Introducing Delays in Multi-Agent Path Finding

Justin Kottinger[1], **Tzvika Geft**[2], **Shaull Almagor**[3], **Oren Salzman**[3], **Morteza Lahijanian**[1]

[1] Department of Aerospace Engineering, University of Colorado Boulder, USA
[2] The Blavatnik School of Computer Science, Tel Aviv University, Israel
[3] The Henry and Marilyn Taub Faulty of Computer Science, Technion, Israel
justin.kottinger@colorado.edu, zvigreg@mail.tau.ac.il, shaull@technion.ac.il, osalzman@cs.technion.ac.il, morteza.lahijanian@colorado.edu

Abstract

We consider a Multi-Agent Path Finding (MAPF) setting where agents have been assigned a plan, but during its execution some agents are delayed. Instead of replanning from scratch when such a delay occurs, we propose *delay introduction*, whereby we delay some additional agents so that the remainder of the plan can be executed safely. We show that finding the minimum number of additional delays is APX-hard, i.e., it is NP-hard to find a $(1+\varepsilon)$-approximation for some $\varepsilon > 0$. However, in practice we can find *optimal* delay-introductions using Conflict-Based Search for very large numbers of agents, and both planning time and the resulting length of the plan are comparable, and sometimes outperform the state-of-the-art heuristics for replanning.

1 Introduction

Multi-Agent Path Finding (MAPF) is a problem in Artificial Intelligence (AI) that asks to find non-colliding paths for a group of agents moving on a graph (Stern et al. 2019; Salzman and Stern 2020). Applications vary from autonomous warehouse management (Wurman, D'Andrea, and Mountz 2008) and factory pipe routing (Belov et al. 2020) to rail planning (Li et al. 2021) and swarm robotics (Ramaithitima et al. 2016). Although MAPF is known to be generally an intractable problem (Yu 2016; Banfi, Basilico, and Amigoni 2017; Nebel 2020; Geft 2023), recent algorithms can scale to thousands of agents, e.g., (Li et al. 2022; Li, Ruml, and Koenig 2021; Okumura 2023). A limiting aspect of these algorithms is the simplifying assumption that, at deployment, agents can synchronously execute a plan. In reality, however, it is common for agents to fall out of sync, e.g., due to delays or model uncertainty. Such incidents may cause the plan to no longer be valid (non-colliding), in which case we must either compute a new plan or repair the old plan quickly. This is challenging since replanning faces the same difficulties as the original MAPF problem, and plan repair is shown to be as difficult as plan generation itself (Nebel and Koehler 1995).

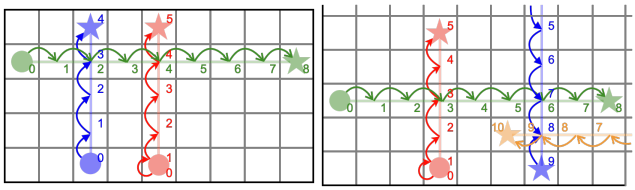
In this work, we propose a simple but effective approach to plan repair that inherits a lot of the benefits of the original plan and can scale to a large number of agents. Our key idea is to use the topology of the original plan and resolve conflicts by allowing agents to stay in place. That is, we repair the plan by introducing *delays* – i.e., requiring some agents to remain in a certain location for a certain amount of time instead of advancing according to the prescribed path. The intuition is that resources are put into generating and validating the original plan. It is hence desirable to maintain at least some of its properties. For example, in safety-critical or ethical situations (e.g., transportation of hazardous materials, air traffic control) plans often need to be approved by a human controller, and therefore, replanning requires the human to accept a new plan, which in turn requires trust. By using the same paths, we gain several benefits, including inheriting the visual explainability of the original plan (i.e., the paths visually remain the same), reducing the search space to a smaller graph than the original one, and existence of a solution when delays are not constrained along the paths.

Specifically, we consider the following setting: we work over an environment modeled as a directed graph, where agents wish to move from their starting vertices to their goals. We further assume that we already have a plan P that drives each agent from start to goal. However, P may contain collisions. Motivationally, we think of P as obtained from a non-colliding plan by having some agents delay in place, resulting in possible collisions. We allow to repair P to a new plan P' by having some agents delay at certain vertices. Furthermore, we want P' to be such that the overall number of delays is minimal.

We also draw motivation from a non-optimization problem formulation closely related to ours by (Abrahamsen et al. 2023). They focus on adding *any* set of delays to P to ensure P' is non-colliding, for which they provide positive and negative complexity results. Notably, they mention studying optimization variants, akin to our problem, as an open extension.

Introducing a minimal set of delays in this setting gives rise to some intricate behaviors, as demonstrated in the following examples. In particular, the choice of when to delay an agent and for how many steps is crucial and nontrivial.

Example 1 (Postponing delays)**.** *Consider the case in Fig. 1a, which shows a plan for three agents. Note that the original plan, which takes the agents straight to their respective goals, is non-colliding. Now, imagine Red is delayed at time 0, resulting in the detection of an upcoming collision*



(a) Example 1 (b) Example 2

Figure 1: Setting of (a) Example 1 and (b) Example 2. Straight lines depict the original plan, and curved edges represent the plan when Red agent is delayed at time 0. (a) It is better to have Green agent delay at time 3. (b) it is better to have Green agent delay for two timesteps.

with Green at time 4. However, delaying Green upon detection (step 1) will cause a collision with Blue at time 3. Instead, it is preferable to postpone the delay of Green to time 3, and let Blue pass through.

Example 2 (Long delays)**.** *Consider the setting in Fig. 1b. Green is again about to collide with Red. However, delaying Green for a single timestep will cause another collision with Blue at time 7. Blue, in turn, passes a train of 100 agents (depicted in orange) just before it crosses at time 8. Thus, if we delay Blue for even a single timestep, this would require either delaying it for another 100 times, or delaying the train of 100 agents. Thus, the optimal solution is to delay Green for 2 timesteps or delay Red for another timestep.*

These examples allude to our two main contributions:

- We show that the problem of *avoiding collisions by introducing delays* (ACID) is NP-complete, and in fact even APX-hard.
- We propose an algorithmic approach to ACID by formulating it as a small instance of MAPF, with a low branching degree, for which existing algorithms can be readily used.

We experimentally evaluate our algorithmic approach through several standard benchmark problems. Specifically, the results show that for a small number of unexpected delays (one delay, in our experiments), the simplicity of the small MAPF problem allows to use optimal algorithms such as Conflict-based Search (CBS) (Sharon et al. 2015) to compute a plan with minimum number of delays fast and scale up to 1000 agents. For more delays (with a large number of agents), CBS does not scale, but Heuristic-based MAPF algorithms perform well.

Related Work: The execution of MAPF plans may present unexpected delays that hinder the system's ability to follow the prescribed plan. Atzmon et al. (2020b) proposed a method to account for these uncertainties by computing k-robust plans (for some user-provided k) that guarantee safe execution even in the presence of up to k delays. Atzmon et al. (2020a) also extended the idea to the probabilistic setting, guaranteeing success with probability at least p, when given a (user defined) probabilistic model of delays. However, these robust planning techniques suffer from being computationally expensive, overly-conservative, and are robust only in expectation (probabilistically).

Plan repair was considered almost three decades ago by Nebel and Koehler (1995), where the authors show that repairing plans is potentially harder than planning from scratch. Interestingly, they show that a bottleneck of repairing plans is choosing the plan that we repair *to*. This challenge can be avoided in specific cases. For example, Tonola et al. (2023) repaires single-agent paths in the presence of dynamic obstacles by connecting pre-computed path-segments together to repair an invalid motion plan. Their work does not consider coordination with additional agents. Komenda and Novák (2011) introduce the generalized problem of multi-agent plan repair and proposed three sub-optimal algorithms. Komenda, Novák, and Pěchouček (2014) present an optimal way to solve the problem but their results only consider up to 10 agents.

Hönig et al. (2016) solves problems associated with delays via a post-processing technique that transforms a MAPF plan into a plan-execution schedule (MAPF-POST). Their setting crucially relies on agents' kinematic abilities. Specifically, the ability to use rational constant speeds allows reducing the problem to a linear program (LP) using simple temporal networks. That work is extended by Berndt et al. (2020) by formalizing the problem as a MILP and solving it sub-optimally online. Similarly, Ma, Kumar, and Koenig (2017) present a probabilistic approach to resolving delays in decentralized systems by employing an approximate expectation-minimization approach. Note that casting our work as an instance of MAPF-POST allows for a simple solution: when an agent is delayed, simply slow down agents that may collide with it. This is a particularly simple case of MAPF-POST. Therefore, our main interest is in the combinatorial aspect of this problem in the discrete case.

There are recent works that are similar to our work. Barták, Švancara, and Vlk (2018) solve traditional MAPF via a scheduling-based approach. Specifically, they use a layered graph to represent delays. If only a single layer is used, no delays are allowed. Secondly, Svancara et al. (2023) present a preliminary work where the goal is to solve MAPF by only introducing delays onto predefined paths. They create an abstract graph where the nodes represent agents and the edges represent choices to wait. Lastly, Abrahamsen et al. (2023) studies the most similar version of our problem. There, however, they study the *feasibility* problem of finding a set of delays that allows agents to reach their targets without colliding along a set of simple paths. They provide an in-depth computational complexity investigation in lieu of empirical results. Specifically, they present a sharp tractability boundary based on a key parameter called *vertex multiplicity (VM)*, defined as the maximum number of paths passing through the same vertex. They present a variant of the problem that is NP-complete for VM $= 3$ and efficiently solvable for VM ≤ 2.

This work differs from all of these works in multiple aspects. First, in contrast to Hönig et al. (2016) and Berndt et al. (2020), we consider a combinatorial problem, which does not admit a reduction to LP. Moreover, our approach allows for constraints that prohibit certain delays. Secondly,

we deviate from Ma, Kumar, and Koenig (2017) by (i) considering a worst-case scenario rather than a probabilistic one and (ii) provide an optimal approach for the setting where the system has a centralized controller rather than proposing execution policies for decentralized systems. The key differences between the work of Svancara et al. (2023) and our work are that (i) we use traditional MAPF algorithms rather than optimization techniques and (ii) our graph is comparatively very small, allowing us to solve much more complex problem instances in a much shorter amount of time.

Additionally, our problem setting is very similar to that of Svancara et al. (2023) but our solving techniques are quite different. Specifically, we avoid the feasibility problem by assuming the predefined paths came from an originally safe plan and solve the problem optimally using out-of-the-box MAPF solvers. And finally, in contrast to Abrahamsen et al. (2023), we study the optimality of plans, i.e., we aim to minimize the number of delays that are introduced, and not merely check for the feasibility of some number of delay introductions. Indeed, in practical settings (as well as in our experimental setting), collisions occur due to agents breaking down. In such cases, it is trivial to repair the plan by introducing delays in all the remaining agents, thus halting execution until the breakdown is fixed (c.f., Remark 3).

In contrast to Atzmon et al. (2020a,b), our method does not pre-compute a robust plan (and hence has no inherent added computational cost), but rather fixes (repairs) an existing plan if a delay occurs. We show theoretically that our setting still incurs the computational hardness presented by Nebel and Koehler (1995) (c.f., Theorem 1). However, we mitigate practical computation by keeping the set of repaired plans relatively small due to strictly limiting the agents to only using delays. Thus, modern MAPF algorithms allow us to practically repair plans.

2 Problem Statement

We start by formulating the general MAPF setting. Consider $n \in \mathbb{N}$ agents, acting in an environment represented by a directed graph $G = \langle V, E \rangle$ where each agent $i \in \{1, \ldots, n\}$ has a source $s_i \in V$ and a goal $g_i \in V$. A *path* in G is a sequence of vertices $\pi = v_1 v_2 \ldots v_m$ such that $(v_k, v_{k+1}) \in E$ for all $1 \leq k < m$. We assume that the vertices of G contain self loops (i.e., for all $v \in V$, we have $(v, v) \in E$), so agents can be delayed. We remark that our results still hold if one allows self loops only on some of the vertices.

Given paths $\pi_1 = v_1 v_2 \ldots v_k$ and $\pi_2 = u_1 u_2 \ldots u_k$ in G, we say that π_1 and π_2 are *non-colliding* if the following conditions are satisfied for all $1 \leq j < k$:

(i) $v_j \neq u_j$ (i.e., no vertex collisions),

(ii) $(v_j, v_{j+1}) \neq (u_{j+1}, u_j)$ (i.e., no edge collisions).

If π_1 and π_2 are of different lengths, we assume the agent with the shorter path remains in the target state for collision-checking purposes.[1]

[1]Changing this to have the agents "disappear" (which is a commonly-used assumption) at the target location does not impact our results in any way.

Given n agents on a graph G and two lists $s_1, \ldots, s_n$ and $g_1, \ldots, g_n$ of source and goal vertices, respectively, a *plan* $P = \{\pi_1, \ldots, \pi_n\}$ is a set of paths such that π_i drives agent i from s_i to g_i for every $i \in \{1, \ldots, n\}$. A plan is called *non-colliding* if π_i and π_j are non-colliding for all $i \neq j \in \{1, \ldots, n\}$. The *length* of the plan, denoted by $\boldsymbol{\ell}(P)$, is the maximal length of a path in P. The *sum-of-costs* (SOC) of P, denoted by $\|P\|$, is the sum of lengths of all the paths in P. The classical *Multi-Agent Path Finding (MAPF)* problem is to find a non-colliding plan[2] P in G with the given source and target vertices.

We now turn to formalize delays and delay-introduction. Consider a path $\pi = v_1 \ldots v_m$ and some $d \in \mathbb{N}$. We say that a path π' is a *d-delay* of π if $\pi' = v_1 v_1^{k_1} v_2 v_2^{k_2} \ldots v_m v_m^{k_m}$, where $v_i^{k_i}$ means repeating v_i for an additional $k_i \geq 0$ times. That is, π' repeats some of the vertices of π so that the total amount of repetitions is d. Note that if $k_i = 0$ for all $i \in \{1, \ldots, m\}$ then $\pi' = \pi$, i.e., π' is a 0-delay path of π. Also, π might already have vertex repetitions (e.g., it could be that $v_1 = v_2$). Thus, we only allow adding repetitions, not removing them.

Problem 1 (Avoiding Collisions by Introducing Delays (**ACID**))**.** *Given a graph $G = \langle V, E \rangle$, a plan $P = \{\pi_1, \ldots, \pi_n\}$ and a budget $D \in \mathbb{N}$, decide whether there exist paths $\pi'_1, \ldots, \pi'_n$ where π'_i is a d_i-delay of π_i for each i, with $\sum_{i=1}^{n} d_i \leq D$ and $P' = \{\pi'_1, \ldots, \pi'_n\}$ is non-colliding.*

Note that ACID is stated as a decision problem, but for algorithmic purposes we consider its optimization variant, in which we want to find a non-colliding plan that minimizes D, i.e., the added length of the plan.

3 Computational Complexity of ACID

We start our investigation of ACID by establishing its computational complexity. Specifically, we show that solvable instances can be solved using a quadratic delay.

Lemma 1. *Consider an ACID instance with plan $P = \{\pi_1, \ldots, \pi_n\}$ and budget D. If the instance is solvable, then it is also solvable with budget $D' = (n-1) \cdot \|P\|$.*

Proof. Intuitively, the maximal number of delays we may need to introduce is such that we "spread" P so that only a single agent moves at each timestep, and the remaining $n-1$ agents are delayed.

Formally, consider a plan P' that is a solution to Problem 1. If there is a time where all agents are delayed simultaneously in P', this delay can be safely removed. Thus, at each step, at least one path advances, so to obtain P' from P, we introduce, for each agent and each step of P, at most $n-1$ delays. Since P comprises a total of $\|P\|$ agent steps, the total delays introduced are at most $(n-1) \cdot \|P\|$. □

Remark 1 (Encoding of the budget D)**.** *ACID can be considered with D encoded either in binary or in unary. We note that this does not affect the computational complexity, since by Lemma 1, we can assume w.l.o.g. that D is polynomially bounded in the size of P (namely in n and $\|P\|$).*

[2]Typically, the plan is required to be optimal with respect to some cost function, e.g., length or sum-of-costs.

We are now ready to establish the complexity of ACID. The complete proof can be found in extended version (Kottinger et al. 2023). For background on hardness of approximation see (Ausiello et al. 1999).

Theorem 1. *ACID is* NP*-complete, and even* APX*-hard.*

Proof sketch. Membership in NP follows immediately from Lemma 1—simply guess a set of (polynomially bounded) delays and check that the resulting plan is non-colliding.

We turn to show NP-hardness by showing a reduction from the Minimum Sum Coloring (MSC) problem, defined as follows. In an MSC instance we are given an undirected graph $G = \langle V, E \rangle$ and a threshold C (encoded in unary), and the goal is to decide whether there exists a coloring $\chi : V \to \mathbb{N}$ such that $\sum_{v \in V} \chi(v) \leq C$. That is, we need to color the vertices of G with natural numbers (including 0), such that every two vertices that share an edge are assigned different colors, and the sum of colors is at most C. MSC was shown to be NP-complete in Kubicka and Schwenk (1989) and is APX-hard (DeHaan and Friggstad 2023).

We start with an intuitive overview of the reduction and depict the reduction in Figs. 2 and 3. Given a graph $G = \langle V, E \rangle$ and $C \in \mathbb{N}$, denote $V = \{1, \ldots, n\}$ and $E = \{e_1, \ldots, e_m\}$. Our ACID instance consists of n agents travelling along a concatenation of $C + 1$ identical *blocks*, constructed as follows. For agent $i \in \{1, \ldots n\}$ we build a path of length m that for the most part is disjoint from all other paths. However, for each edge $e_r \in E$, if $e_r = \{i, j\}$ for some $j \in V$, then node r in the path is shared by the paths of agents i and j. For example, edge e_2 in Fig. 2 appears on both the green and red paths in the blocks in Fig. 3. Each agent starts its traversal from a distinct initial node, and the blocks are concatenated in the natural way. Note that since C is in unary, the reduction is polynomial.

We claim that G can be colored with sum at most C if and only if the resulting ACID instance has a solution with budget C. Intuitively, without introducing delays, for every edge $e_r = \{i, j\} \in E$ agents i and j collide in each block on the node corresponding to e_r. However, if i and j are delayed by different amounts before reaching node e_r, then they do not collide.

Thus, one direction of the proof is easy: if G has a coloring χ of sum at most C, then by delaying agent $i \in V$ for $\chi(i)$ steps (hence keeping within the budget C), the agents do not collide in any block.

The converse direction is more involved. Assume the ACID instance has a solution with budget at most C. Since there are $C + 1$ blocks, it follows that there is at least one block where the agents are not delayed. In the technical appendix we show that we can therefore assume all following blocks contain no delays as well, and moreover – that we can aggregate all the delays to the initial node of each agent. These delays induce a coloring of G of sum at most C, whereby each agent is colored with its number of delays. Since the agents do not collide, this coloring is legal. The hardness of approximation follows from the fact that our reduction preserves the cost of a solution. □

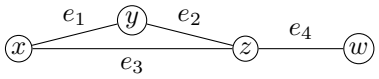


Figure 2: An input graph for the reduction with $C = 3$. Observe that the graph can be colored with sum 3, by $\chi(x) = \chi(w) = 0$, $\chi(y) = 1$ and $\chi(z) = 2$. Note that for clarity, we use x, y, z and w and not $1, \ldots, 4$ (as is done in the reduction) to name the vertices.

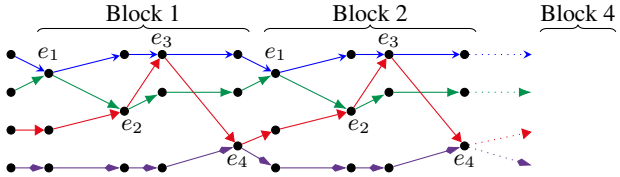


Figure 3: Reduction output. Each agent is represented by a path (e.g., x is the blue path, also distinguished by arrow types). The complete output has $C + 1 = 4$ blocks.

Remark 2 (ACID variants)**.** *The hardness in Theorem 1 holds already for the most "relaxed" version of ACID. However, the upper bound still holds with various restrictions on the delays, such as only allowing a certain number (possibly zero) of delays per agent, or per node. Thus, ACID remains* NP*-complete even if the agents are not allowed to delay in some nodes, or if the budget is specified for each agent, or most generally – if each agent-vertex pair has a budget.*

4 MAPF Formulation of ACID

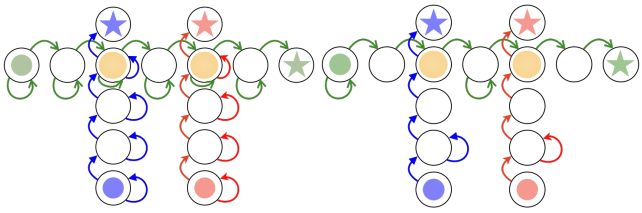
We turn our attention to developing an algorithmic approach for solving ACID. To this end, we reduce ACID to a version of MAPF, and utilize existing solutions for the latter. Crucially, we show that the specific MAPF instances resulting from our reduction have certain favorable properties which render them amenable to scalable optimal solutions.

Before detailing our approach, we present a small modification to the MAPF problem, whereby we allow a different set of edges for each agent. An instance of *MAPF with agent-specific edges* (dubbed *Agent-Edge MAPF*) is a set of vertices V and sets $E_1, \ldots, E_n \subseteq V \times V$ of edges, as well as start and goal vertices for each of the n agents. The remaining definitions are identical to MAPF, with the exception that a path for agent i must use only edges from E_i.

From an algorithmic perspective, solving Agent-Edge MAPF is similar to solving MAPF, in the following sense.

Observation 1. *An algorithm $\mathcal{A}$ for MAPF whose queries to the graph are only stated in the form "what are the edges from vertex v for agent i?" can solve Agent-Edge MAPF and preserve the same optimality/bounded suboptimality/anytime properties of the original algorithm $\mathcal{A}$.*

Note that (i) if $\mathcal{A}$ makes only such queries, it cannot distinguish between a (regular) graph and the agent-specific edge setting and that (ii) most common MAPF solvers such as all A^*-based solvers like CBS (Sharon et al. 2015) and PBS (Ma et al. 2019) satisfy the condition of Observation 1.



(a) Constrained Graph. (b) Improved Constrained Graph.

Figure 4: (a) A Constrained Graph and (b) an Improved Constrained Graph for an MAPF plan with three agents.

4.1 Constrained Graph

Our reduction of ACID to Agent-Edge MAPF is as follows. Consider an ACID instance with graph $G = \langle V, E \rangle$ and a plan $P = \{\pi_1, \ldots, \pi_n\}$ (ignore the budget for now). We construct an Agent-Edge MAPF instance with the vertices $V \times \{1, \ldots, \ell(P)\}$ (i.e., a copy of V for each step in P, up to the longest path), and the edges are defined by the paths in P, as well as self-loops. That is, let $\pi_i = v_1^i, \ldots, v_k^i$. Then we define $E_i = \{((v_j^i, j), (v_{j+1}^i, j+1)) \mid 1 \leq j < k\} \cup \{((v_j^i, j), (v_j^i, j)) \mid 1 \leq j < k\}$. We set the start and goal vertices for agent i as $(v_1^i, 1)$ and (v_k^i, k), respectively.

We refer to the multiple-edgeset graph obtained above as the *Constrained Graph (CG)* (see Fig. 4a). Notice that the Agent-Edge graph does not allow agents to deviate from their original paths. That is, the green agent located at either highlighted vertex must either delay or transition to the next immediate right vertex. Similarly, the red and blue agents located at the same vertices must either delay or move upward.

Note that the out-degree (number of outgoing edges) of each vertex in a CG is at most two. This makes a CG an "easy" candidate for planning since the branching factor used by search algorithms is often (though not necessarily) small, implying low running times.

We later show that an ACID instance has a solution with delay D iff the Agent-Edge MAPF instance on the CG has a solution of length $\|P\| + D$. But first, we present an optimization over the CG which further eliminates redundancy.

4.2 Improved Constrained Graph

Observe that the CG has self loops on all vertices. This, however, may be redundant. For example, the collision from Fig. 1a can be resolved by having the red agent delay at *any* of the vertices along its path prior to the conflict. Thus, it suffices to have a self loop on only one of the vertices along its path prior to the conflict (see Fig. 4b).

Given a CG, we construct an *Improved* CG (ICG) as follows. For each path $\pi_i = v_1^i, \ldots, v_k^i$ in the plan, let $I_i \subseteq \{1, \ldots, k\}$ be the set of indices j for which v_j^i occurs also in some other path π_l. We refer to I_i as the *intersecting indices* of agent i. The ICG is an Agent-Edge MAPF instance obtained from a CG so that between every two intersecting indices we keep exactly one vertex with a self-loop. Formally, for every agent i and $s < t \in I_i \cup \{0\}$ we retain one self loop in the vertices $v_{s+1}^i \ldots v_t^i$.

Note that computing ICG from CG is easy—we simply find the intersecting vertices in quadratic time, and scan the intervals between them.

As we prove in the extended version (Kottinger et al. 2023), using ICG instead of CG is sound and complete, in the following sense.

Theorem 2. *Given an instance of ACID with plan P, let the CG and ICG be as above. Then, the following are equivalent.*

1. *The ACID instance has a solution with budget D.*
2. *The Agent-Edge MAPF of CG has a solution P' with $\|P'\| \leq \|P\| + D$.*
3. *The Agent-Edge MAPF of ICG has a solution P' with $\|P'\| \leq \|P\| + D$.*

Following Theorem 2, we can solve an ACID instance by applying *any* MAPF algorithm that satisfies Observation 1 to the corresponding CG or ICG.

5 Experimental Evaluation

We now provide an empirical evaluation comparing different approaches for plan repair. Our approach is as follows. We take an existing plan P for some MAPF instance, and we introduce delays to it such that the plan becomes colliding. We then consider three approaches to repair the plan:

- In the first approach, we simply try to find a new plan from the agents' current locations to their original goals on the *original graph (OG)* using a MAPF solver.
- In the second (resp. third) approach, we implement our reduction to the *Constrained Graph (CG)* from Section 4.1 (resp. *Improved Constrained Graph (ICG)* from Section 4.2), and use a MAPF solver.

The approaches above are further split according to which MAPF solver we use, as we detail in Section 5.1. An important point is that we test on CBS, which typically does not scale very well without suboptimal heuristics.

We remark that a-priori, the comparison with replanning on OG is not "fair", in that OG allows for shorter plans that are unavailable when only delays can be introduced. Nonetheless, we show that when only a single delay is introduced, our approach is competitive also in this sense — we almost never output longer plans than OG (especially as the number of agents increases) with the exception of a few outliers. Thus, our approach has both the advantage of keeping to the original plan, as discussed in Section 1, as well as in plan length.

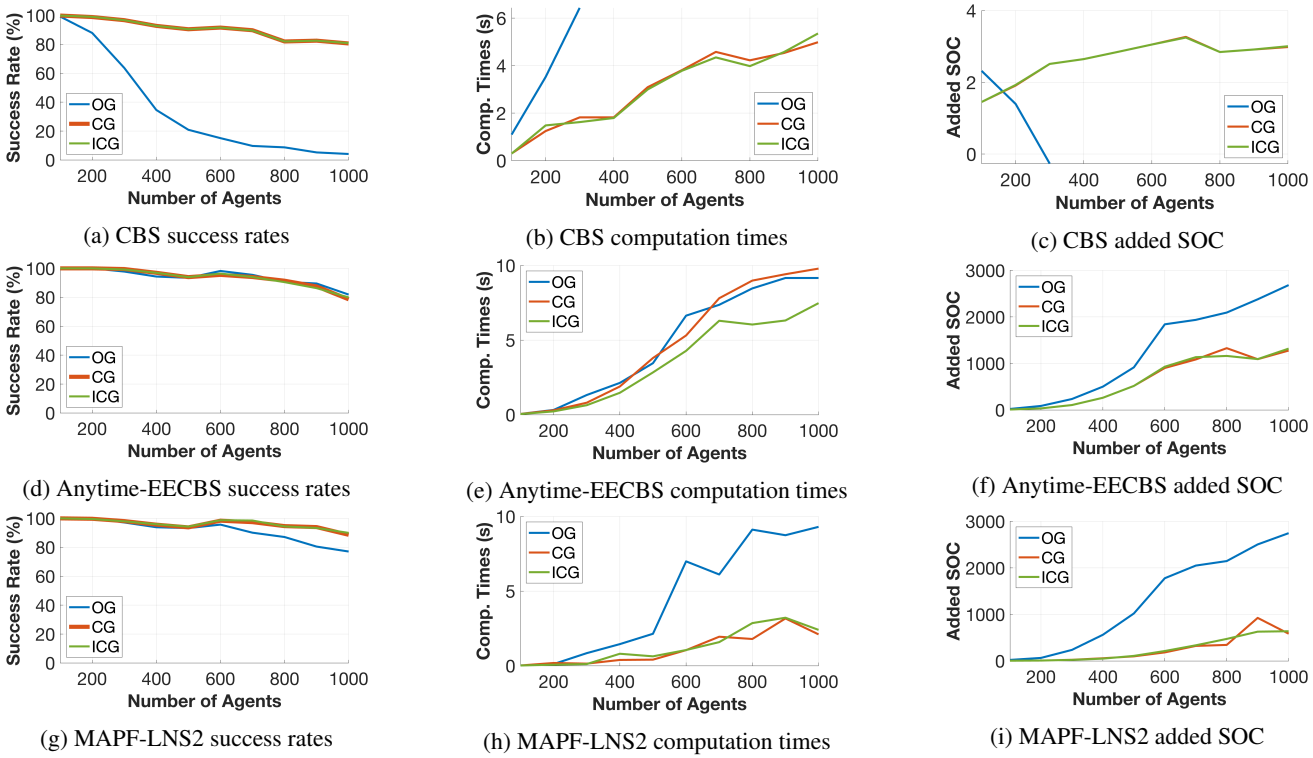
In our setting, we consider environments where delays are enabled on all vertices (c.f. Remark 2). While this is not used by the algorithms, it does allow us to strengthen Lemma 1.

Remark 3 (Upper bound on delays)**.** *Consider a colliding plan P for n agents obtained by experiencing d unexpected delay from a non-colliding plan. We can always repair P by delaying all the remaining agents for d timesteps, hence synchronizing back to the non-colliding plan. This gives an upper bound of $d(n-1)$ on the number of delays necessary to repair a delayed plan.*

We can use Remark 3 as a sanity check on the optimality of solutions obtained in the experiments. We divide our experiments to ones where we introduce a single delay, and ones

(a) Berlin (b) Boston (c) den520d (d) empty 32 (e) Paris (f) random 64 (g) w-coast (h) warehouse-1 (i) warehouse-2

Figure 5: MAPF maps used in the experimental evaluations.



(a) CBS success rates (b) CBS computation times (c) CBS added SOC

(d) Anytime-EECBS success rates (e) Anytime-EECBS computation times (f) Anytime-EECBS added SOC

(g) MAPF-LNS2 success rates (h) MAPF-LNS2 computation times (i) MAPF-LNS2 added SOC

Figure 6: Global averages over all tested instances. CG and ICG lines are coincident in Figs. 6a-6d, and 6g.

where we introduce multiple delays. Conceptually, the former is a "cleaner" study of the effect of a delay. As we show, however, the latter is much more challenging to solve.

5.1 Experimental Setup – Single Delay

We consider nine different MAPF maps from Stern et al. (2019) (see Fig. 5). For each map, we selected 10 random MAPF instances and use Anytime-EECBS (Li, Ruml, and Koenig 2021) to calculate a high-quality MAPF solution P given a time budget of three minutes. If successful, we perform 10 iterations where, in each iteration, we sample a *collision-inducing* delay for a random agent i at a random step $0 < k < m_i$, where m_i is the agent i's path length, such that the resulting plan becomes colliding at $k < t < m_i$. We then attempt to repair this plan on OG, CG and ICG using CBS, Anytime-EECBS, and MAPF-Large Neighborhood Search 2 (MAPF-LNS2) (Li et al. 2022) within a three minute time limit.

We emphasize that our goal is not to compare MAPF algorithms, but to compare the effect of CG and ICG against OG on different MAPF algorithms.

The number of agents was incremented from $n = 100$ in steps of 100 until a maximum number of agents was reached for a particular instance. For most instances, the maximal number of agents was $n = 1000$.

We evaluated our findings using three metrics:

(i) success rate (i.e., percentage of plans for which a solution was computed within the allotted time budget),

(ii) computation time (only on successful instances), and

(iii) added plan length. For CG and ICG, this amounts to the number of delays introduced.

All evaluations were performed on AMD 4.5 GHz CPU and 64 GB of RAM. Our implementation[3] was forked from the MAPF-LNS2 codebase[4] built in C++. Due to space constraints, we provide a representative subset of our results here.

[3]https://github.com/aria-systems-group/Delay-Robust-MAPF

[4]https://github.com/Jiaoyang-Li/MAPF-LNS2.

We remark that we do not seed the MAPF solvers (e.g., MAPF-LNS2) with the colliding plan because we are only interested in the affect of CG/ICG compared to OG. Doing so is an implementation choice that does not affect our conclusions in any significant way.

5.2 Computation Time

The average computation times of every MAPF solver over all of the instances are shown in Figs. 6b, 6e, and 6h. The three lines correspond to OG (blue), CG (orange), and ICG (green). The most obvious computation time improvements are seen in Fig. 6b which show that not only can CBS on CG and ICG scale to 1000 agents, but it also produced very low computation times compared to OG. Similarly, MAPF-LNS2 on CG and ICG performed faster than on OG (see Fig. 6h). Anytime-EECBS on ICG improved its computation times over both CG and OG, especially for large number of agents (see Fig. 6e).

The computation times for all three MAPF algorithms on the most difficult scenarios are shown in Table 1. Combinations that performed at least as good as OG are **bolded**.

CBS on CG and ICG consistently outperformed OG (Table 1, rows 1-9). Interestingly, Anytime-EECBS is generally faster on OG than CG (Table 1, all rows except 8) but is faster on ICG than both OG and CG (Table 1, all rows except 3 and 7). MAPF-LNS2 shows obvious computation-time improvements of up to $14\times$ than when using OG (Table 1, all rows except row 7).

5.3 Success Rate

The average success rates of every MAPF solver over all the instances are shown in Figs. 6a, 6d, and 6g. The results show that CG and ICG perform better than OG across all values of n for both CBS and MAPF-LNS2 while performing equally as well for Anytime-EECBS. CBS improved the most, which succeeded over 80% of the time for all n with CG and ICG but only scaled to 400 agents on OG.

The success rates for all three MAPF algorithms on the most difficult (largest number of agents n) scenarios are shown in Table 2. The entries where CG or ICG performed at least as well as OG are **bolded**.

Observe that using CG and ICG dramatically improved ($9\times$, at least) the success rate of CBS on all tested maps (rows 1-9 of Table 2). In addition, the optimal CBS algorithm becomes a viable candidate for solving instances of ACID up to 1000 agents. Anytime-EECBS provided similar success rates for all three graphs on most examples (rows 1, 2, 5, 7 and 9 of Table 2). There are however, scenarios where Anytime-EECBS both improved the success rate (rows 4 and 6 of Table 2) and hindered it (rows 3, and 8 of Table 2). MAPF-LNS2 produced similar success rates for all three graphs on about half of the tested instances (rows 1, 2, 4, 5, and 9) but did show significant improvements on the other half (rows 3, and 6-8 of Table 2).

5.4 Added SOC

The average added SOC of every MAPF solver over all the maps are shown in Figs. 6c, 6f, and 6i. The results show that both Anytime-EECBS and MAPF-LNS2 generally provided shorter solutions when using CG and ICG compared to OG (see Figs. 6f and 6i). Replanning with CBS on CG and ICG generally results in very small SOC additions (about $4 \ll n$ as per Remark 3) compared to Anytime-EECBS and MAPF-LNS2 (see Fig. 6c).

The added SOC for all three MAPF algorithms on the most difficult (largest number of agents n) scenarios are shown in Table 3. The entries where CG or ICG performed at least as well as OG are shown in **bold**.

In terms of added SOC, we see that in the (*very*) rare occasion that CBS succeeds on OG, it can improve the original plan (rows 1-2, 5-6, and 9 of Table 3). Meanwhile, CBS on CG and ICG generally repair plans with 1000 agents with at most 11 additional delays (all rows of Table 3). Anytime-EECBS produces a larger number of delays compared to CBS and MAPF-LNS2 but using CG and ICG performs better than OG, in general (see rows 1-9 of Table 3). MAPF-LNS2 on CG and ICG generally improved optimality compared to OG (all rows except 7 of Table 3).

Overall, planning on CG and ICG improved all three algorithms in different aspects. CBS on CG and ICG scaled to 1000 agents while minimizing added plan length. CG and ICG also enabled Anytime-EECBS to produce more optimal solutions while occasionally improving success rate. MAPF-LNS2 on CG and ICG greatly improved the added plan length and success rates compared to OG while also improving computation times.

Note that the performance differences on CG and ICG are typically very small, if any. This occurs because as the space becomes congested, most vertices become intersections, and hence CG and ICG become almost identical.

5.5 Experimental Setup – Multiple Delays

Our setup for introducing multiple delays is identical to that of Section 5.1 with some key exceptions: we introduced multiple simultaneous delays, we tested on 3 maps (Figs. 5e, 5f, and 5h), we only considered instances with 600 agents, our timeout was *one* minute, and we considered the anytime properties of Anytime-EECBS and MAPF-LNS2. Table 4 report our results for 10 and 50 delay introductions.

Roughly summarizing our findings, we see that CBS fails almost entirely in the presence of multiple delays. This fact is interesting, as it identifies a class of graphs for which CBS struggles that is not caused due to the number of agents or the branching degree (as CBS works well in the presence of one delay with even more agents).

We see that Anytime-EECBS performs roughly equally well on OG, CG and ICG, but typically elongates the plan more on OG than in CG and ICG (despite having potentially shorter plans!). MAPF-LNS2 performs poorly on OG, but well on CG and ICG, introducing less delays than Anytime-EECBS, when successful.

We conclude that for multiple delays, heuristic algorithms are preferable to the optimal CBS, as expected, but that CG and ICG do assist in minimizing the number of delay.

Row	Map	CBS			Anytime-EECBS			MAPF-LNS2		
		OG	CG	ICG	OG	CG	ICG	OG	CG	ICG
1	Fig. 5a	12.5	**4.8**	**4.6**	10.6	12.7	**6.8**	7.0	**0.8**	**0.5**
2	Fig. 5b	6.5	**4.9**	**4.7**	13.3	14.3	**8.3**	13.4	**1.3**	**2.6**
3	Fig. 5c	—	**9.7**	**15.5**	18.4	20.6	24.0	44.2	**3.1**	**4.0**
4	Fig. 5d	—	**5.7**	**4.8**	3.0	3.6	**1.9**	23.3	**4.5**	**9.6**
5	Fig. 5e	10.3	**4.5**	**4.1**	8.4	11.1	**5.0**	4.3	**0.6**	**0.7**
6	Fig. 5f	65.6	**4.1**	**3.5**	0.8	2.3	2.2	17.7	**4.4**	**13.2**
7	Fig. 5g	—	**23.0**	**22.7**	4.4	20.6	**3.6**	1.4	36.3	31.3
8	Fig. 5h	—	**6.6**	**7.4**	16.7	**16.5**	25.2	35.4	**5.0**	**2.6**
9	Fig. 5i	25.8	**7.2**	**7.3**	6.4	7.8	**4.5**	1.2	**0.4**	**0.4**

Table 1: Computation time means (s) of the initial solutions for all maps on scenes with the most agents.

Row	Map	CBS			Anytime-EECBS			MAPF-LNS2		
		OG	CG	ICG	OG	CG	ICG	OG	CG	ICG
1	Fig. 5a	4.2	**90.3**	**90.3**	100.0	98.6	98.6	100.0	**100.0**	**100.0**
2	Fig. 5b	4.7	**81.3**	**81.3**	100.0	98.4	**100.0**	100.0	96.9	98.4
3	Fig. 5c	0.0	**61.2**	**63.3**	89.8	67.3	73.5	63.3	**89.8**	**93.9**
4	Fig. 5d	0.0	**46.3**	**46.3**	7.3	**26.8**	**26.8**	19.5	17.1	**19.5**
5	Fig. 5e	6.5	**90.9**	**90.9**	98.7	**98.7**	**98.7**	100.0	**100.0**	**100.0**
6	Fig. 5f	2.4	**61.0**	**61.0**	17.1	**31.7**	**34.1**	22.0	**41.5**	**51.2**
7	Fig. 5g	0.0	**33.3**	**33.3**	12.5	**16.7**	**12.5**	16.7	**29.2**	**29.2**
8	Fig. 5h	0.0	**83.6**	**83.6**	67.2	49.2	52.5	44.3	**93.4**	**91.8**
9	Fig. 5i	10.0	**97.1**	**97.1**	97.1	**100.0**	**100.0**	98.6	**100.0**	**100.0**

Table 2: Success rate means (%) of the initial solutions for all maps on scenes with the most agents.

Row	Map	CBS			Anytime-EECBS			MAPF-LNS2		
		OG	CG	ICG	OG	CG	ICG	OG	CG	ICG
1	Fig. 5a	−16	4	4	1,727	**1037**	**1036**	2,977	**171**	**199**
2	Fig. 5b	−16	3	3	4,813	**2803**	**2816**	6,428	**995**	**983**
3	Fig. 5c	—	**4**	**4**	10,305	**5847**	**6141**	9,297	**1019**	**1036**
4	Fig. 5d	—	**11**	**11**	457	**262**	**340**	420	480	**375**
5	Fig. 5e	−8	3	3	1,110	**694**	**695**	2,114	**141**	**144**
6	Fig. 5f	−37	5	5	607	**555**	661	1,340	**726**	**1053**
7	Fig. 5g	—	**5**	**5**	1,657	5,926	**1026**	415	12,950	14,760
8	Fig. 5h	—	**5**	**5**	5,245	**1251**	**1363**	4,074	**540**	**487**
9	Fig. 5i	−11	3	3	506	**293**	**293**	767	**73**	**75**

Table 3: Added SOC means of the initial solutions for all maps on scenes with the most agents.

Delay Introduction	Statistic	CBS			Anytime-EECBS			MAPF-LNS2		
		OG	CG	ICG	OG	CG	ICG	OG	CG	ICG
10	Succ. Rate (%)	1	**32**	**32**	99	98	98	32	**95**	**100**
	Comp. Times (s)	—	**5.3**	**5.3**	1.6	2.1	1.7	5.4	**3.7**	**4.5**
	Added SOC	—	**7**	**7**	587 (168)	**346** (**80**)	**346** (**79**)	230 (66)	**136** (99)	**140** (102)
50	Succ. Rate (%)	0	0	0	100	**100**	**100**	31	**100**	**100**
	Comp. Times (s)	—	—	—	1.8	2.8	2.1	7.1	**5.2**	**5.5**
	Added SOC	—	—	—	660 (318)	758 (**296**)	757 (**314**)	336 (138)	556 (370)	567 (379)

Table 4: Experimental results for 10 and 50 simultaneous delays. Anytime algorithms report two values for Added SOC: the initial solution and the best solution (in parentheses). Computation times are provided for the initial solutions.

6 Conclusion

We address the issue of repairing MAPF plans after encountering unexpected delays. We introduce the ACID problem, and prove it is NP-Complete. We adapt ACID into an MAPF problem using two novel graph formulations, CG and ICG, which confine the graph to the original paths. We empirically show that the CG and ICG improve MAPF algorithms' ability to repair plans compared to traditional MAPF.

Acknowledgements

This research was supported in by the Israeli Ministry of Science & Technology grants No. 3-16079 and 3-17385, the United States-Israel Binational Science Foundation (BSF) grants no. 2019703 and 2021643, the Israel Science Foundation (grant nos. 1736/19 and 2261/23), by NSF/US-Israel-BSF (grant no. 2019754), by the Blavatnik Computer Science Research Fund, the Israeli Smart Transportation Research Center (ISTRC), Israel Science Foundation (grant No. 989/22), and the University of Colorado Boulder.

References

Abrahamsen, M.; Geft, T.; Halperin, D.; and Ugav, B. 2023. Coordination of Multiple Robots along Given Paths with Bounded Junction Complexity. In *AAMAS*, 932–940.

Atzmon, D.; Stern, R.; Felner, A.; Sturtevant, N. R.; and Koenig, S. 2020a. Probabilistic Robust Multi-Agent Path Finding. In *ICAPS*, 29–37.

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N. 2020b. Robust Multi-Agent Path Finding and Executing. *JAIR*, 67: 549–579.

Ausiello, G.; Marchetti-Spaccamela, A.; Crescenzi, P.; Gambosi, G.; Protasi, M.; and Kann, V. 1999. *Complexity and approximation: combinatorial optimization problems and their approximability properties*. Springer.

Banfi, J.; Basilico, N.; and Amigoni, F. 2017. Intractability of Time-Optimal Multirobot Path Planning on 2D Grid Graphs with Holes. *Robot. Autom. Lett.*, 2: 1941–1947.

Barták, R.; Švancara, J. í.; and Vlk, M. 2018. A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs. In *AAMAS*, 748–756.

Belov, G.; Du, W.; de la Banda, M. G.; Harabor, D.; Koenig, S.; and Wei, X. 2020. From Multi-Agent Pathfinding to 3D Pipe Routing. In *SOCS*, 11–19.

Berndt, A.; Duijkeren, N. V.; Palmieri, L.; and Keviczky, T. 2020. A Feedback Scheme to Reorder a Multi-Agent Execution Schedule by Persistently Optimizing a Switchable Action Dependency Graph. arXiv:2010.05254.

DeHaan, I.; and Friggstad, Z. 2023. Approximate Minimum Sum Colorings and Maximum k-Colorable Subgraphs of Chordal Graphs. In *WADS*, volume 14079 of *Lecture Notes in Computer Science*, 326–339. Springer.

Geft, T. 2023. Fine-Grained Complexity Analysis of Multi-Agent Path Finding on 2D Grids. In *SOCS*, 20–28. AAAI Press.

Hönig, W.; Kumar, T. K. S.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Multi-Agent Path Finding with Kinematic Constraints. In *ICAPS*, 477–485.

Komenda, A.; and Novák, P. 2011. Multi-agent plan repairing. In *Decision Making in Partially Observable, Uncertain Worlds: Exploring Insights from Multiple Communities, Proceedings of IJCAI 2011 Workshop*, 1–6.

Komenda, A.; Novák, P.; and Pěchouček, M. 2014. Domain-independent multi-agent plan repair. *J. Netw Comput Appl*, 37: 76–88.

Kottinger, J.; Geft, T.; Almagor, S.; Salzman, O.; and Lahijanian, M. 2023. Introducing Delays in Multi-Agent Path Finding. arXiv:2307.11252.

Kubicka, E. M.; and Schwenk, A. J. 1989. An Introduction to Chromatic Sums. In *ACM Conf. on CS*, 39–45.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2022. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. In *AAAI*, 10256–10265.

Li, J.; Chen, Z.; Zheng, Y.; Chan, S.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2021. Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge. In *ICAPS*, 477–485.

Li, J.; Ruml, W.; and Koenig, S. 2021. EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. In *AAAI*, 12353–12362.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with Consistent Prioritization for Multi-Agent Path Finding. In *AAAI*, 7643–7650.

Ma, H.; Kumar, T. K. S.; and Koenig, S. 2017. Multi-Agent Path Finding with Delay Probabilities. In *AAAI*, 3605–3612.

Nebel, B. 2020. On the Computational Complexity of Multi-Agent Pathfinding on Directed Graphs. In *ICAPS*, 212–216.

Nebel, B.; and Koehler, J. 1995. Plan reuse versus plan generation: A theoretical and empirical analysis. *Art. Int.*, 76(1-2): 427–454.

Okumura, K. 2023. LaCAM: Search-Based Algorithm for Quick Multi-Agent Pathfinding. In *AAAI*, 11655–11662.

Ramaithitima, R.; Whitzer, M.; Bhattacharya, S.; and Kumar, V. 2016. Automated Creation of Topological Maps in Unknown Environments Using a Swarm of Resource-Constrained Robots. *Robot. Autom. Lett.*, 1(2): 746–753.

Salzman, O.; and Stern, R. 2020. Research Challenges and Opportunities in Multi-Agent Path Finding and Multi-Agent Pickup and Delivery Problems. In *AAMAS*, 1711–1715.

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

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, S.; Barták, R.; and Boyarski, E. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *SOCS*, 151–159.

Svancara, J.; Tignon, E.; Barták, R.; Schaub, T.; Wanko, P.; and Kaminski, R. 2023. Multi-Agent Pathfinding with Predefined Paths: To Wait, or Not to Wait, That Is the Question [Extended Abstract]. In *SOCS*, 185–186.

Tonola, C.; Faroni, M.; Beschi, M.; and Pedrocchi, N. 2023. Anytime Informed Multi-Path Replanning Strategy for Complex Environments. *IEEE Access*, 11: 4105–4116.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*, 29(1): 9.

Yu, J. 2016. Intractability of optimal multirobot path planning on planar graphs. *Robot. Autom. Lett.*, 1: 33–40.