Proceedings of the Sixteenth International Symposium on Combinatorial Search (SoCS 2023)

Adapting to Planning Failures in Lifelong Multi-Agent Path Finding

Jonathan Morag[1,2], **Roni Stern**[1], **Ariel Felner**[1]
[1] Ben-Gurion University of the Negev
[2] Get Fabric, Inc.
moragj@post.bgu.ac.il, roni.stern@gmail.com, felner@bgu.ac.il

Abstract

Multi-Agent Path Finding (MAPF) is the problem of finding collision-free paths for multiple agents operating in the same environment. In Lifelong MAPF (LMAPF), these agents continuously receive new destinations, and the task is to constantly update their paths while optimizing for a high throughput over time. Therefore, many MAPF sub-problems must be solved over time in order to solve a single LMAPF problem. LMAPF problems manifest in real-world applications, such as automated warehouses, where strict responsiveness requirements limit the amount of time allocated to planning. MAPF algorithms occasionally fail to produce a plan within the allotted time. We propose a system design for LMAPF that is robust to such planning failures. Then, we explore different approaches to avoid planning failures, reduce their severity, and handle them when they occur. In particular, we describe and analyze different Fail Policies that are applied when planning failures occur and ensure collisions and unnecessary degradation of throughput are avoided. To our knowledge, while such Fail Policies are used in practice in the industry, they have yet to be researched academically.

1 Introduction

Multi-agent Path Finding (MAPF) is the problem of finding a set of non-conflicting paths for a set of agents on a graph. MAPF has recently received significant interest in many research works (Stern et al. 2019; Felner et al. 2017). MAPF is relevant for several existing and emerging practical applications (Ma et al. 2017; Ho et al. 2019; Li et al. 2019; Dresner and Stone 2008). In particular, automated warehouses have recently become a primary use-case for MAPF, with interest from both the industry and academia (Ma et al. 2017; Li et al. 2021b). The type of MAPF problem that is most relevant to automated warehouses is *Lifelong MAPF (LMAPF)*. LMAPF is a version of the MAPF problem in which a sequence of MAPF problems must be solved over time, where an agent receives a new target to find a path to, whenever it reaches its current target. Online LMAPF algorithms such as the *Rolling-Horizon Collision Resolution (RHCR)* (Li et al. 2021b), solve LMAPF problems by repeatedly planning the next sequence of actions to execute based on the agents' current locations and targets.

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In some cases, however, an online LMAPF algorithm may fail to plan the next sequence of actions. We refer to such a situation as a *planning failure*. A planning failure may happen because the current state is unsolvable, because the algorithm itself is incomplete, or because the LMAPF algorithm cannot find a solution within the time allocated to it. In this paper, we consider the following question — how to design a system for solving LMAPF problems that can handle such planning failures? The first contribution of this work is a design for such a robust system. Our system design includes three main components: an *agent selection policy*, a *planner*, and a *fail policy*. The agent selection policy defines which agents should replan. The planner invokes a LMAPF algorithm to replan and update the paths of the selected agents. Importantly, the chosen LMAPF algorithm is modified such that it may return a *partial solution* if it encounters a planning failure. In cases where a planning failure occurs, the fail policy is responsible for quickly remedying the returned partial solution to ensure the system continues to function properly.

Our second contribution is an exploration of different ways to implement each of these system components. For the agent selection policy component, we consider selecting to replan only agents whose current paths are expected to conflict in the near future. This selection policy balances between the overhead of always replanning for all agents and the undesirable myopic behavior of only replanning when the next actions conflict. For the planner component, we propose a simple method to bias RCHR (Li et al. 2021b) towards returning higher quality partial solutions in cases where a planning failure occurs. For the fail policy component, we introduce two fast and simple fail policies, IStay and IAvoid. IStay moves only the non-conflicting agents, and has the conflicting agents stay in place. IAvoid allows conflicting agents to make a single action to avoid conflicts. Both policies are guaranteed to avoid collisions and run in tractable time. To the best of our knowledge, while fail policies are used to some extent in real-world applications, they have never been formally defined.

Our third contribution is a comprehensive experimental evaluation on a standard benchmark of the proposed system design using different implementations of its components. This evaluation reveals the advantages and disadvantages of using different implementations of our system components.

The evaluation revealed a specific configuration of agent selection policy, planner, and fail policy that works best in most problem instances. In some cases, using this configuration can double the system throughput compared to a state-of-the-art baseline.

2 Background and Problem Definition

In the Multi-Agent Path Finding (MAPF) problem, the input is a graph $G = (V, E)$, and a set A of agents, where each agent a_i is associated with a pair of source and target vertices, denoted s_i and g_i respectively. A path $p = (v_1, \ldots, v_{|p|})$ is a sequence of vertices such that each pair of successive vertices in p are either the same vertex or are connected by an edge. The former corresponds to a *wait* action and the latter to a *move* action. A *solution* to a MAPF problem is a mapping of each agent $a_i \in A$ to a path p_i that starts at its source and ends at its target, and no two paths have a *conflict*. Several types of conflicts have been proposed for classical MAPF (Stern et al. 2019). In this work, we consider only *vertex conflicts* and *swapping conflicts*, that is, a pair of paths is said to conflict if two agents following them would occupy the same vertex at the same time, or swap their vertices in one move. The length of a path is defined as the number of vertices (non-unique) in the path, minus one ($len(p) = |p| - 1$). A cost function maps a solution to a numeric cost. *Sum Of Costs* (SOC) is a common cost function for MAPF, defined as the sum of the lengths of all paths in the solution ($SOC(\pi) = \sum(len(p)|p \in \pi)$). Another well known cost function is *Makespan*, defined as the maximum length amongst all paths in the the solution ($makespan(\pi) = \max(len(p)|p \in \pi)$). A solution π is considered *optimal* by some cost function for a given MAPF problem if it is has the minimum cost of all solutions to that MAPF problem.

The *Lifelong MAPF (LMAPF)* problem (Ma et al. 2017; Li et al. 2021b) extends MAPF by considering that after agents reach their targets, they are given new targets to reach, and this process continues indefinitely over time. Following Li et al. (2021b) and others (Švancara et al. 2019), we consider the online version of LMAPF, where at every time step each agent is associated with a single target, and an agent receives a new target only when it reaches its current target. A useful measure used to evaluate the performance of LMAPF algorithms is *throughput* at X, which is the number of times agents reached their targets after X time steps. Note that we use the term "time step" to refer to execution time steps, as opposed to CPU time that is used for planning. Throughout this work, we assume agents' actions are deterministic (i.e., move and wait actions always reach their intended locations), which is a reasonable abstraction in some cases. Also, prior work showed how post-processing of MAPF solutions allows them to be executed even when these assumptions are not met in practice (Hönig et al. 2016).

2.1 Solving LMAPF Problems

Solving the online version of LMAPF requires interleaving planning and execution, and consists of an alternating sequence of planning and execution periods. Some prior works plan only when an agent reaches its target and is assigned a new target, and only plan for that agent (Švancara et al. 2019; Čáp, Vokřínek, and Kleiner 2015), while others replan for all agents every time step (Wan et al. 2018). *Rolling-Horizon Collision Resolution (RHCR)* is a state-of-the-art framework for LMAPF in which replanning occurs every fixed number of time steps and every replanning only considers conflicts between agents within a fixed time horizon, allowing conflicts to occur afterwards. The sizes of the replanning frequency and the time horizon are input parameters for RHCR. We assume an agent's next target becomes known only when it reaches its current target. Therefore, when an agent reaches its target, it will stay there until the next time that replanning occurs.

RHCR is a framework that allows different MAPF algorithms to be plugged in while adapting them to ignore conflicts beyond the RHCR horizon. In this work, we consider RHCR when used with a Prioritised Planning (PrP) algorithm (Latombe 1991; Silver 2005), which is arguably the state of the art in practical applications of large-scale LMAPF. In PrP, the agents are sorted by some (often arbitrary) priority ordering. Then, individual paths are computed for the agents in order of their priority, where each agent avoids the paths of all higher priority agents. PrP is incomplete and sub-optimal, but it is fast (polynomial in the number of agents) and very simple to understand, implement, and extend. A simple and effective improvement to PrP is *Prioritised Planning with Random Restarts* (PrPr) (Bennewitz, Burgard, and Thrun 2001). In PrPr, several iterations of PrP are performed. For each iteration, a different random priority ordering is used. The best solution found during those iterations is returned. This procedure increases the running time linearly while improving the quality of the solution and increasing the likelihood that a solution would be found. In the rest of this paper, we will refer to PrPr simply as PrP.

2.2 Problem Setting

In this work, we consider the online version of LMAPF that is solved within a system in which the time allowed for planning before every execution is bounded. That is, some solution must be produced within a given time limit. In more detail, our system includes two main components: a *controller* and a *planner*. The planner is called by the controller every fixed number of time steps k to decide on the next steps to perform. The planner must return for each agent at least a path of length k, instructing each agent what to do until the next planning period. The controller attempts to execute the returned paths, monitoring the current state and detecting if collisions are about to occur. This setting is based on the real-world commercial autonomous warehouse system of Get Fabric, Inc. A similar setting has already been described in the context of Real-Time MAPF (Sigurdson et al. 2018).

Our main research question is how to design a robust planner within this system that will maximize the system's throughput. A key challenge in using an existing LMAPF solver such as RHCR for this purpose is that it may *fail* to

find a solution within the planning time limit. One reason for such a *planning failure* is that the system has reached a state in which no solution exists. Another reason is that while a solution exists for the current time step's state, the underlying LMAPF solver could not find it either due to its inherent incompleteness or due to the planning time limit. Existing LMAPF algorithms avoided this problem by pausing execution and planning until a solution is found, and, if finding a solution takes an excessive length of time, considering the algorithm to have failed to solve the larger LMAPF problem. Practical LMAPF applications must handle such planning failures in some way, and most do so in an ad-hoc manner. In the next section, we formalize such planning failure situations and define *fail policies* to handle them.

3 A System Design for Robust LMAPF

In this section, we describe our design for a robust LMAPF system that can handle planning failures. We first introduce several required definitions and notations.

3.1 Partial Solutions and Fail Policies

A *partial solution* to a MAPF problem is a mapping of agents to paths such that each agent is either not mapped to any path or mapped to a single path starting from its source (but not necessarily ending at its target). Any solution to a given MAPF problem is also a partial solution to it, and an empty mapping in which no agent is mapped to a path is also a partial solution. Many MAPF and LMAPF algorithms, such as CBS (Sharon et al. 2015), BCP (Lam et al. 2022), PrP (Latombe 1991; Silver 2005), and PBS (Ma et al. 2019), work by iteratively updating a partial solution starting from an empty mapping of agents to paths and continuously adding paths and resolving conflicts between them. In a planning failure situation, it may be beneficial to consider the *partial solution* created so far by the LMAPF algorithm being used.

Invalid agents and types of partial solutions. We say that an agent a_i is *k-invalid* in a partial solution $\hat{\pi}$ if either it is not mapped to any path in $\hat{\pi}$ or the path it is mapped to has a conflict with some other path in $\hat{\pi}$ in the first k steps. A *k-safe solution* is a partial solution that includes a path for every agent and must be free of conflicts within its first k steps, that is, a partial solution in which all agents are not k-invalid.[1] The RHCR algorithm is designed to return a *k*-safe solution where k is its horizon parameter.

Definition 1 (Fail Policy) *A* fail policy *is a function that accepts a MAPF problem* Π*, a partial solution to it* $\hat{\pi}$*, and a planning frequency k, and outputs a k-safe solution.*

Note that the agents' sources in the MAPF problem Π given to the fail policy represent the current locations of the agents, not their original sources. Also, it is possible to define a fail policy that returns a *k'*-safe solution, where k' is different from the planning frequency k. If $k' < k$, we must also check for collisions between planning periods. In our implementation, we set $k = 3$ and aimed for a 3-safe solution.

[1] The concept of a *k*-safe solution is similar to the *partial routes* in the Windowed Hierarchical Cooperative A* algorithm (Silver 2005), except that paths in the partial routes are also required to end at each agent's target.

Algorithm 1: Fail-Robust LMAPF

Input: $\langle G, A\rangle$, a MAPF problem
Input: k, the number time steps between planning periods
Input: T_{max}, planning time limit

1: $\hat{\pi} \leftarrow$ A solution where all agents stay in place
2: **loop**
3: Set new targets for agents that reached their targets
4: $A_{select} \leftarrow$ SelectAgents($\hat{\pi}, k$)
5: $\hat{\pi} \leftarrow$ Plan($G, A_{select}, \hat{\pi}, T_{max}, k$)
6: **if** there exist k-invalid agents in $\hat{\pi}$ **then**
7: $\hat{\pi} \leftarrow$ FailPolicy($\hat{\pi}$)
8: **end if**
9: Execute the first k steps in $\hat{\pi}$
10: **end loop**

3.2 System Design

Algorithm 1 describes the proposed LMAPF system design, which is robust to planning failures and uses the proposed fail policy mechanism. Throughout the search, we maintain an active, possibly partial solution $\hat{\pi}$. Every iteration of the loop in Algorithm 1 corresponds to a planning period (lines 3–8) followed by an actual movement of the agents (line 9). The main invariant is that at the end of a planing period we ensure $\hat{\pi}$ is a k-safe solution. Then, the first k steps of $\hat{\pi}$ are executed. First, any agents that reached their current targets are given new targets. Then, we *select* which agents require updating their paths in $\hat{\pi}$. These agents are sent to a LMAPF planner, considering the paths of all other agents as constraints. In the resulting partial solution $\hat{\pi}$, agents that were not sent to the planner retain their existing plans. If the returned partial solution $\hat{\pi}$ is not *k*-safe, i.e., there are k-invalid agents in $\hat{\pi}$, then we apply a fail policy to update $\hat{\pi}$ to make it *k*-safe. Finally, all agents perform the next k actions according to $\hat{\pi}$. This system design has three main components: an agent selection policy, a planner, and a fail policy. Next, we discuss several ways to implement each of these components.

3.3 Agent Selection Policies

In every planning period, we are given an opportunity to update the paths in $\hat{\pi}$ that are mapped to any subset of agents. We explore two policies for selecting which subset of agents to plan for in each planning period.

- **AllAgents.** Select to plan for all agents.
- **Fail@LH(R).** Select to plan for all R-invalid agents, where $R \geq k$.[2]

The benefit of the AllAgents policy is that it is the most informed approach, allowing updates to non-conflicting paths that can potentially be improved. However, this policy may be wasteful in terms of computational complexity, as the

[2] $R < K$ is also technically possible, since applying the fail policy later would still guarantee a *k*-safe solution.

path already mapped to some agents in $\hat{\pi}$ may be sufficient. Intuitively, Fail@LH(R) reduces the computational complexity as fewer agents must be planned for. However, the specific constraints imposed to avoid existing paths may make the planning more difficult. For example, a constraint created for the path of one of the unselected agents may block a passage in the graph and force other agents to make a large detour, which would increase the cost of their paths, and possibly even cause them to impede the movement of yet more agents. Thus tuning the R parameter, referred to as the *lookahead range*, may have a great impact on the performance of Fail@LH(R).

3.4 Planners

Most LMAPF algorithms are defined in a binary way: either a solution (or in the case of RHCR, a k-safe solution) has been found, or not. Thus, they do not specify what to return when a planning failure occurs. Our system requires a planner that returns a partial solution in such cases. A baseline implementation for such a planner, which we refer to as *Full*, it to run a LMAPF algorithm and return an empty partial solution when a planning failure occurs.

Next, we consider two other approaches for obtaining a non-empty partial solution. We describe these approaches specifically in the context of RHCR that uses the PrP framework, but we believe these approaches can naturally generalize to other LMAPF algorithms.

Recall that in PrP, a solution is generated incrementally by creating a path for each agent according to some order, having each agent constrained to avoid the previously created paths. If such a path cannot be found, the current PrP iteration is halted and the planner restarts with a new agent priority ordering. A planning failure in PrP only occurs when the planning time is exhausted. Thus, a natural way to return a non-empty partial solution is to create one from the paths generated for the agents in one of the PrP iterations that were halted before the planning time was exhausted. We refer to this policy as *Restart*. A different way to obtain partial solutions involves slightly modifying the PrP algorithm. When a path cannot be found for an agent, instead of restarting the search, we ignore this agent in this PrP iteration and continue creating paths for the next agents in the ordering. Thus, a PrP iteration only ends after going over all agents or when the planning time has been exhausted. This method can result in partial solutions that have significantly more agents with paths mapped to them, and consequently, a more effective behavior of the chosen fail policy. We refer to this method as *Persist*.

While Restart directs search efforts towards finding a full solution, Persist aims to find a partial solution that covers as many agents as possible. We also tried several hybrid policies, e.g., perform one iteration with Persist and then all other iterations with Restart, or switch between Persist and Restart depending on how many agents we were unable to find paths for. Their performance in our experiments was similar or inferior to the simpler Restart and Persist policies, so they are not included in the results that we present in this paper.

Algorithm 2: IStay & IAvoid

Input: $\hat{\pi}$, a partial solution

1: $A_{invalid} \leftarrow$ Identify all k-invalid agents in $\hat{\pi}$
2: **while** $A_{invalid}$ is not empty **do**
3: $a \leftarrow$ Choose non-staying agent from $A_{invalid}$
4: Update $\hat{\pi}$ so a stays in place for k time steps
5: **+ if** a is still k-invalid in $\hat{\pi}$ **then**
6: **+** **TryToAvoid**(a, $\hat{\pi}$, 1)
7: **+ end if**
8: $A_{invalid} \leftarrow$ Identify all k-invalid agents in $\hat{\pi}$
9: **end while**
10: **return** $\hat{\pi}$

3.5 Simple Fail Policies

If the planner did not return a full k-safe solution then a planning failure occurs and the fail policy is invoked. Fail policies are not intended to replace the LMAPF planner, and thus it is crucial that their runtime is limited. We propose the following simple fail policies, which are easily computable:

- **AllStay.** All agents stay in their place for k time steps.
- **IStay.** k-invalid agents stay in their places for k time steps. All other agents move along their paths in $\hat{\pi}$.
- **IAvoid.** k-invalid agents either stay in their places for k time steps or make a single move to avoid conflicts with other agents, and then stay for the remaining $k-1$ time steps. All other agents move along their paths in $\hat{\pi}$.

The AllStay fail policy serves as a baseline: it is easy to compute in constant time but it completely ignores the given partial solutions $\hat{\pi}$. The other Fail Policies we propose — IStay and IAvoid— consider $\hat{\pi}$ and aim to allow agents to make progress towards their targets. Specifically, the IStay fail policy, described in Algorithm 2 (excluding lines marked with "+"), iterates over every k-invalid agent and updates their path in $\hat{\pi}$ to stay in place. The updated path may resolve some conflicts but also create others, so some agents that were not initially k-invalid may become so. If this occurs, their path in $\hat{\pi}$ is also updated to stay in place. This ensures IStay outputs a k-safe solution. In the worst case, this iterative process results in all agents staying in their places just like AllStay. In practice, however, after applying this fail policy some agents still maintain their original paths and are thus allowed to continue moving along these paths, potentially getting closer to their targets.

The IAvoid fail policy extends IStay by also allowing k-invalid agents to make a single move in order to reduce the number of conflicts they create, hopefully allowing more agents to continue moving along their paths in $\hat{\pi}$. Algorithm 2 (including lines marked with "+") contains the pseudocode for IAvoid. Just like IStay, the IAvoid fail policy iterates over all k-invalid agents and considers updating their path to stay in place. But, the IAvoid fail policy differs in cases where planning for a k-invalid agent a to stay in place creates a conflict with any other path in $\hat{\pi}$. In such cases, the IAvoid fail policy considers the option that a will make a single move to avoid conflicting with other agents (the **TryToAvoid** method). That is, for every location adjacent to the

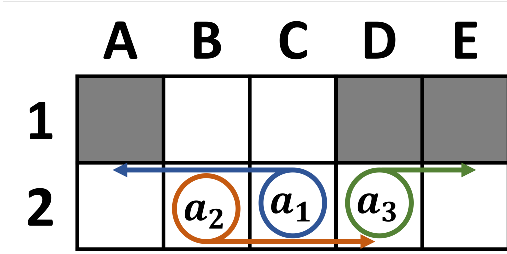

Figure 1: An example where AllStay, IStay, and IAvoid yield different behavior.

current location of a, IAvoid creates a path where a moves to that location and stays there for the next $k-1$ time steps. If this new path does not conflict with any other path in $\hat{\pi}$, it is adopted.

Similarly to the IStay fail policy, an agent may become k-invalid because another k-invalid agent has decided to stay in its place. Moreover, for IAvoid an agent may update its path more than once: first deciding to make a move action to avoid conflicts, and then becoming k-invalid again due to a change of some other agent. Note that this may only happen when another agent decides to stay in place, since the alternative is for the other agent to make a move that does not make any other path k-invalid. To avoid entering an infinite loop, we do not allow an agent to update its path again once it has decided to stay in place. In the worst case, IAvoid will result in all agents staying in their place, like AllStay. Note that each agent may choose to update its path with a move action once per possible move action. Thus, IAvoid performs at most the maximal degree in the graph plus one passes over all agents, and thus its runtime is negligible. Generalizing IAvoid to consider a sequence of more than one move to avoid the other agents' paths is left for future work, as it is expected to increase the fail policy's runtime complexity with the length of the considered sequence.

Figure 1 demonstrates the behavior of the AllStay, IStay, and IAvoid fail policies. Agents a_1, a_2, and a_3 are situated in cells $C2$, $B2$, and $D3$, respectively. The current partial solution $\hat{\pi}$ has a path for all agents, marked in the figure by arrows going out of each agent. The paths in $\hat{\pi}$ mapped for agents a_1 and a_2 have a conflict in the first time step since they are planned to swap their vertices. Using AllStay results in all agents staying in their places for the next k time steps. Using IStay results in agents a_1 and a_2 staying in their places, but agent a_3 is allowed to execute its path unimpeded. Using IAvoid results in either a_1 or a_2 moving one cell up, allowing all other agents to continue executing their paths.

4 Experiments

We implemented the proposed robust LMAPF system design[3] and conducted a set of experiments to evaluate the

[3]Our implementation is publicly available at: https://github.com/J-morag/MAPF/releases/tag/2023_SoCS_Lifelong

Type	Map	Size	Open Cells	Dia-meter	Avg. Dist.
DAO2	lt_gallowstemplar_n	180x251	10,021	287	112.0
Open	empty-48-48	48x48	2,304	94	32.0
Random	random-64-64-20	64x64	3,270	126	44.4
Maze	maze-128-128-10	128x128	14,818	546	197.9
Room	room-64-64-8	64x64	3,232	158	59.3
Warehouse	warehouse-10-20-10-2-1	161x63	5,699	218	82.1
Warehouse	warehouse-20-40-10-2-1	321x123	22,599	438	164.1
Warehouse	warehouse-20-40-10-2-2	340x164	38,756	498	177.7

Table 1: Properties of the maps used in our experiments.

different choices for agent selection policies, planners, and fail policies. We used maps and instances from a common MAPF benchmark (Stern et al. 2019). This benchmark contains a set of varied grid maps. Problem instances in this benchmark contain lists of agents with unique start and target locations. We modified these instances by preserving each agent's start location and adding a queue of random targets sampled uniformly from all locations in the instance. We used a diverse set of 8 different maps from the benchmark, with a preference towards large maps, as those have instances with a larger number of agents which is more aligned with our intended use case of a large warehouse. Table 1 details the properties of each map. Images of the maps can be seen in figure 2. We varied the number of agents between 25 and 1000 (or the maximal number of agents available in the benchmark), increasing in increments of 25. All experiments were run on a large CPU cluster of AMD EPYC 7702P processors, where each experiment was run with 8GB of available RAM and a single CPU core. To further avoid any accidental bias, we fixed all experiment runs related to a single problem instance of a single map, varying only the solvers and the number of agents, to run sequentially on the same processor core.

The underlying LMAPF planner used in all our experiments is RHCR with PrP. We set the time horizon and planning frequency RHCR parameters to 10 and 3 time steps, respectively, which we found to be effective in our experiments. In each experiment, we set a maximum planning time of one second at each time step, and a maximum of 200 time steps. After reaching 200 time steps the experiment is terminated and throughput is measured. Each map had 25 unique instances, and the throughput we report is averaged over these 25 instances. In addition, we also report the *maximal throughput* for each map, computed by examining the different throughput values achieved by a single solver on a single map but with varying numbers of agents, and taking the maximum of these values. Note that this is arguably the most important metric by which to evaluate the performance of a given solver. A system designer would want to achieve the highest possible throughput in a given environment (map), for instance in the case of a given warehouse

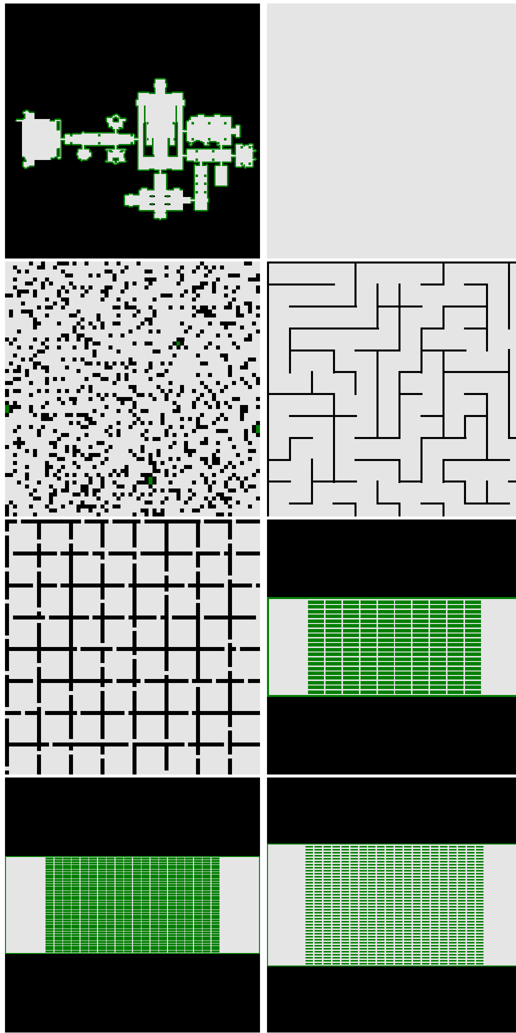

Figure 2: The maps we used. From left-to-right, top-to-bottom: lt_gallowstemplar_n, empty-48-48, random-64-64-20, maze-128-128-10, room-64-64-8, warehouse-10-20-10-2-1, warehouse-20-40-10-2-1, warehouse-20-40-10-2-2.

facility. Adding more agents is desirable so long as adding them increases the system's throughput (Salzman and Stern 2020). All the presented plots conform to the following format: The y-axis shows throughput, and the x-axis shows the number of agents. Different sub-plots show results for different maps. The maximal throughput obtained for each configuration and map, i.e., the peak of each line, is labeled with its throughput.

4.1 Selection Policies

Figure 3 compares the throughput obtained when using the baseline AllAgents and Fail@LH(R) selection policies with different values of R, ranging from 3 to 10. We show a representative subset of R values, namely 3, 5, 7, and 10, and use the Persist planner and the IAvoid fail policy. Similar trends were observed in other configurations.

The results show that in all domains, either Fail@LH(3) or Fail@LH(5) yielded the highest, or nearly the highest throughput, whereas further increasing R tended to produce worse results. Fail@LH(5) usually achieved a higher maximal throughput than Fail@LH(3), whereas Fail@LH(3) often had higher throughput for larger numbers of agents, beyond the number of agents where the maximal throughput was achieved. For example, on map room-64-64-8, maximal throughput was 268 for Fail@LH(5), 257 for Fail@LH(3), 258 for Fail@LH(7), and 245 for Fail@LH(10). An explanation for the poor performance of higher R values is that using an overly large R invalidates the plans of agents whose conflicts are distant and may be resolved incidentally before their time comes, while also increasing the sizes of the groups of agents we attempt to plan each time, reducing the likelihood that a new path would be found for these agents.

The baseline AllAgents policy achieved comparatively low throughput on most maps, especially at higher numbers of agents. For instance, on map warehouse-20-40-10-2-1, it had a maximal throughput of only 276, whereas Fail@LH(5) had a maximal throughput of 450. On maps empty-48-48, random-64-64-20, and warehouse-10-20-10-2-1, AllAgents had throughput similar to that of the solvers that used Fail@LH(R). Generally, is appears that AllAgents performs worse when there are many agents and a large map with many open cells. This is likely because under these conditions AllAgents struggles to find paths for all agents before running out of time, whereas Fail@LH(R) better focuses the search effort to where it is most needed.

4.2 Planning Policies

Figure 4 compares the results of using the Full, Restart, and Persist policies for finding partial solutions. We used Fail@LH(5) and IAvoid in these results, but similar trends are observed in other configurations. The results clearly show that Full achieves a significantly lower throughput than Restart and Persist, especially for higher numbers of agents. The only exception is random-64-64-20, where all methods achieved nearly equivalent throughput. We believe this is due to the limited maximal number of agents available for this map (200), and that at higher numbers of agents Full would perform worse than the other solvers. When comparing the Persist and Restart policies, we see that Persist achieves higher or equivalent throughput on all maps. For example, on map maze-128-128-10, Persist achieved a maximal throughput of 269, whereas Restart achieved a throughput of 234, and Full only achieved 201.

4.3 Fail Policies

Table 2 summarizes the results comparing the different fail policies, namely AllStay, IStay, and IAvoid, when using Persist and Fail@LH(5). Each row corresponds to a different map. The table contains the maximal throughput achieved by each solver. The number of agents at which the maximal throughput was achieved is written in parentheses. Within each row, the highest maximal throughput value is written in bold. The results clearly show that using a non-trivial fail policy, i.e., either IStay or IAvoid, is much better than the baseline AllStay. For instance, on map lt_gallowstemplar_n,

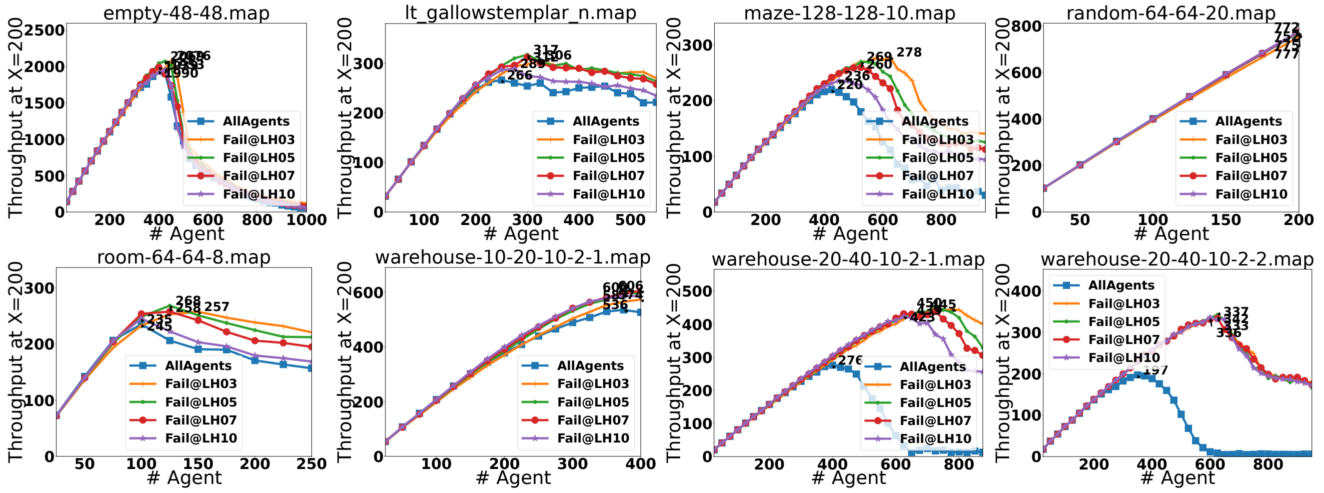

Figure 3: Throughput achieved by using different agent selection policies, as a factor of the number of agents.

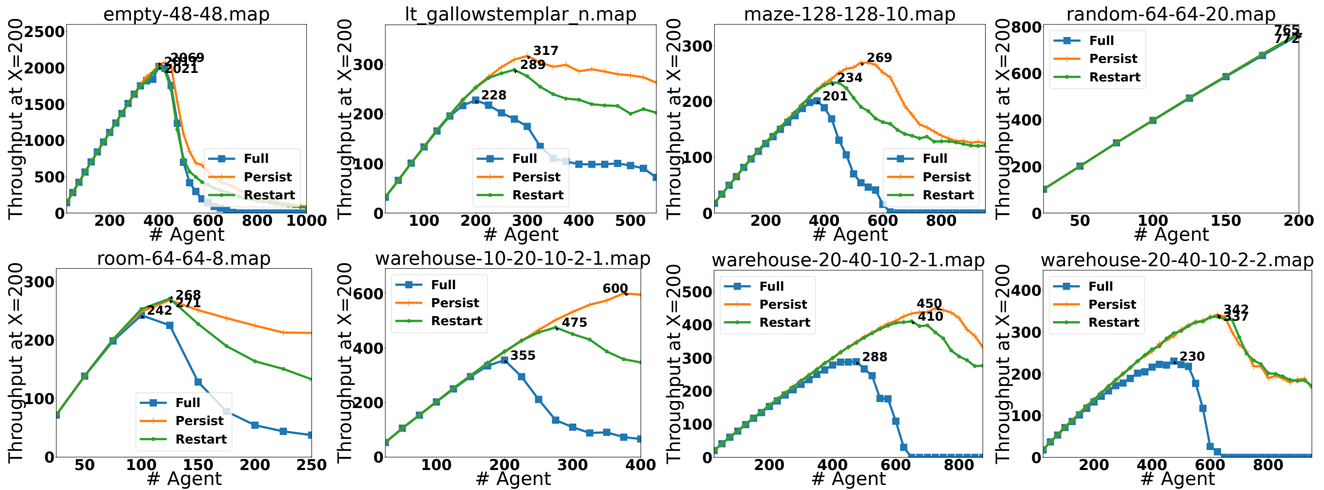

Figure 4: Throughput achieved by using different methods for finding partial solutions, as a factor of the number of agents.

IAvoid achieved a maximal throughput of 317 with 300 agents, whereas IStay achieved 306 (3.4% less), also with 300 agents, and AllStay achieved only 211 (33.4% less than IAvoid) with 175 agents. As expected, IAvoid achieves a higher maximal throughput than IStay, but the observed difference is small. These results are reasonable, as IAvoid, despite being more flexible than IStay, is still quite limited. We think these results highlight fail policies as a possible avenue for continued research.

4.4 Ablation Study

Figure 5 shows a comparison of the throughput achieved by different combinations of agent selection policy and planner. The fail policy here was set to IAvoid. For each component, we considered the best option and the baseline option. That is, for agent selection policies we considered Fail@LH(5) and AllAgents, and for the planners we considered Persist and Full. Generally, Fail@LH(5)+Persist had the highest throughput, followed by Persist and then Fail@LH(5), with AllAgents+Full having the lowest throughput. For example, on map warehouse-10-20-10-2-1, maximal throughput was 600 for Fail@LH(5)+Persist, 536 for Persist, 355 for Fail@LH(5), and 336 for AllAgents+Full. On maps random-64-64-20, and empty-48-48, Fail@LH(5)+Persist achieved similar throughput to that of Persist, following the trends observed in figure 3. AllAgents+Full always had the lowest throughput, except for on random-64-64-20 where it had equivalent throughput. This correlates with the trends seen in Figures 3 and 4, and in Table 2. Persist usually achieved higher throughput than AllAgents+Full or Fail@LH(5). Thus, our results suggest that combining the best performing versions of our selection policies, planners, and fail policies, yielded the most effective configuration for solving LMAPF problems.

5 Related Work

Lifelong MAPD for Online Pickup and Delivery Tasks (MAPD) (Ma et al. 2017) is a similar lifelong problem,

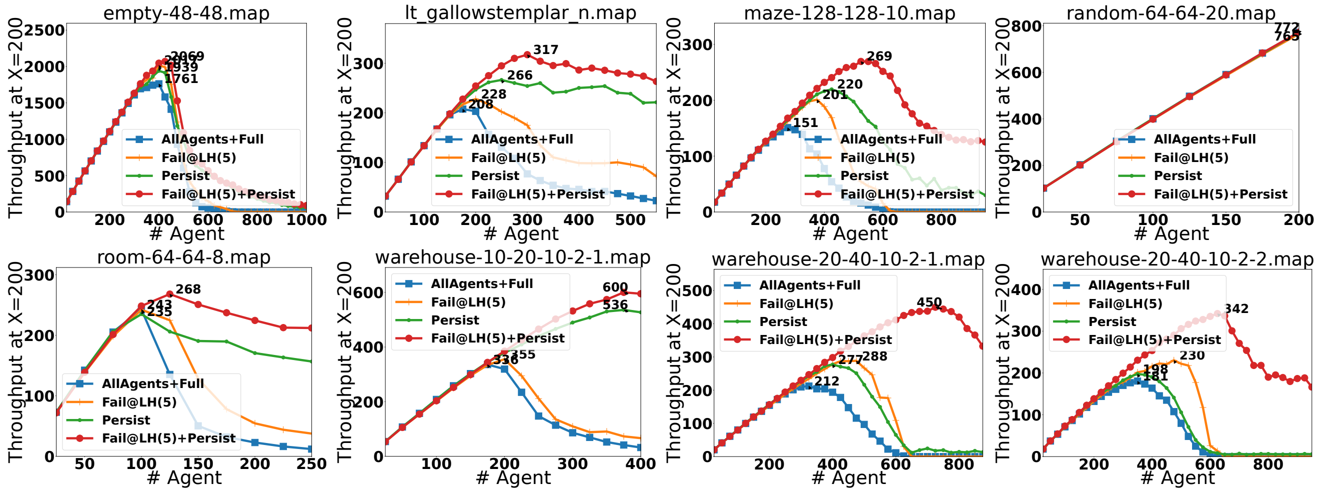

Figure 5: Throughput achieved by the addition of different improvements, as a factor of the number of agents.

	IAvoid	IStay	AllStay
empty-48-48	**2069(425)**	2060(425)	1945(400)
lt_gallowstemplar_n	**317(300)**	306(300)	211(175)
maze-128-128-10	**269(525)**	255(500)	172(325)
random-64-64-20	**772(200)**	769(200)	752(200)
room-64-64-8	**268(125)**	267(125)	209(100)
warehouse-10-20-10-2-1	**600(375)**	598(400)	282(175)
warehouse-20-40-10-2-1	**449(725)**	437(650)	197(425)
warehouse-20-40-10-2-2	**342(625)**	329(625)	227(475)

Table 2: The maximal throughput achieved by the different fail policies, on each map.

where agents are always given paired tasks, where agents must first perform a "pickup" at a certain location, and then deliver it at another. These tasks are treated as a single stream of tasks, from which tasks may be assigned to idle agents, and they incorporate this problem of task assignment into their algorithm. That work does not consider that the planning time per time step may be strictly limited, and the failure that may happen as a result. Li et al. (2021b) proposes to apply Bounded-Horizon Planing (Silver 2005) when solving MAPD problems. Additionally, they consider that it may be possible to know of some future tasks. They show how their suggested approach may be implemented by using several different MAPF algorithms, and demonstrate the advantages of this approach in increasing the number of agents that may participate. Xu et al. (2022) continues this work by incorporating techniques from state-of-the-art MAPF solvers to reduce run time and increase throughput.

Online MAPF (Švancara et al. 2019; Morag et al. 2022) is another online version of MAPF, where each agent receives only one task to perform, but agents disappear after completing their task, and new agents appear over time.

Sigurdson et al. (2018) suggests an algorithm for real-time search in the context of MAPF, with a focus on the application of navigation in video games. They suggested a more decoupled approach in which each agent plans for itself.

MAPF-LNS (Li et al. 2021a) is a recent framework for solving classical MAPF problems. Similarly to our approach, this algorithm considers selecting subsets of the agents in the problem to plan for at any one time, in order to increase the quality of a MAPF solution. *MAPF-LNS2* (Li et al. 2022) extends this approach further by focusing on reducing the number of conflicts in an invalid solution, to quickly find a solution of any quality. They did not consider planning failures, and have been proposed for classical MAPF as opposed to LMAPF.

6 Conclusions & Future Work

In this work, we consider the Lifelong Multi-Agent Path Finding (LMAPF) problem in a system where planning is done online, and the length of time for planning paths is strictly limited. In such a system, planning may fail to find a full solution, yet the agents must perform some actions. We propose a system design for LMAPF algorithms in such systems, which is designed to handle planning failures. We explored different ways to implement key elements of this design, namely selecting which agents to plan for, how to plan for them, and how to handle planning failures. To this end, we introduced the concepts of a partial solution and a fail policy, which provides a way to maintain safety (avoid collisions) while advancing towards the targets. We compared our methods experimentally on a varied benchmark, and showed that using them dramatically increases the throughput achieved. We intend to continue this line of research by considering more advanced fail policies, particularly by optimizing the order in which they are applied to agents, by combining them with methods from Real-Time Search and procedural MAPF algorithms, and by considering longer sequences of actions that can be used to avoid conflicts.

Acknowledgements

This research was partially funded by Get Fabric, Inc.

References

Bennewitz, M.; Burgard, W.; and Thrun, S. 2001. Optimizing schedules for prioritized path planning of multi-robot systems. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 1, 271–276. IEEE.

Čáp, M.; Vokřínek, J.; and Kleiner, A. 2015. Complete decentralized method for on-line multi-robot trajectory planning in well-formed infrastructures. In *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 25, 324–332.

Dresner, K.; and Stone, P. 2008. A multiagent approach to autonomous intersection management. *Journal of artificial intelligence research*, 31: 591–656.

Felner, A.; Stern, R.; Shimony, S. E.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N.; Wagner, G.; and Surynek, P. 2017. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *Tenth Annual Symposium on Combinatorial Search*.

Ho, F.; Geraldes, R.; Goncalves, A.; Rigault, B.; Oosedo, A.; Cavazza, M.; and Prendinger, H. 2019. Pre-flight conflict detection and resolution for UAV integration in shared airspace: Sendai 2030 model case. *IEEE Access*, 7: 170226–170237.

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

Lam, E.; Le Bodic, P.; Harabor, D.; and Stuckey, P. J. 2022. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research*, 144: 105809.

Latombe, J.-C. 1991. Multiple Moving Objects. In *Robot motion planning*, 1–57. Springer.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P.; and Koenig, S. 2021a. Anytime multi-agent path finding via large neighborhood search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

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 *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, 10256–10265.

Li, J.; Gong, M.; Liang, Z.; Liu, W.; Tong, Z.; Yi, L.; Morris, R.; Pasearanu, C.; and Koenig, S. 2019. Departure scheduling and taxiway path planning under uncertainty. In *AIAA Aviation 2019 Forum*, 2930.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. S.; and Koenig, S. 2021b. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 11272–11281.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with consistent prioritization for multi-agent path finding. In *AAAI Conference on Artificial Intelligence*, volume 33, 7643–7650.

Ma, H.; Li, J.; Kumar, T.; and Koenig, S. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *International Conference on Autonomous Agents and Multiagent Systems*.

Morag, J.; Felner, A.; Stern, R.; Atzmon, D.; and Boyarski, E. 2022. Online Multi-Agent Path Finding: New Results. In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 229–233.

Salzman, O.; and Stern, R. 2020. Research challenges and opportunities in multi-agent path finding and multi-agent pickup and delivery problems. In *Proceedings of the 19th International Conference on Autonomous Agents and Multi-Agent Systems*, 1711–1715.

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

Sigurdson, D.; Bulitko, V.; Yeoh, W.; Hernández, C.; and Koenig, S. 2018. Multi-agent pathfinding with real-time heuristic search. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Silver, D. 2005. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, volume 1, 117–122.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. S.; et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Symposium on Combinatorial Search (SoCS)*.

Švancara, J.; Vlk, M.; Stern, R.; Atzmon, D.; and Barták, R. 2019. Online multi-agent pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7732–7739.

Wan, Q.; Gu, C.; Sun, S.; Chen, M.; Huang, H.; and Jia, X. 2018. Lifelong multi-agent path finding in a dynamic environment. In *International Conference on Control, Automation, Robotics and Vision (ICARCV)*, 875–882.

Xu, Q.; Li, J.; Koenig, S.; and Ma, H. 2022. Multi-goal multi-agent pickup and delivery. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9964–9971. IEEE.