doi:10.1017/S1471068425100276 First published online 12 September 2025

A General Framework for Dynamic MAPF Using Multi-Shot ASP and Tunnels[*]

AYSU BOGATARKAN and ESRA ERDEM
Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkiye
(*e-mails:* aysubogatarkan@sabanciuniv.edu, esraerdem@sabanciuniv.edu)

submitted 28 July 2025; revised 28 July 2025; accepted 29 July 2025

Abstract

The multi-agent path finding (MAPF) problem aims to find plans for multiple agents in an environment within a given time, such that the agents do not collide with each other or obstacles. Motivated by the execution and monitoring of these plans, we study dynamic MAPF (D-MAPF) problem, which allows changes such as agents entering/leaving the environment or obstacles being removed/moved. Considering the requirements of real-world applications in warehouses with the presence of humans, we introduce (1) a general definition for D-MAPF (applicable to variations of D-MAPF), (2) a new framework to solve D-MAPF (utilizing multi-shot computation and allowing different methods to solve D-MAPF), and (3) a new answer set programming-based method to solve D-MAPF (combining advantages of replanning and repairing methods, with a novel concept of tunnels to specify where agents can move). We have illustrated the strengths and weaknesses of this method by experimental evaluations, from the perspectives of computational performance and quality of solutions.

KEYWORDS: multi-agent path finding, answer set programming, multi-shot computation

1 Introduction

The multi-agent path finding (MAPF) problem aims to find plans for multiple agents in a shared environment such that the agents do not collide with each other or obstacles, subject to constraints on the total/maximum plan length. The dynamic MAPF (D-MAPF) problem considers the changes that occur in the environment or in the team during the execution of a MAPF plan (e.g., new agents joining the team, existing agents leaving the environment, new obstacles being added, existing obstacles being removed or moved) and aims to find a new plan for agents to reach their goals.

The existing approaches to solving D-MAPF consider different objective functions (e.g., minimizing the makespan or the sum of costs) and changes (e.g., team or environment changes), make different assumptions on entrances and departures of agents (e.g., agents

[*] We would like to thank Volkan Patoglu for the useful discussions about the real-world applications of MAPF, as well as for the helpful feedback and support. We also thank the anonymous reviewers for their valuable comments.

immediately or later appear at their initial locations, agents stay or disappear when they reach their destinations), and investigate different methods to solve D-MAPF.

For instance, Lifelong MAPF (Wan *et al.* 2018; Li *et al.* 2021) considers assignment of new goal locations to the agents who have completed their plans, which can be viewed as adding a new agent. An incremental version of Conflict-Based Search (CBS) algorithm was introduced by Wan *et al.* (2018) such that solutions for new agents are computed while re-using the solutions for the existing agents and they are adjusted only if needed at each time step. The idea of Windowed MAPF was used by Li *et al.* (2021), where the instance is splitted into a sequence of instances, replanning is done periodically after a specific number of steps, and the collisions are resolved only for the steps within a given window.

Online MAPF (Svancara *et al.* 2019) considers the addition of new agents as a dynamic change and allows the agents to appear from a garage after some waiting and disappear from the environment once it reaches its goal. Different methods were introduced for solving this problem with SAT-based and search-based solvers, and these methods are further investigated by Morag *et al.* (2022) in terms of their quality. One of the methods introduced for solving Online MAPF is the *Replan-All* method that discards the existing plan and re-solves MAPF for all agents considering the changes. Ma (2021) investigates Online MAPF problem theoretically, gives complexity results, and provides a classification for online MAPF algorithms. Ho *et al.* (2019) consider an online and three-dimensional setting, with heterogeneous agents being added during execution and solves the problem using modified versions of CBS and Enhanced CBS (ECBS) algorithms.

Different from these studies, in terms of changes, Bogatarkan *et al.* (2019) consider leaving agents and changes in obstacles as well. Their *Revise-and-Augment* method reuses the existing plan: when a change occurs, the plans of existing agents are revised by rescheduling their waiting times, while plans are computed for the new agents. Atiq *et al.* (2020) consider a similar problem. When a change occurs in the environment, a minimal subset of agents having conflicts is identified, and replanning is applied to resolve conflicts.

In this study, we investigate D-MAPF further with the following motivations and theoretical and practical contributions.

- To be able to study rich variations of D-MAPF mentioned above, on a common ground, we introduce a rigorous definition for D-MAPF, which is general enough to cover (1) various changes in the environment and the team of agents over time, (2) different objective functions on plans, and (3) different assumptions on appearances/disappearances of agents, and that is not specifically oriented toward a particular method.
- We introduce a new framework to solve D-MAPF, which is general and flexible enough to allow different replanning and/or repairing methods. With the motivation of a modular architecture and efficient computations, our framework utilizes multi-shot computation (Gebser *et al.* 2019) based on answer set programming (ASP) (Gelfond and Lifschitz 1991; Lifschitz 2008). Recall that multi-shot solving allows changes to the input ASP program in time, by introducing an external control to the ASP system. The external control allows operations, such as adding and grounding new programs, assigning truth values of some atoms, and solving the updated program, while the ASP system is running. Our earlier studies (Bogatarkan *et al.* 2019) utilize single-shot computation.

• We design and implement the Replan-All and Revise-and-Augment methods using multi-shot ASP, and integrate them in the general D-MAPF framework. We empirically observe that multi-shot Replan-All is computationally more efficient but sometimes dramatic changes in the paths of the existing agents occur in the recomputed plans. Such changes are not desired from the perspective of real-world applications. For instance, in a warehouse where robots collaborate with human workers, changes in the routes of robots might be unexpected, distracting, unsafe, and inefficient for human workers.

• We introduce a new method for D-MAPF, called *Revise-and-Augment-in-Tunnels*, which combines the advantages of these two methods. Unlike revise-augment, this method does not require that every existing agent follow their existing paths while revising their plans. Instead, (1) it creates a "tunnel" for each existing agent that consists of the agent's existing path and the neighboring locations within a specified "width," and (2) it allows every existing agent to follow a path within their own tunnel while it revises their plans. So the existing agents do not have to follow their previously computed paths. While revising the plans of existing agents within their tunnels, (3) the Revise-and-Augment-in-Tunnels method computes plans for the new agents and augments these plans with the revised plans, respecting the collision constraints. Note that as the tunnel width gets larger (resp. smaller), the Revise-and-Augment-in-Tunnels method gets closer to the Replan-All method (resp. the Revise-and-Augment method).

• We implement the Revise-and-Augment-in-Tunnels method using multi-shot ASP and integrate it into our D-MAPF framework. We design and perform experiments to better understand the strengths and the weaknesses of this new method, considering computational performance (in time) and quality of solutions (in terms of plan changes).

2 Preliminaries: paths, traversals, plans

We introduce a more general definition for MAPF problem, compared to the earlier definitions (Erdem *et al.* 2013; Stern *et al.* 2019), so that it allows us to explicitly state our assumptions and consider different cost functions depending on the particular application.

Let us first introduce the relevant concepts and notations. Consider an undirected graph $G=(V,E)$. A path $P=\langle v_0,\dots,v_n\rangle$ in G is a sequence of vertices $v_i \in V$ such that, for every $\langle v_i, v_{i+1}\rangle$ in P, there exists an edge $\{v_i, v_{i+1}\} \in E$.

Let A be a set of agents. Every agent $a_i \in A$ is characterized by an initial location $init_i \in V$, a goal location $goal_i \in V$, and a joining time $join_i$ ($join_i \geq 0$).

For every agent a_i and every path P_i, a *traversal* $r_{a_i}^{P_i}$ of path P_i by agent a_i is characterized by a starting time x_i ($x_i \geq 0$), an ending time y_i ($y_i \geq x_i$), and a function f_i that maps every integer t ($x_i \leq t \leq y_i$) to a vertex in P_i, such that, for every v_k, v_{k+1} in P_i and, for every t, if $f_i(t)=v_k$ then $f_i(t+1)=v_k$ or $f_i(t+1)=v_{k+1}$. We denote by $\mathbf{P}_A$ the collection of paths for every agent in A, and by $\mathbf{r}_A^{\mathbf{P}}$ the collection of traversals $r_{a_i}^{P_i}=\langle x_i, y_i, f_i\rangle$ of every agent $a_i \in A$.

For two agents a_i, a_j with traversals $r_{a_i}^{P_i}=\langle x_i, y_i, f_i\rangle$ and $r_{a_j}^{P_j}=\langle x_j, y_j, f_j\rangle$, we say that a_i and a_j collide at a vertex at time t, if for some t where $\max(x_i,x_j) \leq t \leq \min(y_i,y_j)$, $f_i(t)=f_j(t)$. We say that a_i and a_j collide at an edge between times t and $t+1$, if for some t where $\max(x_i,x_j) \leq t < \min(y_i,y_j)$, $f_i(t)=f_j(t+1)$ and $f_i(t+1)=f_j(t)$. These types of collisions are called *vertex conflicts* and *swapping conflicts*, respectively.

MAPF Problem
Input:
A nonempty set $A = \{a_1, \ldots, a_n\}$ of agents, a graph $G = (V, E)$, a set $O \subseteq V$ denoting the obstacles, a function *cost* that maps a set of traversals to a nonnegative integer, and a positive integer τ that specifies an upper bound on the value of *cost*.
Output: For every agent $a_i \in A$,

- a path $P_i = \langle v_{i,0}, \ldots, v_{i,l_i} \rangle$ of finite length l_i that the agent a_i follows to reach its goal ($v_{i,l_i} = goal_i$) from its initial location ($v_{i,0} = init_i$), without colliding with obstacles (i.e., $v_{i,j} \in V \setminus O$), and
- a traversal $r^{P_i}_{a_i} = \langle 0, y_i, f_i \rangle$, such that, for every other agent $a_j \in A$ with a traversal $r^{P_j}_{a_j} = \langle 0, y_j, f_j \rangle$ and for every time t, agents a_i and a_j have neither a vertex conflict at time t nor a swapping conflict between times t and $t+1$,

such that $cost(\mathbf{r}^{\mathbf{P}}_A) \leq \tau$.

Fig. 1. A general MAPF problem definition.

Using these concepts, we introduce a MAPF problem definition, as in Figure 1. A MAPF instance is characterized by the quintuple $\langle A, G, O, cost, \tau \rangle$, and its solution (called a MAPF plan) with a collection $\mathbf{r}^{\mathbf{P}}_A$ of traversals.

Remarks: appearances/disappearances. This MAPF definition is more general than the earlier definitions, as it allows us to explicitly state our assumptions about the appearances/disappearances of agents at their goal/initial locations.

For instance, consider the following two possible behaviors of agents once they reach their goals: (1) every agent waits at its goal until the traversals of all agents are completed (as in our earlier studies), or (2) every agent disappears from the environment (as in Svancara *et al.* (2019)). Both cases can be covered with this MAPF definition. Suppose that $reach_i$ ($x_i \leq reach_i \leq y_i$) denotes the time step at which an agent a_i reaches its goal. If we assume that the agent a_i disappears when it reaches its goal, then $y_i = reach_i$. If we assume that the agent a_i waits at goal until all traversals end, then $reach_i$ is the time step where the agent reaches its goal and stays there.

Similarly, consider the following two different behaviors of agents at their initial location: (1) every agent appears at their initial locations, at the time step the agent joins the team (as in our earlier studies), or (2) the agents that are joining the team are allowed to wait outside the environment until their initial locations are unoccupied (as in Svancara *et al.* (2019)). In the former case, for agent a_i, the starting time x_i of its traversal is the same as its joining time $join_i$. In the latter case, for agent a_i, the joining time $join_i$ is the time at which the agent joins the team, and the starting time x_i of its traversal is the time at which the agent enters the environment.

Remarks: cost functions. Furthermore, this definition of MAPF also allows different cost functions depending on the needs of the particular application.

For an agent a_i with traversal $r^{P_i}_{a_i} = \langle x_i, y_i, f_i \rangle$ and for a time step t, let us first define the *cost of waiting* ($cost^i_w(t)$), and the *cost of moving* to another vertex ($cost^i_m(t)$):

- $cost^i_w(t) = \begin{cases} 1 & \text{if } f_i(t) = f_i(t+1) \text{ for } x_i \leq t < y_i \\ 0 & \text{otherwise} \end{cases}$
- $cost^i_m(t) = \begin{cases} 1 & \text{if } f_i(t) \neq f_i(t+1) \text{ for } x_i \leq t < y_i \\ 0 & \text{otherwise} \end{cases}$

Then we can define the *length of its traversal* and the *cost of its traversal* $r_{a_i}^{P_i} = \langle x_i, y_i, f_i \rangle$ (considering the completion of the task, not the waiting afterward):

- $cost_L^i(r_{a_i}^{P_i}) = \sum_{t=x_i}^{y_i} cost_w^i(t) + cost_m^i(t)$ • $cost_T^i(r_{a_i}^{P_i}) = \sum_{t=x_i}^{reach_i} cost_w^i(t) + cost_m^i(t)$

When the distance traveled by an agent a_i is more important than the time spent by the agent, we can consider the *cost of its path* P_i: $cost_P^i(r_{a_i}^{P_i}) = \sum_{t=x_i}^{y_i} cost_m^i(t) = |P_i|$.

Now the *sum of costs* of a MAPF plan $\mathbf{r}_A^{\mathbf{P}}$ (i.e., the total time spent by all of the agents until they reach their goals), and the *makespan* of a MAPF plan $\mathbf{r}_A^{\mathbf{P}}$ (i.e., the time step where all agents in A reach their goals) are defined as follows:

- $cost_{SOC}(\mathbf{r}_A^{\mathbf{P}}) = \sum_{a_i \in A} cost_T^i(r_{a_i}^{P_i})$ • $cost_M(\mathbf{r}_A^{\mathbf{P}}) = max(cost_T^i(r_{a_i}^{P_i}))$.

The total distance $cost_{SOP}(\mathbf{r}_A^{\mathbf{P}})$ traveled by the agents (that is the *sum of path lengths* of a MAPF plan $\mathbf{r}_A^{\mathbf{P}}$) is defined similarly, by adding up $cost_P^i$.

For more details, please see Appendix A of the supplementary material accompanying the paper at TPLP archive.

3 D-MAPF: problem definition

Consider a given MAPF instance $\langle A, G, O, cost, \tau \rangle$, and its solution (a MAPF plan) $\mathbf{r}_A^{\mathbf{P}}$. D-MAPF considers changes in the environment or in the team while such a given MAPF plan is being executed. We describe these changes by means of events, defined as follows.

An *event* e at a time t ($t \geq 0$) is characterized by a tuple $\langle A_t{\uparrow}, A_t{\downarrow}, O_t{\uparrow}, O_t{\downarrow}, t \rangle$, where

- $A_t{\uparrow}$ (resp. $A_t{\downarrow}$) denotes the set of agents leaving (resp. joining) at time t,
- $O_t{\uparrow}$ (resp. $O_t{\downarrow}$) denotes the set of obstacles removed (resp. added) at time t.

A sequence $C = \langle e_0, e_1, \dots, e_m \rangle$ of events leads to changes if, for every $e_k \in C$, at least one of the sets $A_t{\uparrow}, A_t{\downarrow}, O_t{\uparrow}, O_t{\downarrow}$ is nonempty.

Given a sequence $C = \langle e_0, e_1, \dots, e_m \rangle$ of events, the set A_t^C of agents who are present in the environment at time t is defined as follows:

$$A_t^C = \begin{cases} A & \text{if } t = 0, \text{ and } e_0 \text{ occurs at some time } z > 0 \\ (A \setminus A_t{\uparrow}) \cup A_t{\downarrow} & \text{if } t = 0, \text{ and } e_0 \text{ occurs at time } 0 \\ A_{t-1}^C & \text{if } t > 0, \text{ and no event } e_i (i > 0) \text{ in } C \text{ occurs at time } t \\ (A_{t-1}^C \setminus A_t{\uparrow}) \cup A_t{\downarrow} & \text{if } t > 0, \text{ and some event } e_i (i > 0) \text{ in } C \text{ occurs at time } t \end{cases}$$

Given a sequence of events $C = \langle e_0, e_1, \dots, e_m \rangle$, the set O_t^C of vertices covered by the obstacles present in the environment at time t is defined in a similar way.

Let us denote by $A_{\leq t}^C$ the set of agents that were present in the environment before or at some time t, as a sequence C of events takes place.

We say that a sequence $C = \langle e_0, e_1, \dots, e_m \rangle$ of events is *valid* if the following hold: (i) for the event $e_0 = \langle A_t{\uparrow}, A_t{\downarrow}, O_t{\uparrow}, O_t{\downarrow}, t \rangle$,

- $A_t{\uparrow} \subseteq A$, (i.e., leaving agents are present in the environment initially),
- $A_t{\downarrow} \cap A = \emptyset$ (i.e., joining agents are not already in the environment),

D-MAPF Problem
Input:

- A MAPF instance $\langle A, G, O, cost, \tau \rangle$ and its solution $\mathbf{r}^{\mathbf{P}}_{A}$.
- A function $cost'$ that maps a set of traversals to a nonnegative integer.
- A positive integer τ' (an upper bound on the $cost'$ value).
- A valid sequence C of events.
- A positive integer α (an upper bound on the ending times of traversals)

Output: For every agent $a_i \in A^C_{\leq \alpha}$,

- a path $P'_i = \langle v_{i,0}, \ldots, v_{i,l_i} \rangle$ of length l_i where $v_{i,0} = init_i$ and $v_{i,l_i} = goal_i$, and
- a traversal $r^{P'_i}_{a_i} = \langle x_i, y_i, f_i \rangle$, where $x_i \geq join_i$, such that
  - x_i=0 if $a_i \in A$,
  - $f_i(t) \notin O^C_t$ for all t ($x_i \leq t \leq y_i$) (i.e., no collisions with an obstacle), and
  - for every other agent $a_j \in A^C_{\leq \alpha}$ with $r^{P'_j}_{a_j} = \langle x_j, y_j, f_j \rangle$ where $x_j \geq join_j$ and $y_j \leq \alpha$, and for every t where $a_i, a_j \in A^C_t$, the agents a_i and a_j have neither a vertex conflict at time t nor a swapping conflict between times t and $t+1$,

such that $cost'(\mathbf{r}^{\mathbf{P}'}_{A^C_{\leq \alpha}}) \leq \tau'$.

Fig. 2. A general definition for D-MAPF problem.

- $O_t{\uparrow} \subseteq O$, (i.e., removed obstacles are in the environment initially),
- $O_t{\downarrow} \cap (O \setminus O_t{\uparrow}) = \emptyset$ (i.e., new obstacles are different from the obstacles that are already in the environment), and

(ii) for every event $e_k = \langle A_t{\uparrow}, A_t{\downarrow}, O_t{\uparrow}, O_t{\downarrow}, t \rangle$ $(0 < k \leq m)$,

- $A_t{\uparrow} \subseteq A^C_{t-1}$ (i.e., agents leaving at time t are present there at time $t{-}1$),
- $A_t{\downarrow} \cap A^C_{\leq t-1} = \emptyset$ (i.e., agents joining at time t have never been in the environment previously (with the same id)),
- $O_t{\uparrow} \subseteq O^C_{t-1}$ (i.e., obstacles removed at time t are in the environment at time $t{-}1$),
- $O_t{\downarrow} \cap (O^C_{t-1} \setminus O_t{\uparrow}) = \emptyset$ (i.e., new obstacles added at time t are not already present in the environment), and

(iii) for every two events $e_k = \langle A_t{\uparrow}, A_t{\downarrow}, O_t{\uparrow}, O_t{\downarrow}, t \rangle$ and $e_{k+1} = \langle A_z{\uparrow}, A_z{\downarrow}, O_z{\uparrow}, O_z{\downarrow}, z \rangle$ $(0 \leq k < m)$,

- $t < z$ (i.e., event e_k occurs before event e_{k+1}).

Note that, for every event $e_k = \langle A_t{\uparrow}, A_t{\downarrow}, O_t{\uparrow}, O_t{\downarrow}, t \rangle$ in a valid event sequence C, (1) for every agent $a_i \in A_t{\downarrow}$, $t = join_i$, and (2) for every agent $a_j \in A_t{\uparrow}$, $t = y_j$.

Based on the concepts and notation defined above, we introduce a general definition for the D-MAPF problem, as in Figure 2.

Remarks. Our D-MAPF definition can be easily extended to include different constraints, as needed by applications. For instance, during the applications of our studies at warehouses with mobile robots, we have observed the need for another constraint that prevents conflicts due to robots following each other too closely. For those studies, we have defined *following conflicts* as follows, and extended the D-MAPF definition accordingly: For two agents a_i, a_j with traversals $r^{P_i}_{a_i} = \langle x_i, y_i, f_i \rangle$ and $r^{P_j}_{a_j} = \langle x_j, y_j, f_j \rangle$, respectively, a *following conflict* occurs between a_i and a_j between times t and $t+1$, where $\max(x_i, x_j) \leq t < \min(y_i, y_j)$, if $f_i(t) = f_j(t+1)$.

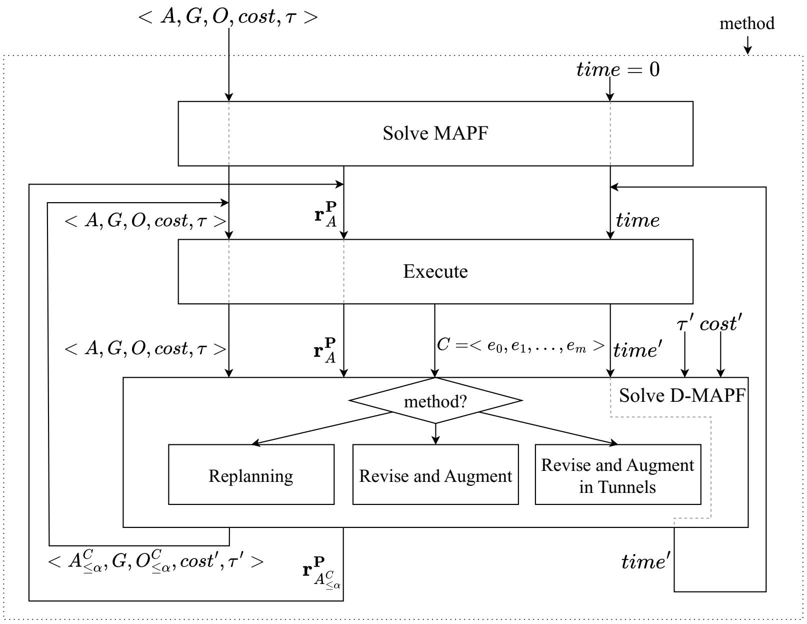

$< A, G, O, cost, \tau >$
method
$time = 0$
Solve MAPF
$< A, G, O, cost, \tau >$
$\mathbf{r}^{\mathbf{P}}_A$
$time$
Execute
$< A, G, O, cost, \tau >$
$\mathbf{r}^{\mathbf{P}}_A$
$C =< e_0, e_1, \ldots, e_m >$
$time'$
τ' $cost'$
Solve D-MAPF
method?
Replanning
Revise and Augment
Revise and Augment in Tunnels
$< A^C_{\leq\alpha}, G, O^C_{\leq\alpha}, cost', \tau' >$
$\mathbf{r}^{\mathbf{P}}_{A^C_{\leq\alpha}}$
$time'$

Fig. 3. Overall architecture of a general framework for D-MAPF.

4 A general framework for solving D-MAPF problems

Overall architecture. We introduce a general framework for D-MAPF problems, based on multi-shot computation, and utilizing one of specified methods to solve D-MAPF. Figure 3 shows an overall architecture of this framework, considering online execution of the computed plans.

In an online execution, the changes that will occur in the environment are not known before the start of the execution. To detect the changes, we consider the environment being monitored by a central agent. When this central agent detects a change in the environment during the execution of a computed plan, a new plan is computed, and the agents start executing the new plan.

Our algorithm takes a MAPF instance $\langle A, G, O, cost, \tau \rangle$ and a method to solve D-MAPF as input.

1. Initially, a MAPF solution is computed for this instance, starting from time 0, shown with the "Solve MAPF" block.
2. Then, the computed solution $\mathbf{r}^{\mathbf{P}}_A$ is executed step by step until a change is detected by the central agent at time $time'$. This execution process is shown with the "Execute" block.
3. The instance $\langle A, G, O, cost, \tau \rangle$ and its solution $\mathbf{r}^{\mathbf{P}}_A$ being executed are the inputs for the next step, "Solve D-MAPF," as well as a sequence of events $C = \langle e_0, e_1, \ldots, e_m \rangle$, the cost function $cost'$ and its upper bound τ'. In this part, D-MAPF problem is solved using the method specified at the beginning of the algorithm.

4. Once a D-MAPF solution $\mathbf{r}^{\mathbf{P}}_{A^C_{\leq \alpha}}$ is found and the current instance is updated as $< A^C_{\leq \alpha}, G, O^C_{\leq \alpha}, cost', \tau' >$, the execution continues from time step $time'$ with the "Execute" block. The execution continues until the end of the computed plan.

Multi-shot solving. For the solving processes "Solve MAPF" and "Solve D-MAPF" blocks, we utilize multi-shot ASP (Gebser *et al.* 2019).

Multi-shot solving aims to handle continuously changing logic programs. A multi-shot ASP program is able to grow and be updated with the changing knowledge to solve a problem. A multi-shot ASP program considers a program splitted into multiple parts. The inclusion of these subprograms into the solving process is maintained by a controller program from the outside. The subprograms generally serve different purposes and can be grouped as: static parts that are grounded once and not changed throughout the program (usually called the `base` program), cumulative parts that can be added multiple times with different parameter values, such as time step in dynamic domains (usually called the `step` program), and volatile parts that are added for a step and removed in the next step, checking whether the stopping condition is satisfied (usually called the `check` program). Enabling and disabling rules in the `check` program at different steps is done with `external` atoms, through the outside controller program.

In our algorithm for solving D-MAPF problem, the `base` program contains the MAPF instance and the initial step of the traversal. The `step` program has a parameter `t`, denoting the time step of the plan, and it generates the plan recursively for time step `t` and adds the collision constraints for time `t`. The `check` program also has a parameter `t` and an external atom called `query` for enabling/disabling the rules in this program. This program verifies whether every agent reached their goals at time `t`. These programs are depicted on the left side of Figure 4 and explained in more detail in Appendix B of the supplementary material accompanying the paper at TPLP archive.

Solving MAPF, using multi-shot ASP. "Solve MAPF" procedure utilizes the programs and solves the problem following these steps:

1. The controller program starts with time step `t = 0`, grounds the `base` program and `check(0)` program, assigns the external atom `query(0)` and solves the current program, if the solver does not return a value (SAT or UNSAT), continues with step 2. Otherwise the computation ends.
2. Time step value `t` is increased by 1, the external atom `query(t-1)` is released and `step(t)` and `check(t)` are grounded, the external atom `query(t)` is assigned and the current program is solved.
3. If the solver returns a value or the makespan limit is reached, solving ends and the solution is returned, otherwise step 2 is repeated.

The solution found in "Solve MAPF" is passed to the "Execute" block and executed step by step until a change is detected by the central agent and D-MAPF is solved. The visualization of these steps of solving can be found in Figure B1 in Appendix B of the supplementary material accompanying the paper at TPLP archive.

Solving D-MAPF, using multi-shot ASP. "Solve D-MAPF" has a similar procedure as described above for "Solve MAPF" but has additional parts for maintaining the new agents. D-MAPF solving uses the existing ground program of MAPF, with the same

```
#program base.
time(0).
plan(A,0,X):- init(A,X), agent(A).

#program step(t).
time(t).
% every agent can stay or move to an adjacent vertex
{plan(A,t,X)}1 :- plan(A,t-1,X), agent(A).
{plan(A,t,Y): edge(X,Y)}1 :- plan(A,t-1,X), agent(A).
% every agent has exactly one location
:- {plan(A,t,X): vertex(X)}0, agent(A).
:- 2{plan(A,t,X): vertex(X)}, agent(A).

% no two agents are at the same vertex
:- 2{plan(A,t,X): agent(A)}, vertex(X), time(t).
% no two agents can swap their locations
plan_to(X,Y,t,0) :- plan(A,t-1,X), plan(A,t,Y),
  edge(X,Y), agent(A).
:- plan_to(X,Y,t,_), plan_to(Y,X,t,_).

% goal is reached, when the goal vertex is visited
goal_reached(A,t) :- goal(A,X), plan(A,t,X), agent(A).
goal_reached(A,t) :- goal_reached(A,t-1).

#program check(t).
#external query(t).
% every agent should reach its goal until time t
:- not goal_reached(A,t), agent(A), query(t).

% last step of the plan of each agent is its goal
% (remark: agents are allowed to leave their goals
% and then come back at the end)
:- not goal(A,X), plan(A,t,X), agent(A), query(t).
```

```
#program newAgent(a,i,g,k).
agent(a).
init(a,i). % initial location of a
goal(a,g). % goal location of a

% plan generation until the current makespan
% (init step starts at k, time/1 is defined till makespan)
plan(a,k,X) :- init(a,X), agent(a).
{plan(a,T,X)}1 :- plan(a,T-1,X), agent(a), time(T), T >= k.
{plan(a,T,Y): edge(X,Y)}1 :- plan(a,T-1,X), agent(a),
  time(T), T >= k.
% every agent has exactly one location
:- {plan(a,T,X): vertex(X)}0, agent(a), time(T), T >= k.
:- 2{plan(a,T,X): vertex(X)}, agent(a), time(T), T >= k.

% no two agents are at the same vertex
:- 2{plan(A,T,X): agent(A)}, vertex(X), time(T), T >= k.
% no two agents can swap their locations
plan_to(X,Y,T,k) :- plan(a,T-1,X), plan(a,T,Y),
  edge(X,Y), agent(a), time(T), T>=k.
:- plan_to(X,Y,T,_), plan_to(Y,X,T,_), time(T), T>=k.

% agent a has reached its goal until time T
goal_reached(a,T) :- goal(a,X), plan(a,T,X),
  agent(a), time(T), T>=k.
goal_reached(a,T) :- goal_reached(a,T-1), time(T), T>=k.

% a should reach its goal until the current time
:- not goal_reached(a,T), agent(a), query(T).
:- not goal(a,X), plan(a,T,X), agent(a), query(T).
```

Fig. 4. The multi-shot ASP programs used for solving MAPF and D-MAPF.

controller program. There is an additional program, called `newAgent`, for each of the three methods, presented on the right side of Figure 4. This program has four parameters: the agent `a`, its initial position `i`, its goal position `g`, and the starting time of its traversal `k`. If multiple agents are added at the same time step, then there will be a different `newAgent` program for each agent.

5 A new method for D-MAPF: Revise-and-Augment-in-Tunnels

In this study, we consider the following two existing approaches to solve D-MAPF problems. The *Replan-All* method (as in Svancara *et al.* (2019)) discards the existing MAPF plan and re-solves MAPF for all agents considering the changes in the team. The *Revise-and-Augment* method (as in Bogatarkan *et al.* (2019)), on the other hand, reuses the existing plan: when a change occurs, the plans of existing agents are revised by rescheduling their waiting times, while plans are computed for the newly joining agents. In this approach, while revising plans, the order of vertices that the agent visits in the MAPF solution is preserved in the D-MAPF solution for every existing agent.

We also introduce a new approach (called *Revise-and-Augment-in-Tunnels*) to solve D-MAPF problems, which combines the advantages of Replan-All and Revise-and-Augment methods. Revise-and-Augment-in-Tunnels aims to reuse the previously computed plans in the spirit of Revise-and-Augment, but in a more relaxed way in the spirit of Replan-All. While revising plans, instead of requiring that every existing agent follow their existing paths only, Revise-and-Augment-in-Tunnels allows each agent to visit

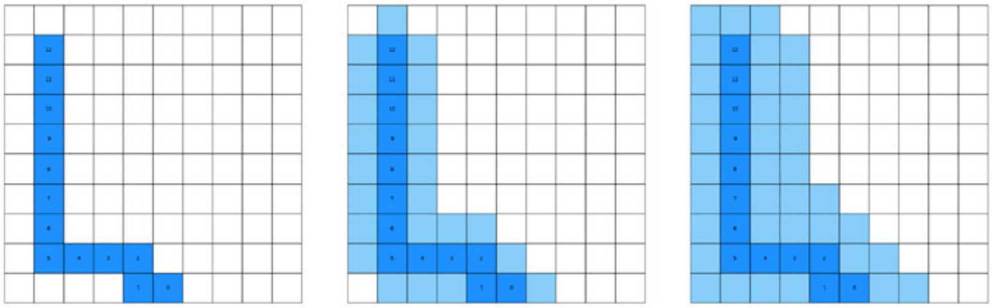


Fig. 5. Tunnels with widths 0 (left), 1 (middle), and 2 (right).

some other vertices neighboring their path. Note that, unlike Revise-and-Augment, this method allows the agents to visit the vertices of their paths in a different order.

Intuitively, each existing agent is allowed to follow some path in a "tunnel" of a specified "width." Such a tunnel consists of the vertices included in the agent's existing path, the neighboring vertices within a Manhattan distance of "width" from the path, and the edges of the graph that connect these vertices. Figure 5 shows sample tunnels with different width values for a sample agent. Note that a tunnel with a zero width contains only the path of the agent and the edges between them.

Formally, the *tunnel* T_i^w of an agent a_i with respect to its path P_i in G, with width $w \geq 0$, is the induced subgraph of the set $\{u : v \in P_i,\ u \in V,\ 0 \leq d_M(u,v) \leq w\}$ of vertices, where d_M denotes the Manhattan distance.

For every agent $a_i \in A$ with a path P_i, Revise-and-Augment-in-Tunnels considers the following *tunnel constraints* while revising plans: every vertex $v_{i,j}$ visited by a_i in the revised plan should be in T_i^w, and, for every $\langle v_{i,j}, v_{i,j+1}\rangle$ followed by a_i in the revised plan, there exists an edge $\{v_{i,j}, v_{i,j+1}\}$ in T_i^w.

Note that, essentially, Revise-and-Augment-in-Tunnels solves D-MAPF extended with such tunnel constraints.

Remarks. To better understand the difference between Revise-and-Augment and Revise-and-Augment-in-Tunnels with tunnel width 0, consider the following example: An agent a_i has a path $P_i = \langle A, B, C, D\rangle$ and its traversal $r_{a_i}^{P_i}$ with $f_i = \langle A, A, B, C, C, D\rangle$. At time $t = 1$, new agents join the environment, and the traversal for a_i will be revised using Revise-and-Augment-in-Tunnels with tunnel width 0. Agent a_i can have a possible revised traversal $r_{a_i}^{P_i'}$ with $f_i = \langle A, A, B, C, B, C, D\rangle$ (if the edges are undirected). This revised traversal is not possible to obtain with the Revise-and-Augment method, since the order of vertices in the path is not allowed to change.

6 Three multi-shot ASP methods for D-MAPF: Replan-All, Revise-and-Augment, and Revise-and-Augment-in-Tunnels

Replan-All, using multi-shot ASP. If the method used for solving D-MAPF is replanning, then only the `newAgent` program and the programs from "Solve MAPF" are used with

```
#program path_constraints(t,k).
#external path_query(t,k). % rescheduling of the plan starts at time k

% identify and count the pairs of different vertices visited consecutively in the plan
plan_pair(A,X,Y,t,k) :- plan(A,T-1,X), plan(A,T,Y), X!=Y, old_agent(A), path_query(t,k).
plan_pair_count(A,X,Y,C1,t,k) :- C1 = #count{X,Y,T: plan(A,T-1,X), plan(A,T,Y), X!=Y},
  plan_pair(A,X,Y,t,k), old_agent(A), path_query(t,k).

% for every vertex pair, its number of occurrences in the plan and in the path should be the same
:- plan_pair_count(A,X,Y,C1,t,k), path_pair_count(A,X,Y,C), C1!=C, old_agent(A), path_query(t,k).

% every vertex appearing in the plan should also appear in the path, and vice versa
:- plan(A,_,X), not path(A,X,_), old_agent(A), path_query(t,k).
:- not plan(A,_,X), path(A,X,_), old_agent(A), path_query(t,k).
```

Fig. 6. The multi-shot ASP program used for solving D-MAPF with Revise-and-Augment.

the following steps, assuming we have an existing ground program until some makespan `m`, and the time of change `k`:

1. For every agent `a` added at time `k`, a `newAgent` program for the agent `a`, its initial and goal locations `i` and `g` and the starting time `k` is grounded. This program recursively computes a plan for the agent starting from time `k` until time `m` and adds the collision constraints for the agent and the goal condition for the agent.
2. Once all the `newAgent` programs are grounded for time `k`, the cumulative ground program containing old and new agents is solved.
3. If the solver returns a solution, it is passed to the execution. Otherwise, the algorithm tries to find a solution with a makespan `m` + `1`. This is done with the same steps in "Solve MAPF." However, instead of starting from `t` = `0`, the procedure starts from `t`=`m` since the time steps from 0 to `m` are already grounded for the old and new agents. Therefore, only steps 2 and 3 in "Solve MAPF" are used.

Since the program does not have any knowledge about the already executed part of the plan, we need to make sure that the existing agents' location at time `k` is the same in the new plan, to avoid them jumping to a disconnected location in the next step. For this purpose, we use the controller program and utilize assumptions of multi-shot *clingo*. We set the truth value of each plan atom in the already executed part of the existing solution to true, making sure that those atoms are included in the answer set in the new solution. This is done after every change in the environment, before starting the procedure of solving D-MAPF.

Revise-and-Augment, using multi-shot ASP. When solving D-MAPF with Revise-and-Augment, in addition to the `newAgent` program and the programs from "Solve MAPF," we have an additional program called `path_constraints` (shown in Figure 6 and explained in more detail in Appendix B of the supplementary material accompanying the paper at TPLP archive). It has two parameters: one of them is the time step `t` that the constraints are being grounded for, and the other one is `k`, the time step of the last change. This program contains an external atom, called `path_query(t,k)` for adding and removing the constraints at each step. When a change is observed during the execution, in addition to the assumptions explained above, we add additional atoms to the *clingo* control, to be used with the `path_constraints` program. These atoms add the knowledge about the order of atoms in the existing paths for the agents and which agents are the relevant ones for preserving the paths. Detailed explanation about these atoms and how they are used in the program can be found in Appendix B. Once the relevant knowledge is added, the following steps are used for computing the revised and augmented solution:

1. `newAgent` programs are grounded same as step 1 of Replan-All.
2. If there is a `path_query` assigned from earlier, it is released. The `path_constraints` program with parameters `m` and `k` is grounded, and its `path_query` atom is assigned with the same parameters.
3. The cumulative program, with all agents and relevant path constraints, is solved.
4. If the solver returns a solution, it is passed to the execution. Otherwise, the algorithm tries to find a solution with a makespan `m + 1`. This is done similarly with "Solve MAPF" but has additional steps for preserving the paths.
   (a) Similar to Replan-All, solving procedure starts with `t=m` from step 2 of "Solve MAPF." Then `t` value is increased by 1, the external atom `query(t-1)` is released, `step(t)` and `check(t)` are grounded, and `query(t)` is assigned.
   (b) Previously used `path_query` is released, the `path_constraints(t,k)` program is grounded and `path_query(t,k)` is assigned. The program containing current step and the constraints for path is solved.
   (c) If the solver returns a value or the makespan limit is reached, solving ends and the solution is returned, otherwise steps (a) and (b) are repeated.

Revise-and-Augment-in-Tunnels, using multi-shot ASP. To solve D-MAPF with this method utilizing multi-shot ASP, one of the following two approaches can be considered.

In the first approach, tunnels are constructed at the plan generation part. In `step` and `newAgent` programs, we update the choice rules for moving to an adjacent vertex:

```
plan(A,t,Y): edge(X,Y), tunnel(A,Y)1 :- plan(A,t-1,X), agent(A).
```

1. Initially, when grounding the `base` program for MAPF, all vertices in the environment are added as part of the tunnel of the agents, with external `tunnel(A,X)` atoms from the controller program, meaning that vertex `X` is in the tunnel of agent `A`. Then, the controller follows the same steps from "Solve MAPF."
2. Once a change in the environment occurs, the paths of the existing agents are extracted from the solution, and the tunnels are computed according to the width value. The tunnel of an existing agent is only computed once, at the first time step when it becomes an old agent after a change in the environment. The external `tunnel` atoms that represent the locations outside of the tunnel of an agent are released. This disables the plan generation rules containing the vertices outside of the tunnel of the agent.
3. After this point, solving process continues the same way with replanning, and, after every change, the tunnels of the agents that become old agents in that time step are updated before moving on with the `newAgent` program.

In the second approach, the locations that are outside an agent's tunnel are considered as forbidden locations for that agent. Accordingly, constraints are added to ensure that no agent visits their forbidden locations in its plan. The procedure for adding these constraints is similar to changes done for the Revise-and-Augment method.

1. The forbidden locations are extracted from the existing solution and added as `forbidden(A,X)` atoms to the program by the controller similar to the `path_order` atoms in Revise-and-Augment.
2. After a change in the environment, a program called `forbidden_locations` containing the constraint of not visiting forbidden vertices is added for every existing agent, if it is not already added before for that agent.
3. The forbidden locations of an agent do not change over time; therefore, there are no query atoms in this program, and it is not added and removed at each step.

7 Experimental evaluations

We evaluate the performance and usefulness of our methods by comparing them with each other, in terms of computation time and quality of solutions. We implement our framework using multi-shot ASP with *clingo* 5.8.0 and Python 3.8.10.
Investigating the computational performance of multi-shot ASP-based D-MAPF methods. With the first set of experiments, our goal is to observe how each of these methods perform in terms of computation time, with different number of agents joining the environment at different times. For this purpose, we generated instances with an empty grid of size 20×20, initially having 20 or 30 agents. We added 5 or 10 new agents to these instances, at different time steps with different group sizes. A total of 25 instances were generated, with 5 different initial and new agent setups.

In these empty-grid instances, the initial locations and the goal locations of the agents are picked such that each initial-goal pair is at the opposite diagonal corners of each other. This setup results with longer plans in empty grids, allowing us to observe the changes in the computation times more easily, since every step added to the plans has an impact on the computation time.

Figure 7 shows the total computation times for 5 different setups of these instances with different methods. Each bar shows the average time of 5 instances with the same setup. For instance, Setup 1 (20 + 5) has initially 20 agents, and 5 agents join at time 1. Setup 5 (30 + 2 + 2 + 2 + 2 + 2) has initially 30 agents; then 2 agents join at time 1, 2 agents join at time 2, 2 agents join at time 3, 2 agents join at time 4, and 2 agents join at time 5.

The computation times consist of two parts, grounding and solving time, taken from *clingo*. We run Revise-and-Augment-in-Tunnels with tunnel constraints (shown as TC) and Revise-and-Augment-in-Tunnels with tunnels in generate (shown as TG) with two different tunnel width values of 0 and 20. Width 0 only uses the vertices of the path of an agent, bringing the solution closer to Revise-and-Augment (shown as R&A), and width 20 covers the whole environment, therefore, the same environment as Replan-All.

Note that in the figure, the computation times greater than 200 s are not displayed exactly. A timeout of 200 s was set for the experiments, and none of the instances was solved with Revise-and-Augment (R&A) within this time limit. The detailed results for each instance and method can be found in Appendix C of the supplementary material

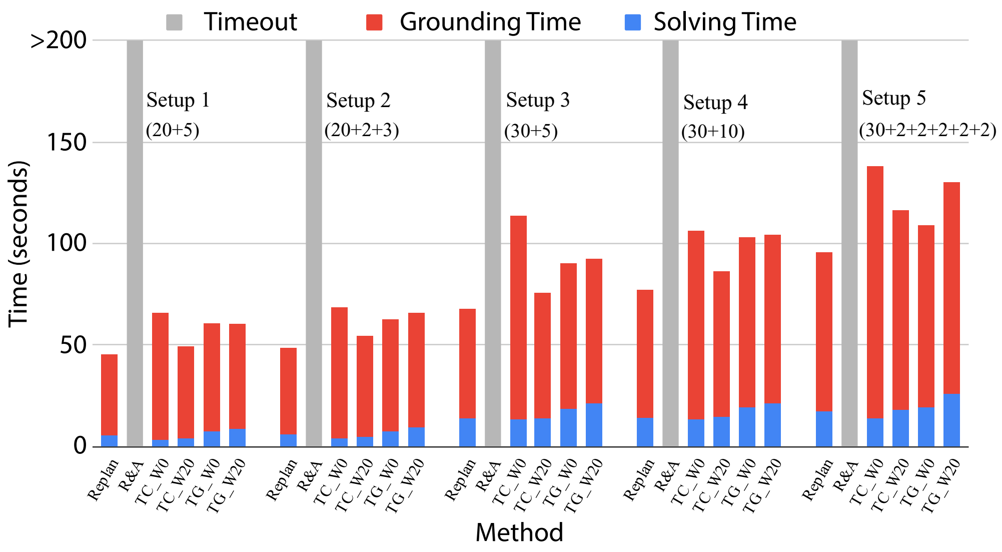


Fig. 7. Average grounding and solving times for 5 setups on an empty 20×20 grid.

accompanying the paper at TPLP archive, including the separate times in each stage of solving D-MAPF, where finding a new solution after a change is called a stage.

Observations and discussions about the computational performance of the methods. From these results, we observe that Revise-and-Augment has the highest computation times among all methods and the fastest performing method is Replan-All. Both approaches for solving Revise-and-Augment-in-Tunnels, TC and TG, perform close to each other and Replan-All, but are not as fast as Replan-All for all of our instances.

We observe that, for TC, when the tunnel width increases, the computation times decrease. This decrease is visible mostly in grounding times, whereas the solving times do not differ significantly for different tunnel widths. This shows the effect of grounding the constraints for the forbidden vertices, which are added for each vertex outside an agent's tunnel, for every existing agent. As the tunnel width increases, the number of forbidden vertices decreases, reducing the number of added constraints and therefore the grounding time for constraints.

On the other hand, for TG, when the tunnel width increases, we observe some increase in computation times. This is due to the higher number of tunnel atoms considered for generating the plans for wider tunnels. As the tunnel width gets larger, the number of tunnel atoms considered for each agent increases accordingly. The number of tunnel atoms directly impacts the number of choice rules being used for generating the new plans, therefore increasing the computation time. The effect of the tunnel atoms on computation times can clearly be observed in the computation times of initial MAPF solving stage. Initially, every vertex is a part of every agent's tunnel, and for our instances, this creates an increase around 10 s in the total time of solving MAPF compared to other methods.

If we consider the case where the MAPF instance is the same and the new agents are added at the same time steps but the number of new agents added at those time steps are different (e.g., Instances 3 and 4), the increase in the computation times with the increase in the number of agents is visible in the results of all methods except R&A. For example, for Replan-All, the computation time for finding a solution after adding 5 new agents in Instance 3 is 19.01 s, while the time for finding a solution after adding 10 new agents in Instance 4 is 28.96 s. Similar increases can be observed for TC and TG.

Consider Instance 4 and Instance 5, where the MAPF instance and the total number of new agents are the same, but they are joining as different groups at different times. For all of our methods, the difference of total times is visible in the results. When the same agents are added at different times, the computation takes longer. This is mainly due to the increase in the makespan of the whole D-MAPF solution, caused by the agents starting their plans at a later time.

Investigating the quality of solutions. In our second set of experiments, we used instances with obstacles adapted from MAPF benchmarks in Stern *et al.* (2019), as follows: (1) We selected three types of 32×32 grids as environments: *random-32-32-10* contains random obstacles in the 10% of the environment, *random32-32-20* has random obstacles in the 20% of the environment, and *room32-32-4* divides the environment into small rooms and connects them through "doors." (2) We scaled these grids to size 20×20 and utilized the random scenarios provided for these grids for creating our instances. (3) From all random scenarios, we extracted the initial and goal location pairs that do not overlap with obstacles in any of these environments. (4) We merged these pairs to a list and generated 5 base MAPF instances by randomly selecting 20 agents from the merged list. (5) Then, for each of these MAPF instances, we created 10 different D-MAPF instances, containing 20 new agents randomly selected from the list, that are not used in its MAPF instance. This resulted in 50 D-MAPF instances that can be used in the selected environments, making 150 instances in total. For these experiments, we assume that all new agents are added at $t = 0$ before the execution but after computing a MAPF plan.

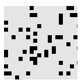
We used these instances to investigate the quality of solutions computed with the tunnels. For the old agents, we examined how their initially computed plans change after the new agents are added to the environment. Table 1 shows a sample result for each environment for the same set of agents.

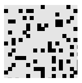
The columns #Plan Changes and #Path Changes show the number of agents that have a different plan and path than their initial solution. A change in a plan is either a change in the time of visiting a location or changing the visited location completely. Here, we consider a path change as visiting a different vertex that does not exist in the original plan. For each instance, the first lines in these columns show the number of agents for Replan-All, the second lines show the number of agents for TC with widths 0, 2, and 5, respectively, and the third lines show the same information for TG.

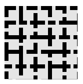
For instance, for the first instance (random 10%), according a D-MAPF solution computed with Replan-All, 12 agents had to change their initial paths, and 14 agents had to change their initial plans. In a D-MAPF solution computed with TC: no agent changes its path if the tunnel width is given as 0, while 6 agents change their plans (i.e., traversals of their paths); 6 agents change their paths if the tunnel width is given as 2, while 7 agents change their plans; and 10 agents change their paths if the tunnel width is given as 5, while 10 agents change their plans.

Table 1. *Results for the same D-MAPF instance in three different 20 × 20 environments*

Grid	MAPF Mkspn	D-MAPF Mkspn	#Plan Changes	#Path Changes	#Diverted Agents [Amount of Divergence in Replan-All]		
					Width-0	Width-2	Width-5

 random 10%	25	27	14
			6, 7, 10
			13, 10, 4
				12
				0, 6, 10
				0, 9, 4
					12
					[6; 8; 8; 17;
					9; 13; 10; 1;
					3; 16; 5; 1]
						7
						[3; 2; 4; 3;
						2; 5; 11]
							1
							[2]

					Width-0	Width-2	Width-5
 random 20%	25	27	6 4, 4, 9 2, 3, 7	3 0, 4, 7 0, 3, 5	3 [1; 1; 1]	0	0
					Width-0	Width-2	Width-5

 room	25	35	20
			19, 20, 20
			19, 19, 19
				19
				0, 19, 19
				0, 19, 19
					19
					[9; 15; 4; 7;
					5; 26; 1; 3;
					7; 3; 27; 15;
					5; 16; 6; 3;
					14; 8; 7]
						11
						[3; 6; 2; 22;
						20; 11; 1; 6;
						7; 6; 1]
							4
							[13; 13; 4; 1]

The last three columns show the number of agents that visit a vertex outside of their tunnels of width 0, 2, and 5, respectively, in a D-MAPF solution computed with Replan-All. The square-bracketed lists show how many such vertices outside of tunnels are visited.

For instance, for the first instance (random 10%), in a D-MAPF solution computed with Replan-All, out of the 12 agents who diverged from their paths, 7 agents went outside of their tunnels of width 2, and 1 of them went further away from its path (i.e., outside of its tunnel of width 5).

Observations and discussions about the quality of solutions. We observe the same amount of increase in the makespan of plans, with all methods. We observe a higher increase in the makespan in the *room* environment, compared to the other environments: all 20 of the existing agents have changed their plans and all except one of the agents have changed their paths. Such higher increases in the makespan (compared to the other two environments) is due to the constrained structure of the environment.

In terms of divergences, we see that, for every tunnel width, the number of vertices visited outside of the agents tunnels is high for some agents. For instance, for *room* with tunnel width 0, one of the agents visits 27 vertices outside its tunnel (in this case, its path). Since the makespan of D-MAPF plan is 35, we can see that the agent diverts from its original path in at least 77% of its new path.

These divergence results demonstrate agents significantly changing their plans with Replan-All. Recall that such significant changes in plans are not desired in real-world applications where robots collaborate with humans, and this was our motivation for introducing the concept of tunnels.

Table 2. *Average run times of 150 D-MAPF instances for different environments/methods*

Grid	Replan-All	TC_W0	TC_W2	TC_W5	TG_W0	TG_W2	TG_W5
random 10%	39.65	47.82	44.62	43.22	48.44	48.53	48.99
random 20%	33.90	39.45	37.11	36.06	41.17	41.60	42.19
room	38.61	39.65	34.44	34.22	42.72	41.98	43.93

Effect of obstacles in computation times. Table 2 shows the average of the total computation times for all D-MAPF instances in the second set of experiments. When we compare computation times in different environments, we observe that *random 20%* has the fastest computation times. This is an expected result due to the environment having fewer empty cells for the agents to move; hence, resulting in a smaller problem size. On the other hand, while *room* environment has even fewer empty cells, the computation times are higher due to the tight passages between the rooms. For TC, we observe a similar pattern with the first set of experiments: as the tunnel size increases, the computation becomes faster. For TG, we also observe similar behavior with the first set of experiments: increasing the tunnel size increases the computation times.

8 Conclusion

D-MAPF problem considers MAPF problem in dynamic environments where changes can occur in the environment or the team of agents. In this study, we introduced a general definition for D-MAPF problem that covers different assumptions on appearance/disappearance of agents and different objective functions studied in the literature, as well as the possible changes that can occur in the environment. We introduced a framework for solving D-MAPF that can use different methods for computing the new solutions after changes in the environment, utilizing multi-shot ASP. In addition to formulating the existing approaches Replan-All and Revise-and-Augment using multi-shot ASP, we introduced a new method called Revise-and-Augment-in-Tunnels that combines the advantages of multi-shot solving and re-using the existing solutions. We integrated this method into our framework and observed its usefulness in terms of computational performance and solution quality.

Supplementary material

The supplementary material for this article can be found at https://doi.org/10.1017/S1471068425100276.

Competing interests

The authors declare no competing interests.

References

ATIQ, B., PATOGLU, V. and ERDEM, E. 2020. Dynamic multi-agent path finding based on conflict resolution using answer set programming. *Electronic Proceedings in Theoretical Computer Science* 325, 223–229.

BOGATARKAN, A., PATOGLU, V. and ERDEM, E. 2019. A declarative method for dynamic multi-agent path finding. In *Proceedings of the 5th Global Conference on Artificial Intelligence 2019*, 54–67. EasyChair.

ERDEM, E., KISA, D. G., OZTOK, U. and SCHUELLER, P. 2013. A general formal framework for pathfinding problems with multiple agents. *Proceedings of the AAAI Conference on Artificial Intelligence* 27, 1, 290–296. AAAI Press.

GEBSER, M., KAMINSKI, R., KAUFMANN, B. and SCHAUB, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1, 27–82.

GELFOND, M. and LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9, 365–385.

HO, F., SALTA, A., GERALDES, R., GONCALVES, A., CAVAZZA, M. and PRENDINGER, H. 2019. Multi-Agent path finding for UAV traffic management. In *Proceedings of the International Conference on Autonomous Agents and MultiAgent Systems 2019*, 131–139, International Foundation for Autonomous Agents and Multiagent Systems.

LI, J., TINKA, A., KIESEL, S., DURHAM, J. W., KUMAR, T. K. S. and KOENIG, S. 2021. Lifelong multi-agent path finding in large-scale warehouses. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 13, 11272–11281.

LIFSCHITZ, V. 2008. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence 2008*, 1594–1597. AAAI Press.

MA, H. 2021. A competitive analysis of online multi-agent path finding. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31, 1, 234–242. AAAI Press.

MORAG, J., FELNER, A., STERN, R., ATZMON, D. and BOYARSKI, E. 2022. Online multi-agent path finding: New results. *Proceedings of the International Symposium on Combinatorial Search*, 15, 1, 229–233. AAAI Press.

STERN, R., STURTEVANT, N. R., FELNER, A., KOENIG, S., MA, H., WALKER, T. T., LI, J., ATZMON, D., COHEN, L., KUMAR, T. K. S., BARTÁK, R. and BOYARSKI, E. 2019. Multi-Agent pathfinding: Definitions, variants, and benchmarks. *Symposium on Combinatorial Search (SoCS)* 10, 1, 151–159. AAAI Press.

SVANCARA, J., VLK, M., STERN, R., ATZMON, D. and BARTAK, R. 2019. Online multi-agent pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 1, 7732–7739. AAAI Press.

WAN, Q., GU, C., SUN, S., CHEN, M., HUANG, H. and JIA, X. 2018. Lifelong multi-agent path finding in a dynamic environment. In *Proceedings of 15th International Conference on Control, Automation, Robotics and Vision (ICARCV) 2018*, 875–882. IEEE.