2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

Should I Replan? Learning to Spot the Right Time in Robust MAPF Execution

David Zahrádka[1,2], David Woller[1], Denisa Mužíková[1,2], Miroslav Kulich[1] and Libor Přeučil[1]

***Abstract*— During the execution of Multi-Agent Path Finding (MAPF) plans in real-life applications, the MAPF assumption that the fleet's movement is perfectly synchronized does not apply. Since some of the agents may become delayed due to internal or external factors, it is often necessary to use a robust execution method to avoid collisions caused by desynchronization. Robust execution methods – such as the Action Dependency Graph (ADG) – synchronize the execution of risky actions, but often at the expense of increased plan execution cost, because it may require some agents to wait for the delayed agents. In such cases, the execution's cost can be reduced while still preserving safety by finding a new plan either by rescheduling (reordering the agents at crossroads) or the more general replanning capable of finding new paths. However, these operations may be costly, and the new plan may not even lead to lower execution cost than the original plan: for example, the two plans may be the exact same, as some losses may not be recoverable at all. Therefore, we estimate the benefit that can be achieved by single replanning in scenarios with delayed agents given an immediate state of the execution with a fully connected feed-forward neural network. The input to the neural network is a set of newly designed ADG-based features describing the execution's state and the impact of potential delays, and the output is an estimated benefit achievable by replanning. We train and test the network on a new labeled dataset containing $12,000$ experiments and show that our proposed method is capable of significantly reducing the impact of recoverable delays.**

I. Introduction

The goal of Multi-Agent Path Finding (MAPF) is to find a set of collision-free paths in a shared environment with discrete space and time, represented as a graph, for a fleet of mobile agents such that the agents reach their given goals. The problem is motivated by applications such as autonomous warehouses [1] and planning for swarms of Unmanned Aerial Vehicles [2]. The goal is often to minimize some cost, such as the time when the last agent reaches its goal (*makespan*) or the sum of lengths of all paths (*Sum of Costs*). While this *optimization variant of MAPF* is an NP-hard problem for many criteria [3], recent solvers are able to solve instances with thousands of agents [4], albeit suboptimally. An example MAPF plan can be seen in Fig. 1.

The plans produced by MAPF solvers assume perfect synchronization of the agent fleet: all agents move at the same time and execute exactly one action in each time step. However, in practical applications, some agents may become unexpectedly delayed compared to the rest of the fleet due to, for example, imprecise control, mechanical issues, or external factors such as a human entering the shared environment and blocking some robots. Safe execution in the presence of unexpected delays can be achieved by using robust execution methods, such as MAPF-POST [5] and Action Dependency Graph (ADG) [6]. These methods take a MAPF plan as an input, find temporal precedences between the actions of the agents and only allow executing an action when all of its precedences are complete, preventing both collisions and deadlocks.

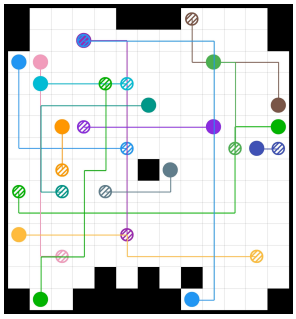


Fig. 1: Example MAPF plan. Full circles are agents, hatched circles their goals, and lines represent paths.

However, with significant delays, it may happen that following the original plan leads to a longer execution than finding and executing a new plan, even if the original plan was optimal. For example, if multiple agents are scheduled to move through a crossroad but the first agent is delayed, the later-scheduled agents may also have to wait, increasing the delay's impact. Changing the order of the agents on the crossroad (*rescheduling*) could reduce it. *Optimization of robust execution* was addressed by the development of robust execution methods capable of rescheduling, such as Switchable ADG (SADG) [7] and Bidirectional Temporal Plan Graph (BTPG) [8]. While SADG requires repeatedly solving a Mixed Integer Linear Problem to determine the agents' order, in BTPG, the ordering is defined by a first-come, first-served manner, although the resulting order may not be optimal. A disadvantage of rescheduling itself is that it is not capable of producing alternative paths, which may be necessary to mitigate the impact of some delays. New paths can be found by *replanning*, which is a more general approach that is also capable of rescheduling [9]. However, it is also more costly.

*This work was supported by European Union under the project Robotics and Advanced Industrial Production (reg. no. CZ.02.01.01/00/22_008/0004590).

[1]Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague, Jugoslávských partyzánů 1580/3, 160 00 Praha 6, Czechia david.zahradka@cvut.cz

[2]Faculty of Electrical Engineering, Czech Technical University in Prague, Karlovo náměstí 13, 121 35 Praha 2, Czechia

A straightforward way to reduce execution duration (cost) is to reschedule/replan often, perhaps even every time step, provided that livelocks are prevented. Since both rescheduling and replanning introduce some overhead, albeit small, frequent re-computation of the plan may cause it to accumulate over time. Furthermore, the same plan may be re-created over and over again, because the solver may not be able to find a plan that would be better than the original one. In some cases, the delay's effect may not even be mitigated at all. Although the negative impact of rescheduling/replanning can be reduced by using a persistent execution method [6], it is still more cost-effective if the frequency of rescheduling/replanning is not high. Predicting the potential benefit that such interventions can bring in order to decide whether to find a new plan or not is, therefore, an important problem which we investigate in this paper.

We use the ADG-based framework proposed in [10] to monitor the progress of a robust execution. We estimate the starting and completion times of all actions and measure the real values as the execution progresses. The measured values are propagated through the ADG to improve the estimate of future actions. For each action, we compute statistics, such as expected and real execution time, and the so-called slack, which represents how long does an agent have to wait for other agents or vice versa [10]. We then use these statistics to train a neural network to predict the potential benefit of replanning, which we use to decide whether to replan or not.

The contributions of this paper are summarized as follows:

1) we augment the ADG with novel metrics that capture the state of execution and the impact of potential delays in real time,
2) we propose a data generation pipeline that produces a labeled dataset, enabling supervised training of a neural network to predict the benefit of replanning,
3) we identify the most informative features and analyze the impact of a dynamic obstacle on execution cost,
4) we experimentally demonstrate that the proposed approach recovers 94.6% of the available cost savings.

II. Related Work

Robust execution methods are used to safely execute MAPF plans in the presence of unexpected delays. A straightforward approach called a Fully Synchronized Policy [11] is to only assign actions belonging to the same time step, and only advancing to the next time step once all agents finished executing their action. However, even independent agents have to wait for the rest of the fleet to complete their actions, increasing the duration of the execution.

More efficient methods include Minimal Communication Policy (MCP) [11], RMTRACK [12], MAPF-POST [5], ADG [6] and Kinodynamic Temporal Plan Graph [13]. MCP [11] synchronizes the agents only on crossroads where the MAPF plan defines a strict order in which the agents move through. RMTRACK [12] facilitates safe execution by ensuring that the trajectories remain in the same homotopy class in configuration space as the original planned trajectories. MAPF-POST [5] constructs a temporal network (TN) with vertices representing events (agents entering locations) and edges representing precedence constraints between the events. An event is scheduled only after all preceding events are scheduled to be completed. A similar approach is used in ADG [6], which is a Temporal Plan Graph (TPG) with vertices representing agents' actions and edges representing their precedences. Safety is ensured by executing an action only after all preceding actions have been completed. Using actions instead of location-related events reduces communication overhead between the central server and the agents. The Kinodynamic TPG [13] introduces kinodynamic constraints into ADG, which allows optimizing the velocities at which the agents move for smoother execution.

Recent research also focuses on **rescheduling** the order in which agents enter crossroads to minimize the duration of the robust execution when an agent is delayed. The problem was formulated as a Job Shop Scheduling Problem and solved with a metaheuristic in [14]. Another approach is to use any standard MAPF solver on a reduced planning graph built from the original paths of the agents [9].

Rescheduling was also integrated into robust execution procedures themselves, allowing to minimize execution cost while maintaining deadlock and collision-free execution. In [7], SADG is introduced, which adds reverse precedences to each precedence between actions of different agents and formulates the problem of which of these two mutually exclusive edges to use as a Mixed Integer Linear Program. In Switchable Edge Search [15], an A*-style algorithm is used on SADG to solve the same problem. Finally, BTPG is proposed in [8], which is built entirely before execution and schedules agents at crossroads in a first-come, first-served manner as the execution progresses.

Machine learning (ML) was successfully applied to many different aspects of MAPF. In [16], multiple image-based ML models were used to select the best planning algorithm from a bank for a given instance. A popular application is to use ML inside existing solvers to improve their efficiency, such as for selecting which part of the solution to destroy in Large Neighborhood Search [17], or to repair the solution in [18]. Learning-based methods were also used as both decentralized solvers via reinforcement learning [19] and centralized solvers using neural networks [20]. In this paper, we use ML to predict the potential benefit of replanning during the robust execution of a MAPF plan.

III. Background & Problem Formulation

We start this section by recalling two fundamental concepts: Multi-Agent Path Finding (MAPF) and Action Dependency Graph (ADG), with notation adopted from [10]. Then, we introduce the Replanning Prediction Problem (RPP), which is the central focus of this paper.

A. Multi-Agent Path Finding

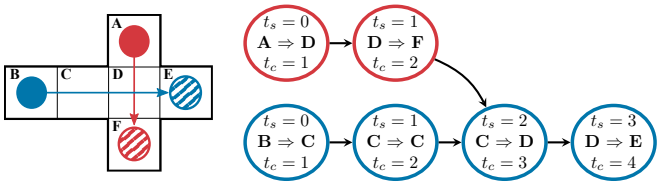
A MAPF instance M is defined as $M = (G, A)$, where $G = (V, E)$ is a graph representing the shared environment and A is a set of agents operating on G. Each agent $k \in A$ has a starting location $s^k \in V$ and a goal location $g^k \in V$.

Then, a plan π^k is a sequence of actions $a_i^k = (v_i, v_{i+1})$, where a_i^k is the i-th action of agent k, representing movement from position $v_i \in V$ to $v_{i+1} \in V$ between discrete time steps i and $i+1$. If $v_i = v_{i+1}$, the agent is waiting in the same position. Finally, the goal of MAPF is to find a set of plans $\{\pi^k = (a_1^k, .., a_{|\pi^k|}^k) \mid k \in A\}$, so that $\forall k \in A : a_1^k = s^k, a_{|\pi_k|}^k = g^k$ and the objective function is minimized. We employ the commonly used criterion Sum of Costs (SOC), defined as

$$SOC = \sum_{k \in A} |\pi^k|, \tag{1}$$

which measures the total execution time across all agents. An example MAPF instance is shown in Fig. 2a.

Each pair of plans must be conflict-free; we prohibit *vertex*, *swap*, *following*, and *cycle* conflicts [21]. Such MAPF solutions are called 1-robust [22], which means that two different agents cannot occupy the same vertex $v \in V$ at the same time step and in two consecutive time steps.



(a) MAPF solution

(b) Corresponding ADG

Fig. 2: Example MAPF instance with two agents

B. Action Dependency Graph

Execution of MAPF plans in practice takes place in the continuous world instead of the discrete environment assumed in MAPF. While it is possible to synchronize the agents so that they perform actions step-by-step, simulating discrete time, it prolongs the duration of the execution [11]. ADG [6] can be used to ensure robust execution of MAPF solutions with uncertain action durations or external factors. It contains the prerequisities of all actions and thus enables executing actions only after their prerequisities are met.

ADG is a directed acyclic graph $G_{ADG} = (V_{ADG}, E_{ADG})$, where V_{ADG} are all actions a_i^k in all plans $\pi_{k \in A}$ and E_{ADG} are edges representing temporal dependencies between different actions. There are two types of edges in E_{ADG}: type 1 edges E_1 for the precedences between actions of a single agent and type 2 edges E_2 for the dependencies between actions of different agents. An example ADG with $E_2 = \{(DF, CD)\}$ is shown in Fig. 2b. The red agent has to move from D to F before the blue agent can move from C to D in the next time step.

An extended version of ADG [10] additionally contains the estimated start time $\hat{t}_s(a_i^k)$ and the estimated completion time $\hat{t}_c(a_i^k)$, where a_i^k is the i-th action of the agent k. These values are periodically updated during the execution to keep track of its progress. Once an action starts, the actual start time $\bar{t}_s(a_i^k)$ is recorded in the ADG and the action is marked as running. After it finishes, the actual completion time $\bar{t}_c(a_i^k)$ is recorded in the ADG, the estimated completion time is set as $\hat{t}_c(a_i^k) = \bar{t}_c(a_i^k)$, the action is marked as completed, and the estimated start and completion times of all subsequent actions are updated. Finally, let us denote the originally planned start and completion times as $t_s(a_i^k)$, $t_c(a_i^k)$, respectively. At the beginning, the values are initialized as follows: $\forall k \in A, \forall a^k \in \pi^k : \hat{t}_c(a^k) = t_c(a^k)$, $\bar{t}_c(a^k) = \infty$, $\hat{t}_s(a^k) = t_s(a^k)$, $\bar{t}_s(a^k) = \infty$.

C. Replanning Prediction Problem

We now define the problem of interest in this paper, the RPP. It aims to predict whether replanning at a given time during the ADG-controlled MAPF execution will be beneficial. To enable this, various metrics within the ADG are monitored and used as input features to the prediction. Formally, let $x \in \mathbb{R}^d$ denote the feature vector. Let $y \in \mathbb{R}$ denote the predicted variable, representing the expected savings in SOC that would be achieved by invoking replanning. The RPP goal is to learn a mapping

$$f : \mathbb{R}^d \to \mathbb{R}, \quad \text{such that } f(x) \approx y. \tag{2}$$

We formulate RPP as a regression problem to allow for flexible thresholding and to preserve interpretability. A binary decision is then obtained by applying a threshold $\tau \in \mathbb{R}$ to the regression output: replan, if $f(x) \geq \tau$. In our experimental setup, the regression target y is defined as

$$y = SOC^{ei} - SOC_t^{eir}, \tag{3}$$

where SOC^{ei} denotes the *executed* SOC (in seconds) of the initial MAPF plan *influenced* by external disturbances, and SOC_t^{eir} denotes the *executed* SOC of the same scenario if *replanning* was triggered at time $t \in \mathbb{R}$. Therefore, a positive value of y indicates that savings are expected. A negative value would mean that replanning is expected to actually increase the execution cost, which may happen if we take into account the time required to find a new plan.

IV. Method

Now, we present the methodology for addressing the RPP. Our main contribution is the definition of the features x (IV-A). Sec. IV-B outlines the generation of the labeled dataset of observations (x, y), and Sec. IV-C describes the learning model used to approximate the regression function $f(x)$.

A. Features definition

To find out which metrics are the best predictors of possible savings, we enumerate various properties of the instance, the solution, and the execution state as scalar features. Most proposed features are derived from the current ADG state during MAPF execution. The feature vector x changes whenever the ADG is updated, ensuring that it reflects the latest execution state. Replanning may be triggered at each ADG update based on the predicted benefit $f(x)$. To ensure that our approach is algorithm-agnostic, we do not use any algorithm-related features.

Table I lists the 18 distinct features along with their formal definitions. Several features are parameterized by n, which

specifies the number of past actions over which the feature is evaluated. For clarity, the table is split into three groups: MAPF instance properties, MAPF solution properties, and current ADG state properties. The main idea is to detect present or expected deviations of ADG-controlled execution at the current time t from the original MAPF plan.

First, we introduce some auxiliary notations and symbols. Let $p^k(t)$ be the *progress* of agent $k \in A$, corresponding to the index of the last finished action at time t in its plan $\pi^k = (a^k_1, .., a^k_{|\pi^k|})$. Then, the last n finished actions of agent k at time t can be defined as $P(k,t,n) = \{a^k_{\max(1,p^k(t)-n+1)}, .., a^k_{p^k(t)}\}$. If no action has finished yet, $p^k(t) = 0$ and $P(k,t,n) = \emptyset$. Analogously, let $e^k(t)$ be the index of the last action assigned to agent k. If $e^k(t) > p^k(t)$, agent k is currently executing the action $a^k_{e^k(t)}$. Otherwise, agent k is either finished (if $p^k(t) = |\pi^k|$) or blocked by the ADG, meaning that he is waiting for another agent. Again, let's denote the last n actions assigned to agent k at time t as $E(k,t,n) = \{a^k_{\max(1,e^k(t)-n+1)}, .., a^k_{e^k(t)}\}$. If no action was assigned yet, $e^k(t) = 0$ and $E(k,t,n) = \emptyset$.

We define several features that capture *action delay*, quantifying the excess execution time of the considered actions. This is computed by measuring the deviation between the planned duration of an action, $d(a) = t_c(a) - t_s(a)$, and its actual duration, $\bar{d}(a) = \bar{t}_c(a) - \bar{t}_s(a)$, or expected duration, $\hat{d}(a) = \hat{t}_c(a) - \hat{t}_s(a)$. In addition, we define *plan delay* features, which evaluate the difference between the actual or expected completion times ($\bar{t}_c, \hat{t}_c$) and the planned completion times t_c. Unlike action delay features, plan delay features also capture indirect effects, such as excess waiting time for agents without an assigned action.

One of the features, *highest slack increase*, is adopted from [10]. It is designed to identify when the current execution delay is likely to cause excess waiting in the future. For every type 2 edge $e \in E^2$, $e = (a^k_i, a^l_j)$ between actions of agents k and l, we define *planned slack* as $\delta(e) = t_c(a^k_i) - t_c(a^l_{j-1})$ and *expected slack* as $\hat{\delta}(e) = \hat{t}_c(a^k_i) - \hat{t}_c(a^l_{j-1})$. For agent l, a positive *planned slack* value means that, according to the initial plan, agent l will wait for agent k for $\delta(e)$ seconds. If the *expected slack* increases relative to *planned slack*, agent l might need to wait longer than originally planned.

Let us also define the set of satisfied dependencies between two different agents in E_{ADG} as $\bar{E}^2$. If an action that is a dependency of another action is completed, it is inserted into $\bar{E}^2$. Finally, let $[\![f]\!] = 1$, if formula f is true, 0 otherwise.

TABLE I: Features x, defined for the RPP

Name	**Description**
map height, map width	Dimensions of the grid environment
agents count	No. of agents in the MAPF instance: $\|A\|$
planned SOC	$SOC = \sum_{k \in A} \|\pi^k\|$
planned makespan	$\max_{k \in A} \|\pi^k\|$
replan time	Current time t; potential replanning time
unfinished agents count	No. of agents not in their goals yet: $\sum_{k \in A} [\![p^k(t) \neq \|\pi^k\|]\!]$
progress gap	Max. progress difference between two agents: $\max_{k \in A}(p^k(t)) - \min_{k \in A}(p^k(t))$
highest plan delay	Highest plan delay among last finished actions: $\max_{k \in A}(\bar{t}_c(a^k_{p^k(t)}) - t_c(a^k_{p^k(t)}))$
highest exp. plan delay	Highest expected plan delay among currently assigned actions: $\max_{k \in A}(\hat{t}_c(a^k_{e^k(t)}) - t_c(a^k_{e^k(t)}))$
total plan delay	Total plan delay over all last finished actions: $\sum_{k \in A}(\bar{t}_c(a^k_{p^k(t)}) - t_c(a^k_{p^k(t)}))$
total exp. plan delay	Total expected plan delay over all currently assigned actions: $\sum_{k \in A}(\hat{t}_c(a^k_{e^k(t)}) - t_c(a^k_{e^k(t)}))$
highest action delay (n)	Highest total delay of last n finished actions: $\max_{k \in A}(\sum_{a \in P(k,t,n)}(\bar{d}(a) - d(a)))$
highest exp. action delay (n)	Highest expected total delay of last n assigned actions: $\max_{k \in A}(\sum_{a \in E(k,t,n)}(\hat{d}(a) - d(a)))$
total action delay (n)	Total delay of last n finished actions: $\sum_{k \in A}(\sum_{a \in P(k,t,n)}(\bar{d}(a) - d(a)))$
total exp. action delay (n)	Total expected delay of last n assigned actions: $\sum_{k \in A}(\sum_{a \in E(k,t,n)}(\hat{d}(a) - d(a)))$
highest slack increase	Highest slack increase over all unmet dependencies between agents: $\max_{e \in E^2 \setminus \bar{E}^2}(\hat{\delta}(e) - \delta(e))$
waiting agents count	Number of agents not currently executing an action: $\sum_{k \in A} [\![e^k(t) = p^k(t)]\!]$

B. Labeled training and testing dataset

In order to train our prediction model, we design a labeled dataset of observations (x, y). Problem instances are randomly generated with varying numbers of agents, map sizes, and multiple random start-goal assignments for the agents. For each instance, we use multiple combinations of varying replanning times t and dynamic obstacles to obtain more results. The entries in the dataset then contain the measured *execution* cost of an optimal 1-robust plan *influenced* by the dynamic obstacle SOC^{ei}, where no replanning occurred, the execution cost of the same plan disturbed by the same obstacle, when *replanning* occurred at time t: SOC^{eir}_t, and the feature vector x as measured at the time t. From these, the regression target y is computed (Eq. 3).

Additionally, we measure the execution time without any disturbance: SOC_e and with replanning, including the *planning overhead* caused by the replanning process:

$$SOC^{eirp}_t = SOC^{eir}_t + t_r \times \sum_{k \in A} [\![p^k(t) \neq |\pi^k|]\!], \quad (4)$$

where t_r is the runtime of the solver and the summation counts the agents that have not yet finished at time t. This term quantifies the additional SOC lost while replanning. We do not use SOC^{eirp}_t to define the regression target y in Eq. 3, as we aim to keep the key experiments independent of the specific solver employed. It is used only to provide further insight into the performance of our method.

C. Learning model

We employ a lightweight, fully connected feed-forward neural network, implemented using scikit-learn [23] and

TensorFlow [24], to learn the mapping $f : \mathbb{R}^d \to \mathbb{R}$. It consists of an input layer, three hidden layers with 64, 32, and 16 neurons, respectively, each using ReLU activation, and a single linear output neuron predicting the expected SOC savings $f(x)$. Prior to training, we apply the `RobustScaler` from scikit-learn to both input x and y, centering their distributions at zero and scaling it so that the interquartile range equals one. The network is trained to minimize the mean absolute error (MAE) loss:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{N} \sum_{i=1}^{N} \left| f(x_i) - y_i \right|, \tag{5}$$

where N is the number of training samples. Together, robust scaling and MAE loss reduce the model's sensitivity to outliers. We use the Adam optimizer with the step decay learning rate $\eta(s)$ defined as

$$\eta(s) = \eta_0 \, \gamma^{\lfloor s / s_{\text{decay}} \rfloor}, \tag{6}$$

where s is the current training step, η_0 the initial learning rate, γ the decay rate, and s_{decay} the decay step size. Training is performed in mini-batches of size B with early stopping based on validation to prevent overfitting. The final model was designed and tuned using k-fold cross-validation.

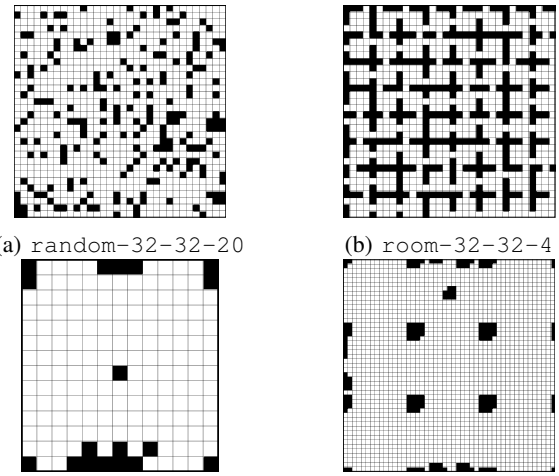
V. Experimental Setup

In this chapter, we describe the experimental environment (Sec. V-A), detail the instance generation (Sec. V-B), and present the parameters related to the dataset handling, learning model design, and the tuned hyperparameters (Sec. V-C).

A. Experimental environment

We use a 1-robust ECBS [25] solver with suboptimality bound 1.0 to find both optimal initial plans and new optimal plans when replanning. To measure their execution cost, we use a simulated environment [10]. It consists of a central robust execution server that manages threads of all agents. The server employs ADG to decide whether each action is safe to execute and also to monitor the execution by recording various execution metrics. Each action takes exactly $1\,\text{s}$ and the agents execute them perfectly, with only a small overhead generated by the centralized execution architecture. However, while AGD prevents inter-agent collisions, a dynamic obstacle (an intrusion) may appear in the environment and block a vertex that is currently free and no agent is moving in or out of it. Agents avoid colliding with the dynamic obstacle by checking whether the goal location of an action is free when starting its execution. If there is an obstacle, the agent starts moving only after it clears. This way, delays are introduced to the otherwise perfectly moving agents.

The dynamic obstacle appears and disappears at random times in a random vertex. We select a random appearance time at least $3\,\text{s}$ before the plan is estimated to be completed, as indicated by makespan. It appears in a currently unoccupied vertex that is to be visited by an agent $3\,\text{s}$ after the appearance time. The $3\,\text{s}$ are a time buffer that ensures that the dynamic obstacle has sufficient opportunities to enter the vertex. The obstacle remains in the vertex before its randomized disappearance time elapses, which is no sooner than $3\,\text{s}$ after it appears to guarantee that it will affect the blocked agent, and no later than the plan makespan. The appearance and disappearance times and the vertex are all controlled by the dynamic obstacle randomization seed.



(a) `random-32-32-20` (b) `room-32-32-4` (c) `lab` (d) `arena`

Fig. 3: Maps used in experiments

Each measured execution cost is obtained through an independent simulator run. To mitigate stochastic errors caused by multithreading, which may influence the results measured by the simulator, we repeat each experiment three times and select the minimal measured values.

B. Instances

We used four maps of various sizes: `random-32-32-20` and `room-32-32-4`, which are maps with 32×32 cells taken from the MovingAI dataset [21] and `arena` and `lab` from [10] with dimensions 49×49 and 13×14, respectively. The maps can be seen in Fig. 3. For each map, we used three different numbers of agents: $\{15, 20, 25\}$ for `arena` and $\{5, 10, 15\}$ for `lab`, `random` and `room` maps. For each map and number of agents, we generated 20 instances that were solvable with the optimal 1-robust solver within $60\,\text{s}$. We run each instance with 5 dynamic obstacle randomization seeds and 10 replanning seeds, producing 3000 different experiments per map, thus 12000 experiments in total. Exactly one replanning time t was always generated at a random time before the obstacle's appearance and the remaining 9 are generated at random times after it appears. We use this bias to reduce the number of instances in which replanning is guaranteed to provide no benefit. The cost SOC_t^{eir} obtained by replanning at time t also serves as our random replanning baseline biased towards effectiveness. After generation, the dataset is split into training set $\mathcal{D}_{train}$ and test set $\mathcal{D}_{test}$, so that $|\mathcal{D}_{train}| = 8400$ and $|\mathcal{D}_{test}| = 3600$.

C. Learning parameters

We employed 5-fold cross-validation on $\mathcal{D}_{train}$ for model design and hyperparameter tuning. The final model is trained

with a batch size of $B = 64$, meaning that weight updates are computed after processing 64 samples. Training is performed for at most $E_{\max} = 500$ epochs, with early stopping triggered if the validation loss does not improve for $E_{\text{stop}} = 100$ consecutive epochs. The validation loss is computed on a randomly sampled subset of the training set using the validation split ratio $\rho_{\text{val}} = 0.2$. The remaining hyperparameters, that were already introduced in Sec. IV-C, were set as follows: $\eta_0 = 0.001$, $\gamma = 0.96$, and $s_{\text{decay}} = 100$.

VI. Results & Discussion

We evaluate the complete pipeline for the RPP through a series of experiments and compare it to the random replanning baseline method. First, we compare different execution scenarios (Sec. VI-A) to assess the influence of intrusion, random replanning and computational overhead. Next, we analyze the prediction performance of the proposed model (Sec. VI-B). Then, we evaluate the realized savings when the model is used to trigger replanning (Sec. VI-C). Finally, we assess the contribution of individual features (Sec. VI-D).

A. Execution Scenarios

Three simulator scenarios are required to generate a single experiment, corresponding to one labeled dataset entry. Fig. 4 compares these scenarios in terms of the Sum of Costs (SOC). The following analysis is performed on the entire dataset of 12000 experiments.

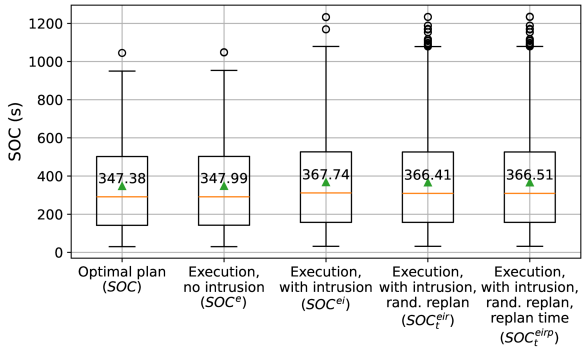


Fig. 4: SOC across all scenarios

The average SOC of the initial plans is 347.38 s. When executed in the real-time simulator, the average measured SOC^e increases only slightly to 347.99 s, which means that the simulator does not introduce fundamental errors into the execution. The minor increase can be attributed to the overhead introduced by the ADG, which coordinates up to 25 parallel threads corresponding to individual agents.

Once the dynamic obstacle is introduced, the average SOC^{ei} increases to 367.74 s, representing a 5.6% increase compared to SOC^e. The absolute increase ($SOC^{ei} - SOC^e$) can be seen in Fig. 5. The plot shows that, for most runs, the dynamic obstacle has only a minor impact: in 50% of all runs, the increase is below 10.90 s. However, in 5% of the runs, the increase exceeds 73.93 s, corresponding to more than 20% of the average SOC^e. The first objective of the RPP is therefore to identify during execution these relatively rare but high-impact cases where the dynamic obstacle causes significant disturbance.

Let us now focus on the fourth boxplot in Fig. 4, which visualizes SOC_t^{eir} - the total execution cost in the presence of a dynamic obstacle when replanning is triggered at the randomly sampled time t and the resulting plan is executed until completion. The distribution is nearly identical to SOC^{ei}, leading to an important conclusion: although the solver is optimal, blindly replanning at an arbitrary time yields almost no benefit, even without considering the computational overhead. Moreover, SOC_t^{eir} exhibits a higher number of outliers, indicating that in rare cases, replanning can be harmful. Two factors may explain this. First, the dynamic obstacle is unobservable. Thus, the new plan may be disrupted even more than the original one. Second, some agents may be in the middle of executing an action at time t, causing the ADG to delay the start of the new plan until those are completed, which can further increase the total cost.

Finally, the last boxplot in Fig. 4 visualizes SOC_t^{eirp}, which also accounts for the computational overhead caused by replanning, as defined in Eq. 4. Its distribution is nearly identical to SOC_t^{eir}, which is a positive result: even though the solver is configured to return optimal solutions, it introduces only a negligible increase in the average execution cost for a single replanning. This observation, however, may not hold for significantly larger maps than those considered in this study, where each replanning may be more costly.

Since random replanning does not significantly improve the average case, the key question is whether it can provide benefits in extreme cases. Fig. 6 addresses this question by showing the histogram of the regression target y, defined in Eq. 3, which represents the achieved savings. For a

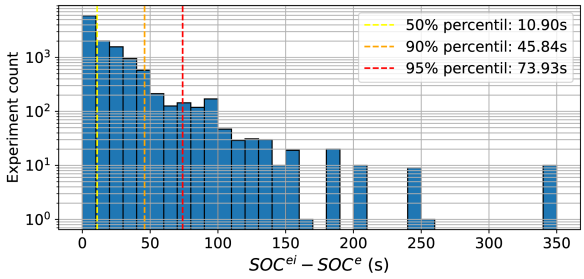


Fig. 5: SOC increase caused by dynamic obstacle

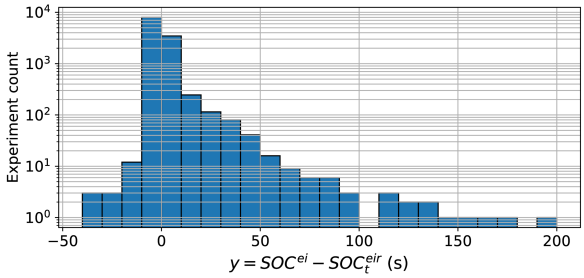


Fig. 6: SOC savings achievable by random replanning

subset of 7.28% of all instances, replanning yields savings of up to 200 s - substantial given that the average $SOC^e = 347.99$ s. Thus, the RPP aims not only to detect whether the disturbance has a significant impact on SOC, but also to determine whether replanning can mitigate it.

B. Predicted SOC Savings

In this section, we evaluate the prediction performance of the proposed approach. All experiments are conducted on the $\mathcal{D}_{test}$, which contains 3600 samples, including 254 positive cases (P), where replanning achieves a saving, and 3364 negative cases (N). The trained model achieves $\mathcal{L}_{\text{MAE}}^{\text{train}} = 1259.66$ on the training set and $\mathcal{L}_{\text{MAE}}^{\text{test}} = 1275.70$ on the test set. Fig. 7 visualizes the predictions on the test set alongside their classification into positive and negative categories. The classification threshold is set to $\tau = 1$ s to avoid treating small variations in runtime as meaningful savings.

The model achieves satisfactory sensitivity of 0.906 and a specificity of 0.979. Thus, selected positive and negative cases are very likely to be true positives and true negatives. This means that while the prediction accuracy among the true positive cases may be lower ($\mathcal{L}_{\text{MAE}}^{\text{TP}} = 14065$), the classification was accurate. The precision is 0.764, indicating a relatively high proportion of false positives. However, these false positives correspond to instances with true savings close to zero, meaning they trigger unnecessary replanning, but do not have any harmful effect on final execution cost. The resulting F1 score is 0.829.

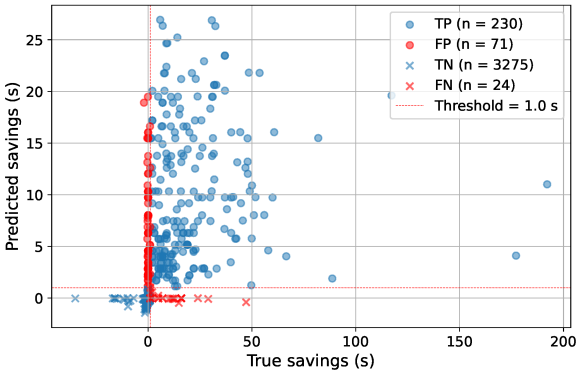
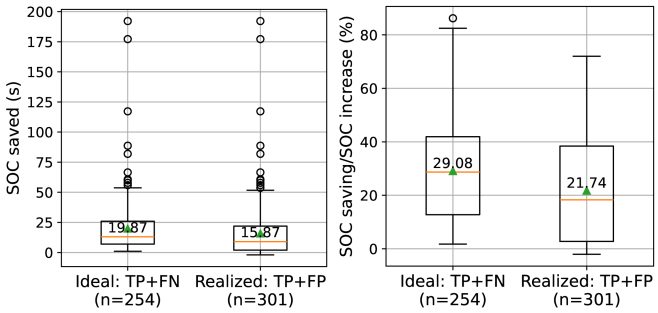


Fig. 7: True vs. predicted SOC savings (test set)

C. Realized SOC Savings

We now examine the *realized SOC savings*, i.e., the actual reduction in SOC^{ei} achieved when replanning is triggered according to the tuned model. These results are summarized in Fig. 8a. When aggregated over $\mathcal{D}_{test}$, the realized replanning (TP + FP) achieved total savings of 4775.366 s out of a potential 5047.336 s (TP + FN), corresponding to a 94.6% recovery rate. In practice, our approach triggered replanning in 301 cases (TP + FP), including 71 false positives, where the realized saving is below the threshold $\tau = 1$ and can even be negative. In the ideal case of perfect prediction on the test set, only the 254 positive cases (P = TP + FN) would be identified, yielding an average saving of 19.87 s



(a) Absolute (b) Relative

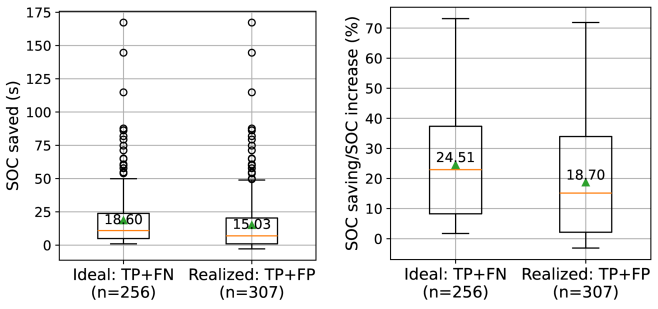
Fig. 8: Realized SOC savings per replanning

per experiment in which replanning is invoked. The average saving per realized replanning, which aggregates TP and FP, was 15.87 s. We also report the SOC savings relative to the SOC increase in Fig. 8b. In the ideal case, replanning would recover on average only 29.08% of the SOC increase across the 254 experiments where a saving greater than the threshold τ is possible. With our approach, we achieve an average relative saving of 21.74% over the 301 experiments in which replanning was actually invoked. In extreme cases, the realized replanning recovered more than 70% of the SOC increase, corresponding to nearly 200 s. Conversely, the worst false positive resulted in a loss of only 2 s.

Fig. 9 shows the realized SOC savings when the planning overhead is included, i.e., when y from Eq. 3 is redefined as $y = SOC^{ei} - SOC_t^{eirp}$. The results closely match those obtained when considering only the dynamic obstacle's impact (Fig. 8). This is a noteworthy finding, as the newly proposed features are designed to only quantify the disturbance caused by the dynamic obstacle, not to assess the computational difficulty of the replanning task or the associated overhead.

D. Features importance

In the final experiment, we analyze the importance of the individual features. Table I lists the 18 proposed feature types, some of which are further parameterized with $n \in \{1, 3, 5, 7, 10, 15, 20\}$, resulting in a total of 42 input features. Fig. 10 shows the permutation importance computed for



(a) Absolute (b) Relative

Fig. 9: Realized SOC savings per replanning, including solver time

the model trained on the test set. Permutation importance quantifies the increase in prediction error when the values of a given feature are randomly shuffled. For clarity, only features with an importance greater than 0.1 are displayed.

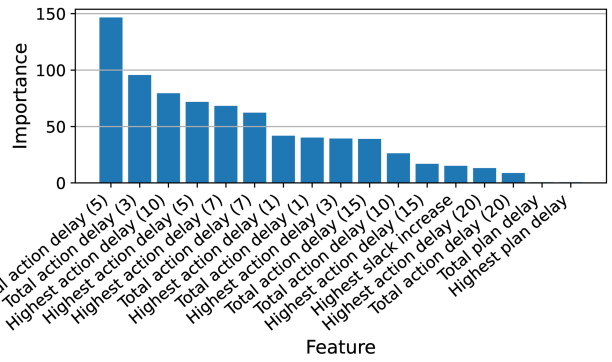


Fig. 10: Features - permutation importance (≥ 0.1)

The results indicate that only three feature types are strong predictors for triggering replanning: total action delay, highest action delay, and highest slack increase. Slack increase, which is relatively unimportant, was the only metric previously used to invoke replanning [10]. Interestingly, static instance or plan properties, expected plan delays, and features describing the current execution progress, appear to be irrelevant. The decision to replan is primarily driven by the magnitude of the already observed execution delays and their future propagation, reflected in slack increase.

VII. Conclusions & Future Work

We formulated the *Replanning Prediction Problem (RPP)* and showed that a simple, planner-agnostic regression model can reliably decide when to replan during ADG-controlled MAPF execution with an unobservable dynamic obstacle. We introduced a bank of 18 feature types and, using a dataset of 12000 experiments on four maps, demonstrated that random replanning offers little average benefit, whereas a minority of cases exhibit large recoverable losses. Our learned model effectively captures these high-impact cases: with a single replanning opportunity, it recovers 94.6% of the available savings across the entire test dataset. In extreme cases, it recovers over 70% of the execution cost per run, while false positives lead to negligible losses. Feature-importance analysis shows that decisions are driven primarily by observed action delays and their propagation (slack increase), whereas static instance or plan attributes contribute little. Notably, the method remains effective even when replanning computation overhead is included. Future work will include spatial features and extend the setting to sequential decisions with multiple replannings, larger maps where planning costs may dominate, and richer disturbance models (e.g., multiple or partially observable dynamic obstacles or failures).

References

[1] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses," *AI Magazine*, vol. 29, no. 1, pp. 9–9, 2008.

[2] W. Hönig, *et al.*, "Trajectory Planning for Quadrotor Swarms," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 856–869, 2018.

[3] P. Surynek, "An Optimization Variant of Multi-Robot Path Planning Is Intractable," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, pp. 1261–1263, 2010.

[4] K. Okumura, "Engineering LaCAM*: Towards Real-time, Large-scale, and Near-optimal Multi-agent Pathfinding," in *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '24. International Foundation for Autonomous Agents and Multiagent Systems, 2024, pp. 1501–1509.

[5] W. Hönig, *et al.*, "Multi-Agent Path Finding with Kinematic Constraints," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 26, pp. 477–485, 2016.

[6] W. Hönig, *et al.*, "Persistent and robust execution of mapf schedules in warehouses," *IEEE Robotics and Automation Letters*, vol. 4, pp. 1125–1131, 4 2019.

[7] A. Berndt, *et al.*, "Receding Horizon Re-Ordering of Multi-Agent Execution Schedules," *IEEE Transactions on Robotics*, vol. 40, pp. 1356–1372, 2023.

[8] Y. Su, R. Veerapaneni, and J. Li, "Bidirectional Temporal Plan Graph: Enabling Switchable Passing Orders for More Efficient Multi-Agent Path Finding Plan Execution," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, pp. 17 559–17 566, 2024.

[9] J. Kottinger, *et al.*, "Introducing Delays in Multi Agent Path Finding," *Proceedings of the International Symposium on Combinatorial Search*, vol. 17, pp. 37–45, 2024.

[10] D. Zahrádka, *et al.*, "A Holistic Architecture for Monitoring and Optimization of Robust Multi-Agent Path Finding Plan Execution," Sept. 2025. [Online]. Available: https://arxiv.org/abs/2509.10284

[11] H. Ma, T. K. S. Kumar, and S. Koenig, "Multi-Agent Path Finding with Delay Probabilities," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[12] J. Gregoire, M. Čáp, and E. Frazzoli, "Locally-optimal multi-robot navigation under delaying disturbances using homotopy constraints," *Autonomous Robots*, vol. 42, no. 4, pp. 895–907, 2018.

[13] J. Yan, S. F. Smith, and J. Li. WinkTPG: An Execution Framework for Multi-Agent Path Finding Using Temporal Reasoning.

[14] D. Zahrádka, D. Kubišta, and M. Kulich, "Solving Robust Execution of Multi-Agent Pathfinding Plans as a Scheduling Problem," 2023.

[15] Y. Feng, *et al.*, "A Real-Time Rescheduling Algorithm for Multi-robot Plan Execution," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 34, pp. 201–209, 2024.

[16] W. Chen, *et al.*, "No Panacea in Planning: Algorithm Selection for Suboptimal Multi-Agent Path Finding." [Online]. Available: https://arxiv.org/abs/2404.03554

[17] T. Huang, *et al.*, "Anytime Multi-Agent Path Finding via Machine Learning-Guided Large Neighborhood Search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 9, pp. 9368–9376, 2022.

[18] Y. Wang, *et al.*, "LNS2+RL: Combining Multi-agent Reinforcement Learning with Large Neighborhood Search in Multi-agent Path Finding," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 22, pp. 23 343–23 350, 2025.

[19] M. Damani, *et al.*, "PRIMAL2: Pathfinding via Reinforcement and Imitation Multi-Agent Learning – Lifelong," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 2666–2673, 2021. [Online]. Available: http://arxiv.org/abs/2010.08184

[20] Y. Tang, *et al.*, "RAILGUN: A Unified Convolutional Policy for Multi-Agent Path Finding Across Different Environments and Tasks (Extended Abstract)," *Proceedings of the International Symposium on Combinatorial Search*, vol. 18, pp. 273–274, 2025.

[21] R. Stern, *et al.*, "Multi-agent pathfinding: Definitions, variants, and benchmarks," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 10, 2019, pp. 151–158.

[22] D. Atzmon, *et al.*, "Robust multi-agent path finding and executing," *Journal of Artificial Intelligence Research*, vol. 67, pp. 549–579, 3 2020.

[23] F. Pedregosa, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[24] M. Abadi, *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Mar. 2016. [Online]. Available: https://arxiv.org/abs/1603.04467

[25] M. Barer, *et al.*, "Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem," in *Seventh Annual Symposium on Combinatorial Search*, July 2014.