Solving Robust Execution of Multi-Agent Pathfinding Plans as a Scheduling Problem

David Zahrádka,[1,2] **Daniel Kubišta,** [1] **Miroslav Kulich**[2]

[1] Faculty of Electrical Engineering, Czech Technical University in Prague
[2] Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague
david.zahradka@cvut.cz, kubisdan@fel.cvut.cz, kulich@cvut.cz

Abstract

Multi-Agent Path Finding (MAPF) is the problem of planning collision-free trajectories for a team of synchronized mobile robots. In real-life applications, however, unexpected delays may occur during the execution of the plans, leading to desynchronization of the agents and possible collision risk. State-of-the-art methods are able to ensure safe execution of MAPF plans in the presence of delays by enforcing the order of agents' actions as given by the MAPF plan to maintain synchronization. This increases execution time, since it requires all agents scheduled to visit a vertex after the agent that is delayed to wait, accumulating the same delay. We propose a scheduling-based approach based on the Job Shop Scheduling Problem (JSSP) as a method for robust execution of MAPF plans. Our proposed approach is able to re-order agents visits to vertices, reducing the impact a delayed agent has on the plan's execution cost. Furthermore, we present a Variable Neighborhood Search method with a newly designed random operator to solve the proposed scheduling problem.

1 Introduction

Multi-Agent Pathfinding (MAPF) is the problem of planning collision-free trajectories for a team of mobile robots (agents) such that each agent moves from its starting position to its given goal position. The agents are assumed to be synchronized, meaning that all the agents move at the same time, and are commonly assumed to be homogeneous. In real-life applications, however, it is possible that otherwise similar robots may behave differently due to inaccuracies in control. This can lead to desynchronization of the agents, and in turn, the execution of the plan may differ from the plan itself. The deviation from the original plan can cause collisions or cause robots to get stuck (Barták et al. 2018). An overview of the challenges in the real-life application of MAPF is available in (Ma et al. 2017).

A possible solution is to re-plan the trajectories whenever a delay of an agent is detected. The problem can be solved optimally using, e.g., Conflict Based Search (CBS) (Sharon et al. 2015), which however may be both time and memory consuming. Therefore, bounded suboptimal solvers, such as Enhanced CBS (ECBS) (Barer et al. 2014), which guarantee that the returned solution is within a constant factor of the optimal solution, are often used. However, even when using faster solvers, replanning is time consuming and often ineffective, since a delay of one agent might not require replanning all agents. Therefore, in order to deal with possible delays during execution, robust planning and execution methods that guarantee safe execution even in the presence of delayed agents are of interest.

Robust planning methods, such as k-robust MAPF methods (Atzmon et al. 2020), guarantee robustness to delays up to some length. For example, in case of k-robust planning, the length of the maximum safe delay is k time steps. This is achieved by ensuring safety distances between agents such that if any agent is delayed, the plan remains collision-free. However, this negatively affects solution cost and when the delay of any agent becomes too large, safe execution is no longer guaranteed.

The state-of-the-art methods for robust execution are able to ensure safe execution of MAPF plans for delays of any duration. They do so by enforcing the order of agents' actions as they were determined by the MAPF plan. However, delays of agents during execution negatively impact the resulting cost of the solution, such as the time when the plan execution is finished.

Consider an example where two agents need to move through a crossroad, but only one agent fits at a time. One agent will necessarily move through earlier than the other. The order in which the agents move through the crossroad is given by the MAPF plan, and the second agent may not enter the crossroad before the first agent leaves it. However, by enforcing the originally planned order in which the agents cross it, a delay of the agent scheduled earlier causes the second agent to accumulate the same delay. Meanwhile, if the earlier-scheduled agent is sufficiently delayed, the second agent may safely pass through the crossroad, inverting their scheduled order of visits. The state-of-the-art methods are unable to perform this, as they are unable to modify the plan that is being executed. This is detrimental to the resulting execution time. By developing methods which are able to divert from the order of agents' visits to vertices as given by the original plan, better execution can be achieved. This can increase the efficiency of the mobile robot fleet application.

We propose formulating the problem of robust execution as a scheduling problem inspired by the Job Shop Scheduling Problem (JSSP). The newly proposed formulation is capable of reordering the agents' visits to vertices while ensuring robust execution by expressing each visited cell as a shared resource. In case of the crossroad example, our proposed problem formulation is able to let the second agent pass the crossroad if the first agent is delayed. Furthermore, we propose a method based on the Variable Neighborhood Search (VNS) (Mladenović and Hansen 1997) to solve the resulting scheduling problem. We show that it outperforms current state-of-the-art robust execution methods regarding the resulting cost of the solution after execution by minimizing the impact the agents' delays have.

2 Related Works

There are several methods that address the problem of robustness by incorporating information about possible delays during execution into path planning itself (Ma, Kumar, and Koenig 2017; Atzmon et al. 2020). A method to plan the paths for agents centrally while considering the possibility of delays was presented in (Ma, Kumar, and Koenig 2017). It combines a new problem formulation of MAPF with delay probabilities, which restricts agents moving into vertices that were occupied by another agent in the previous time step, together with two decentralized robust execution policies, which rely on agent-to-agent communication. The concept of k-robust MAPF exists, in which a plan is robust as long as no agent is delayed by more than k time steps (Atzmon et al. 2020). As such, it preempts execution issues due to imperfect agent synchronization by adding additional constraints on the agents. However, this is reflected in the cost of the solution.

Another approach is to use robust execution methods, which are able to ensure safe execution of MAPF plans even in the presence of delays and kinematic constraints, which would otherwise cause desynchronization and possible collision risk. Current state-of-the-art robust execution methods include MAPF-POST (Hönig et al. 2016) and Action Dependency Graphs (Hönig et al. 2019). Both methods enforce synchronization by detecting dependencies between the agents actions and allowing the agents to start moving into a vertex (location) only after all preceding actions relating to the vertex are finished. In other words, agents enter locations in the order that is given by the MAPF plan. An example of this can be delaying an agent that needs to traverse a crossroad until another agent, which was scheduled to enter the crossroad earlier in the plan, finishes leaving it. The difference between the two methods is that MAPF-POST constructs a Simple Temporal Network and encodes the precedence constraints for the agents' states, whereas the Action Dependency Graph encodes them for agents' actions instead, requiring less communication at runtime, since the agents only report finishing an action in contrast to reporting their current state.

3 Problem Formulation

3.1 MAPF

MAPF with n agents $A_1, \dots, A_n$ is specified by a triple (G, s, g), where $G = (V, E)$ is an undirected graph, $s : [A_1, \dots, A_n] \to V$ is a function mapping an agent to the source vertex, and $g : [A_1, \dots, A_n] \to V$ maps an agent to the goal vertex. Each vertex represents a location in a discretized space. Time is discretized into time steps, and in every time step, each agent is situated in one of the graph vertices and performs a single action. An action is a function $a : V \to V$ that maps a vertex where an agent is currently located to the vertex where the agent will be in the next time step. An agent located in a vertex $v \in V$ can perform two types of actions: *move* to another vertex $v', (v, v') \in E$, or *wait* to remain in v. Both actions have uniform time cost. For a sequence of actions $\pi = (a_1, \dots, a_k)$ and an agent A_j, we denote by $\pi_j[x]$ the location of the agent after executing the first x actions (Stern et al. 2019; Felner et al. 2017). The locations of the agents are assumed to be known at all times, and the agents are assumed to be synchronized – that is, all agents move at the same time and perform exactly one action per time step. The goal is to find a set of collision-free paths (paths without conflicts) such that each agent starts at its starting location and ends at its goal location.

There are two main types of conflicts that can occur: (i) vertex conflict, and (ii) swapping conflict. A vertex conflict occurs when two or more agents occupy the same vertex at the same time. Two agents have a swapping conflict if they swap their location over the same edge at the same time. In some cases, two more types of conflicts are recognized (Stern et al. 2019): (iii) following conflict, where an agent is planned to enter a vertex in time $t + 1$ that another agent occupies in time i, and (iv) cycle conflict, where a group of agents plans to move at the same time in a circular pattern.

3.2 Robust Execution

The problem of robust execution of MAPF plans deals with the problem of safely executing the collision-free plans in the presence of unexpected delays. A formulation for robust execution was presented in (Hönig et al. 2019), where an execution is considered robust if no collision occurs even in the event of varying execution times of robot actions. We use an equivalent problem formulation: we consider an execution of a MAPF plan *robust* if an unexpected delay of an agent during execution does not cause a collision (conflict).

4 Robust Execution as a Scheduling Problem

The vertices which the agents occupy can be viewed as shared resources. An agent that is visiting a vertex consumes the resource, and until the resource is released, no other agent can be scheduled to visit it. Therefore, the problem of robust execution can be formulated as a scheduling problem, such as a variant of JSSP.

In Section 4.1, we present a problem formulation of JSSP. Afterwards, in Section 4.2, we present an extension to JSSP that allows to formulate the problem of robust execution of MAPF plans as a scheduling problem, and in Section 4.3 we detail how to represent a MAPF execution schedule. Finally, in Section 4.4, we formulate the plan repair problem in the context of robust execution.

4.1 JSSP

A JSSP instance consists of a set of m machines $M_1, \dots, M_m$, and a set of n jobs $J_1, \dots, J_n$. Since JSSP is a multi-operation machine scheduling problem, each job $j \in J$ consists of a sequence of $\mu(j)$ operations $\mathcal{O}(j) = \{O_{1,j}, \ldots, O_{\mu(j),j}\}$. An operation $O_{i,k}$ is identified by two indices. The index k specifies the job it belongs to, and the index i specifies that an operation $O_{i,k}$ must be completed before every operation $O_{u,k}, u > i$. Each operation is assigned to a machine: $M_{\lambda(i,j)}(\lambda(i,j) \in 1, \ldots, m)$, where it consumes the resource for the duration of its processing time $p_{i,j}$. A single machine can process multiple jobs, but only one at a time. These parameters are the input parameters of a JSSP instance.

For each operation $O_{i,j}$ with processing requirement $p_{i,j}$ we define start time $R_{i,j}$ and completion time $C_{i,j}$. These variables satisfy equation $C_{i,j} = R_{i,j} + p_{i,j}$. The feasible solution is a solution that complies with the following rules.

1. There is at most one operation scheduled on each machine at any given time. For operations $O_{i,j}, O_{k,l}$ assigned to the same machine ($\lambda(i,j) = \lambda(k,l)$), the following must stand:
$$(R_{i,j}, C_{i,j}) \cap (R_{k,l}, C_{k,l}) = \emptyset \tag{1}$$
2. No two time intervals allocated to the same job overlap and operations of each job are scheduled in the predetermined order. For operations $O_{i,j}, O_{k,j}$ from job J_j, where $i < k$:
$$C_{i,j} \leq R_{k,j} \tag{2}$$
3. The minimum allowed start time is 0. For every operation $O_{i,j}$:
$$R_{i,j} \geq 0 \tag{3}$$

The goal of JSSP is to find an operation processing schedule that satisfies Constraints 1-3 and minimizes a selected objective function. There are multiple possible objective functions, such as *makespan*, which is the time required to finish processing all jobs. In other words, *makespan* corresponds to the time it takes to finish the job with longest processing time.

4.2 Extending JSSP for MAPF

Consider a path p_1 for a single agent seen in Fig. 1, $p_1 = \{(0,1), (1,1), (2,1), (2,1), (2,2)\}$. In each time step, the agent can move to the center of a neighboring cell that is free, or wait in place. Since the cells have equal sizes, we can simplify the movement of the agent and assume that starting at the center of the origin cell, the agent takes half of the time step to reach its border, and the remaining half to reach the center of the destination cell. The agent's path can then be represented as positions in different time intervals as seen in 2. For simplicity, we assume that the agent occupies exactly one cell at all times. During a *wait* action, the agent occupies a single cell during the whole time step, whereas during a *move* action, the occupied cell changes in the middle of the time step.

The representation of the agent's path from Fig. 2 can then be easily translated into a job shop schedule. Each cell will be a JSSP machine, each agent will be a job and each of

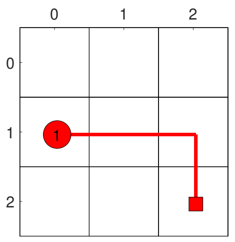


Figure 1: Example path.

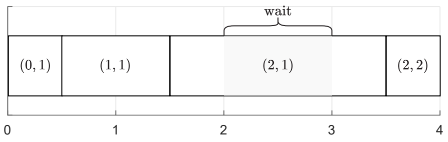


Figure 2: Time intervals of example path.

the job's operations will correspond to the agent occupying a cell during some time interval. The constraints of JSSP are also directly applicable in MAPF: at most one agent can occupy each cell in any distinct time step, the order an agent occupies the cells is fixed and minimum start time is also 0. The job shop schedule representing the path p_1 can be seen in Fig. 3.

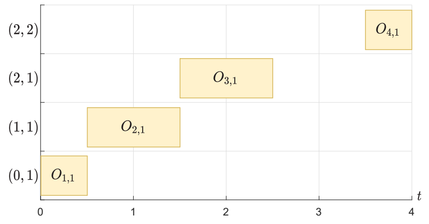


Figure 3: Job shop schedule of the example path.

However, a feasible JSSP solution may not necessarily be a feasible MAPF schedule. One key difference is that unlike jobs in JSSP, an agent may block a machine even after the scheduled operation is processed. This is due to the ability of agents to wait in place to avoid collisions. Therefore, we need to define the occupancy time interval as:

$$\Omega_{i,j} = \begin{cases} (R_{i,j}, R_{i+1,j}), & \text{if } i < \mu(j) \\ (R_{i,j}, \infty), & \text{if } i = \mu(j) \end{cases} \tag{4}$$

Additionally, in JSSP, two jobs may swap machines. In MAPF, this would correspond to two agents swapping cells during one time step, or in other words, a swapping conflict. Therefore, it is necessary to extend the constraints of JSSP with MAPF-specific constraints:

1. First operation $O_{1,j}$ of every job J_j must start at time 0.
2. Each machine can be occupied up to one operation at every moment. For operations $O_{i,j}, O_{k,l}$ assigned to the same machine, i.e. $\lambda(i,j) = \lambda(k,l)$, must stand:
$$\Omega_{i,j} \cap \Omega_{k,l} = \emptyset \tag{5}$$

3. Two jobs must not exchange/swap machines they occupy. For operations $O_{i,j}, O_{k+1,l}$ that are assigned to the same machine $M_{\lambda(i,j)} = M_{\lambda(k+1,l)}$, and operations $O_{i+1,j}, O_{k,l}$ assigned to the machine $M_{\lambda(i+1,j)} = M_{\lambda(k,l)}$:

$$\Omega_{i,j} \cap \Omega_{k,l} \neq \emptyset \implies \Omega_{i+1,j} \cap \Omega_{k+1,l} = \emptyset \qquad (6)$$

The first additional constraint ensures that all agents are present in their start cells at time 0. The second constraint ensures that there is no *vertex* conflict in any cell, i.e., in any given moment and any given cell, there is at most one agent occupying the cell. The last constraint ensures that there is no *swapping* conflict.

4.3 Solution Representation

After formulating robust MAPF plan execution scheduling as a JSSP-based scheduling problem, it is necessary to define a representation of the schedule. The first option is to use operation-based encoding (Cheng, Gen, and Tsujimura 1996). An operation of a job is represented by a symbol called gene. A sequence of multiple genes forms a chromosome. While a gene only carries information about the job it belongs to, it is possible to decode which operation each gene represents using the rule that there is a fixed order of operations for each job. The sequence is read left-to-right with the leftmost gene having the highest priority (being scheduled first).

However, operation-based encoding can encode only active schedules (schedules where all operations are scheduled as soon as possible). This makes it a bad fit for encoding MAPF schedules, since an agent may have to wait before entering a vertex even before the vertex is blocked to let another agent pass. Scheduling the visit to the vertex as soon as possible could block the other agent, leading to a deadlock.

The second option is to use the preference list-based representation (Davis et al. 1985). The chromosome of this representation is formed by a list of operation sequences, one for each machine. The genes in the chromosome then encode the schedule of the operations for the given machine. Consider an example problem shown in Table 1. Its schedule, seen Fig. 4, can be encoded as the chromosome $[1, 2, 2, 3, 4, 3, 4, 4, 1, 1]$. However, this representation may be ambiguous, meaning that the same schedule can be encoded by multiple different chromosomes (such as $[2, 2, 1, 3, 4, 3, 4, 4, 1, 1]$ in this case). Therefore, we modify the preference list-based representation to be unambiguous. This modification encodes the same example schedule as the chromosome $[O_{2,2}, O_{1,4}, O_{2,1}]$, $[O_{1,2}, O_{1,3}, O_{2,4}, O_{3,1}]$ and $[O_{1,1}, O_{2,3}, O_{3,4}]$, containing schedules for machines M_1, M_2 and M_3, respectively.

Table 1: Example problem.

Job	Operations			Assigned machine			Processing time		
J_1	$O_{1,1}$	$O_{2,1}$	$O_{3,1}$	M_3	M_1	M_2	3	2	3
J_2	$O_{1,2}$	$O_{2,2}$	-	M_2	M_1	-	2	2	-
J_3	$O_{1,3}$	$O_{2,3}$	-	M_2	M_3	-	3	2	-
J_4	$O_{1,4}$	$O_{2,4}$	$O_{3,4}$	M_1	M_1	M_3	4	2	3

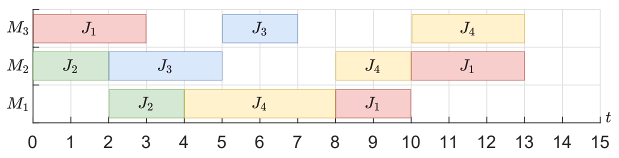


Figure 4: Decoded schedule.

Recognizing infeasible chromosomes In order to verify whether a chromosome produces a feasible schedule or not, it must be decoded, which generates computational overhead. Therefore, predicting feasibility without decoding is beneficial. We achieve this by leveraging our unambiguous schedule representation. An unambiguous representation not only encodes preference of operations, but completely determines their order. This means that each solution can be encoded by exactly one chromosome. However, this comes at a cost: some chromosomes cannot be decoded into any feasible schedule.

The order of the operation $O_{i,j}$ on the preference list of machine $M_{\lambda(i,j)}$ is given by the *preference number* $\rho(i, j)$. As an example, consider a problem with one machine, three jobs J_1, J_2, J_3, three operations $O_{1,1}, O_{1,2}, O_{1,3}$ and a preference list $[O_{1,3}, O_{1,1}, O_{1,2}]$. The preference number $\rho(i, j)$ expresses an order in which $O_{i,j}$ occurs in the preference list, therefore $\rho(1, 1) = 2$, $\rho(1, 2) = 3$, $\rho(1, 3) = 1$.

A chromosome that encodes a feasible schedule then satisfies the following set of conditions:

1. The first operation $O_{1,j}$ of the job J_j must be the first item in the preference list of the corresponding machine $M_{\lambda(i,j)}$, i.e., $\rho(1, j) = 1$.
2. The last operation $O_{\mu(j),j}$ of the job J_j must be the last item in the preference list of the corresponding machine $M_{\lambda(i,j)}$, i.e., $\rho(\mu(j), j)$ is equal to the length of the preference list.
3. For operations $O_{i,j}$ and $O_{k,j}$ of the job J_j on the same machine, where $O_{i,j}$ precedes O_{kj} $(i < k)$, preference number $\rho(i, j)$ must be less that preference number $\rho(k, j)$.
4. For operations $O_{i,j}, O_{k,l}$ assigned to the same machine $M_{\lambda(i,j)}$, and operations $O_{i+1,j}, O_{k+1,l}$ assigned to the same machine, it must stand:

$$\rho(i, j) < \rho(k, l) \iff \rho(i + 1, j) < \rho(k + 1, l) \qquad (7)$$

5. For operations $O_{i,j}, O_{k,l}$ assigned to the same machine $M_{\lambda(i,j)}$, and operations $O_{i+1,j}, O_{k-1,l}$ assigned to the same machine, it must stand:

$$\rho(i, j) < \rho(k, l) \iff \rho(i + 1, j) < \rho(k - 1, l) \qquad (8)$$

Condition 1 means that each agent must start in a unique starting position. Condition 2 means that each agent remains occupying its goal position after reaching it. Condition 3 applies in order to preserve the precedence constraint of operations within a job.

Consider agents A_j and A_l sharing two cells that occur consecutively on both paths. The agents traverse from one cell to another in the same direction. The violation of Condition 4 would mean that the agent A_j visits the first cell earlier than the agent A_l, but visits the second cell after A_l does.

Violation of Condition 5 corresponds to two agents swapping their locations over the same edge, leading to either a swapping conflict or a vertex conflict. This depends on whether the swap takes place in one time step or more.

Figures 5 and 6 show a useful visual representation, using which it is possible to detect violations of Conditions 1-5. For each job J_j (agent A_j), we can show preference lists of machines $M_{\lambda(1,j)}, \ldots, M_{\lambda(\mu(j),j)}$. This sequence corresponds to the path of the agent A_j. The sequence of machines is shown on the horizontal axis, whereas the vertical axis shows the preference of operations. Lower operations have higher preference. To visually verify that Conditions 4 and 5 are satisfied, we can connect all successive operations $O_{i,j}$ and $O_{i+1,j}$ of each job J_j by an arrow pointing from $O_{i,j}$ towards $O_{i+1,j}$ and determine if any of the two arrows cross each other or not. If they do, either Condition 4 and 5 is not satisfied. If we label each operation as $O_{i,j}$, it is even possible to verify Conditions 1-3. To completely check a chromosome, we must examine the path of each agent.

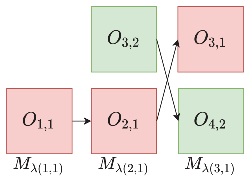


Figure 5: Preference list lined up in order of agent's A_1 path.

Condition 4 violation. Consider the sequence of machines corresponding to the path of the agent A_1 in Figure 5. The preference number of the operation $O_{2,1}$ is equal to 1, while the other operation $O_{3,2}$ on the same machine $M_{\lambda(2,1)}$ has preference number equal to 2 – therefore, $\rho(2,1) < \rho(3,2)$. The operation $O_{3,1}$, following the operation $O_{2,1}$, belongs to the machine $M_{\lambda(3,1)}$. $M_{\lambda(3,1)}$ also contains operation $O_{4,2}$, which follows the operation $O_{3,2}$ and has preference number equal to 1. Therefore $\rho(3,1) > \rho(4,2)$. The two given inequalities do not satisfy Condition 4.

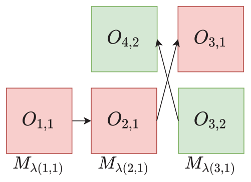


Figure 6: Preference list lined up in order of agent's A_1 path.

Condition 5 violation. Consider the sequence of machines corresponding to the path of the agent A_1 in Figure 6. The preference number of the operation $O_{2,1}$ is equal to 1, while the other operation $O_{4,2}$ on the same machine $M_{\lambda(2,1)}$ has preference number equal to 2. Therefore $\rho(2,1) < \rho(4,2)$. The operation $O_{3,1}$, following the operation $O_{2,1}$, belongs to the machine $M_{\lambda(3,1)}$. The machine $M_{\lambda(3,1)}$ contains operation $O_{3,2}$, which precedes the operation $O_{3,2}$ and has preference number equal to 1. Therefore $\rho(3,1) > \rho(3,2)$. The two given inequalities do not satisfy Condition 5.

Cycle Conflict In addition to the Conditions 1-5 described previously, it is necessary to deal with the so-called *cycle conflicts*: situations where four or more agents mutually exchange their positions in one time step. Such situations can occur in plans generated by state-of-the-art solvers, such as CBS or ECBS. A *dependency operation* $O_D(M)$ of a machine M is the operation $O_{k+1,l}$, where $O_{k,l}$ is the last scheduled operation on the machine $M = M_{\lambda(k,l)}$. In other words, it is an operation whose starting time $R_{k+1,l}$ is the earliest time we can schedule a new operation on machine M. However, in the situation of cycle conflict, there are operations situated in the cycle which cannot be scheduled this way. Consider the chromosome described in Table 2. Agent A_1 is delayed by one time step, and agent A_3 is delayed by two time steps. This problem can be solved only by allowing a cycle conflict. After scheduling the first operation of each job, the dependency operation of the machines are as follows: $O_D(2,2) = O_{2,4}$; $O_D(1,2) = O_{2,3}$; $O_D(1,1) = O_{2,2}$ and $O_D(2,1) = O_{2,1}$. As we can see, neither dependency operation is scheduled, so we cannot schedule any other operation in the usual way. However, if we identify a cycle and determine the minimal start time $T_{\text{min}} = 2.5$, that would not violate the constraints of the feasible schedule, we can schedule one of the operations $O_{i,j}$ located in the cycle to the time interval $(T_{\text{min}}, T_{\text{min}} + p_{i,j})$ and then continue with regular scheduling. The paths of the agents are shown in Figure 7 and a part of the schedule is shown in Figure 8.

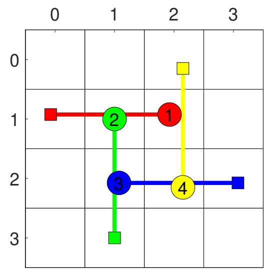


Figure 7: Paths of four agents leading to a cycle conflict with delayed agents 1 and 3.

Table 2: Preference lists of cycle problem (Fig. 7).

Machine	Operation order	Machine	Operation order
$(0,1)$	$[O_{3,1}]$	$(1,1)$	$[O_{2,1}, O_{1,2}]$
$(2,0)$	$[O_{3,4}]$	$(2,1)$	$[O_{1,1}, O_{2,4}]$
$(1,3)$	$[O_{3,2}]$	$(1,2)$	$[O_{1,3}, O_{2,2}]$
$(3,2)$	$[O_{3,3}]$	$(2,2)$	$[O_{1,4}, O_{3,2}]$

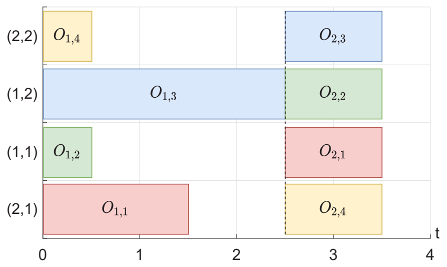


Figure 8: Part of the schedule in the example with cycle conflict.

Cycle detection algorithm. The detection of a cycle and the computation of the minimum start time is described in Algorithm 1.

Algorithm 1: Cycle detection

```
INPUT: $O_{\text{input}}$
 1: $O_{i,j} \leftarrow O_{\text{input}}$
 2: $T_{\text{min}} \leftarrow 0$
 3: $k \leftarrow 0$
 4: do
 5:     $O_{\text{dep}} \leftarrow$ dependency operation of $M_{\lambda(i,j)}$
 6:     $O_{\text{next}} \leftarrow$ nextOperation($M_{\lambda(i,j)}$)
 7:     $O_{\text{last}} \leftarrow$ lastOperation($M_{\lambda(i,j)}$)
 8:     if $O_{\text{next}} = O_{i,j}$ and $O_{\text{dep}}$ exists and is not scheduled then
 9:         $O_{i,j} \leftarrow O_{\text{dep}}$
10:         $T_{\text{min}} \leftarrow \max(T_{\text{min}}, C_{\text{last}})$
11:         $k \leftarrow k + 1$
12:     else
13:         return (false, -1)
14:     end if
15: while $O_{i,j} \neq O_{\text{input}}$
16: return ($k \geq 4$, $T_{\text{min}}$)
```

Function `nextOperation` takes a machine as input and returns the next operation that should be scheduled according to the corresponding preference list; in other words, it returns the operation $O_{i,j}$, which has the lowest preference number $\rho(i,j)$ of all unscheduled operations belonging to the machine $M_{\lambda(i,j)}$. The function `lastOperation` takes a machine as input and returns the last scheduled operation on that machine.

The input of Algorithm 1 is the operation $O_{i,j}$ that is the next operation that should be scheduled according to the precedence constraints of the job J_j. Algorithm 1 gradually checks whether the operation $O_{i,j}$ is next to be scheduled according to the preference list, and whether the dependency operation of machine $M_{\lambda(i,j)}$ is not scheduled. The operation $O_{i,j}$ is either the input operation or the dependency operation of some machine, therefore it is implicitly checked that the operation $O_{i,j}$ is also next operation that should be scheduled based on the precedence constraints of the job J_j. If all the conditions are satisfied, $O_{i,j}$ is redefined to be the dependency operation of the machine that contains the current operation $O_{i,j}$. In the case where the chromosome contains a cycle, the process is successfully repeated until the input operation is reached again. To separate the swap and cycle conflict, it is checked that the cycle consists of at least four operations.

4.4 Plan Repair

In the context of this work, the plan repair problem means having a feasible solution that cannot be executed as planned due to one or more delayed agents. These delays in execution may cause collisions. We assume that each agent A_j has accumulated some delay $d_j \in \mathbb{N}$.

In the plan repair problem, we have a MAPF plan, which consists of a sequence of vertices for each agent. This sequence contains information about the order the agent must visit the vertices, but does not contain timing. The goal is to add timing that would result in a feasible solution that minimizes the cost function.

We can look at assigning the timing to each item of a path as a scheduling problem. Processing time $p_{i,j}$ of each operation $O_{i,j}$ is used to separate occupancy time interval into *fixed* and *optional* disjoint subintervals, where $(R_{i,j}, C_{i,j})$ is fixed subinterval and $\Omega_{i,j} \setminus (R_{i,j}, C_{i,j})$ is optional subinterval. The size of the fixed interval $|C_{i,j} - R_{i,j}| = p_{i,j}$ is known prior to the scheduling and is a property of the problem. The size of the optional interval $|\Omega_{i,j}| - p_{i,j}$ is determined by scheduling procedure and is a property of the solution.

Proposed processing requirements are as follows: first operation $O_{1,j}$ of each job J_j has a processing time equal to waiting in the center of start cell for d_j time steps plus going from the center to the border of start cell, i.e. $p_{1,j} = d_j + 0.5$. Processing requirement of last operation $O_{\mu(j),j}$ is equal to traveling from border of goal cell to its center, i.e. $p_{\mu(j),j} = 0.5$. Other operations $O_{i,j}$, $1 < i < \mu(j)$, have processing requirements $p_{i,j} = 1$ that correspond to going from the border of current cell to its center and then going to border that links current cell and the next cell.

Consider the plan in Fig. 3 delayed by two time steps. This example would have properties shown in Table 3.

Table 3: Properties of example problem.

agent	plan				processing times			
A_1	(0,1)	(1,1)	(2,1)	(2,2)	2.5	1	1	0.5

5 Method

In this section, we propose a VNS optimization method for optimization of MAPF execution schedules. First, in Section 5.1, we describe the *shake* and *local search* procedures of the proposed method. Next, in Section 5.2, we present a procedure to decode chromosomes representing a schedule for a MAPF solution. Afterwards, in Section 5.3, we propose an initial solution generation procedure, and in Section 5.4 we describe our proposed random operator for the VNS method.

5.1 Shake and Local Search

The VNS has two alternating phases that are used to gradually optimize the initial solution: *shake* and *local search*.

Shake is used to escape local minima, and *local search optimizes* attempts to reach a local minimum. Implementation of VNS proposed in (Sevkli and Aydin 2006) used functions `insert` and `exchange` for the local search, and used a sequence of random applications of the operators in the shake procedure. Therefore, we need an operator for *local seach* and *shake*. We will use only one operator, and thus, the shake procedure will call the same operator N times, where N is an input parameter of the method.

5.2 Decoding

The decoding Algorithm 2 takes a chromosome as an input and outputs a schedule. It uses functions `schedule`, and function `forceSchedule`. The function `schedule` takes an operation $O_{i,j}$ as an input. Then it finds the operation $O_{k,l}$ with the greatest completion time that is scheduled to the same machine as operation $O_{i,j}$. If $O_{k,l}$ is the last operation of its job, the procedure terminates with failure, since it is impossible to schedule another operation into the partial schedule. If it is not, it checks whether $O_{k+1,l}$ has been allocated. In such case, $O_{i,j}$ is allocated to the earliest possible feasible time interval – a time interval that does not violate the constraints defined in Sections 4.1 and 4.2. Otherwise the procedure does not schedule $O_{i,j}$. The function `forceSchedule` schedules the input operation $O_{i,j}$ to time interval $(T, T + p_{i,j})$, where T is the second input argument.

Algorithm 2: Decoding procedure

```
INPUT: Chromosome
 1: do
 2:     counters ← $\{0, \dots, 0\}$
 3:     scheduleChanged ← false
 4:     for $j \in \{1, \dots, n\}$ do
 5:         $i$ ← counters[$j$]
 6:         $O_{k,l}$ ← nextOperation($M_{\lambda(i,j)}$)
 7:         if $O_{i,j} \neq O_{k,l}$ then continue
 8:         else
 9:             success ← schedule($O_{i,j}$)
10:             if success then
11:                 scheduleChanged ← true
12:                 counters[$j$] ← counters[$j$] + 1
13:             else
14:                 cycleDetected, $T_{\text{min}}$ ← detectCycle($O_{i,j}$)
15:                 if cycleDetected then
16:                     forceSchedule($O_{i,j}, T_{\text{min}}$)
17:                     scheduleChanged ← true
18:                     counters[$j$] ← counters[$j$] + 1
19:                 end if
20:             end if
21:         end if
22:     end for
23:     if scheduleChanged = false then,
24:         terminate decoding                ▷ decoding not successful
25:     end if
26: while counters $\neq \{\mu(1), \dots, \mu(n)\}$
27: return Schedule
```

Algorithm 2 attempts to schedule the operations of a job one by one. It checks whether the operation satisfies the constraints of the corresponding preference list and attempts to schedule it. If the operation is scheduled successfully, we move on to the next operation of the job. Otherwise, it checks whether the failure was because a cycle conflict occurred. If it had, the operation is scheduled into the calculated time interval and the algorithm moves on to the next operation of the job. In the event that the operation cannot be scheduled, we move on to another job. Chromosomes that do not produce any schedule are detected when it is not possible to add any other decoded operation.

5.3 Initial Solution Construction

At the start, VNS needs an initial feasible solution. A trivial solution is to use the original collision-free plan without delays. The original plan consists of a sequence of time steps, where each time step contains information about the current position of each agent. First, we create a list of jobs $J_1, \dots, J_n$ for the team of agents $A_1, \dots, A_n$. The original plan can then be translated into a chromosome by iterating over each time step. Then we perform the following two actions for each agent:

1. If the machine corresponding to the current position is not in the list of machines, add it to the list.
2. If the current time step is not equal to 0 and the agent moved since the last time step, add a new operation assigned to the current machine to the job representing the agent.

After the chromosome is encoded, it is necessary to add processing requirements to each operation. The last operation $O_{\mu(j),j}$ of each job J_j has processing time $p_{\mu(j),j} = 0.5$. The first operation $O_{1,j}$ of each job has processing time $p_{1,j} = 0.5 + d_j$, where d_j denotes the initial delay of agent A_j. All the other operations have processing requirements equal to 1. Afterwards, we can obtain the initial feasible schedule by decoding the chromosome.

5.4 MAPF-JSSP Operator

The idea of the proposed operator is to randomly select one agent and change its preference number on the machines it visits while satisfying all necessary Conditions 1-5. The operator starts from the first cell of the agent's path and iterates over all visited cells in the predetermined order.

To satisfy Constraints 1 and 2, the first operation O_{1j} must necessarily have $\rho(1, j) = 1$ for any $j \in J$. In the same manner, the last operation $O_{i,j}, i = \mu(j)$ must have $\rho(i, j)$ equal to the length of the corresponding preference list. If there is any operation O_{1l} of job J_l on the machine, then $\rho(1, l)$ is the lower bound. Vice versa, if there is any operation $O_{\mu(l),l}$ which is the last operation of the job J_l, then $\rho(\mu(l), l)$ serves as an upper bound.

The Condition 3 is enforced by tightening the upper bound. When determining the preference number $\rho(i, j)$ for the operation $O_{i,j}$ on a preference list that contains another operation of the same job O_{kj}, where $i > k$ ($O_{i,j}$ precedes O_{kj}) then $\rho(i, j)$ acts as an upper bound. Note that if $i < k$, $\rho(i, j)$ is not used as a lower bound. This is to allow temporary violation of the constraint by reordering on the preference list. Feasibility is then ensured by consistently applying the rule for $i < k$.

In order to satisfy Constraint 4, when selecting the preference number for operation $O_{i,j}$, we must take a look at the other operations$O_{k,l}$ on the same machine. If there is an operation $O_{k-1,l}$ where $M_{\lambda(k-1,l)} = M_{\lambda(i-1,l)}$, then the preference number $\rho(k,l)$ serves as a lower bound (when $\rho(k-1,l) < \rho(i-1,l)$) or an upper bound (when $\rho(k-1,l) > \rho(i-1,l)$). Constraint 5 is then satisfied by using the $\rho(k,l)$ if there is $O_{k+1,l}$ such that $M_{\lambda(k+1,l)} = M_{\lambda(i-1,l)}$ as a lower bound (when $\rho(k+1,l) < \rho(i-1,l)$) or an upper bound (when $\rho(k+1,l) > \rho(i-1,l)$).

Based on the type of constraint and the direction in which agents travel the section of their path, it can happen that the constraints exclude each other and there is no $\rho(i,j)$ that would satisfy all of them. Such cases can be solved by *backtracking*: the process of iterating over a selected job in a descending order while searching for an assignment of preference numbers that satisfies the constraints.

```
Algorithm 3: Proposed operator
 1: $j \leftarrow$ randomly select integer from $\{1,\dots,n\}$
 2: **for** $i \in \{2,\dots,\mu(j)\}$ **do**
 3:     $P \leftarrow$ preference list corresponding to machine $M_{\lambda(i,j)}$
 4:     $X \leftarrow$ permissible preference numbers
 5:     **if** $X$ is empty **then**
 6:         $i \leftarrow$ backtrack()
 7:     **else**
 8:         $\rho \leftarrow$ randomly select from $X$
 9:         $P \leftarrow$ reorder$(O_{i,j}, \rho)$
10:     **end if**
11: **end for**
```

First, a random job J_j (corresponding to the agent A_j) is chosen. Then, the operator iterates over its operations, starting from $O_{2,j}$. Recall that the first operation should always have the preference number $\rho(i,j) = 1$ and we can skip it. For each operation, we determine a list of preference numbers that would meet all the previously described constraints. If the constraints allow us to select a new preference number $\rho(i,j)$, we randomly select a permissible preference number and reorder operations on the machine accordingly. Otherwise, the procedure backtracks until it reaches the point where it is possible to choose a different, permissible order in the preference list.

Consider applying the proposed operator to a problem with at least three agents. In this example, it is enough to differentiate jobs (agents) by colours and label the first and last operations of each job by S and G. In the first step of the operator procedure, the blue agent is chosen. The original path of the blue agent is shown in Figure 9. The operator then iterates over the machines corresponding to the blue agent's path and reorders the operations in preference lists. One of the possible outcomes of the application of the operator is shown in Figure 10. A green background indicates a set of preference numbers from which the algorithm chooses the new preference number. The grey background indicates forbidden preference numbers. To determine what preference numbers are forbidden in the preference list corresponding to the machine M_k, the preference list of the machine M_{k-1} must be known. In this example, the operator made two decisions, first in the cell corresponding to the machine M_2, and then in the cell corresponding to the machine M_4.

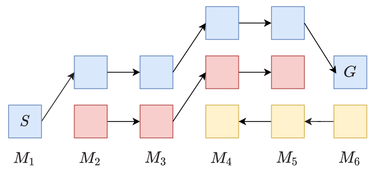


Figure 9: The original path of the blue agent.

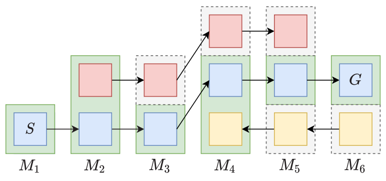
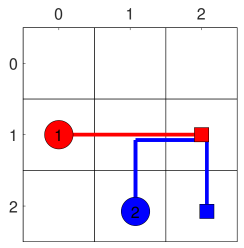


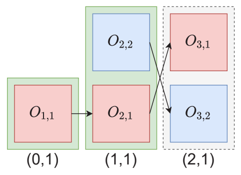
Figure 10: The path of the blue agent after applying the operator.

The following examples illustrate two different situations that lead to the necessity of applying the `backtrack` function:

Example 1. Assume the situation shown in Figure 11. The operator tries to change the order in which the agents finish. When the algorithm reaches the third operation of the job J_1 (corresponding to the agent A_1), it should exchange the operations $O_{3,1}$ and $O_{3,2}$ on machine $(2,1)$ in order to preserve the relations set in the previous machine $(1,1)$ by Constraint 4. However, this is not possible because if the operations were exchanged, the last operation of the agent A_1 would not be the last operation of the preference list. Therefore, a violation of Constraint 2 would occur.



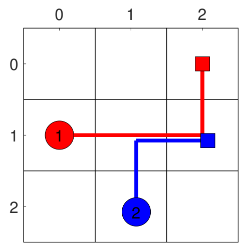
(a) Paths of both agents.



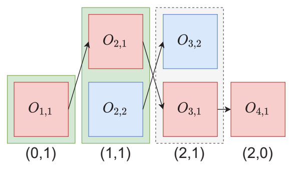
(b) Path of the agent A_1.

Figure 11: First example of necessary backtracking.

Example 2. Consider the situation shown in Figure 12. When the algorithm reaches the third operation of the job J_1, to preserve the relations set in the previous machine $(1,1)$ by Constraint 4, it should exchange the operations $O_{3,1}$ and $O_{3,2}$ on machine $(2,1)$. However, again that is not possible, because if the operations were exchanged, there would be a violation of Constraint 2.



(a) Paths of both agents.



(b) Path of the agent A_1.

Figure 12: Second example of necessary backtracking.

6 Experimental Results

For the experimental verification, we used the MovingAI benchmark dataset (Stern et al. 2019). We used three maps in our experiments: a map with randomly generated obstacles (`random-64-64-20`), a map resembling rooms in a building (`room-64-64-16`), and a warehouse-like map (`warehouse-10-20-10-2-1`). For simplicity, we refer to the maps as `random`, `room` and `warehouse`. First, we investigate the quality of solutions found by VNS in relation to the number of iterations. We also evaluate the performance of our proposed method in comparison to Action Dependency Graph (ADG) (Hönig et al. 2019). Next, we study the evolution of cost with increasing number of iterations of the method.

On each map, ten instances, that is, the start and goal positions of all agents, were randomly generated. Each instance was solved using ECBS (Barer et al. 2014) with the makespan criterion and suboptimality factor $w = 1.05$. We used the ECBS implementation provided in (Okumura, Tamura, and Défago 2021), available online[1].

Model problem that we aim to solve is a situation in which some of the agents are delayed at their starting position. To examine the effect of the total delay (a sum of delays of all agents), we carried out experiments with the following parameters:

- 100 agents in total
- K random integers, which represent delays in time steps are selected $\forall K \in \{0, 10, \dots, 390, 400\}$. Each delay $d_a \in \{1, \dots, 10\}$ is assigned to a randomly selected agent a. Each agent can be selected more than once.

In each run, we recorded the resulting sum of costs and the computation time.

The VNS takes the shake intensity parameter N as an input, which determines how many times will the shake procedure call the shake operator. We performed experimental analysis of the parameter, which showed that $N = 1$ is the best-performing value. However, the differences were not significant.

6.1 VNS Performance

We compare the performance of our proposed VNS method to results obtained using ADG (Hönig et al. 2019). The ADG constructs a graph containing the action-precedence relations and ensures that actions with dependencies on other actions are executed in the same order as they were planned in the original solution. While ADG can deal with cycle conflicts, it cannot repair such plans. The success rate of ADG on each tested map is shown in Table 4.

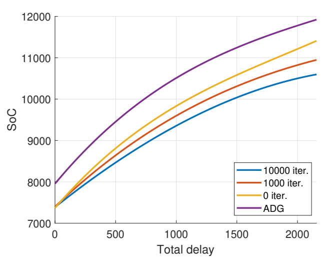
map	success rate (%)
`room`	70
`random`	100
`warehouse`	100

Table 4: Success rate of ADG on different maps.

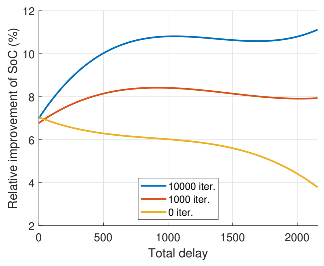
Our proposed VNS algorithm was run with a total of 10000 iterations, and the SoC of each run was recorded at 0, 1000, 10000 iterations. The solution at 0 iterations is equivalent to the initial feasible solution. The measured data for both VNS and ADG were fitted using a polynomial and the results are shown in Figures 13, 14 and 15. The figures show the absolute SoC values and the improvement relative to the solution found by ADG, defined as:

$$\mathrm{RI} = \frac{f_{\mathrm{ADG}} - f_{\mathrm{VNS}}}{f_{\mathrm{ADG}}} \tag{9}$$

where f denotes the objective function (in this case the sum of costs). The time that each run of VNS took seemed to be independent of the total delay; therefore, we utilize the average time of each experiment. The average times are shown in Table 5.



(a) SoC for various number of iterations of VNS and SoC found by ADG.



(b) Improvement of SoC after applying VNS relative to solutions found by ADG.

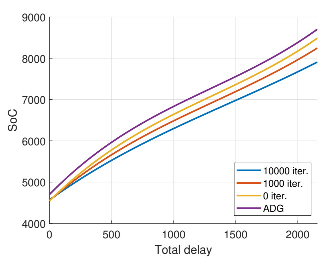
Figure 13: Quality of solutions found by VNS and ADG on map `room`.

Maps	`room`	`random`	`warehouse`
Iterations	Time [ms]	Time [ms]	Time [ms]
10000	5096.34	3043.92	6186.36
1000	836.56	390.34	825.19
0	78.07	36.25	70.34

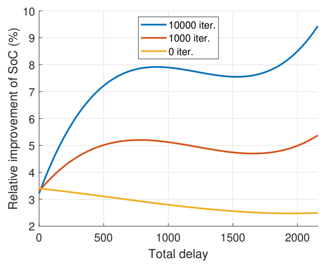
Table 5: Measured CPU time of running VNS on maps `room`, `random` and `warehouse`.

From the results, it is clear that VNS significantly outperforms ADG on the map `room` both from the perspective

[1]`https://github.com/Kei18/mapf-IR`

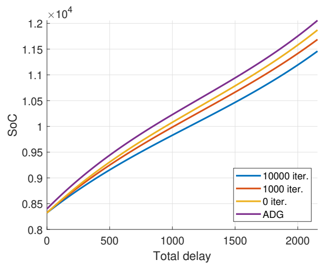


(a) SoC for various number of iterations of VNS and SoC found by ADG.

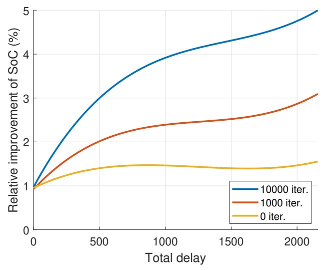


(b) Improvement of SoC after applying VNS relative to solutions found by ADG.

Figure 14: Quality of solutions found by VNS and ADG on map `random`.



(a) SoC for various number of iterations of VNS and SoC found by ADG.



(b) Improvement of SoC after applying VNS relative to solutions found by ADG.

Figure 15: Quality of solutions found by VNS and ADG on map `warehouse`.

of success rate and relative improvement of a solution. On the other two maps, VNS still outperforms the competing method, but with a smaller margin.

To evaluate the usability of the VNS algorithm, the computation time should be taken into account. The instances plans on maps `random` and `warehouse` were all calculated using ECBS in less than a second. However, the time required to solve problems on map `room` ranged from 3 seconds up to 300 seconds, and in one case, of the ten tested, ECBS did not find a sufficient solution in the entire dedicated 300 seconds. From this point of view, the computation times of VNS on this map seem to be reasonable and VNS can be considered to be used on maps similar to `room`. Furthermore, VNS is an anytime algorithm. That means that it is able to output a solution at any point after initial solution construction, which is fast. The method can even be modified to use maximum execution time as a terminating condition. Our method offers improvement even at lower runtimes, i.e., 1000 iterations take less than a second, while still offering improvement of 5% to 8% on the `random` and `room` maps.

6.2 Evolution of Solution's Quality

The last experiment serves to demonstrate the evolution of the solution quality with increasing iterations. The experiment was carried out on the map `room` in a randomly generated instance. The agents were randomly delayed and the total delay was 989 time steps. In total, 50000 iterations of VNS were run and the best solution was recorded in each iteration. The result of the experiment is shown in Figure 16. The graph shows the evolution of the SoC criterion and the improvement relative to the initial solution. It is clear that most of the improvement was achieved at the beginning of the search: approximately 94% of the total improvement was obtained in the first 10000 iterations.

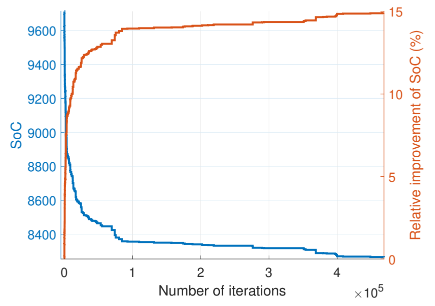


Figure 16: Evolution of solution with increasing iterations.

7 Conclusion

We formulated the problem of robust execution of MAPF plans as a scheduling problem by extending JSSP with newly designed MAPF-specific constraints. Next, we presented a method enabling unambiguous representation of MAPF schedules and their decoding. Furthermore, we proposed an optimization-based VNS method as a solution to the robust execution scheduling problem, which utilized a random operator tailored to the problem at hand. The proposed method was evaluated on execution of MAPF solutions of benchmark dataset maps and compared to state-of-the-art ADG method. Based on experimental results, our new method outperforms ADG in terms of resulting solution cost in presence of unexpected delays of agents at the beginning of the plan. This comes at a cost of execution time, where ADG runs faster. However, our proposed method is still significantly faster than replanning. Furthermore, our method was not optimized for execution time, and we expect that we will be able to decrease its runtime, which is the subject of future work. For additional future work, we want to conduct extended experiments on a larger array of maps and develop more operators for the robust execution scheduling problem.

Acknowledgments

The research was supported by the Czech Science Foundation Grant No. 23-05104S and by the Grant Agency of the Czech Technical University in Prague, Grant No. SGS23/180/OHK3/3T/13. Computational resources were supplied by the project ”e-Infrastruktura CZ” (e-INFRA CZ LM2018140) supported by the Ministry of Education, Youth and Sports of the Czech Republic.

References

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2020. Robust Multi-Agent Path Finding and Executing. *Journal of Artificial Intelligence Research*, 67: 549–579.

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Seventh Annual Symposium on Combinatorial Search*.

Barták, R.; Švancara, J.; Škopková, V.; and Nohejl, D. 2018. Multi-Agent Path Finding on Real Robots: First Experience with Ozobots. In Simari, G. R.; Fermé, E.; Gutiérrez Segura, F.; and Rodríguez Melquiades, J. A., eds., *Advances in Artificial Intelligence - IBERAMIA 2018*, Lecture Notes in Computer Science, 290–301. Cham: Springer International Publishing. ISBN 978-3-030-03928-8.

Cheng, R.; Gen, M.; and Tsujimura, Y. 1996. A tutorial survey of job-shop scheduling problems using genetic algorithms—I. representation. *Computers & Industrial Engineering*, 30(4): 983–997.

Davis, L.; et al. 1985. Job Shop Scheduling with genetic algorithms. In *Proceedings of an international conference on genetic algorithms and their applications*, volume 140.

Felner, A.; Stern, R.; Shimony, S.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N.; Wagner, G.; and Surynek, P. 2017. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *International Symposium on Combinatorial Search*, volume 8.

Hönig, W.; Kumar, T. K.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Multi-Agent Path Finding with Kinematic Constraints. *Proceedings of the International Conference on Automated Planning and Scheduling*, 26: 477–485.

Hönig, W.; Kiesel, S.; Tinka, A.; Durham, J. W.; and Ayanian, N. 2019. Persistent and Robust Execution of MAPF Schedules in Warehouses. *IEEE Robotics and Automation Letters*, 4(2): 1125–1131.

Ma, H.; Koenig, S.; Ayanian, N.; Cohen, L.; Hönig, W.; Kumar, T. K. S.; Uras, T.; Xu, H.; Tovey, C.; and Sharon, G. 2017. Overview: Generalizations of Multi-Agent Path Finding to Real-World Scenarios. arxiv:arXiv:1702.05515.

Ma, H.; Kumar, T. K. S.; and Koenig, S. 2017. Multi-Agent Path Finding with Delay Probabilities. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1).

Mladenović, N.; and Hansen, P. 1997. Variable neighborhood search. *Computers & Operations Research*, 24(11): 1097–1100.

Okumura, K.; Tamura, Y.; and Défago, X. 2021. Iterative Refinement for Real-Time Multi-Robot Path Planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9690–9697.

Sevkli, M.; and Aydin, M. E. 2006. A Variable Neighbourhood Search Algorithm for Job Shop Scheduling Problems. In Gottlieb, J.; and Raidl, G. R., eds., *Evolutionary Computation in Combinatorial Optimization*, 261–271. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-33179-7.

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

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks.