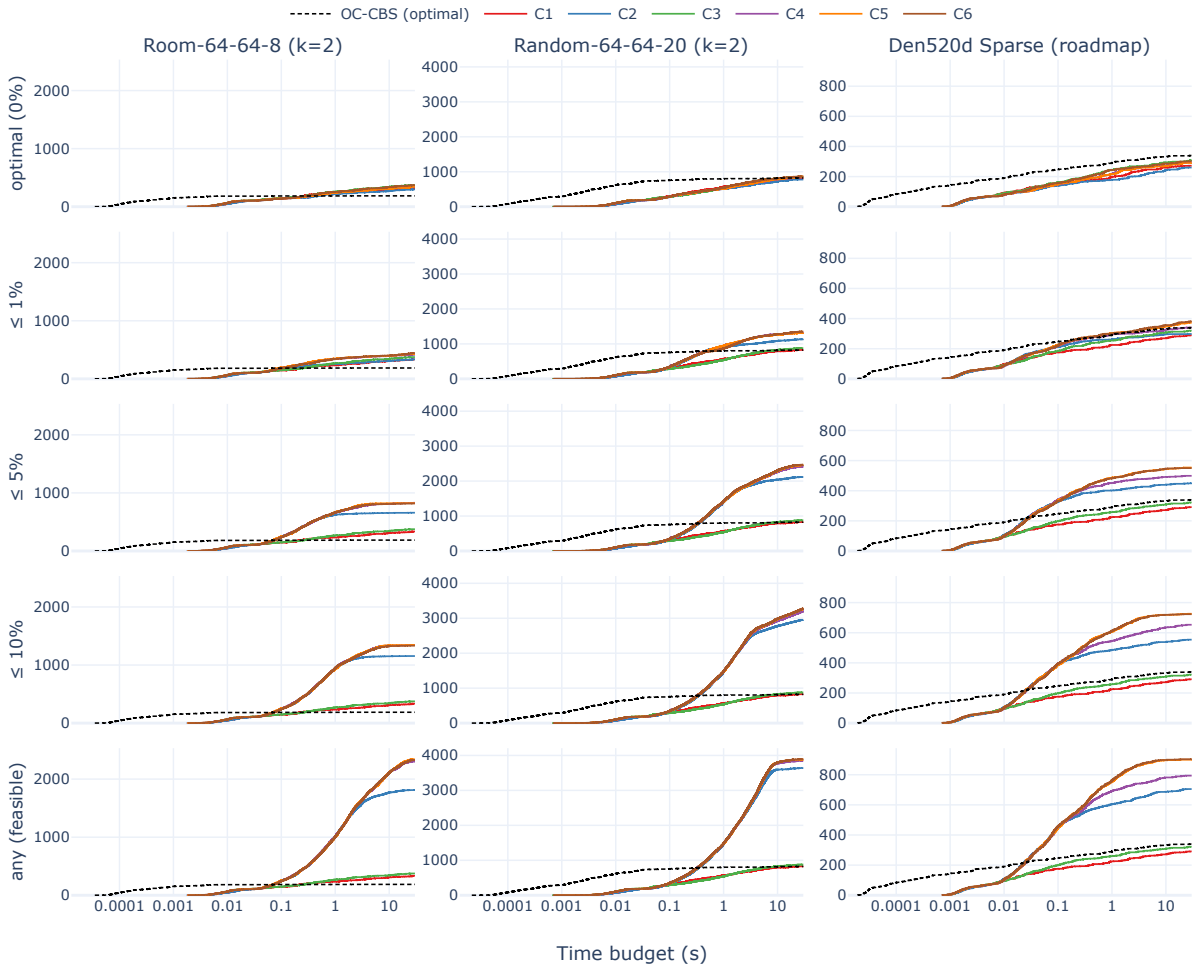


Figure 6: Number of problems solved to a given optimality gap upper bound Δ^{UB} within a given time budget, for AOC-CBS configurations **C1**–**C6** and OC-CBS. Columns are maps; rows are Δ^{UB}, from proven optimal (top) to any feasible solution (bottom). The vertical axis is an absolute instance count, shared down each column and independent across columns; the horizontal axis is logarithmic.

Figure 6 visualises instead the number of problems that each configuration can solve within a given runtime, for a given optimality gap and map. These results highlight how fast the solvers are at solving problems. The per-map ordering between **C1**, **C3** and OC-CBS is the same as in Figure 5, but the runtime axis shows how it arises: OC-CBS solves the easy instances one to two orders of magnitude faster, consistent with a C++ constant-factor advantage, while **C1** and **C3** close most of that gap over the remaining budget. Among the repair-using configurations, **C5** and **C6** are equivalent, **C4** is equivalent on the grid maps (*Room* and *Random*) but lags slightly on the road map (*Den520d Sparse*), and **C2** is just behind. Taken over the full 30 second budget, **C6** finds a feasible solution to 6.9× as many problems as **C1** on *Room* (2329 versus 336), 4.7× as many on *Random* (3896 versus 827), and 3.1× as many on *Den520d Sparse* (904 versus 291). Specifically on *Random*, 4.7× is a lower bound since 11 of the 25 scenarios reach the 200-agent limit rather than a failure. The multi-core effect is far smaller at this aggregate level: **C3** ends 6–15% ahead of **C1** in solved instances on each map. The next section shows that this modest aggregate figure hides a large effect concentrated in the hardest instances.

A natural question that arises from looking at the results in Figures 5 and 6 is: *what is the advantage of using multiple policies (CPU processors), and where can that advantage be found?* The performance difference between **C2** and **C4** could possibly be explained by **C4** running TP-SIPP more often to sample more random repair orders, thereby finding better solutions through chance. **C1** and **C3**, on the other hand, do not use TP-SIPP. Therefore, the additional policies in **C3** contribute primarily toward the search for, and verification of, an optimal solution, yet Figures 5 and 6 show only a modest aggregate difference between the two. The following analysis shows that this aggregate view conceals a large effect confined to the hardest instances.

The results of our investigation into the above question are shown in Figure 7 and Table 3. Figure 7 visualises, for each map, the speedup of using **C3** over **C1** for each instance, plotted over the time it took **C1** to solve. That is, we use **C1**'s solve time $t_{\mathbf{C1}}$ as a proxy for a problem's difficulty, and then show how much faster **C3** could solve the same problem, $t_{\mathbf{C1}}/t_{\mathbf{C3}}$. The 67 instances that **C3** was able to solve to optimality but not **C1** (there were no cases of the opposite) are shown as lower bounds on the speedup, using the time limit $t_{\mathbf{C1}} = 30$ seconds. The geometric mean curve is also shown, computed by dividing the data into 12 log-spaced bins and computing the geometric mean over the per-instance speedup within the bin, with bins containing less than 8 instances dropped. All three maps share the same shape—at or below break-even on the easiest instances, a dip below it, then a monotone rise—but differ in where the crossover falls and how large the final gain is. On *Den520d Sparse* the trend crosses break-even at roughly 0.15 seconds and reaches 7.0×; on *Room* at roughly 0.5 seconds, reaching 4.7×; and on *Random* only at roughly 2 seconds, reaching 2.3×. The multi-policy portfolio thus pays off earliest and most on the road map and latest and least on the random grid. Note also that the trend is conservative at its right end: the censored instances, where **C3** did best, are by construction excluded from the bins, as are the 123 instances that only **C3**'s sweep reached and which therefore have no **C1** counterpart to pair with.

Table 3 aggregates the data from all maps, showing for each bucket the geometric mean

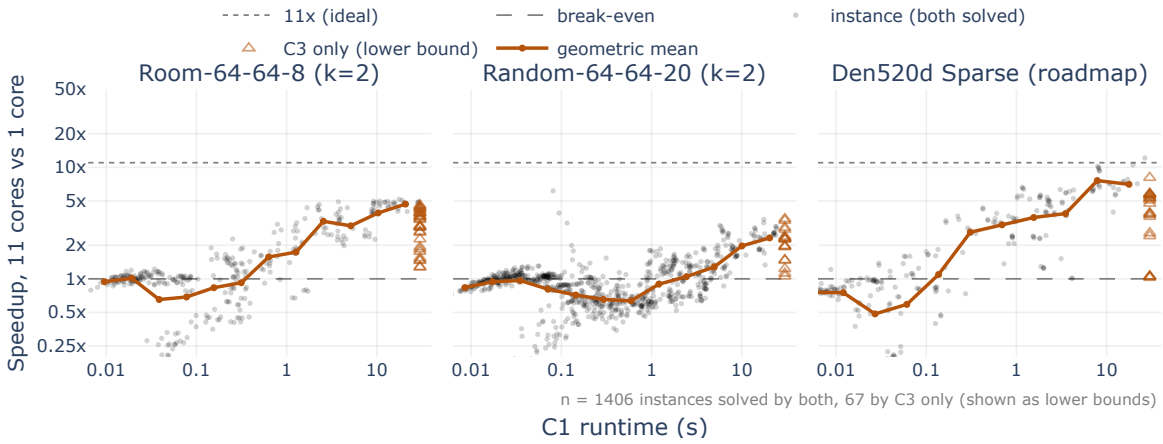


Figure 7: Per-instance speedup of **C3** (11 policies) over **C1** (1 policy), $t_{\mathbf{C1}}/t_{\mathbf{C3}}$, plotted against **C1**'s solve time as a proxy for instance difficulty. Both axes are logarithmic, and the vertical axis is symmetric about break-even so that equal speedups and slowdowns are equally far from it. Each dot is one instance solved optimally by both configurations; open triangles are the 67 instances solved optimally by **C3** alone, drawn at the lower bound $30\,\mathrm{s}/t_{\mathbf{C3}}$ with the true value somewhere above. The amber curve is the geometric mean over 12 log-spaced difficulty bins, excluding bins with fewer than 8 instances and excluding the censored instances—so it understates the advantage at its right end.

bucket	n	$t_{\mathbf{C1}}$	$t_{\mathbf{C3}}$	speedup	throughput	overhead
0–0.1	572	0.02	0.03	0.78×	0.93×	1.19×
0.1–0.5	311	0.22	0.28	0.78×	2.02×	2.59×
0.5–2	231	1.06	0.91	1.16×	3.68×	3.16×
2–5	145	3.18	1.87	1.70×	4.01×	2.36×
5–15	101	8.30	2.88	2.88×	4.43×	1.54×
15–30	46	19.65	5.76	3.41×	4.77×	1.40×
>30 (timeout)	67	>30	10.88	>2.76×	–	–

Table 3: Paired comparison of **C1** and **C3** over all three maps. Instances are grouped into buckets by **C1**'s solve time, with n the number of instances in each bucket. $t_{\mathbf{C1}}$ and $t_{\mathbf{C3}}$ are geometric mean solve times, so $t_{\mathbf{C1}}/t_{\mathbf{C3}}$ equals the speedup column. The remaining columns are geometric means over the instance-wise ratios between **C3** and **C1**: solve time $t_{\mathbf{C1}}/t_{\mathbf{C3}}$ (speedup), CT nodes expanded per unit time (throughput), and total CT nodes expanded (overhead), so that speedup = throughput / overhead. The final row holds instances **C1** did not solve optimally within the limit; their speedups are lower bounds and no node ratios are available. There were no instances solved optimally by **C1** alone.

solve time for **C1** and **C3**, the speedup, the increase in node expansion rate (throughput), and the increase in the total number of expanded nodes (overhead). Since all three are geometric means over the same instances, the speedup is exactly the throughput divided by the overhead, which separates the two competing effects of running 11 policies in parallel.

The throughput shows the raw benefit of using multiple processors: more nodes are expanded per unit time, peaking at 4.77× that of **C1** in the 15–30 second bucket. This is well short of the 11× ideal, giving at most 43% parallel efficiency on our hardware, which we attribute to the cost of managing a pool of policies on separate processes with synchronization at every iteration. On the easiest instances that cost dominates outright: at a geometric mean solve time of 0.02 seconds the pool expands nodes *slower* than a single process (0.93×), as the whole solve is shorter than the time needed to make the pool productive.

The overhead shows that the parallel search expands more CT nodes than the single-core search to solve the same problem, so part of the throughput is spent on nodes that are not needed to find and verify an optimal solution. This is not surprising: expanding one node at a time means every choice is made with the latest information, whereas expanding several nodes in parallel means that all but the first are chosen without it. The overhead is largest on mid-difficulty problems (3.16× in the 0.5–2 second bucket) and falls to 1.40× on the hardest, where the CT is large enough that the parallel policies explore genuinely distinct regions rather than duplicating a short search.

The speedup is what remains after these two effects cancel. On the two easiest buckets the overhead exceeds the throughput and **C1** is the faster solver (0.78×); from the 0.5–2 second bucket onward the ordering reverses and the advantage grows monotonically, reaching 3.41× in the 15–30 second bucket. Both factors move favourably with difficulty—throughput rises while overhead falls—which is why the effect compounds rather than saturating.

The most consequential row is the last. On 67 instances **C3** found and verified an optimal solution within the time limit where **C1** did not, with no instance where the reverse held, raising coverage of the paired instances from 92.0% to 96.3%. The principal benefit of the additional policies is therefore not that the same problems are solved faster, but that problems out of reach of a single core come into reach.

6.2.2 Ablation on Conflict Selection

Every configuration in Section 6.2.1 uses the *Best-of-*∞ conflict selection strategy. This section isolates that choice. All five configurations below are **C5** with only the conflict-selection rule varied, so any difference between them is attributable to that rule alone. Recall from Section 4.1.1: *Arbitrary* branches on the first conflict found and performs no probing; *Semi-/Cardinal* scans the conflict list and returns the first conflict for which at least one child's cost increases; *Best-of-n* probes up to n candidate conflicts and returns the one maximising the smaller of the two children's cost increases. We test **C5** using *Arbitrary*, *Semi-/Cardinal*, and *Best-of-n* with $n \in \{8, 16, \infty\}$. The ablation is run on *Room* (2535 instances over 25 scenarios) and *Den520d Sparse* (968 instances over 25 scenarios).

Figure 8 visualises the number of problems solved using each of the conflict selection strate-

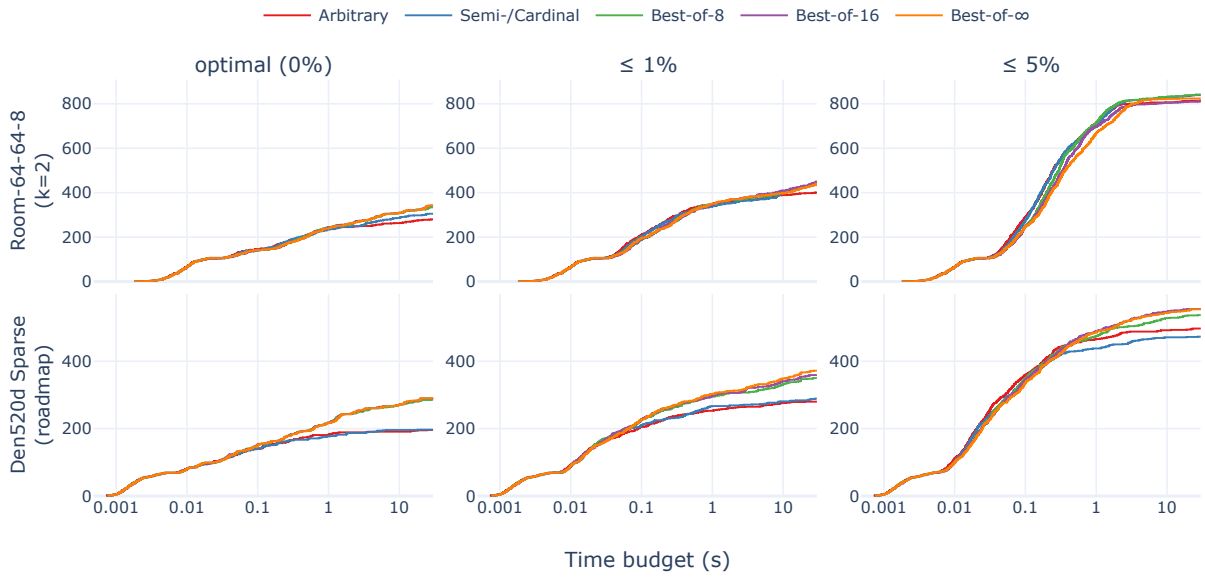


Figure 8: Conflict-selection ablation: instances proved to within each optimality gap as a function of the time budget, on *Room* (top row, 2535 instances) and *Den520d Sparse* (bottom row, 968 instances). All five configurations are **C5** with only the conflict-selection strategy varied. At the optimal threshold the curves fall into two groups on both maps, with the three *Best-of-n* rules mutually indistinguishable and *Arbitrary* below them; the separation narrows as the gap threshold loosens, and on *Room* it has closed entirely by $\leq 5\%$.

gies, for various Δ^{UB}-values. At the optimal threshold a clear separation emerges on both maps between *Arbitrary* and the *Best-of-n* strategies. Within the 30-second time limit on *Den520d Sparse*, *Arbitrary* and *Semi-/Cardinal* prove optimality on 196 and 197 instances respectively, while *Best-of*-8, *Best-of*-16 and *Best-of*-∞ reach 286, 291 and 291—a 48% increase over branching at random. On *Room* the same ordering holds but with a smaller margin: 280 and 305 against 334, 342 and 342, a 22% increase. The advantage shrinks as the threshold loosens. On *Den520d Sparse* it survives at $\leq 1\%$ (280 and 289 against 350, 359 and 372) and at $\leq 5\%$ (497 and 473 against 537, 555 and 554). On *Room* it is already marginal at $\leq 1\%$ (401 and 434 against 438, 450 and 441) and absent at $\leq 5\%$, where all five configurations lie within 4% of one another (815 and 839 against 842, 811 and 824) and *Best-of*-16 is nominally the weakest. These results suggest that reasoning over conflict cardinality is most useful when aiming for optimal solutions, however, once a few percent of suboptimality is acceptable, the choice of conflict to branch on matters less, and on the map where good feasible solutions are easy to come by it stops mattering sooner.

The behaviour of *Semi-/Cardinal* is map-dependent, and this is the one place the two maps disagree qualitatively. On *Den520d Sparse* it is not merely a weak heuristic but a net loss: it costs 0.84× against *Arbitrary* in paired geometric-mean runtime, expands 18% *more* constraint-tree nodes, and at $\leq 5\%$ falls behind *Arbitrary* outright (473 against 497). On *Room* it is a modest gain instead, at 1.05× paired geometric-mean runtime with 10% *fewer* nodes, and ahead of *Arbitrary* at every threshold. What is consistent across both maps is that it captures only a small part of the available benefit: it recovers 25 of the 62-instance gap to *Best-of*-16 on *Room* and 1 of 95 on *Den520d Sparse*. We attribute this to the rule satisficing rather than maximising. Accepting the first conflict that raises either child's cost is barely more informative than just accepting the first conflict, yet it pays for a probe at every candidate it scans along the way, and unlike *Best-of-n* that scan is not capped—so whether it comes out slightly ahead or slightly behind depends on how expensive the uncapped scan happens to be on a given map. The benefit of cardinality reasoning in this setting comes from *ranking* candidate conflicts, not from *classifying* them.

These results suggest that the *Best-of-n* candidate sample saturates almost immediately, and this holds on both maps. At the optimal threshold *Best-of*-8, *Best-of*-16 and *Best-of*-∞ are separated by 5 and 0 instances on *Den520d Sparse* and by 8 and 0 on *Room*. Pairwise, their geometric-mean speedups are within 3% of each other on both maps (1.02× and 1.00× on *Den520d Sparse*, 1.00× and 1.03× on *Room*), with node counts within 2%. Since every probe of a candidate conflict costs two low-level searches (to determine the raise in cost), and the list of candidate conflicts grows with the number of agents, this matters in practice: apparently even a small candidate conflict set ($n \in \{8, 16\}$) suffices to find a good candidate, while ensuring that the amount of work is bounded, unlike with $n = \infty$.

Figure 9 and Table 4 compare *Arbitrary* and *Best-of*-16 on speedup, throughput, and overhead over $t_{Arbitrary}$ as a proxy for instance difficulty. Probing costs node throughput on both maps: *Best-of*-16 expands nodes at 0.63 to 0.89 times *Arbitrary*'s rate on *Den520d Sparse* and 0.75 to 0.97 times on *Room*. What it buys is a smaller constraint tree, and it buys progressively more of it as instances get harder—down to 0.77, 0.41 and 0.23 of *Arbitrary*'s node count on *Den520d Sparse*, and 1.05, 0.93 and 0.28 on *Room*. The net effect is therefore a

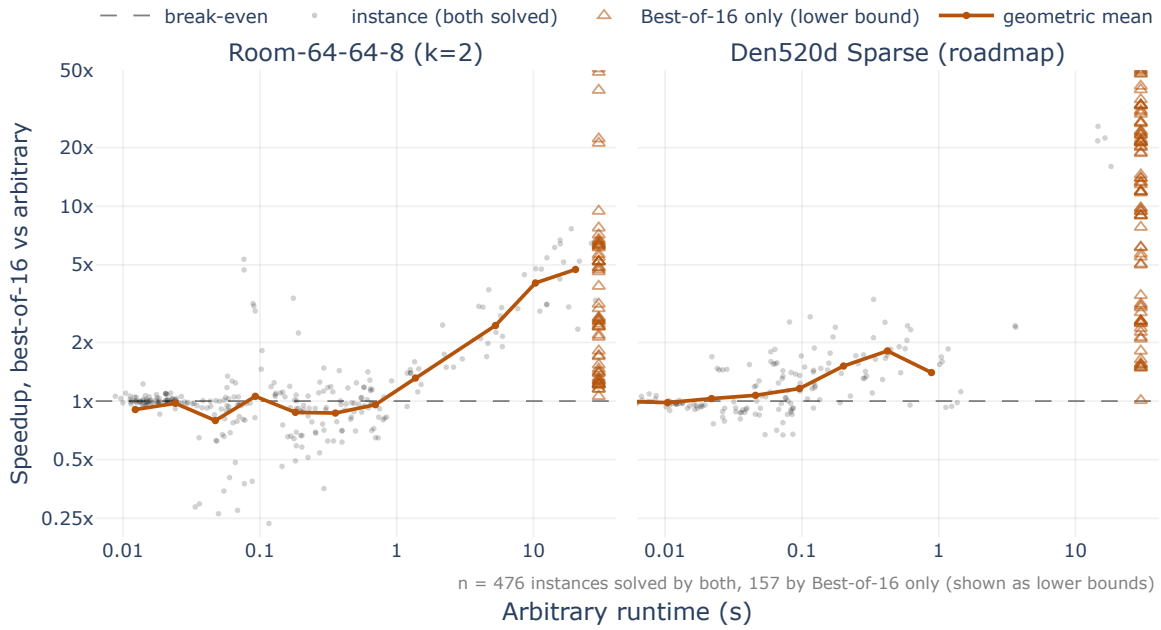


Figure 9: Per-instance speedup of *Best-of*-16 over *Arbitrary* against instance hardness, on *Room* and *Den520d Sparse*. Open triangles at the right edge are the instances only *Best-of*-16 proved optimal—62 on *Room* and 95 on *Den520d Sparse*—drawn at the lower bound $30\,\mathrm{s}/t_{\textit{Best-of-16}}$; there are none in the opposite direction on either map. The trend line covers only jointly-solved instances.

bucket	n	$t_{Arbitrary}$	$t_{Best\text{-}of\text{-}16}$	speedup	throughput	overhead
Room						
0–0.1	134	0.03	0.03	0.92×	0.97×	1.05×
0.1–1	102	0.31	0.35	0.90×	0.84×	0.93×
1–30	44	5.60	2.10	2.67×	0.75×	0.28×
> 30 (timeout)	62	> 30	8.74	$> 3.43\times$	–	–
Den520d Sparse						
0–0.1	137	0.02	0.02	1.03×	0.79×	0.77×
0.1–1	44	0.24	0.15	1.55×	0.63×	0.41×
1–30	15	3.35	0.85	3.94×	0.89×	0.23×
> 30 (timeout)	95	> 30	2.30	$> 13.06\times$	–	–

Table 4: Paired comparison of *Best-of*-16 against *Arbitrary* by instance hardness. Speedup decomposes as throughput divided by search overhead, where throughput is the ratio of node expansion rates and search overhead the ratio of nodes expanded. All entries are per-instance ratios averaged geometrically within a bucket. The final row of each block holds instances *Arbitrary* did not prove optimal within the budget, whose speedups are right-censored and reported as lower bounds.

crossover rather than a uniform gain, and *Room* shows it more starkly than *Den520d Sparse*: there, probing does not repay its cost at all below one second (0.92× and 0.90×), and only on instances taking *Arbitrary* more than a second does it pay off, at 2.67×. On *Den520d Sparse* the corresponding progression is 1.03×, 1.55× and 3.94×. Consequently the aggregate geometric mean over jointly-solved instances—1.25× on *Den520d Sparse* and 1.08× on *Room*—understates the effect substantially, being dominated by the trivial instances that make up 70% and 48% of those sets respectively. The strongest evidence is the censored row. Of the instances *Arbitrary* failed to prove optimal within the budget, *Best-of*-16 proved 95 on *Den520d Sparse*, taking a geometric-mean 2.30 seconds and therefore at least 13× less time, with extreme cases exceeding 100×; on *Room* it proved a further 62 at a geometric-mean 8.74 seconds, at least 3.4× less time, with extreme cases exceeding 50×. On neither map is there a single instance in the opposite direction.

6.3 Comparisons with OC-CBS on Additional Maps

We apply the benchmarking procedure of Section 6.2 to size MovingAI gridmaps and the three sampled roadmaps of Figure 3, using configuration **C6** with *Best-of*-16 conflict selection.

Figure 10 compares AOC-CBS and OC-CBS on the gridmaps. At $\Delta^{UB} = 0$ the two are roughly equivalent, AOC-CBS proving optimality on 3 414 instances against OC-CBS's 3 269. AOC-CBS is ahead on five of the six maps, by as much as 2.0× on *Room-64-64-8* (376 against 187) and 1.7× on *Room-32-32-4* (440 against 262), and behind on *Warehouse-10-20-10-2-2* (949 against 1 296). Given that OC-CBS is implemented in C++ and AOC-CBS in Python, we read this as evidnce that the implementation optimizations described

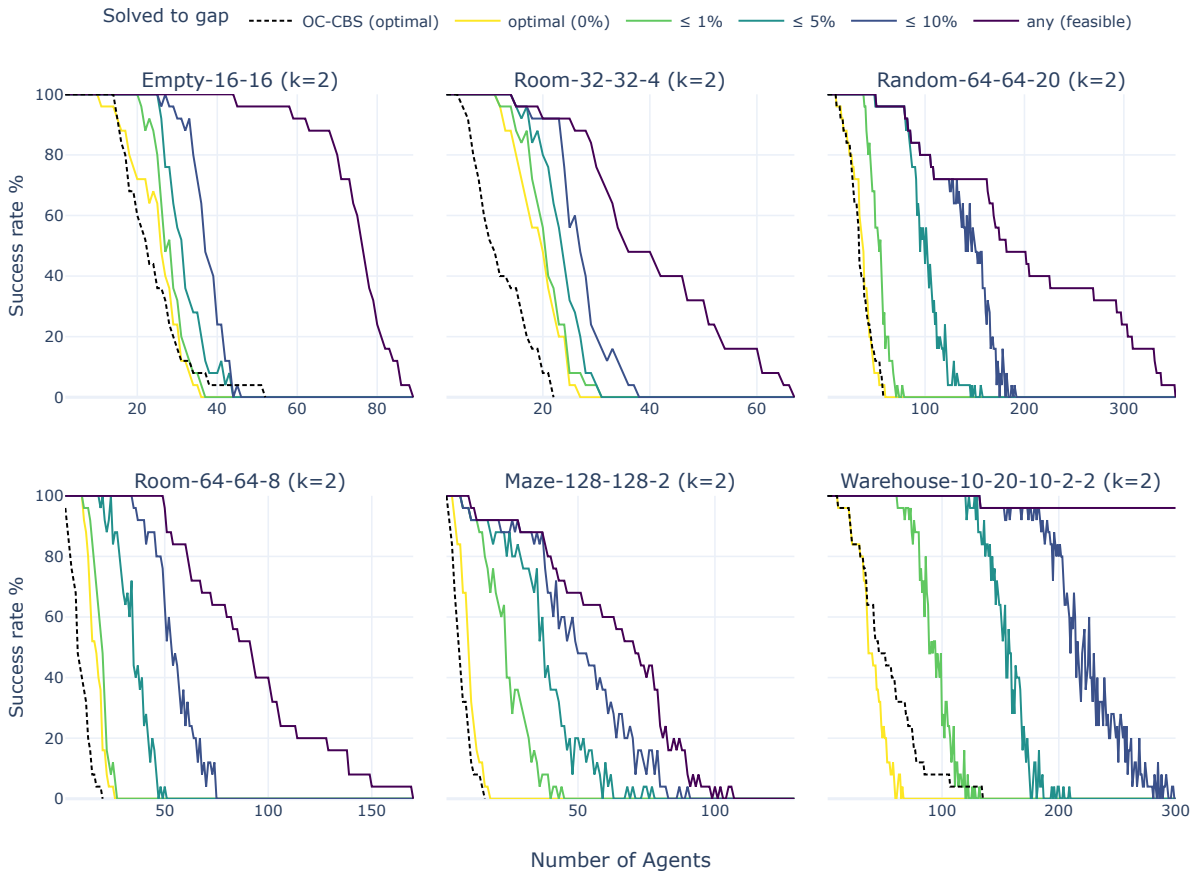


Figure 10: A comparison between AOC-CBS and OC-CBS on success rate at various optimality gap upper-bound values, on six different gridmaps. The map *Warehouse-10-20-10-2-2* was tested only up to 300 agents (experiments are ongoing).

in Appendix A offset the language gap, rather than as a claim of superiority at proving optimality.

The separation appears once suboptimality is permitted. Within the same 30 second time limit, AOC-CBS finds a feasible solution to 5.8× as many instances as OC-CBS solves optimally (19 078 against 3 269), and the agent count at which half the scnearios are still solved rises accordingly: from 22 to 76 on *Empty-16-16*, 7 to 91 on *Room-64-64-8*, 6 to 70 on *Maze-128-128-2*, and 33 to 181 on *Random-64-64-20*. On *Warehouse-10-20-10-2-2*, AOC-CBS solves every scenario at 300 agents (the limit of these preliminary experiments, reached by 24 of the 25 scenarios) while OC-CBS is down to a single scenario beyond 124 agents and solves none at 135. That map's 5.6× is therefore a lower bound.

Figure 11 repeats the comparison on the roadmaps, where the same pattern holds more starkly. At $\Delta^{UB} = 0$ the two methods are equivalent: AOC-CBS proves optimality on 616 instances against 592, ahead on all three maps but by margins that do not extend the reachable range: the success-rate curves reach zero at 7 agents on *Maze-32-32-2* and 19

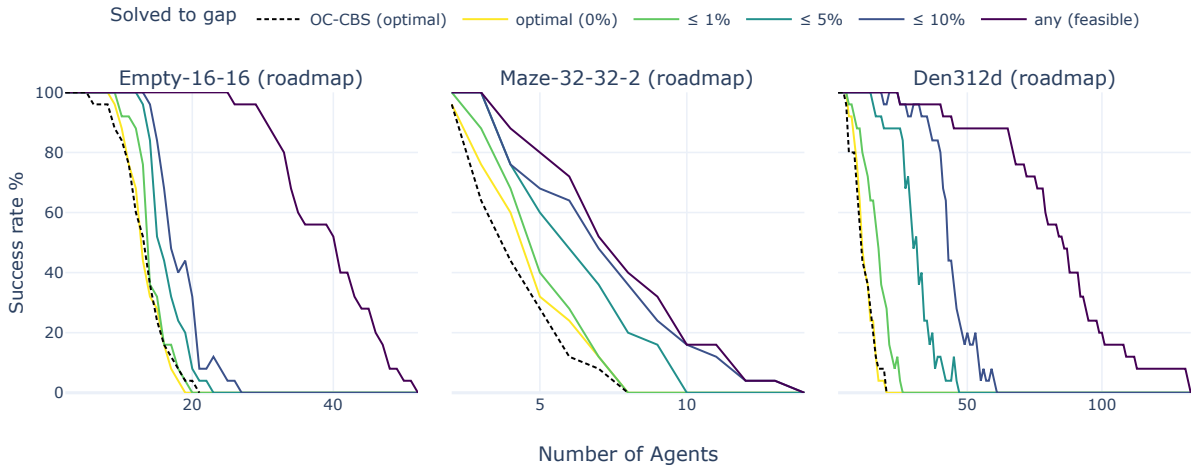


Figure 11: A comparison between AOC-CBS and OC-CBS on success rate at various optimality gap upper-bound values, on three different roadmaps.

on *Den312d* for both methods, and at 18 against OC-CBS's 20 on *Empty-16-16*. This is consistent with Section 6.2.1, where **C1** outperformed OC-CBS on both gridmaps but lagged on the roadmap *Den520d Sparse*.

Permitting suboptimality again separates them, by 5.4× overall (3 220 instances against 592). On *Den312d* the agent count at which half the scenarios are solved rises from 10 to 85, and the deepest scenario reaches 132 agents against OC-CBS's 19. Across both figures, the benefit of relaxing Δ^{UB} grows with the size of the map. On the smallest maps (*Empty-16-16*, *Room-32-32-4*, and the *Maze-32-32-2* roadmap), AOC-CBS solves 2.8 to 3.8× as many instances as OC-CBS, while on the largest (*Room-64-64-8*, *Maze-128-128-2*, and the *Den312d* roadmap), the factor is 8.6 to 12.3×. Accepting a bounded optimality gap is thus most valuable precisely where optimal methods scale worst.

6.4 Demonstration of Non-Convex Agents and Smooth Motion

This section describes the demonstration in Figure 1 that was first introduced in Section 1. The problem includes four agents, two of which are forklift-shaped and the other two are crane-shaped. The problem uses the implementation in Section A.3.3, which supports arbitrary 2D polygonal shaped agents and arbitrary connected and continuous trajectories.

Preprocessing consist of two parts, the the graph information (Section A.2) and the conflict information (Section A.1). All agents move on the same graph, with preprocessing of this graph taking 27 milliseconds. The conflict information is computed for each unique agent-type pair: between two forklifts (175 seconds), between two cranes (180 seconds), and between a forklift and a crane (306 seconds), all to a precision of 10^{-2} second. This graph and conflict preprocessing is a one-time cost that is valid for any instance on this same map, with any number of forklift and crane agents. If new types of agents are introduced, only the information involving these new agents must be computed.

AOC-CBS took 5.50 milliseconds to find the optimal solution.

7 Discussion

The comparison in Section 6 shows that AOC-CBS is able to solve around 5× as many problems across the entire benchmark compared to OC-CBS. It is important to remember, however, that OC-CBS is designed to only return optimal solutions. Read as a comparison of exact solvers, the two are roughly equivalent. AOC-CBS proves optimality on 3 414 gridmap instances against 3 269, and on 616 roadmap instances against 592, only differing by about 4%. The result we draw from this is therefore not that AOC-CBS is the stronger optimal solver, but that accepting a bounded optimality gap buys roughly a 5× increase in the number of instances solvable within the same budget (19 078 against 3 269 on the gridmaps, 3 220 against 592 on the roadmaps), and that no existing continuous-time solver offers that trade at a known optimality gap upper bound and gurantees of eventual optimality. The size of the gain grows with the map: 2.8 to 3.8× on the three smallest maps against 8.6 to 12.3× on the three largest, so bounded suboptimality is worth most where optimal methods scale worst. Note however that the reported optimality gap is an upper bound, so the true suboptimality of the returned solutions is unmeasured and may be considerably smaller.

The ablations point consistently to a single design principle: finding a solution and proving it optimal are different workloads, and no fixed configuration serves both. The repair function TP-SIPP extends the reachable agent count several-fold at loose Δ^{UB}, yet at $\Delta^{UB} = 0$ every repairing configuration is worse than its non-repairing counterpart on all three maps (**C2** below **C1**, **C4** below **C3**), because time spent repairing is time not spent expanding CT nodes and raising the lower bound. Careful conflict selection, on the other hand, appears to greatly benefit the search for optimal solutions, which is consistent with prior work [50]. This effect is less noticeable at higher Δ^{UB}. Parallelism improves solver performance with instance difficulty. From our limited experiments across this dimension, the throughput increase outweighs the overhead of managing a pool of multiple processes only beyond a certain problem difficulty threshold. Since Δ^{UB} is observable at runtime, a solver can schedule its policies against it, which is what **C6** does, matching **C5** at loose Δ^{UB} while recovering **C3**'s behaviour at $\Delta^{UB} = 0$, which no static configuration achieves at both ends. We expect this to apply to anytime bounded-suboptimal search more broadly than to AOC-CBS specifically.

The benefit of multiple policies is coverage rather than speed, and the aggregate statistics obscure this. Eleven workers reach at most 4.77× the node expansion rate of one, at most 43% parallel efficiency, and part of that is spent on nodes a sequential search would not expand (1.4 to 3.2× overhead). Resultingly, **C3** solves only 6–15% more instances than **C1**, and is the slower solver below roughly half a second. What the policies do buy is 67 instances that **C3** proved optimal and **C1** did not, with none in the opposite direction, raising coverage of the paired instances from 92.0% to 96.3%. For an anytime solver the relevant question is which instances become solvable at all, not how much faster the already-solvable ones are handled.

The conflict-selection results suggest that the value of cardinality reasoning lies in ranking candidate conflicts rather than classifying them, and that a small candidate sample suffices.

Semi-/Cardinal, which accepts the first conflict raising either child's cost, recovers only 1 of the 95-instance gap between *Arbitrary* and *Best-of*-16 on *Den520d Sparse* and 25 of 62 on *Room*, while paying for an uncapped amount of work; whether it is a net gain or a net loss depends on the map. By contrast *Best-of*-8 lies within 5 instances of *Best-of*-∞. Since each probe costs two low-level searches and the candidate list grows with the agent count, a bounded probe budget is both cheaper and, in these experiments, no less effective. Both observations apply to CBS variants in general, including the discrete-time case.

As detailed in Appendix A, the generality of AOC-CBS rests on moving domain-specific reasoning offline. Geometry, agent shape, and the conflict definition are distilled during pre-processing into information the high-level search queries without interpreting, which is why the same solver handles circular agents on straight-line edges, arbitrary continuous edge trajectories, and non-convex polygonal agents without change to the main solver. The cost of generality is therefore paid in pre-processing rather than in solving. This also bounds what we can currently claim: the pre-processing measurements in Figure 4 cover only the simple model of circular agents with straight-line traversals at constant speed, and establishing the corresponding cost for curved trajectories and polygonal agents is ongoing work.

Finally, AOC-CBS is implemented in Python and is level at $\Delta^{UB} = 0$ with a C++ implementation of OC-CBS. We read this as evidence that data-structure and pre-processing choices dominate language choice at this scale, rather than as a claim about the algorithms, and it suggests that a compiled implementation of AOC-CBS would extend the range of solvable instances further. We leave an AOC-CBS C++ implementation for future work.

Further work could also investigate the incorporation of disjunctive conflicts [50] and conflict bypassing [51]. Just as with careful constraint selection, both aim to reduce the size of the CT and thereby reach an optimal solution in fewer iterations. Incorporating disjunctive constraints would require a careful analysis of branching soundness to preserve the correctness guarantees, whereas conflict bypassing should require comparatively little effort. Conflict bypassing means to quickly search for a conflict-free path with an equivalent cost instead of branching in the CT. We suspect that such a strategy works best on gridmaps where many paths of equivalent cost exist, but not on roadmaps with non-unit length edges. Furthermore, TP-SIPP is only one repair function among possibly many, and AOC-CBS supports several repair functions within the same portfolio. Such repair functions could be paired with the selection policy best suited to them, or invoked at different stages of the search; some may perform better early on, for instance, while others may dominate later, once the underlying OC-CBS search has reached deeper CT nodes closer to a solution.

Further investigation into how AOC-CBS scales with the number of CPU cores would also be valuable. We expect performance to improve with additional cores up to a certain point. Consider the shallowest CT node representing an optimal solution, at depth N. If every leaf node were expanded at each iteration, as in a breadth-first search, this node would be found within N iterations, but doing so would require an impractically large 2^N cores. The CT node-selection policies instead focus the search on the most promising nodes, so we expect an optimal solution to be reachable in close to N steps without requiring as many as 2^N cores. Where this saturation point lies remains an open empirical question.

Finally, deployment on physical hardware remains the ultimate test for a MAPF solver intended for robotics. AOC-CBS allows for many types of agents, such as land-based robots, aerial vehicles, robotic arms, even non-physical agents such as jobs in a job-shop scheduling problem. As long as an agent has an associated state space which it moves through according to a discrete graph, AOC-CBS can plan its path to desired positions and trajectories, in the same system as other types of agents. Future work could evaluate how planning directly over feasible motions, which AOC-CBS enables at higher scalability, affects robustness and solution quality under real-world execution.

8 Conclusion

This work generalised the MAPF problem to heterogeneous agents on individual graphs, continuous time, abstract pairwise conflicts, sequences of edge and vertex tasks, and agents free to move after completing them. For this generalised problem we presented AOC-CBS, a solver that combines the guarantees of an exact solver with the responsiveness of a suboptimal one. Like OC-CBS, AOC-CBS is exact and solution complete: given enough time, it is guaranteed to return an optimal solution. Unlike OC-CBS, it does not require that time to elapse before returning anything: at every point during its search, AOC-CBS makes available an incumbent solution together with a certified upper bound on how far that incumbent can be from optimal, a bound that tightens as the search continues and reaches zero exactly when optimality is reached and verified. This anytime behaviour comes from decoupling the search that establishes the lower bound from a configurable portfolio of repair functions that searches for incumbents, one instance of which, TP-SIPP, we introduced to handle agents that must vacate a completed task; the same decoupling also allows multiple policies to be expanded in parallel across processor cores.

Our preliminary experiments show that AOC-CBS matches OC-CBS at finding and verifying optimal solutions, while solving several times as many problem instances once a bounded optimality gap is accepted, extending scalability from the tens to the hundreds of agents. The ablations further show that the portfolio's components trade off along different axes rather than uniformly. Repair functions markedly extend the reachable agent count at loose optimality gaps, but the time they spend searching for incumbents is time not spent raising the lower bound, making them a net cost once the goal is a proven optimum rather than a fast solution. Additional policies benefit harder instances more than easy ones, since managing a pool of processes costs more than it returns on instances a single core already solves quickly; their main effect is not raw speed but coverage, bringing instances that are otherwise out of reach within reach. Careful conflict selection helps throughout the search rather than only near the end, and its advantage grows with instance difficulty, but that advantage is largest when the goal is a proven optimum and narrows, without disappearing, as the required gap loosens. This motivates a portfolio that shifts its configuration as the optimality gap closes, which our gap-cadence configuration does.

Existing continuous-time MAPF solvers force a choice: an exact solver that returns nothing until it has proven optimality, or a suboptimal one that returns quickly but with no guarantee on solution quality, nor of eventual optimality. AOC-CBS removes this trade-off. By pairing

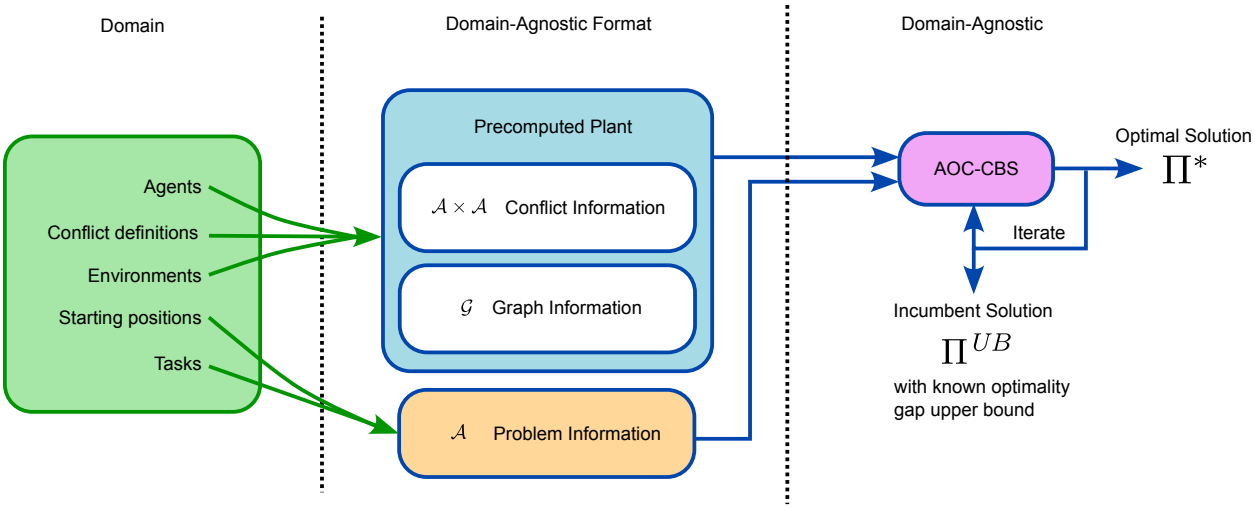


Figure 12: The architecture of the framework surrounding AOC-CBS.

OC-CBS's exactness with an anytime incumbent and a certified optimality gap, it is usable both as a fast heuristic solver and as an exact one, without having to commit to either role in advance.

A Implementation

As discussed in section 3.1, much of the the problem formulation's generality stems from the abstract function *InConflict* separating the AOC-CBS solver from the specifics of an application. This section introduces a practical framework surrounding this abstract *InConflict* function, and additionally details our implementation of this framework under multiple different assumptions.

Figure 12 provides an overview of the architecture surrounding AOC-CBS. On the left is the specific domain, containing the agents, what they are and how they move in their state spaces, the conflict definitions (e.g., geometric collisions, mutually exclusive resources, etc.), the environments in which the agents exist, the agents' starting positions, and their tasks. This domain-specific information is converted into a domain-agnostic format (center of Figure 12), consisting of two parts: the *plant* information and the *problem* information. The plant information is assumed to be known before-hand, including what each agent is and the graph it moves on. For instance, we assume that a warehouse knows how many agents they have and what the agents are before the start of operations, and that this information is static. The precomputed information contains conflict information between agents as well as information about the graphs themselves that is useful during single-agent path planning. The problem information is not known before-hand and contains each agent's starting position and its sequence of tasks; this information cannot be utilized outside of the runtime. The plan and problem information are specific to the domain, encoding all that is required about the specific domain, but is saved in an domain-agnostic format. That is, the format of this information is the same, regardless of if the domain is a logistical warehouse,

and manufacturing cell of robotic arms, or a swarm of drones. Finally, the plant information and problem information is fed to AOC-CBS, upon which the iterations begin. At every iteration, an incumbent solution Π^{UB} (once one has been found) with a known optimality gap upper bound is made available. Within a finite amount of time, AOC-CBS halts with an optimal solution Π^*.

A.1 Precomputed Conflict Information

The conflict information distils the domain-specific conflict definitions into a set of time intervals, an *annotation* technique first introduced in [28]. These time intervals can be precomputed and stored in lookup tables for use during runtime. This conflict information is stored in a domain-agnostic format and is what AOC-CBS consumes.

For any agent pair $\langle i, j \rangle \in \mathcal{A} \times \mathcal{A} : i \neq j$, the following conflict information can be computed and stored in lookup tables:

1. **Vertex-Vertex**: $\forall \langle v^i, v^j \rangle \in \mathcal{V}^i \times \mathcal{V}^j$, is $\mathit{InConflict}\,(i, v_s^i, j, v_s^j)$?
2. **Vertex-Edge**:
   - $\forall \langle v^i, e^j \rangle \in \mathcal{V}^i \times \mathcal{E}^j$, for which times $t \in [0, e_D^j)$ is $\mathit{InConflict}\,(i, v_s^i, j, e_\tau^j(t))$?
   - $\forall \langle e^i, v^j \rangle \in \mathcal{E}^i \times \mathcal{V}^j$, for which times $t \in [0, e_D^i)$ is $\mathit{InConflict}\,(i, e_\tau^i(t), j, v_s^j)$?
3. **Edge-Edge**: $\forall \langle e^i, e^j \rangle \in \mathcal{E}^i \times \mathcal{E}^j$, for which start times $\delta \in [-e_D^i, e_D^j)$ for i to traverse e^i does there exists a $t \in [\max(0, \delta), \min(e_D^i, e_D^j))$ where $\mathit{InConflict}\,(i, e_\tau^i(\delta + t), j, e_\tau^j(t))$?

Informally, the Vertex-Vertex case simply tell us which vertices can the two agents not be located at simultaneousness, the Vertex-Edge cases tell us during what times do the agents conflict if one waits at a vertex and the other begins traversing its edge at time $t = 0$, and the Edge-Edge case tells us at what times can one agent begin traversing an edge given that the other begins traversing its edge at time $t = 0$. This conflict information is what [27] refers to as *intersection intervals* and what [34] refers to as *conflict intervals*. The bounds on t for the Vertex-Edge cases and δ and t for the Edge-Edge case exist since we do not consider conflicts outside of the time during which the edge trajectories are defined.

The Vertex-Vertex case results in a countable and bounded list of mutually exclusive vertices, however, the remaining cases work over continuous times and therefore have certain requirements for these computations to be practically feasible. Under a finite-precision implementation with a minimum precision ϵ_{mach}, two real values differing by less than ϵ_{mach} are indistinguishable and by only a few ϵ_{mach} indistinguishable from rounding errors. Any conclusions at these scales unreliable. For this reason, we assume that all conflicts have a duration of at least a few order of magnitude above ϵ_{mach}. We then know that over the searchable intervals ($t \in [0, e_D^j)$ and $t \in [0, e_D^i)$ for Vertex-Edge, $\delta \in [-e_D^i, e_D^j)$ for Edge-Edge) the times at which conflicts occur are grouped into maximally connected, non-zero-sized intervals. Searching for these intervals can be done using established methods (e.g., using Lipschitz bounds as described in Section A.1.1 below) to a specified precision. Furthermore, conservative estimations of these interval can be used, where the size of the conflict intervals

are inflated, to include a safety margin to also allow for courser precision when searching for these intervals.

This conflict information contains all that is needed to implement conflict avoidance. The Vertex-Vertex information provides mutually exclusive vertex-pairs that are used to avoid wait-wait conflicts. Although, we note that wait-wait conflicts are not reasoned over in the main AOC-CBS algorithm since, assuming the agents start in conflict-free positions, wait-wait conflicts must always be preceded by a move-wait or move-move conflict. However, this information is still useful for various optimizations. The Vertex-Edge information provides times when an agent cannot wait at a vertex if another traverses and edge at time $t = 0$; since the graphs are time-invariant, this information can be shifted in time. Likewise for the Edge-Edge information that assumes one agent traverses its edge at $t = 0$, this information can be shifted in time.

Although it is stated in the above that these computations must be done for each ordered agent-pair $\langle i, j\rangle \in \mathcal{A}\times\mathcal{A} : i \neq j$, in practice the above information computed for $\langle i, j\rangle$ is easily converted for $\langle j, i\rangle$: this is obvious for Vertex-Vertex, Vertex-Edge are already bi-directional, and for Edge-Edge the intervals can be negated and inverted since it is the relative time difference between the traversals that matters. That is, if i cannot begin its traversal in $[t_1, t_2)$ when j begins its traversal at $t = 0$, then if instead i begins its traversal at $t = 0$ then j cannot begin its traversal in $[-t_2, -t_1)$. Thus, the worse-case number of agent pairs to compute for is reduced by roughly half.

Furthermore, coarse filtering can be applied to quickly reduce the number of graph element pairs to check for collisions. For instance, if two graph elements are too far from each other for two agents to possibly conflict while one each respective element, then computationally intensive conflict detection is not required. This further reduces the computational work for preprocessing.

Many applications may have multiple agents of the same *agent model*, ρ^a for agent a. Although it is sufficient for our framework to let ρ^a remain abstract, ρ^a could for instance specify that a is a robotic arm, as well as all the attributes required to determine how it interacts with other agents. Importantly, if $\mathit{InConflict}\,(i, s_i, j, s_j)$ then for another agent k with $\rho^k = \rho^i$ we know that $\mathit{InConflict}\,(k, s_k, j, s_j)$. Given this, the conflict information does not need to be computed for every (unordered) agent pair, but instead every unordered pair of unique $\langle \rho^a, \mathcal{G}^a\rangle$ over $a \in \mathcal{A}$. In the best case, all agents have the same model and operate on the same graph (a common assumption in the MAPF literature), requiring only a single instance of the conflict information.

Since this conflict information is the only connection between AOC-CBS and the specific application over which AOC-CBS plans, all correctness guarantees that we provide here and the validity of any returned solution is with respect to this conflict information.

A.1.1 Lipschitz-Bounded Conflict Search

The Vertex-Edge and Edge-Edge conflict information consists of intervals along a continuum of time or offset, and reliably computing such an interval requires verifying that every point

within it corresponds to a conflict, without evaluating each of the infinitely many points it contains. This section develops a general method for doing so using Lipschitz bounds, verifying whole intervals from a small number of pointwise evaluations.

Conflict is determined using two components: a *conflict predicate* and a *Lipschitz bound*. A conflict predicate $f(i, s_i, j, s_j)$ encodes whether agent i in state s_i and agent j in state s_j are in conflict, via $f(i, s_i, j, s_j) \leq 0 \Leftrightarrow \mathit{InConflict}\,(i, s_i, j, s_j)$. A Lipschitz constant L_f bounds how fast f can change with respect to the relative position between the two agents' centres, i.e. $|f(i, s_i, j, s_j) - f(i, s_i', j, s_j')| \leq L_f \cdot |\Delta\mathbf{p} - \Delta\mathbf{p}'|$ for any two relative positions $\Delta\mathbf{p}$ and $\Delta\mathbf{p}'$.

These three components are used together to efficiently partition time into conflicting and non-conflicting intervals, without needing a closed-form solution for when conflicts occur. Consider agents i and j following trajectories e_τ^i and e_τ^j, with durations e_D^i and e_D^j, and known maximum speeds $|\mathbf{v}^i|_{\max}$ and $|\mathbf{v}^j|_{\max}$ respectively. At some time t, the predicate can be evaluated as $f(t) = f(i, e_\tau^i(t), j, e_\tau^j(t))$. Suppose $f(t) > 0$, i.e. at time t the agents are not in conflict. Since time is continuous, ruling out only the single instant t is not useful on its own. Instead, a window around t can be ruled out directly using the Lipschitz bound. The relative position between the agents can change at a rate of at most $|\mathbf{v}^i|_{\max} + |\mathbf{v}^j|_{\max}$, and by the Lipschitz bound, f can therefore change by at most $L_f \cdot (|\mathbf{v}^i|_{\max} + |\mathbf{v}^j|_{\max})$ per unit time. Consequently, over a duration of magnitude at most

$$\Delta t = \frac{f(t)}{L_f \cdot (|\mathbf{v}^i|_{\max} + |\mathbf{v}^j|_{\max})}$$

f cannot have decreased to 0, so no conflict can occur within the interval $[t - \Delta t, t + \Delta t]$. This interval can be skipped without further evaluation, and the process repeated from its boundary, allowing the conflict-free and conflicting regions of time to be identified using only pointwise evaluations of f. The same procedure symmetrically holds if $f(t) \leq 0$ (i.e., a conflict at time t) instead; $[t - \Delta t, t + \Delta t]$ is an interval of only conflict.

This method can be repeated at each boundary of verified conflict/no conflict interval until a desired precision is reached. In practice, this method is not run to converge on a single point since that would require unboundedly many evaluations near any boundary where $f(t)$ approaches 0. Any interval that remains unresolved when the search terminates (that is, the small intervals between a verified conflict and verified non-conflict interval of size less than the desired precision) is conservatively treated as a conflict interval. This yields an estimate where no genuinely conflicting interval is misclassified. The cost of this guarantee is that a small margin of conflict-free interval is classified as conflicting. However, for the intended use of this conflict information, a false conflict merely costs some avoidable caution, while the opposite would be unsound.

A.2 Precomputed Graph Information

Besides precomputing conflict information from above, we also precompute information pertaining to a single graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$. The low-level MG-CSIPP (Section 4.2) and TP-SIPP (Section 4.3) perform A* searches to construct a lowest-cost plan for a single agent. The

heuristic used in these A* searches take advantage of the minimum cost from any vertex to any other vertex, which is precomputed and stored. The runtime cost of computing the all-pairs shortest paths in a non-negative weighted graph is $\mathcal{O}(|\mathcal{V}|^3)$ with the Floyd-Warshall algorithm or, on sparse graphs where $|\mathcal{E}| \ll |\mathcal{V}|$, using Dijkstra's with a Fibonacci heap gives $\mathcal{O}(|\mathcal{V}|^2 \log |\mathcal{V}| + |\mathcal{V}||\mathcal{E}|)$ [52]. The memory cost is $\mathcal{O}(|\mathcal{V}|^2)$.

In our implementation, this graph information is computed and stored on disk. When a particular problem is given, we load into RAM only the shortest-path distances to the source vertex of every agent's tasks, as that is the only information needed during the A* searches. This reduces the amount of RAM storage needed (significant for large graphs) while avoiding computing this information at runtime.

A.3 Agent and Environment Models

The conflict information (Section A.1) and graph information (Section A.2) build on an underlying model of the agents, the graphs, and the conflict definitions. We currently provide implementations for the following MAPF-grounded models

> **Model 1** (Section A.3.1): the common MAPF assumptions—agents are circular, omnidirectional, and without acceleration constraints. Edge trajectories are straight lines at constant speed, and all vertices are waitable. Conflicts are based on geometric collision. Unlike most MAPF problems, agents here may still have unique radii and move on unique graphs.
>
> **Model 2** (Section A.3.2): extends Model 1 with arbitrary connected and continuous trajectories. Vertices may be non-waitable.
>
> **Model 3** (Section A.3.3): extends Model 2 with arbitrary 2D polygonal-shaped agents.

Work is ongoing to extend these models into 3D space.

A.3.1 Model 1

This model follows the common MAPF assumptions where agents and graphs are defined in 2D Euclidian space, the agents are circular (although here they have different radii and exist on separate graphs), and edge traversals are done in straight lines at constant speed from the source to target vertex. An agent's state space consists of 2D position only; orientation does not matter for circular agents, and velocity is not necessary since all vertices are waitable. Conflicts include only geometric collision: two agents conflict when their circles overlap.

This setting allows for the closed-form computation of conflict information [53, 28, 54] (see [54] for a visual description), making this model the fastest to precompute out of those provided in this work. Vertex-Vertex information reduces to simply checking if the distance d between two agents at their respective vertices is less than the sum of their radii, $d < r_1 + r_2$, and Vertex-Edge requires the same check for the distance d from the stationary agent's vertex to the nearest point on the line segment of the moving agent's traversal. Edge-Edge collisions instead form a conic section $d^2(t, \delta)$ representing the squared distance between the two agents, as a function of time t and the one agent's starting time δ (the other agent

starting at $t = 0$). The interior of the ellipse $d^2(t, \delta) = 0$ in the (t, δ)-plane represents a guaranteed collision only if the agents travel along the infinite line between their respective source and target vertices. The time when both agents are actually traversing their respective line segments (instead of the infinite lines) form a parallelogram in the (t, δ)-plane. For every point (t, δ) within both the parallelogram and the ellipse, the agents will be colliding with each other at time t if the one agent traverses its edge starting at time δ. The points minimizing and maximizing δ within this intersection form a single interval, which is what we store as the Edge-Edge conflict information.

A.3.2 Model 2

This model extends Model 1 to include arbitrary connected and continuous edge trajectories. We describe how trajectories are represented and how the conflict information can be computed efficiently.

A state in this model is a pair $(\mathbf{p}, \mathbf{v}) \in \mathbb{R}^2 \times \mathbb{R}^2$ consisting of a position $\mathbf{p}$ and velocity $\mathbf{v}$; orientation is omitted since agents are circular. Each vertex v corresponds to a specific state $v_s = \langle \mathbf{p}_v, \mathbf{v}_v \rangle$, and is waitable if $\mathbf{v}_v = \mathbf{0}$. The trajectory $e_\tau(t)$ of an edge $e \in \mathcal{E}$ from $v \in \mathcal{V}$ to $u \in \mathcal{V}$ is defined by a pair of cubic splines, one for each of the two dimensions, interpolated through a sequence of position-time waypoints with clamped boundary conditions. At the first waypoint, $e_\tau(0) = \mathbf{p}_v$ with $e'_\tau(0) = \mathbf{v}_v$, and at the last, $e_\tau(e_D) = \mathbf{p}_u$ with $e'_\tau(e_D)) = \mathbf{v}_u$. The trajectory passes through each intermediate waypoint at the position and time it specifies, with the velocity and acceleration of the incoming segment matching those of the outgoing segment.

Unlike in the previous model where straight-line trajectories allow for the closed-form computation of conflict information, here the arbitrarily shaped trajectories warrant the use of the Lipschitz bounded method (Section A.1.1). Additionally, since conflict is collision-based and therefore depends on the spatial separation of agents, we additionally make use of an *interaction radius* which is the largest possible center-to-center separation between two agents at which a conflict remains possible. That is, if the agents' centres are further apart than the sum of their interaction radii, no conflict can occur regardless of their orientation or state.

In this model, agents are circular, and conflict is defined by circle separation: two agents with radii r_i and r_j are in conflict when the distance between their centres is at most $r_i + r_j$. This gives the straight-forward conflict predicate

$$f(i, s_i, j, s_j) = |\mathbf{p}_j - \mathbf{p}_i| - (r_i + r_j),$$

where $\mathbf{p}_i$ and $\mathbf{p}_j$ are the positions extracted from s_i and s_j. Under this predicate, the interaction radius is exactly $r_i + r_j$, since $f \leq 0$ is possible only when the centres are within this distance. The Lipschitz constant is $L_f = 1$, since f is a norm of the relative position minus a constant, and the norm is 1-Lipschitz.

The search for finding all necessary conflict information consists of two levels: the *broadphase* and the *narrowphase*. The broadphase quickly filters out graph element pairs that cannot be involved in a conflict; two edges on separate sides of the environment do not require expensive conflict detection if they are too far away for agents on each edge to possibly

conflict with each other. Each vertex and edge gets a swept bounding box, inflated by the interaction radius; non-overlapping pairs are discarded without evaluating f since the agents never get within the interaction radius. For edges, this box is computed exactly via root-finding on the spline's derivative polynomials. Non-waitable vertices (those with non-zero velocity components) are excluded from Vertex-Vertex and as Vertex-Edge sources, since they cannot produce a waiting conflict.

The narrowphase evaluates every graph element pair that passed the broadphase. Vertex-Vertex needs one evaluation on f, just as in Model 1. For Vertex-Edge, the waiting agent i is kept stationary at v_s^i and we use the Lipschitz-bounded conflict search to find all $T \in [0, e_D^j) : \varphi(T) \leq 0$ where $\varphi(T) = f(i, v_s^i, j, e_\tau^j(T))$ For Edge-Edge, a nested search is required instead. We fix j to begin traversing e^j at time $t = 0$, and seek all start times δ for i to begin traversing e^i such that the agents conflict while traversing their respective edges. The outer search therefore is done over i's start times $\delta \in [-e_D^i, e_D^j)$ using the predicate

$$h(\delta) = \min_t g(t, \delta),$$

with the inner search over $t \in [\max(0, \delta), \min(e_D^i, \delta + e_D^j)]$ using the predicate

$$g(t, \delta) = f(i, e_\tau^i(t), j, e_\tau^j(t - \delta)).$$

Each evaluation of $h(\delta)$ requires an inner minimization over t, since two agents traversing with offset δ are conflict-free only if $g(t, \delta) > 0$ for every t in the valid window, which is exactly the condition $\min_t g(t, \delta)$. However, the inner search does not need the true minimum, only its sign: as soon as any evaluated t yield $g(t, \delta) \leq 0$, a conflict at offset δ is already certified, and the search terminates immediately. Only when no such t is found does the search need to continue until every remaining cell has been certified $g > 0$, at which point δ is certified conflict-free. This asymmetry means a conflicting δ is typically resolved after a few evaluations, while a conflict-free δ requires exhausting the entire inner domain.

Both searches rely only on a Lipschitz bound on their respective objective, obtained from, obtained from L_f by the chain rule. For the inner search, both agents move, giving $|\partial g/\partial t| \leq L_f \cdot (|\mathbf{v}^i|_{\max} + |\mathbf{v}^j|_{\max})$; for the outer search, only e^j's traversal shifts with δ, giving $|\partial h/\partial \delta| \leq L_f \cdot |\mathbf{v}^j|_{\max}$. The smaller outer constant lets each evaluation of h certify a wider band of δ at once than a comparable evaluation of g could certify for t, and since each evaluation of h triggers an entire inner search, fewer outer evaluations directly reduce the total number of inner minimisations needed.

Given a Lipschitz constant L for either search, a cell $[a, b]$ with midpoint m and width w can be certified through only an evaluation at m: if $\varphi(n) - Lw/2 > 0$ the cell is conflict-free and discarded. If instead $\varphi(m) + Lw/2 \leq 0$ the cell is entirely conflicting and kept whole, otherwise it is bisected further. The Vertex-Edge search, and the outer Edge-Edge search over δ, apply this test directly to find the sublevel set $\{\delta : h(\delta) \leq 0\}$. The inner Edge-Edge minimisation over t uses the same bound within a branch-and-bound search that always expands the cell with the lowest lower bound $\varphi(m) - Lw/2$.

The resulting conflict information is a one-sided over-approximation: a cell is discarded only when the Lipschitz certificate proves $f > 0$ throughout it, so a conflicting offset δ is never

certified conflict-free, though a marginally conflict-free offset may conservatively be treated as conflicting.

A.3.3 Model 3

This model extends Model 2 by replacing the circular agent shape with a 2D polygon, which may be non-convex. Everything else about the framework is unchanged: the same broadphase/narrowphase search and the same Lipschitz-bounded certified sampling produce the conflict information. What changes is the state space, the set of edge trajectories, and the conflict predicate.

A polygon is not rotation-invariant, so orientation becomes part of the state: a state is a triple $\langle \mathbf{p}, \varphi, \mathbf{v} \rangle$ consisting of a position $\mathbf{p} \in \mathbb{R}^2$, an orientation $\varphi \in \mathbb{R}$, and a velocity $\mathbf{v} \in \mathbb{R}^2$. An agent's shape is given as an ordered list of polygon vertices in the agent's body frame, relative to its reference point; placing the agent in a state rotates those vertices by φ about the reference point and translates them by $\mathbf{p}$. As in Model 2, each vertex of the graph corresponds to a specific state, and is waitable if its velocity is zero.

Adding orientation admits edge trajectories that Model 2 cannot express, since a trajectory must now prescribe a heading in addition to a position at every t. Nothing in the method depends on the specific choice of trajectory, only on the properties it exposes: a trajectory need only be continuous, connect its endpoint states, and report its duration, a bounding box, and a bound on how fast the agent's body can move. This abstraction accommodates trajectory types ranging from a simple turn in place or straight-line traversal at a constant held heading, to curvature-constrained curves (e.g., Dubins) or a position spline paired with an independently prescribed orientation schedule.

Conflict is again geometric collision, but between two placed polygons rather than two circles. We use the signed separation

$$f(i, s_i, j, s_j) = \begin{cases} -d & \text{if the placed polygons overlap,} \\ +d & \text{otherwise,} \end{cases}$$

where d is the minimum Euclidean distance between the two placed polygons' boundaries, taken as the minimum over all pairs of boundary segments, one from each agent. Overlap is detected by two tests: whether any boundary segment of one agent transversally crosses a boundary segment of the other, and whether either polygon contains a vertex of the other. Containment uses even-odd ray casting, which is correct for any simple polygon, so no convexity is assumed anywhere and no convex decomposition is performed. Disjoint agent shapes therefore give $f = d > 0$, one shape fully inside the other gives $f = -d < 0$, and a partial overlap, where the boundaries touch, gives $d = 0$ and hence $f = 0$. The set $\{f \leq 0\}$ is thus exactly the overlap set, which is what soundness requires. Note that f is a boundary distance and not a penetration depth, so it does not necessarily continue to decrease as a partial overlap deepens; the only consequence is that the certified search bisects somewhat more near deep overlaps than a true penetration depth would require.

The interaction radius is the sum of the two agents' circumradii, where an agent's circumradius is the largest distance from its reference point to any of its polygon vertices. Beyond

that separation the agent shapes cannot touch, whatever their orientations. The Lipschitz constant is again $L_f = 1$, since a boundary distance changes by at most the distance any body point moves.

This last point is where Model 3 differs from Model 2 in a way that matters for the certified search. In Model 2 the chain rule converts L_f into a bound in time using each agent's maximum translational speed, because a circular agent shape is completely described by its centre. A rotating polygon, by contrast, sweeps area even while its reference point is stationary, so the relevant quantity is the speed of the fastest point of the body: the translational speed plus the angular speed multiplied by the circumradius. Each trajectory reports this quantity. A turn in place has zero translational speed and reports (angular speed $\times$ circumradius); a Dubins curve travelling at speed v with turning radius R reports $v(1 + r_{\text{body}}/R)$; a spline with an orientation schedule reports the spline's maximum speed plus the maximum of the schedule's angular rate times the circumradius. With this bound in place, the Vertex-Edge and Edge-Edge searches of Model 2, including the nested inner minimisation over t for Edge-Edge, are used verbatim, and inherit the same guarantee: the resulting conflict information is a one-sided over-approximation, never certifying a conflicting offset as conflict-free.

The broadphase likewise carries over: every vertex and edge gets a bounding box inflated by the interaction radius, and non-overlapping pairs are discarded. A turn-in-place edge has a point-sized box, since its reference point does not move, and the inflation by the two circumradii covers the whole area the rotating body sweeps. A Dubins edge's box is obtained from its straight segments' endpoints together with the bounding boxes of its arcs' supporting circles.

Polygon separation is by far the most expensive of the predicates we implement: a single evaluation performs a segment-to-segment distance computation for every pair of boundary segments of an agent's shape, one from each agent, and is therefore quadratic in the agent shapes' vertex counts, where the circle predicate is a single norm. Since the certified search calls the predicate many times per graph element pair, this makes Model 3 the most expensive model to precompute, and is the reason Model 1 and Model 2 retain the circle predicate rather than treating a circle as a many-sided polygon.

Finally, a polygonal agent shape does not require a graph with orientation in its states. A polygon held at a fixed orientation can be used directly on a Model 1 graph, and polygon-with-circle agent pairs are supported, so a polygonal ground robot can share an environment with a circular agent. Whichever route is taken, the resulting conflict information is written in the same format, and AOC-CBS cannot distinguish the two.

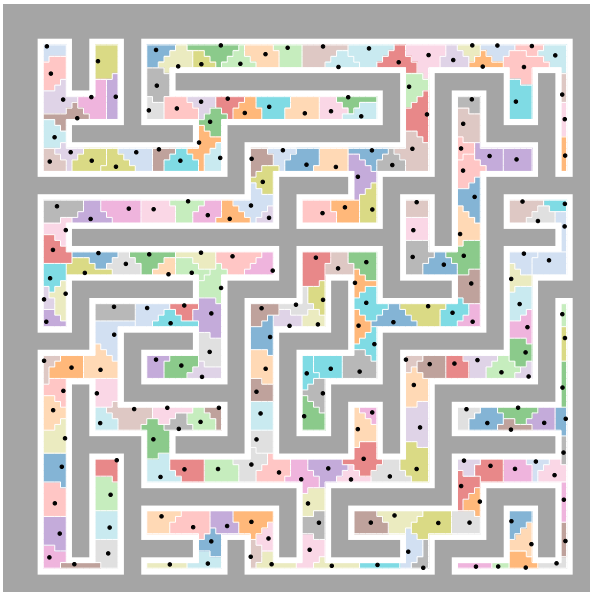
B Roadmap Generation

The MovingAI gridmaps [29] are the standard benchmark for discrete-time MAPF. These gridmaps are represented by a connected, undirected graphs with unit-length edges. Roadmap instances for continuous-time MAPF exist in prior work, e.g., the *Den520d* roadmaps used by CCBS [20] and OC-CBS [27], and PRM-sampled instanced from [28], but neither offers

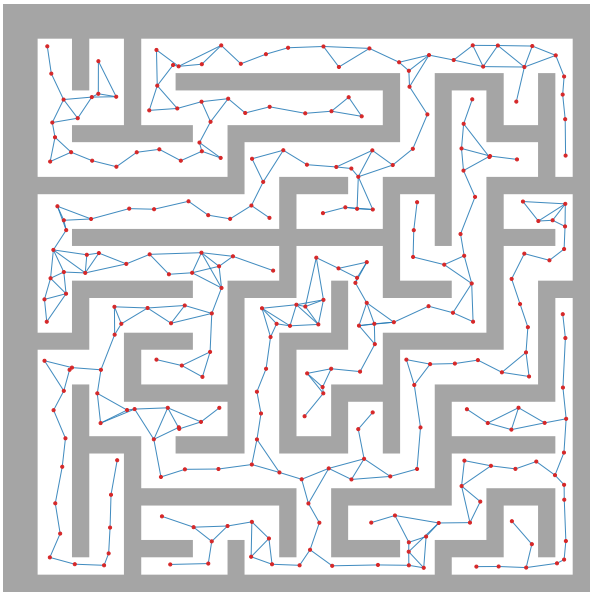
a systematic scheme for deriving roadmaps from the standard MovingAI benchmark suite with guaranteed connectivity, obstacle clearance, and bounded stretch. We present such a scheme, apply it across the full suite, and release the resulting roadmaps together with matching scenario files.

B.1 Sampling Roadmaps From Gridmaps

Given a MovingAI map, its free cells define a free space $X_{\text{free}} \subset \mathbb{R}^2$, with every unit cell centred on an integer coordinate and everything outside the map counted as blocked. To turn this into a roadmap $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, we require four parameters: a target vertex density $\rho > 0$, a clearance radius $r > 0$, a stretch factor $\kappa \geq 1$, and a maximum edge length $l_{\max} > 0$. Figure 13 provides an illustration of the sampling steps. Informally, the free cells are divided into finer grid cells; these fine cells within r from blocked cells are removed. Vertices are scattered evenly across the free space at roughly the requested density ρ. Each pair of vertices whose free-space Voronoi regions are adjacent (Figure 13a) is joined along a route through the boundary between them, reduced to as few straight segments as possible (Figure 13b); and shortcut edges shorter than $l_{\max}$ are added wherever this graph would otherwise force a detour between two vertices that is more than κ times the straight-line distance. The resulting roadmaps using three different densities on the *Den312d* gridmap are shown in Figure 14. We detail each of the steps in turn.

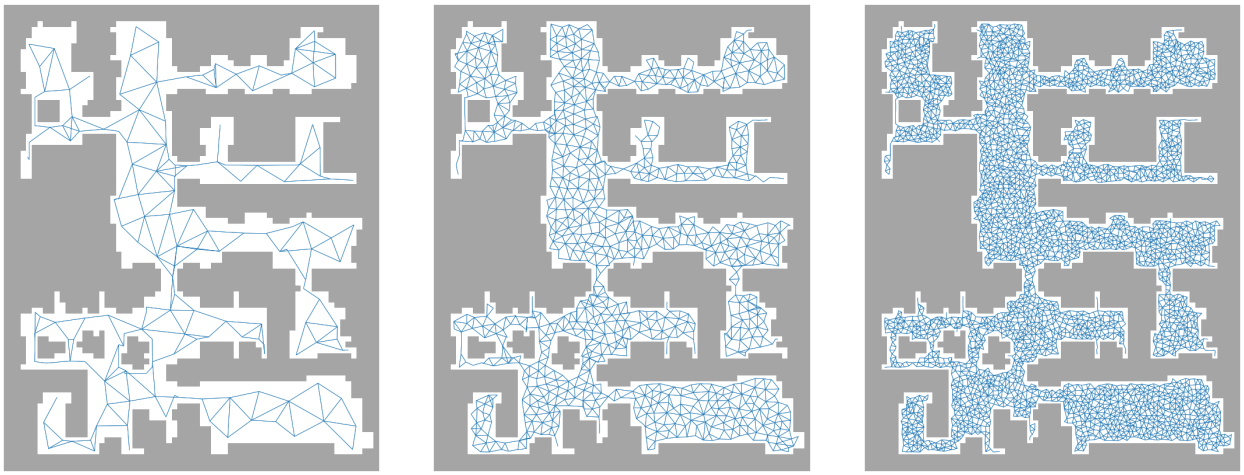


(a) The free space is divided into finer cells (individual coloured squares); fine cells are removed if too close to blocked cells; vertices (dots) are scattered to roughly match a given density; and each vertex claims the fine cells around it (coloured regions).



(b) Any two vertices that share a border are connected by as few straight edges as possible while ensuring no edge gets too close to a blocked cell.

Figure 13: A roadmap sampled from the MovingAI gridmap *Maze-32-32-2*.



(a) Density $\rho = 0.05$, with $|\mathcal{V}| = 223$ and $|\mathcal{E}| = 824$.

(b) Density $\rho = 0.5$, with $|\mathcal{V}| = 1\,607$ and $|\mathcal{E}| = 7\,894$.

(c) Density $\rho = 2.0$, with $|\mathcal{V}| = 4\,582$ and $|\mathcal{E}| = 24\,442$.

Figure 14: The MovingAI gridmap *Den312d*, with $|\mathcal{V}| = 2\,445$ and $|\mathcal{E}| = 8\,782$, sampled with various densities. The remaining parameters are $r = 0.5$, $\kappa = 1.5$, and $l_{\max} = 4\delta$.

From the target density ρ we derive a minimum separation $\delta = \sqrt{0.8/\rho}$ between sampled vertices. Placing vertices on a regular grid with spacing $\sqrt{1/\rho}$ would achieve exactly the target density ρ; however, rejection sampling under a minimum-separation constraint yields a lower density than a regular grid at the same separation, so we shrink the separation below $\sqrt{1/\rho}$ to partially compensate, using the empirically tuned factor 0.8. The region construction and the routes between vertices (below) are computed on a refined grid obtained by subdividing each map cell; every clearance test itself remains an exact geometric test of a point or segment against the original map cells. The subdivision factor is found by rounding $\sqrt{2}/\delta$ up to the nearest odd integer value, so that the diagonal of a refined cell is shorter than δ and no two sampled vertices can therefore land in the same refined cell, letting each vertex seed its own region. The refinement is additionally chosen odd so that every map cell's centre remains at the centre of a refined cell; with an even factor, refinement can shift the fine grid off the centreline of a corridor which may hold the only traversable line through it. Refining relative to the density ρ also removes any upper bound on how dense a roadmap can be requested, independent of the original map resolution.

A fine cell is admissible if its centre lies at least r from every blocked cell. We compute the set of admissible fine cells once and restrict it to its largest 4-connected component, since this is the largest region a disc of radius r can move through without leaving X_{free}; sampling outside this component would place vertices an agent of radius r could never reach from the rest of the map.

Vertices are placed by rejection sampling. A candidate refined cell is drawn at random from the admissible region, and a point is drawn uniformly at random within it. The point is accepted only if it lies at least δ from every previously accepted vertex; since only the cell's centre was certified admissible, the point must additionally, and independently, lie at least r from every blocked cell. If both conditions hold, the point is added to $\mathcal{V}$. Sampling stops

once 500 consecutive candidates in a row have been rejected, rather than continuing until the admissible region is fully saturated; the achieved density therefore falls somewhat short of ρ even after the 0.8 correction in the separation, which only partially compensates for this. The result is an even, non-clustered covering of X_{free}.

To connect the vertices, a breadth-first search over 4-adjacent admissible cells is run simultaneously from every vertex's cell, with each step claiming an unclaimed admissible cell; every admissible cell thereby ends up owned by the vertex reachable in fewest steps, a discretized Voronoi diagram of $\mathcal{V}$ over the admissible region under this graph distance. Two vertices are neighbours when their regions contain adjacent cells. For each neighbouring pair $\langle u, v\rangle$, we take the shortest route, by number of cells, among those passing through a boundary cell between their two regions, obtained by following the search trees back from that boundary cell to u and to v; this route is generally longer than the true shortest clearance-respecting path between u and v. The route, a dense sequence of fine-grid positions, is then reduced to a small number of straight segments by repeatedly jumping to the furthest later position still reachable by a segment that keeps clearance r throughout; this greedy reduction is not guaranteed to be minimal, since visibility along the route need not be monotone, for instance when rounding a corner into open space lets a later position be clear while an intermediate one is not. Where u and v have such a clear segment directly between them, the reduction collapses to a single edge $e = \langle u, v\rangle \in \mathcal{E}$; otherwise, the retained turning positions, each lying at the centre of a fine cell, become additional vertices of $\mathcal{V}$, and the connection becomes a short chain of edges bending around the obstacle, with a turning position shared by several routes becoming a single shared vertex. The reduction is guaranteed to succeed for every interior segment of the route, since consecutive fine-grid positions along it are adjacent and the segment between two admissible adjacent positions is itself clear; this argument does not cover the first and last segments, which run from a sampled vertex, at an arbitrary position within its cell, to that cell's centre. If the reduction fails on one of these two endpoint segments, the corresponding connection is dropped, and the roadmap's connectivity is instead checked after construction.

The graph produced so far is purely local and can force long detours between vertices that are close in a straight line but far apart along $\mathcal{G}$. We consider candidate pairs $u, v \in \mathcal{V}$ whose straight-line distance is at most $l_{\max}$, processed in increasing order of that distance. For each candidate, if its distance in the current graph, that is $\mathcal{G}$ together with all shortcuts accepted so far in the sweep, exceeds κ times the straight-line distance between u and v, and the straight segment between them is clear, we add a shortcut edge $e = \langle u, v\rangle$. Processing pairs in increasing order of straight-line distance means an accepted shortcut removes the need for longer shortcuts covering the same detour, while restricting candidates to those within $l_{\max}$ from the outset, rather than testing and discarding longer pairs, ensures every shortcut remains local regardless of how large a detour it removes.

Every edge $e \in \mathcal{E}$ is traversable in both directions. Clearance is guaranteed by construction: every vertex and edge keeps clearance r from every blocked cell, independent of the parameters. Connectivity of $\mathcal{G}$ is not guaranteed in the same sense; it is instead verified after construction, and a parameter setting that fails to yield a connected roadmap, for instance too low a density or too large a clearance relative to the map, is rejected and reported rather

than returned. The stretch factor κ and the target density ρ jointly control the size and quality trade-off: values of κ near 1 and higher ρ yield roadmaps whose distances closely approximate true free-space distances at the cost of more vertices and edges, while looser values yield sparser graphs on which paths detour more.

B.2 Scenario Generation

Given a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ and an agent model, we construct a scenario as follows. Because roadmap vertices are samples in continuous space, distinct vertices do not guarantee non-conflicting agents: two agents at different vertices may still occupy overlapping space, and two agents may be placed so that neither can move without conflicting with the other. Scenario generation must exclude both cases, and other similarly unsolvable configurations.

Two vertex sets are derived from $\mathcal{G}$ and the agent model prior to sampling, and reused across all scenarios on that graph. A vertex qualifies as a start if it is waitable (an agent may remain there indefinitely) and lies in the graph's largest strongly connected component (SCC). We require the SCC condition rather than only a path from each start to its own goal, because an agent must reach its goal not only from its start but from any vertex it may be displaced to while yielding to another agent; the SCC condition guarantees this, a per-pair path check does not. Although this excludes some vertices that admit a feasible start-goal path on their own, such a path offers no guarantee once an agent must yield to another: none of the checks introduced below examine intermediate positions, so an unrecoverable vertex would go undetected until execution.

A vertex qualifies as a goal only if removing it, together with everything its permanent occupation would put out of reach, leaves the remaining admissible vertices mutually reachable. This follows from treating a reached goal as held for the remainder of the scenario: an agent that lets others pass before occupying a cut vertex still severs the graph, only later. The condition is stricter than necessary, since it excludes a vertex regardless of whether any agent in a given instance would cross it, but it is the only safeguard against this failure mode.

For each admissible vertex, we also record the vertices and edges unavailable to a second agent while it is occupied, determined again by spatial overlap.

Goals are drawn first, by shuffling the admissible goal vertices and accepting each in turn unless it conflicts with an accepted goal, until the requested count is reached or the pool is exhausted (failure). Starts are drawn second, one per goal in draw order: the admissible start vertices (the full set, not only those also valid as goals) are shuffled once and scanned per agent, accepting the first vertex that does not conflict with an accepted start, and is not that agent's goal. A vertex rejected for one agent may be accepted for another, so start-goal pairing need not follow the shuffle order; exhausting the pool for any agent fails the attempt.

A scenario generation attempt failing any of the following conditions is discarded and redrawn, up to a redraw budget. Starts must be pairwise non-conflicting, as must goals. However, a start and goal of different agents may coincide, since the former agent departs before the latter arrives. If there exists some group of agents where every agent's outgoing edges are blocked by other agents in the group, then that scenario is discarded. The mirror

condition applies to incoming edges: a scenario is also discarded if some group of agents cannot make their final approach, with every incoming edge to each member's goal blocked by another member of the group already there.

All scenarios within a set (on the same graph) are kept at a common agent count. Scenario seeds within a set are drawn independently and distinctly, so scenarios within and across sets do not repeat.

References

[1] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. "Multi-agent pathfinding: Definitions, variants, and benchmarks". In: *Proceedings of the Twelfth International Symposium on Combinatorial Search (SoCS 2019)*. Vol. 10. 1. 2019, pp. 151–158.

[2] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. 3rd. Springer Publishing Company, Incorporated, 2008. ISBN: 0387789340.

[3] Yang-Kuei Lin and Chen-Hao Yen. "Genetic algorithm for solving the no-wait three-stage surgery scheduling problem". In: *Healthcare*. Vol. 11. 5. MDPI. 2023, p. 739.

[4] Yorai Shaoul, Itamar Mishani, Maxim Likhachev, and Jiaoyang Li. "Accelerating Search-Based Planning for Multi-Robot Manipulation by Leveraging Online-Generated Experiences". In: *Proceedings of the International Conference on Automated Planning and Scheduling* 34.1 (May 2024), pp. 523–531. DOI: 10.1609/icaps.v34i1.31513.

[5] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating hundreds of cooperative, autonomous vehicles in warehouses". In: *AI Magazine* 29.1 (2008), pp. 9–9.

[6] Hang Ma and Sven Koenig. "AI buzzwords explained: multi-agent path finding (MAPF)". In: *AI Matters* 3.3 (Oct. 2017), pp. 15–19. DOI: 10.1145/3137574.3137579.

[7] Kyle Brown, Oriana Peltzer, Martin A Sehr, Mac Schwager, and Mykel J Kochenderfer. "Optimal sequential task assignment and path finding for multi-agent robotic assembly planning". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 441–447.

[8] Sumanth Varambally, Jiaoyang Li, and Sven Koenig. "Which MAPF model works best for automated warehousing?" In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 15. 1. 2022, pp. 190–198.

[9] Erik Brorsson, Kristian Ceder, Ze Zhang, Sabino Francesco Roselli, Endre Erős, Martin Dahl, Beatrice Alenljung, Jessica Lindblom, Thanh Bui, Emmanuel Dean, Lennart Svensson, Kristofer Bengtsson, Per-Lage Götvall, and Knut Åkesson. *Infrastructure-based Autonomous Mobile Robots for Internal Logistics – Challenges and Future Perspectives*. 2025. arXiv: 2512.15215 [cs.RO].

[10] Jamie Snape, Stephen J. Guy, Ming C. Lin, Dinesh Manocha, and Jur Van den Berg. "Reciprocal Collision Avoidance and Multi-Agent Navigation for Video Games". In: *MAPF@AAAI*. 2012. URL: https://api.semanticscholar.org/CorpusID:10768014.

[11] Hang Ma, Jingxing Yang, Liron Cohen, T. K. Kumar, and Sven Koenig. "Feasibility Study: Moving Non-Homogeneous Teams in Congested Video Game Environments". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 13.1 (Oct. 2017), pp. 270–272. DOI: 10.1609/aiide.v13i1.12919.

[12] Daniel Harabor, Ryan Hechenberger, and Thomas Jahn. "Benchmarks for pathfinding search: Iron harvest". In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 15. 1. 2022, pp. 218–222.

[13] Jiaoyang Li, The Anh Hoang, Eugene Lin, Hai L. Vu, and Sven Koenig. "Intersection Coordination with Priority-Based Search for Autonomous Vehicles". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 37.10 (June 2023), pp. 11578–11585. DOI: 10.1609/aaai.v37i10.26368.

[14] Christofel Rio Goenawan. *ASTM : Autonomous Smart Traffic Management System Using Artificial Intelligence CNN and LSTM*. 2025. arXiv: 2410.10929 [cs.LG].

[15] Robert Morris, Corina S Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, TK Satish Kumar, and Sven Koenig. "Planning, Scheduling and Monitoring for Airport Surface Operations." In: *AAAI Workshop: Planning for Hybrid Systems*. 2016, pp. 608–614.

[16] Jiaoyang Li, Mimi Gong, Zi Liang, Weizi Liu, Zhongyi Tong, Liangchen Yi, Robert Morris, Corina Pasearanu, and Sven Koenig. "Departure Scheduling and Taxiway Path Planning under Uncertainty". In: *AIAA Aviation 2019 Forum*. DOI: 10.2514/6.2019-2930.

[17] Alexander Krawciw, Luka Antonyshyn, Sven Lilge, Nicolas Olmedo, Faizan Rehmatullah, Maxime Desjardins-Goulet, Pascal Toupin, and Timothy D. Barfoot. *Sharing the Load: Autonomous Multi-Rover Cargo Transport*. 2026. arXiv: 2510.18766 [cs.RO].

[18] Liron Cohen, Tansel Uras, T. K. Kumar, and Sven Koenig. "Optimal and Bounded-Suboptimal Multi-Agent Motion Planning". In: *Proceedings of the International Symposium on Combinatorial Search* 10.1 (Sept. 2021), pp. 44–51. DOI: 10.1609/socs.v10i1.18501. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18501.

[19] Noy Gabay, Jonathan Morag, Ariel Felner, and Roni Stern. "The Reachability Objective in Multi-Agent Path Finding". In: *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems*. AAMAS '26. Paphos, Cyprus: International Foundation for Autonomous Agents and Multiagent Systems, 2026, pp. 1165–1173. DOI: 10.65109/OOLP5568.

[20] Anton Andreychuk, Konstantin Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern. "Multi-agent pathfinding with continuous time". In: *Artificial Intelligence* 305 (2022), p. 103662. DOI: 10.1016/j.artint.2022.103662.

[21] Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, T. K. Satish Kumar, and Sven Koenig. "Multi-Agent Path Finding for Large Agents". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 7627–7634. DOI: 10.1609/aaai.v33i01.33017627.

[22] Pavel Surynek. "Multi-Goal Multi-Agent Path Finding via Decoupled and Integrated Goal Vertex Ordering". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.14 (May 2021), pp. 12409–12417. DOI: 10.1609/aaai.v35i14.17472.

[23] Sabino Francesco Roselli, Kristofer Bengtsson, and Knut Åkesson. "SMT Solvers for Job-Shop Scheduling Problems: Models Comparison and Performance Evaluation". In: *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. 2018, pp. 547–552. DOI: 10.1109/COASE.2018.8560344.

[24] Ishan Bansal, Oktay Günlük, and Richard Shapley. *Parallel Token Swapping for Qubit Routing*. 2025. arXiv: 2411.18581 [cs.DS]. URL: https://arxiv.org/abs/2411.18581.

[25] Leo D'Amato, Paola Pellegrini, and Vito Trianni. *Decentralised multi-agent coordination for real-time railway traffic management*. 2025. arXiv: 2502.08324 [cs.MA].

[26] Antoine Deza, Chris Dickson, Tamás Terlaky, Anthony Vannelli, and Hu Zhang. "Global Routing in VLSI Design: Algorithms, Theory, and Computational Practice". In: *JCMCC. The Journal of Combinatorial Mathematics and Combinatorial Computing* 80 (Jan. 2012).

[27] Alvin Combrink, Sabino Francesco Roselli, and Martin Fabian. *Optimal Multi-agent Path Finding in Continuous Time*. 2025. arXiv: 2508.16410 [cs.MA].

[28] Kazumi Kasaura, Mai Nishimura, and Ryo Yonetani. "Prioritized Safe Interval Path Planning for Multi-Agent Pathfinding With Continuous Time on 2D Roadmaps". In: *IEEE Robotics and Automation Letters* 7.4 (2022), pp. 10494–10501. DOI: 10.1109/LRA.2022.3187265.

[29] Nathan R. Sturtevant. "Benchmarks for Grid-Based Pathfinding". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.2 (2012), pp. 144–148. DOI: 10.1109/TCIAIG.2012.2197681.

[30] Francesco Popolizio, Martina Vinetti, Alvin Combrink, Sabino Francesco Roselli, Maria Pia Fanti, and Martin Fabian. "Online Conflict-Free Scheduling of Fleets of Autonomous Mobile Robots". In: *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2024, pp. 3063–3068.

[31] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. *Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks*. 2017. arXiv: 1705.10868 [cs.AI].

[32] Jonathan Morag, Noy Gabay, Daniel koyfman, and Roni Stern. *Transient Multi-Agent Path Finding for Lifelong Navigation in Dense Environments*. 2024. arXiv: 2412.04256 [cs.MA].

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

[34] Andy Li, Zhe Chen, Mor Vered, and Danial Harabor. *Revisiting Conflict Based Search with Continuous-Time*. 2025. arXiv: 2501.07744 [cs.MA].

[35] Mike Phillips and Maxim Likhachev. "SIPP: Safe interval path planning for dynamic environments". In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 5628–5635. DOI: 10.1109/ICRA.2011.5980306.

[36] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[37] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. "Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem". In: *Proceedings of the international symposium on combinatorial Search*. Vol. 5. 1. 2014, pp. 19–27.

[38] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernandez, Ariel Felner, T. K. Satish Kumar, and Sven Koenig. "Anytime Focal Search with Applications". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 1434–1441. DOI: 10.24963/ijcai.2018/199.

[39] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. "Anytime multi-agent path finding via large neighborhood search". In: *International joint conference on artificial intelligence 2021*. Association for the Advancement of Artificial Intelligence (AAAI). 2021, pp. 4127–4135.

[40] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. "MAPF-LNS2: Fast repairing for multi-agent path finding via large neighborhood search". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 9. 2022, pp. 10256–10265.

[41] Taoan Huang, Jiaoyang Li, Sven Koenig, and Bistra Dilkina. "Anytime multi-agent path finding via machine learning-guided large neighborhood search". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 9. 2022, pp. 9368–9376.

[42] Thomy Phan, Taoan Huang, Bistra Dilkina, and Sven Koenig. "Adaptive anytime multi-agent path finding using bandit-based large neighborhood search". In: *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on*

Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence. AAAI'24/IAAI'24/EAAI'24. AAAI Press, 2024. DOI: 10.1609/aaai.v38i16.29701.

[43] Keisuke Okumura. "Improving LaCAM for scalable eventually optimal multi-agent pathfinding". In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*. IJCAI '23. Macao, P.R.China, 2023. DOI: 10.24963/ijcai.2023/28.

[44] Alvin Combrink, Sabino Francesco Roselli, and Martin Fabian. "Prioritized Planning for Continuous-time Lifelong Multi-agent Pathfinding". In: *2025 11th International Conference on Control, Decision and Information Technologies (CoDIT)*. Vol. 1. 2025, pp. 1454–1459. DOI: 10.1109/CoDIT66093.2025.11321711.

[45] Keisuke Okumura, Guang Yang, Zhan Gao, Heedo Woo, and Amanda Prorok. "Concrete multi-agent path planning enabling kinodynamically aggressive maneuvers". In: *npj Robotics* 4.1 (Mar. 2026), p. 20. DOI: 10.1038/s44182-026-00083-2.

[46] Akmaral Moldagalieva, Joaquim Ortiz-Haro, Marc Toussaint, and Wolfgang Hönig. "db-CBS: Discontinuity-bounded conflict-based search for multi-robot kinodynamic motion planning". In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2024, pp. 14569–14575.

[47] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig. "Lifelong multi-agent path finding in large-scale warehouses". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 13. 2021, pp. 11272–11281.

[48] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, Oded Betzalel, David Tolpin, and Eyal Shimony. "ICBS: The Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding". In: *Proceedings of the International Symposium on Combinatorial Search* 6.1 (June 2015), pp. 223–225. DOI: 10.1609/socs.v6i1.18343.

[49] Farzeen Zehra, Maha Javed, Darakhshan Khan, and Maria Pasha. "Comparative Analysis of C++ and Python in Terms of Memory and Time". In: *Preprints* (Dec. 2020). DOI: 10.20944/preprints202012.0516.v1.

[50] Anton Andreychuk, Konstantin Yakovlev, Eli Boyarski, and Roni Stern. "Improving continuous-time conflict based search". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 13. 2021, pp. 11220–11227.

[51] Eli Boyarski, Ariel Felner, Guni Sharon, and Roni Stern. "Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 25. 2015, pp. 47–51.

[52] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 2022. ISBN: 9780262046305.

[53] Thayne T Walker and Nathan R Sturtevant. "Collision detection for agents in multi-agent pathfinding". In: *arXiv preprint arXiv:1908.09707* (2019).

[54] Alvin Combrink. "Advances in Multi-Agent Path Finding: Scalable lifelong planning and optimal continuous-time solutions". Licentiate thesis. Gothenburg, Sweden: Chalmers University of Technology, 2025. URL: https://research.chalmers.se/publication/549458.