EPiC Series in Computing

Volume 65, 2019, Pages 54–67

GCAI 2019. Proceedings of the 5th Global Conference on Artificial Intelligence

A Declarative Method for Dynamic Multi-Agent Path Finding

Aysu Bogatarkan, Volkan Patoglu, and Esra Erdem

Sabancı University, Faculty of Engineering and Natural Sciences, İstanbul, Turkey
{aysubogatarkan,vpatoglu,esraerdem}@sabanciuniv.edu

Abstract

The multi-agent path finding (**MAPF**) problem is a combinatorial search problem that aims at finding paths for multiple agents such that no two agents collide with each other. We study a dynamic variant of **MAPF**, called **D-MAPF**, which allows changes in the environment (e.g., some existing obstacles may be removed from the environment or moved to some other location, or new obstacles may be included in the environment), and/or changes in the team (e.g., some existing agents may leave and some new agents may join the team) at different times. We introduce a new method to solve **D-MAPF**, using answer set programming.

1 Introduction

The Multi-Agent Path Finding (**MAPF**) problem aims to find a plan for multiple agents to reach their destinations in a certain environment with static obstacles, subject to some constraints on the maximum or the total plan length. Every agent can be considered as a dynamic obstacle for other agents. Therefore, the obstacles and the agents lead to some constraints for the executability of the plan: agents cannot pass through obstacles, and agents cannot collide with each other. Although single-agent shortest pathfinding can be solved in polynomial time [6], **MAPF** (with constraints on the plan length) is an intractable problem [25] due to the latter constraint that no two agents can be in the same location at the same time. Yet, **MAPF** has been studied in various domains, such as robotics [16], video games [28], autonomous aircraft towing vehicles [22], traffic control [7] and autonomous warehouse systems [33].

During the execution of a computed plan for **MAPF** in a dynamic environment (e.g., a warehouse that is not completely autonomous), changes may occur: existing obstacles may be removed from the environment or moved to some other location in the environment, existing agents may leave the environment, or new agents may be included in the team with new tasks. Then the goal is to find a new solution for the new team of agents in the modified environment. We call this problem *Dynamic Multi-Agent Path Finding Problem (***D-MAPF***)*. Note that **D-MAPF** inherits the intractability of **MAPF**, subject to the constraints on plan lengths.

One of the possible solutions for **D-MAPF** is replanning: consider a new **MAPF** instance defined by the current locations and goal locations of both the existing and the new agents, and the updated environment, and compute a solution for this instance. Although replanning finds a solution, if one exists, it does not re-use the plans of the existing agents and may not be computationally efficient.

With this motivation, we propose a novel method to solve **D-MAPF**, using Answer Set Programming (ASP) [21, 24, 17, 2, 3] (based on answer sets [10, 12]). The main idea (and novelty) of this method is, instead of replanning for all the agents right away, to *revise and augment* the existing **MAPF** solution: (*revise*) try to schedule the waiting times of existing agents as they traverse the rest of their paths, (*augment*) while computing paths for the new agents within a given makespan (i.e., the length of the plan). In this way, the paths for the existing agents can be re-used as part of the new plan. As observed by our experimental evaluations, the re-use of plans as proposed by our method improves the computational efficiency in timings significantly compared to replanning.

2 R-MAPF: Revising and Augmenting MAPF

D-MAPF can be viewed as a generalization of **MAPF** in a dynamic environment, that considers what changes in the environment, such as adding new agents, removing some of the existing agents, adding or removing obstacles or changing their locations, as well as what does not change, such as the itineraries of existing agents. We propose to solve **D-MAPF** by revising and augmenting the existing **MAPF** solution.

Let us first introduce some concepts and notation before we define the problem of revising and augmenting an existing **MAPF** solution (called **R-MAPF**).

A *traversal f of a path $P = \langle w_1, w_2, \ldots, w_n \rangle$* in a graph G within some time t ($t \in \mathbb{Z}^+$) is an onto function that maps every nonnegative integer less than or equal to t to a vertex in P, such that, for every w_i and w_j in P and for every $x < t$, if $f(x) = w_i$ and $f(x{+}1) = w_j$, then $w_j = w_i$ or $w_j = w_{i+1}$.

We denote by $f(P)$ a traversal f of a path P (within time t). We denote by $\mathbf{P}_A$ the collection of paths P for every agent in A, and by $\langle \mathbf{P}, \mathbf{f} \rangle_A$ the collection of pairs $\langle P_i, f_i(P_i) \rangle$ of paths and their traversals for every agent a_i in A.

Let f_i and f_j be traversals of two different paths P_i and P_j, respectively, in a graph G within some time t. We say that the traversals f_i and f_j *do not collide with each other* within time t if,

- for every times $x, x' \leq t$, the following holds: if $f_i(x) = f_j(x')$ then $x \neq x'$ (i.e., if the same vertex is visited by paths P_i and P_j, then it should be visited at different times — no two agents can be at the same location at the same time);
- for every time $x{<}t$, the following holds: if $f_i(x) = f_j(x{+}1)$ then $f_i(x{+}1) \neq f_j(x)$ (i.e., an edge cannot be visited by paths P_i and P_j in reverse directions at the same time — no two agents can swap their locations at the same time).

MAPF problem can be defined as in Figure 1. A **MAPF** instance can be characterized by a sextuple $\langle A, G, init, goal, O, \tau \rangle$, and its solution by a collection $\langle \mathbf{P}, \mathbf{f} \rangle_A$ of pairs of paths P_i and their traversals $f_i(P_i)$ within some time $u{\leq}\tau$ for agents a_i in A.

For a path $P = \langle v_1, \ldots, v_n \rangle$ and its traversal $f(P)$ within time u, and a nonnegative integer $k \leq u$, the *k-suffix* of P with respect to f (denoted $P^{k,f}$) is the subsequence $\langle v_i, \ldots, v_n \rangle$ such that $v_i = f(k)$. Intuitively, the k-suffix of a path P describes the part of the path visited after time k (included).

For a collection $\langle \mathbf{P}, \mathbf{f} \rangle_A$ of pairs of paths and their traversals, we denote by $\mathbf{P}_A^{k,f}$ the collection of k-suffixes of paths in $\mathbf{P}_A$ with respect to their traversals in $\langle \mathbf{P}, \mathbf{f} \rangle_A$.

Let us now define the **R-MAPF** problem as in Figure 2, subject to the *obstacle assumption*:

> The currently observed obstacles are not on the paths of the existing agents.

Note that the obstacle assumption is not restrictive, since the existing agents whose paths go through the new obstacles can be considered as new agents. This assumption is considered to make the definition easier to follow.

MAPF Problem

Input:

- A nonempty set $A = \{a_1, \ldots, a_n\}$ of agents ($n > 0$).
- A graph $G = (V, E)$ (to describe an environment where the agents move around).
- A function *init* : $A \mapsto V$ (to describe the initial locations of agents).
- A function *goal* : $A \mapsto V$ (to describe the goal locations of agents).
- A set $O \subseteq V$ (to denote the obstacles in the environment).
- A positive integer τ (to specify the maximum makespan—plan length).

Output: For every agent $a_i \in A$, for some positive integer $u \leq \tau$,

- a path $P_i = \langle w_{i,1}, \ldots, w_{i,n_i} \rangle$ of length n_i ($n_i \leq u$)
  - that the agent a_i will follow to reach its goal location from its initial location (i.e., $w_{i,1}$=*init*(a_i) and w_{i,n_i}=*goal*(a_i)),
  - without colliding with any obstacles (i.e., $w_{i,j} \in V \setminus O$), and
- a traversal f_i of the path P_i within time u, such that
  - for every other agent $a_j \in A$ with a path P_j and its traversal f_j within u, $f_i(P_i)$ and $f_j(P_j)$ do not collide with each other.

Figure 1: **MAPF** problem definition.

R-MAPF Problem

Input:

- A **MAPF** instance $\langle A, G, init, goal, O, \tau \rangle$ ($G = (V, E)$), with a solution $\langle \mathbf{P}, \mathbf{f} \rangle_A$.
- A nonnegative integer $k < \tau$ (to denote the current time).
- A nonempty subset $A{\uparrow} \subseteq A$ of agents (who leave the environment by time k).
- A nonempty set $A{\downarrow}$ ($A{\downarrow} \neq A$) of new agents (who join the remaining team $A \setminus A{\uparrow}$ at time k).
- A function *init*$\downarrow$: $A{\downarrow} \mapsto V$ (to describe the initial locations of the new agents).
- A function *goal*$\downarrow$: $A{\downarrow} \mapsto V$ (to describe the goal locations of the new agents).
- A set $O{\downarrow}$ of current obstacles (as observed at time k) that do not occupy any vertices of paths in $\mathbf{P}$.

Output:

- For the new agents (i.e., $A{\downarrow}$): A collection $\langle \mathbf{P}_{new}, \mathbf{f}_{new} \rangle_{A{\downarrow}}$ of pairs of paths and their traversals within time $\tau{-}k{+}1$, such that $\langle \mathbf{P}_{new}, \mathbf{f}_{new} \rangle_{A{\downarrow}}$ is a solution for $\langle G, A{\downarrow}, \mathit{init}{\downarrow}, \mathit{goal}{\downarrow}, O{\downarrow}, \tau{-}k{+}1 \rangle$.
- For the other agents (i.e., $A \setminus A{\uparrow}$): A collection $\langle \mathbf{P}^{k,f}, \mathbf{f'} \rangle_{A \setminus A{\uparrow}}$ of pairs of the k-suffixes $P^{k,f}$ of their paths $P \in \mathbf{P}$, and their new traversals $f'(P^{k,f})$ within time $\tau{-}k{+}1$ that do not collide with each other or the traversal f_{new} of a new agent's path.

Figure 2: **R-MAPF** problem definition.

R-MAPF takes as input the current locations and goal locations of the new agents, the updated environment, and the **MAPF** instance and solution for the existing agents. It aims to find a solution that revises the traversals of the paths of the existing agents, and augments with new paths and their traversals for the new agents. Meanwhile, it ensures that no agent collides with the obstacles or the other agents, and it respects the original maximum makespan.

3 Solving D-MAPF using ASP

Since the agents do not disappear once they reach their locations, **R-MAPF** may not be always solvable even when the corresponding **D-MAPF** is solvable [31]. For that reason, we introduce an algorithm for **D-MAPF**, based on our solution to **R-MAPF** using ASP.

3.1 Our Algorithm

D-MAPF takes as input (as in **R-MAPF**) the current locations and goal locations of the new agents, the updated environment, and the **MAPF** instance and solution for the existing agents. It aims to find a solution (as in **MAPF**) that consists of paths and their traversals for all current agents, ensuring that no agent collides with the obstacles or the other agents, and respecting the original maximum makespan. Note that **D-MAPF** is more general than **R-MAPF** since it does not require the re-use of the existing paths. Therefore, a solution to **R-MAPF** is a solution to **D-MAPF**, but a solution to **D-MAPF** (e.g., found by replanning) may not be a solution to **R-MAPF**.

Our solution is based on the formalization of **R-MAPF**, subject to the obstacle assumption, as an ASP program. According to this ASP program, the waiting times of the existing agents are scheduled relative to their paths within a given makespan, while paths and their traversals are computed for the new agents within that given makespan. Meanwhile, it is ensured that there are no collisions with obstacles or between robots.

Our algorithm for **D-MAPF** utilizes the ASP formalization of **R-MAPF** as follows.

1. During the execution of a given **MAPF** plan according its traversal, if a new obstacle is detected that is on the path of an existing agent, then that agent is considered as a new agent along with other new joining agents. This step ensures that the obstacle assumption is satisfied.
2. A solution to **R-MAPF**, subject to the obstacle assumption, is computed using the ASP program with an ASP solver, like CLINGO [9], within the makespan of the given **MAPF** plan.
3. If a solution cannot be found, the algorithm tries to find a solution to **R-MAPF** by increasing the makespan incrementally until the given maximum makespan is reached.
4. If a solution still cannot be found, then the algorithm applies replanning: it solves **MAPF** for all agents by minimizing the maximum makespan.

Figures 3–6 present an example to illustrate the overall idea.

3.2 ASP Formalization of R-MAPF

Let us describe our ASP program that formalizes **R-MAPF**. We refer the reader to relevant sources [12, 1, 11] for the syntax and semantics of programs in ASP. In the following, we use the mathematical representation of programs, so the lowercase letters (in the arguments of predicates) denote schematic variables, whereas the uppercase letters denote constants.

Notation. Consider the following notation about the input:

- the roads in the environment (viewed as a graph G) are described by atoms of the form $\mathit{edge}(x, y)$ — there is a road from location x to location y;
- a **MAPF** solution (i.e., the plan being executed) is described by atoms of the form $\mathit{path}(a, t, x)$ — existing agent a is expected to be at location x at time step t;
- the maximum makespan of the **MAPF** plan being executed is T;
- the current time step is K;

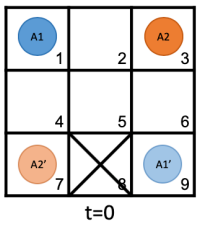

Figure 3: Initially, two agents are at $A1$ and $A2$; their goals are to reach $A1'$ and $A2'$.

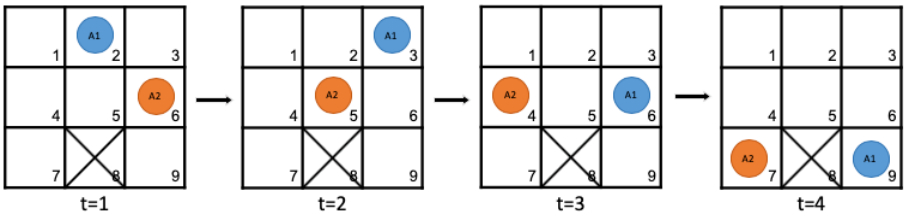

Figure 4: A solution to **MAPF** instance described in Fig. 3: $A1$ starts at 1, moves to 2, 3, 6, and reaches 9; and $A2$ starts at 3, moves to 6, 5, 4, reaches 7.

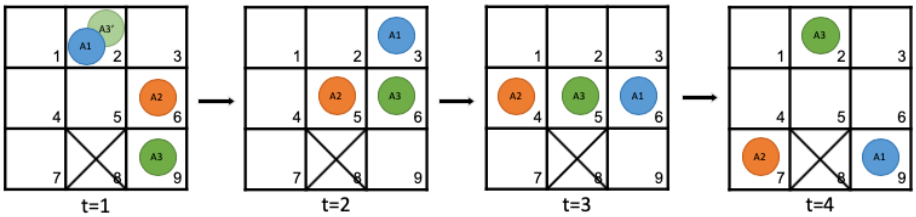

Figure 5: While executing the plan shown in Fig. 4, at time step $t = 1$, another agent $A3$ joins the team with goal $A3'$. A solution to this **D-MAPF** instance is computed, and a path for $A3$ is found, as shown above: $A1$ is at 2, moves to 3, 6, 9; $A2$ is at 6, moves to 5, 4, 7; $A3$ starts at 9, moves to 6, 5, 2.

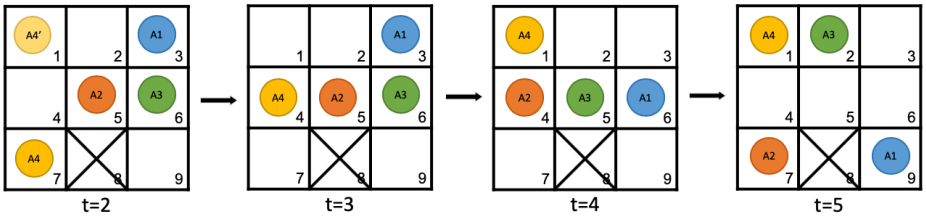

Figure 6: While executing the plan shown in Fig. 5, at time step $t = 2$, another agent $A4$ joins the team with goal $A4'$. A solution to this **D-MAPF** instance is computed. No solution is found with $t = 4$, so the makespan is increased by 1. The new solution with makespan $t = 5$ is found by scheduling the waiting times of $A1$, $A2$ and $A3$, and by computing a path for $A4$, as shown above: $A1$ is at 3, waits at 3, then moves to 6,9; $A2$ is at 5, waits at 5, then moves to 4,7; $A3$ is at 6, waits at 6, then moves to 5,2; $A4$ starts at 7, moves to 4, 1, and waits at 1.

- the vertices x occupied by obstacles observed at time step K are described by atoms of the form $\mathit{obstacle}(x)$;
- the initial and the goal location for every new agent a joining the team at time step K are described by atoms of the forms $\mathit{init}(a, x)$ and $\mathit{goal}(a, x)$, respectively.

Revise the plans for existing agents. Potential waiting schedules for existing agents are generated over the rest of their plans starting from their current positions, to obtain new traversals of the remaining parts of their paths.

First, the current and the goal locations of every existing agent $a \in (A \setminus A\uparrow)$ are identified:

$$\mathit{goal}(a,x) \leftarrow \mathit{path}(a,T,x) \quad (a \in (A \setminus A\uparrow)).$$
$$\mathit{init}(a,x) \leftarrow \mathit{path}(a,K,x) \quad (a \in (A \setminus A\uparrow)).$$

Suppose that the new plan is described by atoms of the form $\mathit{plan\_new}(a,t,x)$. A new plan for every existing agent $a \in (A \setminus A\uparrow)$ then should start at its current location at time K:

$$\mathit{plan\_new}(a,K,x) \leftarrow \mathit{init}(a,x) \quad (a \in (A \setminus A\uparrow)).$$

At every time step $t{-}1$ ($K{\leq}t{-}1{\leq}T{-}1$), every existing agent a may either wait at its location x, or move to the next location y (visited after x) according to its existing path:

$$1\{\mathit{plan\_new}(a,t,x);\ \mathit{plan\_new}(a,t,y) :\ \mathit{path}(a,t_1,y), \mathit{path}(a,t_1{-}1,x), \mathit{edge}(x,y), K{\leq}t_1\}1 \leftarrow$$
$$\mathit{plan\_new}(a,t{-}1,x) \quad (a \in (A \setminus A\uparrow), K{+}1{\leq}t{\leq}T).$$

Note that the paths of the existing agents are utilized inside the choice rules and thus lead to more intelligent generation of new traversals, compared to the straightforward approach of generating many new plans and then eliminating the ones that do not characterize traversals of existing paths. In that sense, this is a novel and elegant contribution from the modeling perspective.

As new traversals are generated, we can add the **MAPF** plan of existing agents as a constraint to generate a different waiting schedule for them.

Augment with plans for new agents. For every new agent $a \in A\downarrow$, plans are generated recursively as suggested by [8]:

$$\mathit{plan\_new}(a,K,x) \leftarrow \mathit{init}(a,x) \quad (a \in A\downarrow).$$
$$1\{\mathit{plan\_new}(a,t,x);\ \mathit{plan\_new}(a,t,y) :\ \mathit{edge}(x,y)\}1 \leftarrow$$
$$\mathit{plan\_new}(a,t{-}1,x). \quad (a \in A\downarrow, K{+}1{\leq}t{\leq}T)$$

These new plans, described by atoms of the form $\mathit{plan\_new}(a,t,x)$ for every agent a, should characterize paths. Due to the cardinality constraint above, there is an outgoing edge from their initial locations.

There should be an incoming edge to their goals to make sure every agent reaches its goal:

$$\leftarrow \{\mathit{plan\_new}(a,t,y) : \mathit{edge}(x,y), \mathit{goal}(a,y)\}0 \quad (a \in A\downarrow, K{+}1{\leq}t{\leq}T).$$

To ensure that the plan of a new agent a ends at the goal, i.e., there is no outgoing edge from the goal, we further add the following constraints:

$$\leftarrow \mathit{plan\_new}(a,t,x), \mathit{plan\_new}(a,t{+}1,y), \mathit{edge}(x,y), \mathit{goal}(a,x) \quad (a \in A\downarrow, K{\leq}t{\leq}T).$$

No collisions. Now that we have generated new plans for all agents, that characterize paths, we need to ensure that there are no collisions.

No two agents a_1 and a_2 are at the same vertex x at the same time t.

$$\leftarrow \mathit{plan\_new}(a_1,t,x), \mathit{plan\_new}(a_2,t,x) \quad (a_1{\neq}a_2, K{\leq}t{\leq}T).$$

No two agents a_1 and a_2 swap their locations at the same time.

$$\leftarrow \textit{edge}(x,y), \textit{plan\_new}(a_1,t,x), \textit{plan\_new}(a_2,t,y), \\ \textit{plan\_new}(a_1,t{-}1,y), \textit{plan\_new}(a_2,t{-}1,x) \quad (a_1{\neq}a_2, K{+}1{\leq}t{<}T).$$

No agent a visits a vertex x containing an obstacle:

$$\leftarrow \textit{plan\_new}(a,t,x), \textit{obstacle}(x) \quad (K{\leq}t{\leq}T).$$

Optimizations. While revising the existing plans and augmenting new plans, our formulation takes into account an upper bound for the makespan, so as to minimize the makespan in our algorithm for **D-MAPF**. We can consider further optimizations. For instance, with the following weak constraints, total waiting time (for all agents a) can be minimized:

$$\overset{\sim}{\leftarrow} \textit{plan\_new}(a,t,x), \textit{plan\_new}(a,t{+}1,x), \textit{not goal}(a,x).[1@1,a,t]$$

4 Experimental Evaluations

We have implemented our **D-MAPF** algorithm described in Section 3.1, based on the ASP formalization of **R-MAPF** explained in Section 3.2, using Python 2.7.3 and CLINGO 4.5.4, and performed experiments on a Linux server with dual 2.4 GHz Intel E5-2665 CPUs and 64 GB memory. Recall that our algorithm tries to re-use the plans of the existing agents by scheduling their waiting times, to allow the new agents to reach their destinations.

We have compared our algorithm that makes use of the previously computed plans with the replanning approach that computes new plans for all agents, by means of some experiments, to observe the trade-off between computation time and solution quality. Recall that in the replanning approach, when some changes are noticed in the team or the environment, a new **MAPF** instance is created by the current locations and goal locations of both the existing and the new agents, and the updated environment, and a solution is computed for this instance. If no solution is found, then the makespan is incremented as in our algorithm until the given maximum makespan.

Note that replanning does not re-use the plans of the existing agents, so we expect to observe from our experiments that the replanning approach may not be computationally as time-efficient as our algorithm. On the other hand, our algorithm tries to re-use the paths of the existing agents, so we expect to observe that the quality of solutions (characterized by their makespans) computed by our algorithm may not be as good as the quality of solutions computed by the replanning approach. Let us investigate these questions.

For experimental evaluations, we have generated random **MAPF** instances over various grid sizes for 20 agents: 10×10, 20×20, 30×30, 40×40, 50×50. For each **MAPF** instance, we have computed a solution whose makespan is smaller than some given value less than T. Then, we have generated **D-MAPF** instances, by extending the team with 1–5 new agents. For convenience, it is assumed that each agent is added at time $t = 0$, and there are no obstacles. The results are presented in Table 1.

Computation time vs. makespan. Let us consider the **D-MAPF** instance on a grid of size 10×10, which starts with a **MAPF** solution of makespan 18, and 1 new agent joins the team. For this **D-MAPF** instance, a new solution is computed by our algorithm in 0.200 seconds, whereas a new solution is computed by the replanning approach in 2.710 seconds. When 2 agents join the team, then our algorithm cannot find a new solution with makespan 18; it takes 0.250 seconds for our algorithm to terminate without any solution. Then our algorithm increases the makespan to 19, and computes a new

Table 1: Experimental evaluations: Our algorithm vs. Replanning.

Initial instance	# of new agents	Our algorithm		Replanning	
		CPU time [s]	Found [Y/N]	CPU time [s]	Found [Y/N]
	1	0.200	Y	2.710	Y
	2	0.250	N	2.810	Y
	2	0.290	Y (19)	3.260	Y
20 agents	3	0.330	N	3.160	Y
10×10 grid	3	0.420	Y (19)	3.910	Y
18 makespan	4	0.400	N	3.180	Y
	4	0.480	Y (19)	3.750	Y
	5	0.490	N	3.710	Y
	5	0.580	Y (19)	4.530	Y
	1	1.510	Y	32.660	Y
20 agents	2	2.490	Y	38.540	Y
20×20 grid	3	3.450	Y	32.180	Y
38 makespan	4	4.510	Y	49.680	Y
	5	5.350	Y	43.490	Y
	1	1.670	Y	38.600	Y
20 agents	2	2.820	Y	34.390	Y
20×20 grid	3	4.300	Y	42.140	Y
40 makespan	4	4.980	Y	41.550	Y
	5	5.970	Y	52.660	Y
	1	7.200	Y	112.290	Y
20 agents	2	8.540	Y	110.800	Y
30×30 grid	3	11.860	Y	123.400	Y
58 makespan	4	15.370	Y	103.630	Y
	5	15.430	Y	119.790	Y
	1	4.630	Y	96.430	Y
20 agents	2	7.970	Y	114.740	Y
30×30 grid	3	10.880	Y	134.320	Y
60 makespan	4	14.090	Y	141.300	Y
	5	14.020	Y	146.720	Y
	1	12.480	Y	280.710	Y
20 agents	2	25.180	Y	348.930	Y
40×40 grid	3	35.460	Y	335.320	Y
78 makespan	4	50.670	Y	398.460	Y
	5	63.530	Y	344.470	Y
	1	14.100	Y	261.290	Y
20 agents	2	25.960	Y	467.840	Y
40×40 grid	3	39.010	Y	371.400	Y
80 makespan	4	58.530	Y	395.510	Y
	5	63.090	Y	443.040	Y
	1	19.990	Y	510.570	Y
20 agents	2	40.920	Y	493.590	Y
50×50 grid	3	55.150	Y	577.660	Y
98 makespan	4	72.120	Y	681.480	Y
	5	93.780	Y	587.230	Y
	1	18.800	Y	737.070	Y
20 agents	2	32.630	Y	545.840	Y
50×50 grid	3	46.820	Y	1150.140	Y
100 makespan	4	63.770	Y	753.140	Y
	5	85.330	Y	774.070	Y

solution with makespan 19 in 0.290 seconds. Meanwhile, when two agents join the team, the replanning approach finds a new solution with makespan 18 in 2.810 seconds.

From these results, we observe the strengths and the weaknesses of our approach (as expected), showing the trade-off between computation time and solution quality. Our algorithm finds solutions significantly faster than the replanning approach. On the other hand, the replanning approach may find solutions with slightly shorter makespans.

Computation times vs. number of new agents. Let us now consider the **D-MAPF** instance on a grid of size 50×50, which starts with a **MAPF** solution of makespan 98. When 1 new agent joins the team, our algorithm computes a solution with makespan 98 in 19.990 seconds (reusing the existing **MAPF** solution) whereas the replanning approach takes 510.570 seconds since it has to plan for every agent. When 2 agents join, our approach computes a solution with makespan 98 in 40.920 seconds, whereas the replanning approach takes 493.590 seconds. When 4 agents join, our approach computes a solution with makespan 98 in 72.120 seconds, whereas the replanning approach takes 681.480 seconds.

From these results, we observe that the underlying idea of reusing existing solutions may be quite efficient in terms of computation time, in particular, when the number of existing agents is much larger than the number of new agents.

As more agents join the team, the computation times increase in general for both approaches. The growth rate of the computation time of our algorithm is greater than that of the replanning approach. This observation is not surprising: Since our algorithm computes new paths and their traversals for the new agents, the computation time increases as the number of new agents increases; since the replanning approach computes new paths and their traversals for all agents, the computation time increases as the number of all agents increases. The increase in the number of new agents is larger than the increase in the number of all agents.

Computation times vs. grid sizes. Finally, let us also consider the scalability of these two approaches as the grid gets larger. We can observe from the experimental results that the computation times increase for both approaches, and that the growth rates are similar.

Larger instances with obstacles. It is important to note that the results illustrate the usefulness of our revise and augment approach (i.e., solving **R-MAPF** as part of our algorithm for **D-MAPF**): for most of the instances, no replanning is done as part of **D-MAPF** algorithm (i.e., Step 4 of the algorithm) if we allow a slight increase of the maximum makespan.

These observations do not change even when we increase the number of new agents (e.g., when 50% more agents join the team) and/or include obstacles on the grid (e.g., where 4 blocks of 4x4 square obstacles are placed on a grid of size 20x20 as in Figure 7(a), or a large cross obstacle placed on a grid of size 20x20 dividing the grid into 4 rooms with narrow door passages in between as in Figure 7(b)), as can be seen in Table 2.

Similar observations can be made in Table 3 for instances with more obstacles (e.g., where 30 blocks of 2x2 square obstacles are placed on a grid of size 20x20 as in Figure 7(c), or a large cross obstacle with narrow door passages and 20 blocks of 2x2 obstacles are randomly placed on a grid of size 20x20 as in Figure 7(d)).

Note that our experiments involve relatively large instances (in terms of grid size, number of agents, and obstacle occupancy) comparable with the ones reported in the related work. For instance, the evaluation of online **MAPF** [31] is performed over grids of sizes 3x5 (with at most 30% obstacle occupancy) and 3x10 (with at most 40% obstacle occupancy) with 10–25 new agents, and a grid of size 16x16 (with 39 % obstacle occupancy) with 60–70 new agents.

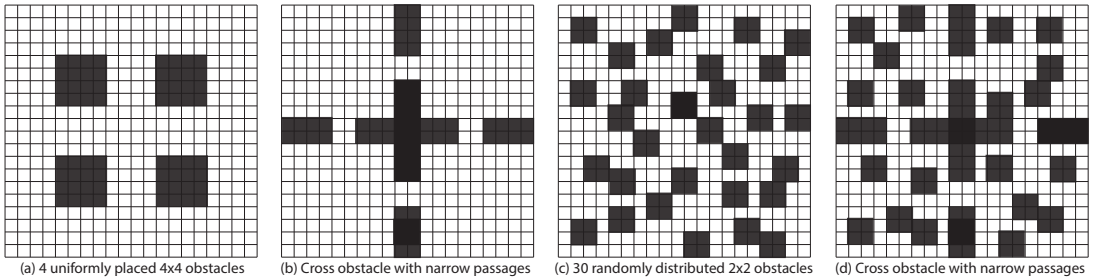

Figure 7: Four different environments (grids of size 20x20) used in our experiments: (a) 4 blocks of 4x4 square obstacles are placed on the grid, (b) a large cross obstacle is placed on the grid dividing it into 4 rooms with narrow door passages in between, (c) 30 blocks of 2x2 square obstacles are randomly placed on the grid, (d) both a large cross with narrow passages and 20 blocks of 2x2 square obstacles are placed on the grid.

Table 2: Experimental results on a 20x20 grid without obstacles, with block obstacles as in Figure 7(a), and with a cross obstacle as in Figure 7(b).

Initial instances	MAPF Makespan	# of New agents	no obstacle		block obstacles		cross obstacle	
			makespan	time (s)	makespan	time (s)	makespan	time (s)
20 x 20 grid 20 agents	36	2	38	16.784	38	16.995	38	17.234
		4	38	52.963	38	56.650	38	54.589
		6	38	99.363	38	117.534	38	107.296
		8	38	161.680	38	177.387	38	185.088
		10	38	240.157	38	257.768	38	269.664
20 x 20 grid 30 agents	36	2	38	21.780	38	23.150	38	24.629
		4	38	68.159	38	69.294	38	75.503
		6	38	119.183	38	124.589	38	135.194
		8	38	190.121	38	208.926	38	196.780
		10	38	273.019	39	416.618	39	467.090

Further remarks on reusability of existing plans. Recall that the main motivation of our revise and augment algorithm for **D-MAPF** is to be able to reuse the existing plans while trying to keep the makespan as small as possible. For that purpose, we can try to minimize the waiting time for each existing agent so that they do not lengthen the traversals of their paths and thus the makespan. With

Table 3: Experimental results with more obstacles as in Figures 7(c) and (d).

Initial Instances	# of New Agents	30 randomly distributed 2x2 obstacles			Cross with 20 random 2x2 obstacles		
		MAPF	makespan	time(s)	MAPF	makespan	time(s)
20x20 20 agents	2	32	32	5.345	25	25	5.201
	4		32	10.826		25	7.468
	6		32	33.166		25	11.609
	8		32	53.223		25	29.007
	10		32	82.527		25	44.907
20x20 30 agents	2	32	32	7.143	25	25	5.118
	4		32	19.062		26	19.553
	6		32	40.716		26	37.548
	8		32	63.402		26	74.474
	10		32	97.442		26	112.574

such an optimization, as in the results discussed above, we observe in Table 4 that the reusability of existing plans is high (i.e., the makespan does not change in most of the cases) and the percentage of change in the total plan length is small. Despite no change in the makespan and small changes in the total plan length, note that **D-MAPF** instances still require revisions of existing plans.

Table 4: Change in the total plan length of the existing agents.

# of New Agents	10x10 grid		20x20 grid		30x30 grid		50x50 grid	
	20 agents	30 agents	20 agents	30 agents	20 agents	30 agents	20 agents	30 agents
1	-2.25%	0.41%	-1.53%	3.42%	0.44%	2.67%	0.67%	0.45%
2	3.13%	1.67%	0.14%	3.70%	0.00%	2.32%	0.83%	0.79%
3	-1.25%	2.29%	1.53%	4.07%	0.09%	1.42%	1.51%	-0.20%
4	3.44%	3.34%	0.97%	3.79%	-0.44%	2.20%	0.78%	0.41%
5	3.44%	3.96%	0.69%	3.89%	-0.80%	2.26%	0.88%	0.41%

5 Related Work

There are several problems that are related to **D-MAPF**, although they are different in general.

MAPF. There are mainly two kinds of **MAPF** solvers: some of them use search-based problem solving (mostly based on a variant of A* search), and some of them use declarative problem solving.

For instance, Silver [27] introduces an incremental method where the paths of agents are computed one by one with A* [13]; once a path is found for an agent, it is considered as an obstacle for other agents. Luna and Bekris [18] propose to compute the paths of agents independently, and then resolve the conflicts (i.e., when two agents collide with each other) with respect to some push-and-swap rules (e.g., there should be at least two free vertices in the graph). Chouhan and Niyogi [5, 4] propose a similar solution where the paths are computed independently; but the conflicts are resolved differently by assigning priorities to agents. Other search-based algorithms, like [7, 32, 15], also compute paths independently; in case a collision occurs, it is resolved by replanning one of the conflicting agents' route. Sharon et al. [26] propose a different method that performs a search on a tree based on the conflicts between agents.

The declarative methods reduce **MAPF** to some formalisms (e.g., ILP, SAT, ASP) and use general problem solvers to find plans. Yu and Lavalle [34] model **MAPF** as a network flow problem and use an ILP solver to optimize the makespan (the time when the last robot reaches its goal) or the distance (the total distance traveled by all robots). Surynek et al. [29, 30] reduce **MAPF** to SAT and use a SAT solver to optimize the makespan or the sum of costs. Erdem et al. [8] model **MAPF** as a logic program and use an ASP solver to optimize the makespan or the distance.

Considering that part of **D-MAPF** includes **MAPF**, it is not surprising that our ASP-based solution for **D-MAPF** builds upon Erdem et al.'s ASP-based solution for **MAPF**, for computing paths for the new agents. Though, the idea of replanning the waiting times of existing agents and the part of our ASP program that decides which existing agent should wait for how long and where meanwhile, are novel.

MAPF-POST. Hoenig et al. [14] study a simpler problem (called **MAPF-POST**, and solved in polynomial time) to postprocess **MAPF** solutions before execution: given a solution for **MAPF** (i.e., paths of the robots), the goal is to schedule the moves of robots according to their kinematic constraints, like velocity, to avoid collisions between robots, subject to the assumptions that the agents follow their paths and they visit common locations in the same order as suggested by their paths. For instance, suppose that a vertex v is visited by two robots r_1 and r_2, and v is the second vertex in the path of r_1 and the fifth vertex in the path of r_2. While scheduling the moves of these robots, **MAPF-POST** ensures that r_1 visits v before r_2.

D-MAPF (thus our ASP-based solution) is similar in that it schedules the waiting times of the existing agents. On the other hand, it does not pose a constraint about the latter assumption, so an existing agent does not have to visit a location before another existing agent although their paths suggest otherwise. For instance, in the example above, it is not required that r_1 visits v before r_2.

TAPF and G-TAPF. Target Assignment and Path Finding problems (**TAPF** and **G-TAPF**) [19, 23] are variants of **MAPF**: they aim to partition the agents into teams of agents, assign some goals to each team, and solve **MAPF**. Although this problem is different, Nguyen et al.'s solution to **G-TAPF** is related since it also utilizes ASP in the spirit of Erdem et al.'s [8] solution for **MAPF**. However, like Erdem et al.'s [8] solution, Nguyen et al.'s solution to **G-TAPF** do not consider changes in the environment. Our solution for **D-MAPF** does.

MAPD. Like **TAPF** and **G-TAPF**, Multi-Agent Pickup and Delivery (**MAPD**) [20] also involves assignment of tasks (i.e., pick up and delivery) to agents but over time, so it requires an online solution. While **MAPD** has a dynamic aspect of emerging new tasks, it does not allow the team or environment to change (e.g., no existing agent leaves, no new agent joins, no new obstacle appears). On the other hand, **D-MAPF** allows for changes in the team or environment, but does not allow new tasks over time.

Online MAPF. Online **MAPF** [31] considers the addition of new agents to the team while a plan is being executed, but no other changes in the team or environment (e.g., no existing agent leaves, no new obstacle appears). Moreover, it is assumed that agents disappear when they reach their goal and that new agents may wait before entering their initial location in the environment. These assumptions relax the **D-MAPF** problem: the new agents may enter the environment one at a time, and they provide more space for the other agents when they disappear.

To solve online **MAPF** with these assumptions, Svancara et al. investigate algorithms that rely on replanning (e.g., for all agents) and conflict-resolution (e.g., planning for the new agents one at a time ignoring others, and then resolving conflicts by replanning). Our approach does not accommodate such assumptions and tries to find a solution for **D-MAPF** by revising the schedules of the plans of existing agents while augmenting the plans of new agents; replanning is triggered only when revising and augmenting is not sufficient to find a solution.

6 Conclusion

While executing a **MAPF** plan in a dynamic environment, new agents may join the team whereas some may leave, and the environment may change by new obstacles. In such cases, we need to solve a variation of **MAPF**, which we call **D-MAPF**.

We have proposed a declarative solution to **D-MAPF**, based on the idea of re-using the existing solution for **MAPF** (i.e., re-using the paths of the existing agents and scheduling their waiting times to obtain their new traversals, instead of computing new paths and traversals for them) while computing new plans for the new agents. We have realized this method using advantages of ASP: the guidance provided by the generate-and-test representation methodology, the special constructs and capabilities (such as aggregates, weak constraints, choice rules, and recursive definitions), and the efficient ASP solver CLINGO.

We have observed from the results of experiments that the underlying idea of our proposed method (i.e., revising and augmenting the plan being executed) improves the computational efficiency in timings significantly compared to replanning, and is promising for further investigations.

References

[1] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.

[2] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *ACM Commununications*, 54(12):92–103, 2011.

[3] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming: An introduction to the special issue. *AI Magazine*, 37(3):5–6, 2016.

[4] Satyendra Singh Chouhan and Rajdeep Niyogi. DMAPP: A distributed multi-agent path planning algorithm. In *Proc. of AI*, pages 123–135, 2015.

[5] Satyendra Singh Chouhan and Rajdeep Niyogi. DiMPP: a complete distributed algorithm for multi-agent path planning. *J. Exp. Theor. Artif. Intell.*, 29(6):1129–1148, 2017.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, 1959.

[7] Kurt M. Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *J. Artif. Intell. Res. (JAIR)*, 31:591–695, 2008.

[8] Esra Erdem, Doga Gizem Kisa, Umut Oztok, and Peter Schueller. A general formal framework for pathfinding problems with multiple agents. In *Proc. of AAAI*, 2013.

[9] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Communications*, 24(2):107–124, 2011.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of ICLP*, pages 1070–1080. MIT Press, 1988.

[11] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, New York, NY, USA, 2014.

[12] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[13] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, 37:28–29, 1972.

[14] Wolfgang Hoenig, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *Proc. of ICAPS*, pages 477–485, 2016.

[15] Renee Jansen and Nathan Sturtevant. A new approach to cooperative pathfinding. In *Proc. of AAMAS*, pages 1401–1404, 2008.

[16] Jae-Yeong Lee and Wonpil Yu. A coarse-to-fine approach for fast path finding for mobile robots. In *Proc. of IROS*, pages 5414–5419, 2009.

[17] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[18] Ryan Luna and Kostas E. Bekris. Efficient and complete centralized multi-robot path planning. In *Proc. of IROS*, pages 3268–3275, 2011.

[19] Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *Proc. of AAMAS*, pages 1144–1152, 2016.

[20] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proc. of AAMAS*, pages 837–845, 2017.

[21] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[22] Robert Morris, Corina S. Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Planning, scheduling and monitoring for airport surface operations. In *Planning for Hybrid Systems, Papers from AAAI Workshop.*, 2016.

[23] Van Nguyen, Philipp Obermeier, Tran Cao Son, Torsten Schaub, and William Yeoh. Generalized target assignment and path finding using answer set programming. In *Proc. of IJCAI*, pages 1216–1223, 2017.

[24] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[25] Daniel Ratner and Manfred K. Warmuth. Finding a shortest solution for the n $\times$ n extension of the 15-puzzle is intractable. In *Proc. of AAAI*, pages 168–172, 1986.

[26] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.*, 219:40–66, 2015.

[27] David Silver. Cooperative pathfinding. In *Proc. of AIIDE*, pages 117–122, 2005.

[28] Trevor Scott Standley and Richard E. Korf. Complete algorithms for cooperative pathfinding problems. In *Proc. of IJCAI*, pages 668–673, 2011.

[29] Pavel Surynek. On propositional encodings of cooperative path-finding. In *Proc. of ICTAI*, pages 524–531, 2012.

[30] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *Proc. of ECAI*, pages 810–818, 2016.

[31] Jiri Svancara, Marek Vlk, Roni Stern, Dor Atzmon, and Roman Bartak. Online multi-agent pathfinding. In *Proc. of AAAI*, 2019.

[32] Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *Proc. of ICAPS*, pages 380–387, 2008.

[33] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1):9–20, 2008.

[34] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multiple robots on graphs. In *Proc. of ICRA*, pages 3612–3617, 2013.