Bachelor Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Plan Repair for a Team of Mobile Agents

Daniel Kubišta

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Cybernetics and Robotics
May 2022

Acknowledgements

I would like to thank RNDr. Miroslav Kulich, Ph.D., for the great discussions and his guidance, especially during the writing of this work.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 20. May 2022

Abstract

Multi-Agent Path Finding (MAPF) is the problem of planning optimal collision-free trajectories for a team of mobile robots. During the execution of the plan, unexpected events may occur that prevent the plan from being performed as intended. Plan correction on fixed paths can be viewed as a variant of Job Shop Scheduling (JSS). JSS is a NP-hard combinatorial optimisation problem that can be solved by the metaheuristic method Variable Neighbourhood Search (VNS). This thesis examines the possibilities of representing MAPF plans as a JSS and proposes modification of the VNS algorithm, so that it is suitable for solving MAPF.

Keywords: Multi-Agent Path Finding, Job Shop Scheduling, Variable Neighbourhood Search

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Intelligent and Mobile Robotics, CIIRC, Jugoslávských partyzánů 1580/3, 160 00 Praha 6

Abstrakt

Cílem multiagentního plánování (MAPF) je najít pro tým mobilních robotů bezkolizní trajektorii. Během realizace plánu se ale mohou vyskytnout situace, které způsobí, že plán nemůže být dokončen tak, jak byl navrhnut. Oprava plánu za předpokladu fixních tras robotů může být viděna jako Job Shop Scheduling (JSS). JSS je NP-těžký problém kombinatorické optimalizace, který je možné řešit pomocí metaheuristické metody Variable Neighbourhood Search (VNS). Tato práce se zabývá možnostmi reprezentace MAPF plánů jako JSS a navrhuje modifikaci VNS algoritmu tak, aby byl vhodný pro řešení MAPF.

Klíčová slova: Multiagentní plánování, Job Shop Scheduling, Variable Neighbourhood Search

Překlad názvu: Oprava plánu pro team mobilních agentů

Contents

Figures

Tables

Chapter 1

Introduction

Multi-Agent Path Finding (MAPF) is the problem of planning optimal collision-free trajectories for a team of mobile robots (agents) that brings all robots from their starting position to the given goal position. During the execution of the plan, unexpected events may occur, such as temporary malfunction of the robot, which causes the robot to stall. The incurred delay may cause a collision of some robots. To avoid that, it is necessary to monitor the robots' positions during the plan execution. If we detect that some robot is delayed, we may need to correct the plan so that no collisions occur.

A possible solution is to re-plan the trajectories whenever a delay of an agent is detected. The problem can be solved optimally using Conflict Based Search (CBS) [15], which however may be both time and memory consuming. Therefore, suboptimal solvers, such as Enhanced Conflict Based Search (ECBS) [1], which guarantee that the returned solution is within a constant factor of the optimal solution, are often used.

An alternative is to reuse the original collision-free plan and reduce it to a sequence of positions (path) that an agent must visit to achieve their goal. The aim of algorithms that use this approach is to turn the paths into trajectories by finding the right time to visit the appropriate cells. This problem can be viewed as a scheduling problem - a task of assigning time slots to the so-called operations, while satisfying given precedence constraints.

The general objective of this thesis is to find a way to represent MAPF plans as a Job Shop Scheduling (JSS) solution and to solve the problem of repairing the plan using a metaheuristic algorithm Variable Neighbourhood Search (VNS) [11]. The concrete objectives of this thesis are:

- Find a way to represent the MAPF problems as a JSS model.
- Explore possibilities of genetic representation of a job shop schedule and find the most suitable one for representing MAPF.
- Propose a random operator that could be used as a neighbourhood function in the VNS algorithm.
- Experimentally verify the functionality of VNS applied to the MAPF problem.

The rest of the thesis is structured as follows. Chapter 2 introduces Job Shop Scheduling, Variable Neighbourhood Search and presents an implementation of VNS for solving JSS. Chapter 3 further introduces Multi-Agent Path Finding, shows a modified job shop model suitable for representing MAPF problems. The third chapter also examines various genetic representations of JSS and offers modifications that will allow us to use them to represent MAPF problems. In the end, Chapter 3 presents a variant of VNS that is suitable for Multi-Agent Path Finding. Chapter 4 experimentally verifies the functionality and properties of VNS deployed on JSS and MAPF problems. Finally, the Chapter 5 summarises the results and suggests future work.

Chapter 2

Job Shop Scheduling and Variable Neighbourhood Search

In this chapter, the reader is introduced to the Job Shop Scheduling problem. Later, commonly used optimality criteria are listed. Finally, the algorithm Variable Neighbourhood Search that can be used to solve job shop problems is described.

2.1 Model description

This section defines Job Shop Scheduling and other related terms. All presented definitions were taken from [3, 5, 10, 14].

Job Shop Scheduling is a combinatorial optimisation problem that belongs to *deterministic machine scheduling* problems. These problems arise when limited resources must be allocated over time to a set of activities. Every instance of a scheduling problem involves a set of m machines $M_1, \dots, M_m$ forming the *machine environment*, and set of n jobs $J_1, \dots, J_n$.

In the case of *multi-operation* model problem, each job consists of $\mu(j)$ operations $O_{1j}, \dots, O_{\mu(j)j}$. The index i of the operation O_{ij} indicates precedence of that operation within the job J_j defined by the other index j. It is required that each operation O_{ij} must be performed on specified machine $M_{\lambda(i,j)}$ ($\lambda(i,j) \in 1, \dots, m$) for amount of time equal to *processing time* p_{ij}. All the mentioned parameters are constant and are known a priori.

Job shop model belongs to the multi-operation domain. The operations of each job J_j are sorted into a sequence that determines the order in which they must be performed. Each machine can serve one job more than once and each job may consist of any number of operations.

2.1.1 Feasible solution

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

A1. There is utmost one operation scheduled on each machine at any given time. For operations O_{ij}, O_{kl} assigned to the same machine $\mu_{ij} = \mu_{kl}$, the following must stand:

$$(R_{ij}, C_{ij}) \cap (R_{kl}, C_{kl}) = \emptyset \tag{2.1}$$

A2. No two time intervals allocated to the same job overlap. For operations O_{ij}, O_{kj} from job J_j:

$$(R_{ij}, C_{ij}) \cap (R_{kj}, C_{kj}) = \emptyset \tag{2.2}$$

A3. Operations of each job are scheduled in the predetermined order. For operations O_{ij}, O_{kj} of job J_j, where $i < k$:

$$C_{ij} \leq R_{kj} \tag{2.3}$$

A4. The minimum allowed start time is 0. For every operation O_{ij}:

$$R_{ij} \geq 0 \tag{2.4}$$

Note that in some literature such as [10], any solution is also called schedule. While in other, for example [5], only feasible solution is called schedule. In this text, I use *schedule* and *solution* interchangeably.

2.1.2 Optimality criteria

Aim of Job Shop Scheduling is to find a schedule that minimises the chosen objective function. For the purpose of evaluating solutions, we define *completion time of a job J_j* as $C_j = C_{\mu(j)j}$. In other words, completion time of the whole job is equal to the completion time of the last operation that forms the job.

The following list contains functions used later in this text. Other optimality criteria can be found in [10].

- The often used criterion is called *makespan* or *length of schedule*. This function is defined as:

  $$C_{\max} = \max_{1 \leq j \leq n} C(J_j) \tag{2.5}$$

  This criterion is equal to the time required to process all jobs. Problems using this cost function are called *minmax*.

- Other possible criterion is *total completion time*, defined as:

  $$\sum C_j = \sum_{j=1}^{n} C(J_j) \tag{2.6}$$

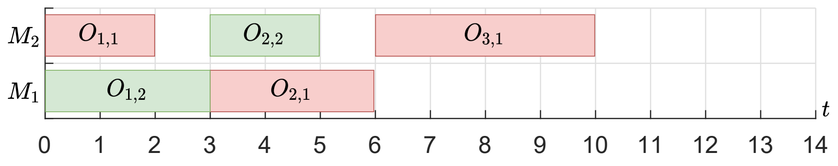
  Problems using this criterion are called *minsum*.

Example 2.1. Consider an example of the job shop problem with two machines, two jobs, and properties given in Table 2.1.

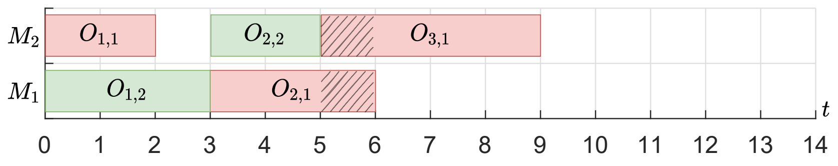
Job	Operations			Assigned machines			Processing times		
J_1	$O_{1,1}$	$O_{2,1}$	$O_{3,1}$	M_2	M_1	M_2	2	3	4
J_2	$O_{1,2}$	$O_{2,2}$	-	M_1	M_2	-	3	2	-

Table 2.1: Properties of an example problem.

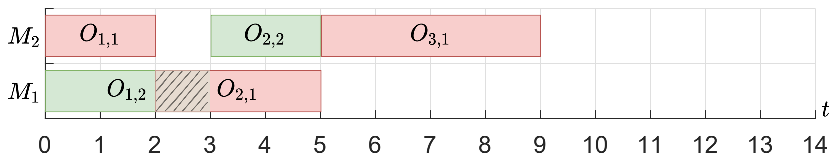
Figure 2.1 shows four different solutions. A feasible schedule with completion times $C(J_1) = 10, C(J_2) = 5$, makespan $C_{\max} = 10$ and total completion time $\sum C_j = 15$ is shown in Figure 2.1a . Figure 2.1b shows unfeasible solution, operation $O_{3,1}$ starts at time 5, while operation $O_{2,1}$ ends in time 6, since both operations belong to job J_1, this is not a feasible schedule. Figure 2.1c shows an unfeasible solution with time overlap of operations $O_{1,2}$ and $O_{2,1}$ on machine M_1. Figure 2.1d shows another unfeasible solution, there is no time overlap, but a violation of the order occurs in job J_1 - operation $O_{3,1}$ precedes operation $O_{2,1}$.



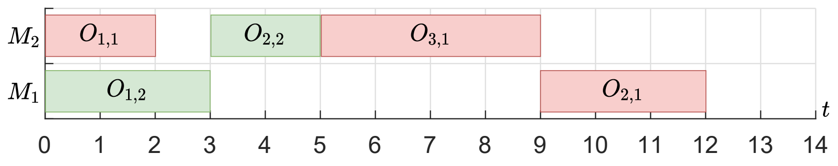
(a) : Feasible solution.



(b) : Unfeasible solution with time overlap.



(c) : Unfeasible solution with time overlap.



(d) : Unfeasible solution with violation of order.

Figure 2.1: Examples of feasible and unfeasible solutions.

2.2 Variable neighbourhood search

The job shop problem is an NP-hard combinatorial problem, which is often solved using metaheuristic methods. The method that I use is called VNS and was proposed in [11]. In [14], the authors implemented the Variable Neighbourhood Search method to solve the Job Shop Scheduling problem.

VNS is designed to avoid getting trapped in local optimums with poor value by systematically changing the subregions of the search space.

For solution x, we define a neighbourhood function or a neighbourhood structure $N(x)$ as a function that returns a set of solutions $\{x'|N(x) = x'\}$. This set is called the neighbourhood of solution x, solutions of the neighbourhood are in proximity of the solution x.

Using a *shake* function, we transfer the search to a different part of the search space, by randomly choosing a solution x' from the neighbourhood of the current best solution x. We then apply a *local search* to obtain a local optimum x'' of a solutions in close proximity of solution x'. When comparing the local optimum to the best so far found solution, in case the local optimum is better, we accept it as a new best solution. There can be more neighbourhood structures used for shake function, in that case, the neighbourhood structures are used sequentially, and every time a new best solution is found the sequence starts over again from the first neighbourhood function. This process is repeated until stopping condition is met. The pseudocode for this method, adopted from [13], is shown in Algorithm 1.

Algorithm 1 General Variable Neighbourhood Search

```
1:  Initialize:
        Choose cost function $f$, select the set of neighbourhood
        structures $N_k$, for $k = 1, \dots, k_{\max}$, that will be used in the
        search; find an initial solution x; choose a stopping
        condition;
2:  while Stopping condition not met do
3:      $k \leftarrow 1$
4:      while $k \leq k_{\max}$ do
5:          $x' \leftarrow$ shake$(x, k)$                          ▷ $x' \in N_k(x)$
6:          $x'' \leftarrow$ localSearch$(x')$
7:          if $f(x'') < f(x)$ then
8:              $x \leftarrow x''$
9:              $k \leftarrow 1$
10:         else
11:             $k \leftarrow k + 1$
12:         end if
13:     end while
14: end while
```

2.3 Representation of job shop scheduling

In the modified version of VNS for solving the Job Shop Scheduling problem [14], *operation-based representation* is used. This representation, was presented in [6] and uses a sequence of symbols called *chromosome*, symbols are being referred to as *genes*. Each gene stands for one operation, and the symbol corresponds to the job to which it belongs. A gene on its own does not carry information about concrete operation, but because there is a predetermined order of operations for every job, genes may be interpreted and assigned to a particular operation according to the order of occurrence in the given chromosome.

For the JSS problem with n jobs $J_j, j \in 1, \ldots, n$ each consisting of $\mu(j)$ operations, there will by $\sum_{j=1}^{n} \mu(j)$ genes in total. Each permutation of the sequence yields a feasible solution. This representation is ambiguous, so the solution may be represented by more than one chromosome.

Example 2.2. To illustrate the translation of a chromosome into a sequence of operations, consider a JSS problem with two jobs J_1, J_2. The job J_1 consists of two operations $O_{1,1}$, $O_{2,1}$, while the job J_2 consists of the operations $O_{1,2}, O_{2,2}, O_{3,2}$ and the chromosome is $[1, 2, 2, 1, 2]$. The translation starts with the leftmost gene and it is necessary to keep track of how many operations of each job have we encountered. The first gene corresponds to the job J_1 and because it is the first operation of J_1, we have encountered, and so this gene is translated to $O_{1,1}$. Similarly, the next gene is translated to the operation $O_{1,2}$. The third gene belongs to the job J_2 and is the second operation of this job, we have encountered, therefore this gene is translated into the operation $O_{2,2}$. In a similar matter, the fourth and fifth genes are translated to operations $O_{2,1}$ and $O_{3,2}$. The resulting sequence is $[O_{1,1}, O_{1,2}, O_{2,2}, O_{2,1}, O_{3,2}]$.

When decoding a schedule from the chromosome, genes are first translated from the chromosome to an ordered list of operations. Then the operations are scheduled one by one. Each operation is allocated to the earliest possible time, so that it meets the requirements of a feasible schedule. The process is repeated until all operations are scheduled and the schedule is complete. A schedule generated by the procedure can be guaranteed to be an active schedule, which is a schedule where we cannot start any operation earlier without delaying the start of another one.

job	Operations			Assigned machines			Processing times		
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

Table 2.2: Properties of example problem.

Example 2.3. Consider a JSS problem from [4], with 4 jobs and 3 machines and properties shown in Table 2.2. Suppose chromosome $[1, 2, 2, 3, 4, 3, 4, 4, 1, 1]$, which can be translated into ordered sequence of operations:

$$[O_{11}, O_{12}, O_{22}, O_{13}, O_{14}, O_{23}, O_{24}, O_{34}, O_{21}, O_{31}]$$

Operations are allocated to the schedule in the order determined from the sequence, the leftmost operation (O_{11}) has the highest priority, while the rightmost has the lowest priority. The decoded schedule is shown in Figure 2.2. Note that due to the ambiguity of the operation-based representation chromosomes $[2, 2, 1, 3, 4, 3, 4, 4, 1, 1]$, $[2, 2, 3, 4, 4, 1, 1, 1, 3, 4]$ and others would also be decoded as the same schedule.

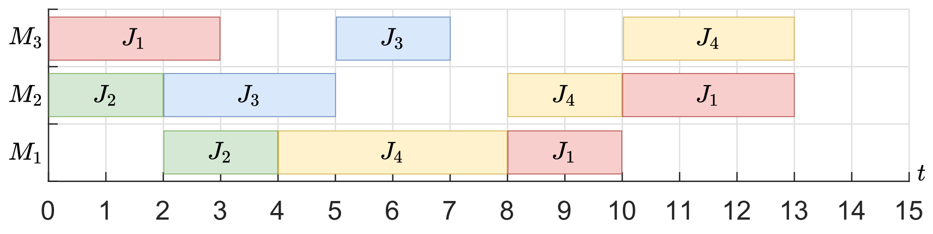


Figure 2.2: Decoded schedule.

2.3.1 Modified VNS

Modified version of variable neighbourhood search used for solving job shop scheduling was proposed in [14]. For a local search, the authors used a version of *first descent* heuristics - a random solution is explored within a neighbourhood of the current solution, if the explored solution is better, the algorithm moves to that solution. In the concrete implementation, used in [14], authors employ two neighbourhood functions: `exchange` (denoted as $N_1^{\mathrm{LS}}(x)$) and `insert` (denoted as $N_2^{\mathrm{LS}}(x)$).

The process of applying the `exchange` neighbourhood function follows: It randomly selects two indexes from the chromosome and swaps genes on those indexes. For example, suppose that we apply *exchange* on chromosome $[1, 2, 2, 1, 3, 4]$, randomly picked indexes are 1 and 5, then the resulting chromosome would be $[3, 2, 2, 1, 1, 4]$.

In the concrete implementation function `insert` selects two indexes i_s and i_d, then removes gene at index i_s from sequence and inserts it in at index i_d. Consider applying *insert* on chromosome $[1, 2, 2, 1, 3, 4]$, with indexes $i_s = 5$, $i_d = 2$, this would result to $[1, 3, 2, 2, 1, 4]$.

Pseudocode of local search is provided in Algorithm 2. During the local search, random solutions from the first neighbourhood $N_1^{\mathrm{LS}}(x')$ are repeatedly explored as long as improved solutions are obtained, each time x' moves to the improved solution. If there is no improvement, the algorithm selects a solution from the next neighbourhood $N_2^{\mathrm{LS}}(x')$, if the solution is better, then the procedure starts over again. If the solution is not better, then the local search procedure ends.

Algorithm 2 Local Search

```
Input: Solution $x'$
 1: $l \leftarrow 1$
 2: $l_{\max} \leftarrow 2$
 3: while $l \le l_{\max}$ do
 4:     if $l = 1$ then
 5:         $x'' \leftarrow \text{exchange}(x')$                    ▷ $x'' \in N_1(x')$
 6:     else if $l = 2$ then
 7:         $x'' \leftarrow \text{insert}(x')$                      ▷ $x'' \in N_2(x')$
 8:     end if
 9:     if $f(x'') < f(x')$ then
10:         $x' \leftarrow x''$
11:         $l \leftarrow 1$
12:     else
13:         $l \leftarrow l + 1$
14:     end if
15: end while
```

As a shake function, the authors used different combinations of `exchange` and `insert` functions. One of the implementations is a function made from the sequence: *exchange*, *insert*, *exchange*, as shown in Figure 2.3.

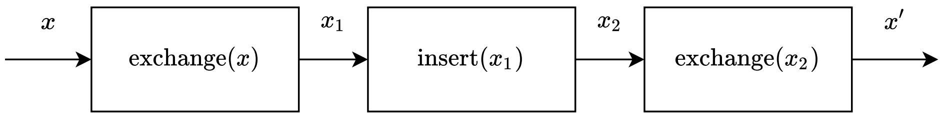


Figure 2.3: Diagram of perturb function.

Chapter 3

Multi-Agent Path Finding and plan repair

3.1 Problem definition

The definition of Multi-Agent Path Finding was adopted from [8, 16].

Multi-Agent Path Finding problem with n agents $A_1, \ldots, A_n$ is specified by a triple (G, s, g), where $G = (V, E)$ is an undirected graph,

$$s : [A_1, \ldots, A_n] \to V \tag{3.1}$$
$$g : [A_1, \ldots, A_n] \to V \tag{3.2}$$

The function s maps an agent to the start vertex and g maps an agent to the goal vertex. Time is discretised 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. Two types of actions are distinguished, *move* and *wait*. A *move* action means that the agent moves from its current vertex v to an adjacent vertex v' in the graph, that is $(v, v') \in E$. A *wait* action means that the agent stays in its current vertex for another time step. Both *move* and *wait* have uniform time cost. A sequence of actions $\pi_j = (a_1, \ldots, a_k)$ that brings the agent A_j from its start vertex to its goal vertex is called the *single-agent plan* of the agent A_j. A *joint plan* is a set $\Pi = (\pi_1, \ldots, \pi_n)$ describing the single-agent plans of all agents $A_1, \ldots, A_n$. Based on the used model of multi-agent path finding, when an agent reaches their goal, it can either disappear from the graph (*disappear-at-target* behaviour) or stay in the goal vertex until the end of a simulation (*stay-at-target* behaviour). This text primarily focuses on the stay-at-target behaviour, but the concepts presented further in the text can adjusted to match the disappear-at-target behaviour.

3.1.1 Objective functions

For the purpose of evaluating *joint plan*, generally one of the two following objective functions is used:

- For a joint plan $\Pi = (\pi_1, \ldots, \pi_n)$, *makespan* is the number of time steps until all agents reach their goal vertex. Formally:

$$\mathrm{M}(\Pi) = \max_{1 \leq j \leq n} |\pi_j| \tag{3.3}$$

- For a joint plan $\Pi = (\pi_1, \ldots, \pi_n)$, *sum of costs* is the sum of time steps of each agent, in which each reaches its goal vertex:

$$\mathrm{SOC}(\Pi) = \sum_{j=1}^{n} |\pi_j| \tag{3.4}$$

3.1.2 Conflicts

In this text, I assume two types of conflicts: *vertex conflict* and *swapping conflict*. Vertex conflict occurs when two or more agents occupy the same vertex at the same time. Two agents have swapping conflict if they swap their location over the same edge at the same time.

3.2 Plan as a job shop schedule

In the physical or simulated world, we consider working with a maps as a rectangular grid formed by the same-size squares called *cells*. Each can be either *accessible* (denoted as a white square) or *inaccessible* (denoted as a black square). The vertex can be interpreted as the centre of the cell and the edge as a line between two adjacent accessible cells connecting the centres of cells. The time needed to traverse an edge is uniform and takes a unit time step. If we consider that the agent moves at a constant speed from one centre of the cell to another, then it spends half of the time step in both cells. An example of a map translated into a graph is shown in Figure 3.1.

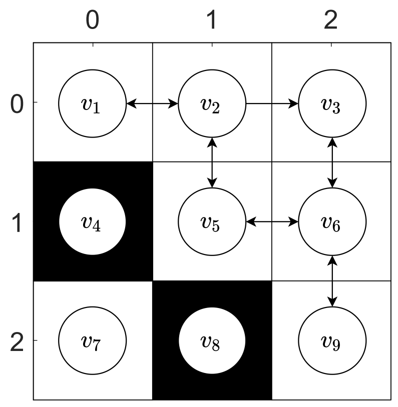


Figure 3.1: Map represented as a graph.

An agent's plan can be described as a sequence of time intervals assigned to a cell that expresses the time spent in that cell.

agent	plan					
A_1	cells	(0,1)	(1,1)	(2,1)	(2,1)	(2,2)
	actions	right	right	wait	down	-

Table 3.1: Plan of the problem from Example 3.1.

Example 3.1. Consider an example with a plan for one agent described by a sequence of actions shown in Table 3.1 and also by the time intervals shown in Figure 3.2. The path of the agent's plan is shown in Figure 3.3. The agent starts in the centre of the cell $(0, 1)$ and immediately moves towards the cell $(1, 1)$. Then, in time 0.5, the agent enters cell $(1, 1)$ crosses the centre of that cell and continues towards the cell $(2, 1)$. In time 1.5, agent enters the cell $(2, 1)$ and reaches the centre of cell in time 2 where it waits until time 3 and then the agent goes towards the cell $(2, 2)$. The agent arrives in the centre of the cell $(2, 2)$ in time 4 and stays there until the end of the simulation.

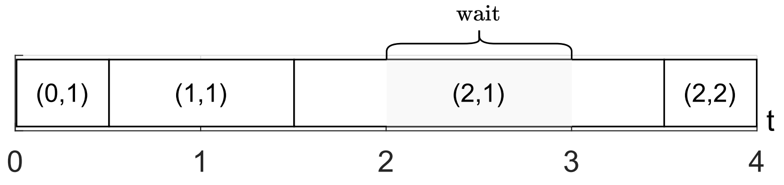


Figure 3.2: Time intervals from Example 3.1.

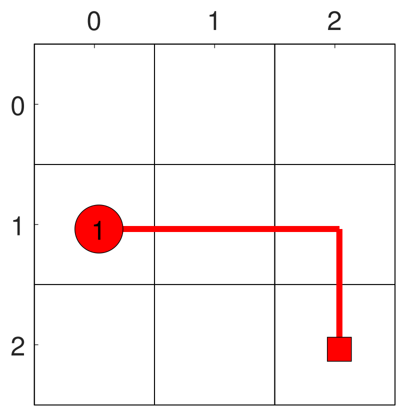


Figure 3.3: Agent's plan from Example 3.1.

A plan as a sequence of actions can be derived from time intervals by looking at the timeline at discrete times $0, 1, 2, \ldots$. An agent occupying the same cell in two following time steps can be translated to a wait action. If the following cells differ, then the resulting action is move. The concrete move action (left, right, up, down) is determined by the relative position of those cells.

From representation as time intervals with cells assigned to them, we can easily represent agents' plans as a job shop schedule. Machines may be interpreted as cells and jobs as agents, and operation is assigned to the machine's corresponding cell. For interpreting operations, I will first define the *occupancy time interval* Ω_{ij} of operation O_{ij} as the whole time agent spends in a cell determined by function $\lambda(i, j)$, which assigns operation to a machine. Therefore, it satisfies:

$$\Omega_{ij} = \begin{cases} (R_{i,j}, R_{i+1,j}), & \text{if } \mu(j) < i \\ (R_{i,j}, \infty), & \text{if } \mu(j) = i \end{cases} \tag{3.5}$$

For a model with *disappear-at-target* behaviour, the last operations of each job $O_{\mu(j)j}$ could be defined as interval $\Omega_{\mu(j)j} = (R_{\mu(j)j}, C_{\mu(j)j})$. Note that the only restriction for processing requirement p_{ij} of operation O_{ij} is that $p_{ij} \leq |\Omega_{ij}|$. The choice of meaningful processing requirements will be provided in the following section.

Example 3.2. Consider the same plan as in Example 3.1. The problem has one job J_1 corresponding to the only agent, and the machines are named after cell coordinates. In this example, the occupancy time intervals are: $\Omega_{1,1} = (0, 0.5)$, $\Omega_{2,1} = (0.5, 1.5)$, $\Omega_{3,1} = (1.5, 3.5)$, $\Omega_{4,1} = (3.5, \infty)$, these time intervals can be easily translated to a sequence of actions by applying the same procedure described above. The resulting plan is shown in Table 3.1.

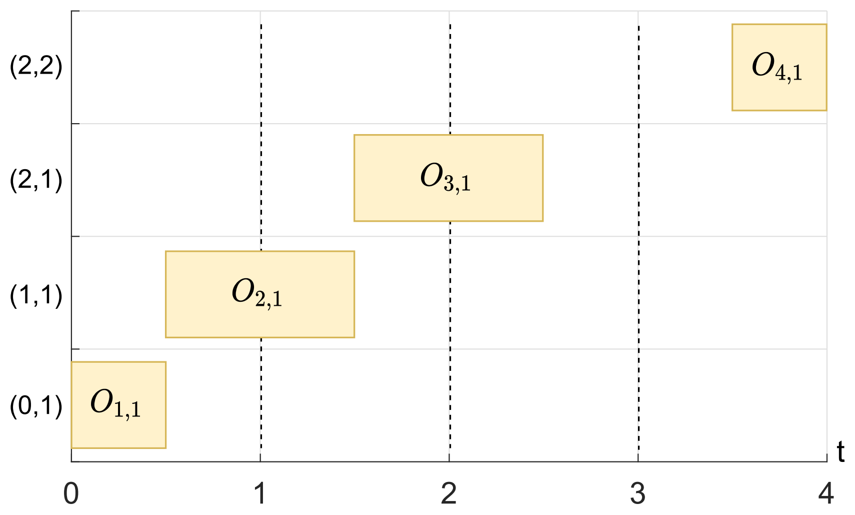


Figure 3.4: Schedule of the plan shown in Table 3.1.

3.3 Plan repair

In the plan repair problem, we consider having a feasible solution that cannot be executed, for various reasons, as planned. In this text, I assume that the solution cannot be executed because some agents were delayed in their start position. Each agent A_j has assigned delay $d_j \in \mathbb{N}$ to it. The modified plan with delayed agents may lead to an unfeasible solution with collisions, therefore it is necessary to correct the plan.

I assume reuse and benefit from having a plan that, without delays, would be feasible and ideally optimal. The plan is reduced into a *path*, which is a sequence of cells that each agent visits to reach their goal. Contrary to the term *plan*, a plan contains information about the order in which an agent must visit the nodes, but does not contain timing. The goal is to add a timing to the path that would result in a feasible solution that minimises the cost function.

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

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

Example 3.3. Consider the plan from Example 3.1 delayed by two time steps. This example would have properties shown in Table 3.2.

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

Table 3.2: Properties of the example problem.

3.4 Representations and decoding

This section presents different representations and decoding of the job shop schedule for the plan repair problem.

3.4.1 Naive approach

The naive approach uses operation-based representation and the same decoding as described in the previous chapter. However, this approach fails to represent some possible solutions, as shown in Example 3.4.

Example 3.4. Consider an example with two agents and their paths shown in Figure 3.5. The plan represented as a sequence of actions is shown in Table 3.3. The solution as a job shop schedule is shown in Figure 3.6. This schedule corresponds to one of two optimal plans, therefore we certainly want to be able to represent this schedule with used representation and decoding. However, note that the second operation of agent A_2 could be placed in front of the fourth operation of agent A_1, for example to interval $(0.5, 1.5)$, without violating the rules of a feasible solution. This implies that this schedule is not active, however, operation-based decoding, as stated in [6], decodes only

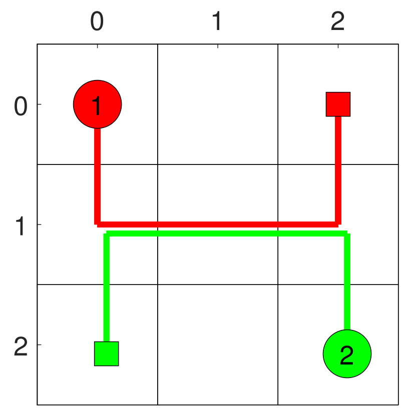


Figure 3.5: Paths of agents in Example 3.4.

active schedules. This means that we cannot represent this solution with the original operation-based approach. To address this problem, a new modified job shop model is introduced in the next part.

agent	plan								
A_1	cells	(0,0)	(0,1)	(1,1)	(2,1)	(2,0)	(2,0)	(2,0)	(2,0)
	actions	down	right	right	up	-	-	-	-
A_2	cells	(2,2)	(2,2)	(2,2)	(2,2)	(2,1)	(1,1)	(0,1)	(0,2)
	actions	wait	wait	wait	up	left	left	down	-

Table 3.3: Plan of agents from Example 3.4.

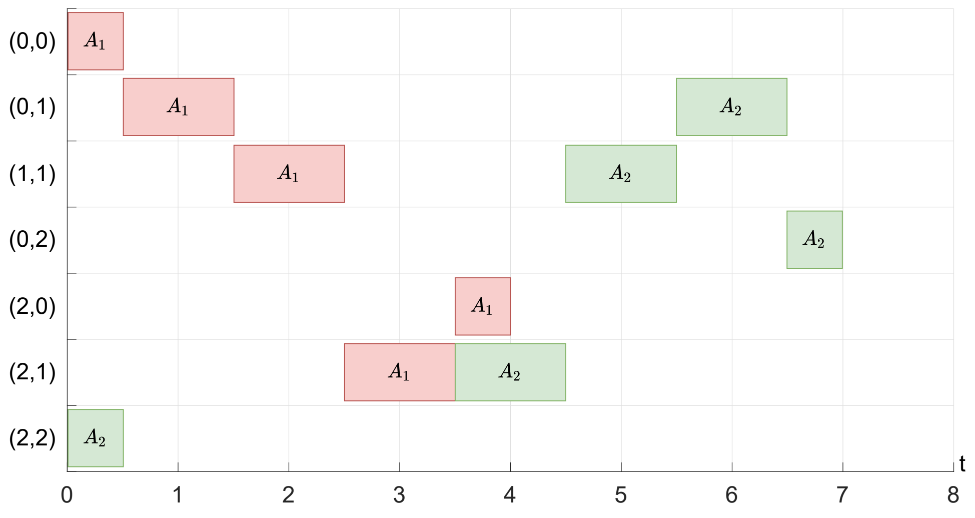


Figure 3.6: Schedule of plan in Example 3.4.

3.4.2 Modified job shop model

The proposed modified model works with the same parameters as the classical job shop model but adds constraints for a feasible solution. Additional constraints are:

A5. First operation O_{1j} of every job J_j must start at time 0.

A6. Each machine can be occupied up to one operation at every moment. For operations O_{ij}, O_{kl} assigned to the same machine, i.e. $\lambda(i,j) = \lambda(k,l)$, must stand:

$$\Omega_{ij} \cap \Omega_{kl} = \emptyset \tag{3.6}$$

A7. 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 $M_{\lambda(i+1,j)} = M_{\lambda(k,l)}$:

$$\Omega_{i,j} \cap \Omega_{k,l} = \emptyset \implies \Omega_{i+1,j} \cap \Omega_{k+1,l} \neq \emptyset \tag{3.7}$$

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 utmost one agent occupying the cell. The last constraint ensures that there is no *swapping* conflict.

3.4.3 The modified operation based representation

In order to solve problems of the naive approach and to satisfy additional constraints listed in the previous section, I proposed a new approach that modifies the decoding and content of the operation based chromosome.

To satisfy the Constraint A5, the first operation O_{1j} of each job J_j is scheduled at the very beginning of the decoding procedure. Scheduling of these operations is not a property of a solution, and therefore they are not represented in the chromosome.

Example 3.5. Consider a simple example with only one agent and a path consisting of three cells, which are represented by operations $O_{1,1}, O_{2,1}, O_{3,1}$. Then the chromosome will be $[1, 1]$, which can be translated into a sequence of operations $[O_{2,1}, O_{3,1}]$.

Note that in order to determine the occupancy time interval $\Omega_{ij} = (R_{i,j}, R_{i+1,j})$ of operation O_{ij}, both operations O_{ij} and $O_{i+1,j}$ must be already allocated to the schedule so that start times $R_{i,j}$ and $R_{i+1,j}$ are known.

Operations are again allocated one by one. Consider that we want to allocate operation O_{ij}. To satisfy the Constraint A6, we must ensure that for every two different operations O_{ij}, O_{ab} assigned to the same machine $M_{\lambda(i,j)} = M_{\lambda(a,b)}$ their occupancy time intervals Ω_{ij} and Ω_{ab} are disjoint. At the time of allocating operation O_{ij} the occupancy time interval is never known, because neither O_{ij} or $O_{i+1,j}$ is yet allocated. On the other hand, the occupancy time interval Ω_{ab} may or may not be known, depending whether operation $O_{a+1,b}$ has been already scheduled. From this information we can derive a rule for allocating an operation that helps to satisfy Constraint A6. The rule is that, when scheduling the operation O_{ij} it must stand that:

$$R_{i,j} \geq R_{a+1,b} \tag{3.8}$$

where operation $O_{a,b}$ is any different operation that belongs to machine $M_{\lambda(a,b)} = M_{\lambda(i,j)}$. Equivalently:

$$R_{i,j} \geq R_{k+1,l} \tag{3.9}$$

where operation O_{kl} has the greatest completion time and belongs to the same machine, that is, $\lambda(i, j) = \lambda(k, l)$. There is an exception in the case that $O_{k,l}$ is the last operation of the job J_l $(\mu(l) = k)$, by this rule it is not possible to allocate O_{ij} without violating the Constraint A6. From the point of view of path planning, it corresponds to the situation where we want the agent A_j to visit the cell adequate to the machine $M_{\lambda(i,j)} = M_{\lambda(\mu(l),l)}$ at the time, when agent A_l has already executed all its actions and now waits in its goal cell. This situation cannot happen without causing a vertex conflict.

The mentioned rule is used in the proposed schedule procedure described in Algorithm 3.

Algorithm 3 Schedule function

```
Input: Operation $O_{ij}$
1: $O_{kl} \leftarrow$ operation with the greatest completion time $C_{kl}$ on the machine $M_{\lambda(i,j)}$.
2: if $\mu(l) = k$ then
3:     Terminate decoding.
4: else if $O_{k,l}$ does not exist then
5:     $R_{ij} = C_{i-1,j}$
6:     Allocate $O_{ij}$ to interval $(R_{ij}, R_{ij} + p_{ij})$.
7:     Return true.
8: else if $O_{k+1,l}$ has been already allocated then
9:     $R_{ij} = \max(C_{i-1,j}, R_{k+1,l})$
10:    Allocate $O_{ij}$ to interval $(R_{ij}, R_{ij} + p_{ij})$.
11:    Return true.
12: else
13:    Return false.
14: end if
```

Schedule function takes operation O_{ij} as an input. First, operation O_{kl} that has the greatest completion time and belongs to the same machine as operation O_{ij}, i.e. $\lambda(i,j) = \lambda(k,l)$, is determined (Line 1). It is checked whether the operation O_{kl} is the last operation of the job J_l, that is, $\mu(l) = k$ (Line 2). If it is true, by allocating additional operations we cannot make a feasible solution from this partial schedule, therefore the decoding procedure is terminated (Line 3). It is also checked whether operation $O_{k+1,l}$ has been allocated (Line 8). If it has, we can allocate operation O_{ij} to the earliest possible time interval that does not violate constrains, i.e. $R_{ij} = \max(C_{i-1,j}, R_{k+1,l})$. If the operation $O_{k+1,l}$ has not been scheduled, the function returns false. If the operation O_{kl} does not exist, then the operation O_{ij} can be scheduled to the earliest time interval $(R_{i-1,j}, R_{i-1,j} + p_{ij})$ that satisfies precedence constraint of operations within the job J_j. (Line 4-7).

The proposed decoding procedure is described by pseudo-code in Algorithm 4.

Algorithm 4 Modified decoding

```
Input: chromosome
 1: Initialize:
        Translate chromosome to operations sequence
 2: for all first operations of job $O$ do
 3:     schedule($O$)
 4: end for
 5: $k \leftarrow 1$
 6: while stopping condition is not met do
 7:     $O \leftarrow$ operations[$k$]
 8:     success $\leftarrow$ schedule($O$)
 9:     if success then
10:         remove $O$ from operations
11:         $k \leftarrow 1$
12:     else
13:         $k \leftarrow k + 1$
14:     end if
15: end while
```

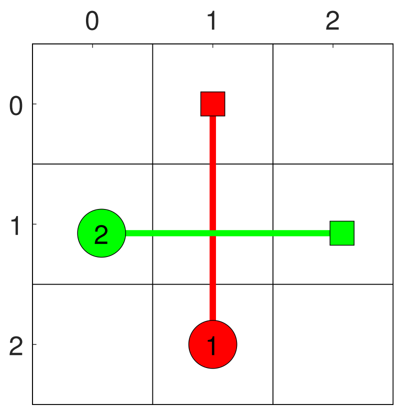
The decoding procedure starts with translating the chromosome into a sequence of operations. The first operation O_{1j} of each job J_j is then scheduled. A counter is used to address the operation we currently want to allocate. The operation determined by the list of operations and the counter is attempted to schedule. If successful, that operation is removed from the list, and the counter is set to point to the first operation of the operation list. If scheduling of the current operation is not successful, the counter is set so that it points to the next operation in the list. This process is repeated until the stopping condition is met.

There are three possible ways to meet the stopping conditions:

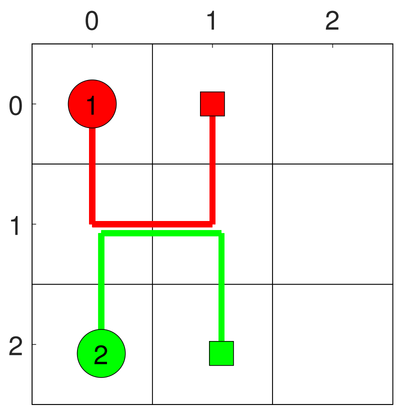
1. The list of operations is empty.
2. Operation O_{ij} is attempted to schedule on machine $M_{\lambda(i,j)} = M_{\lambda(k,l)}$ but there is already scheduled operation O_{kl} and this operation is the last of job J_l, i.e. $k = \mu(l)$.
3. After the last successful allocation of an operation, each item in the operation sequence has been attempted to schedule unsuccessfully.

If the first condition is met, decoding produced a feasible solution, in case of the other two, the produced partial schedule is unfeasible. The first stopping condition is used Examples 3.6 and 3.7. The second stopping condition is applied in Example 3.9 The last stopping condition indicates a swapping conflict, an example of this situation is shown in Example 3.8.

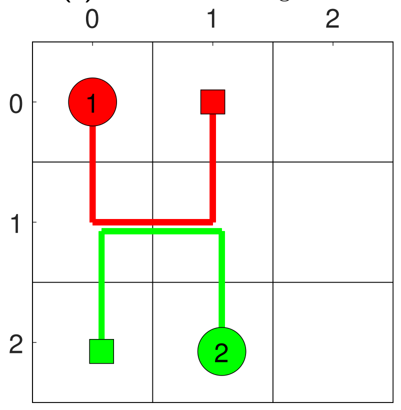
To better illustrate the decoding procedure, consider the four Examples 3.6, 3.7, 3.8 and 3.9. We assume that the machines corresponding to each cell are named after the coordinates of these cells.



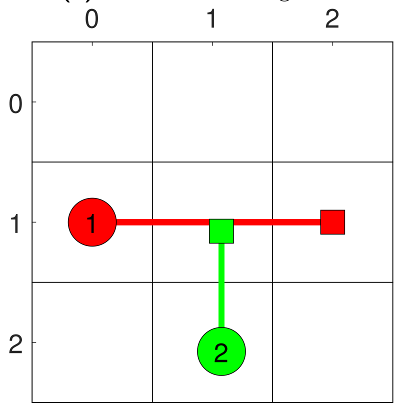
(a) : Paths of two agents.



(b) : Paths of two agents.



(c) : Paths of two agents.



(d) : Paths of two agents.

Figure 3.7: Four path scenarios used in examples.

Example 3.6. Consider two agents, their paths shown in Figure 3.7a and a chromosome $[1, 1, 2, 2]$. First, the chromosome is translated into a sequence of operations $[O_{2,1}, O_{3,1}, O_{2,2}, O_{3,2}]$. Operations $O_{1,1}, O_{1,2}$ are scheduled automatically, then $O_{2,1}$ will be attempted to schedule to cell $(1, 1)$, because in this cell there are no other operations allocated yet, this operation can be allocated to interval $(R_{2,1}, C_{2,1}) = (0.5, 1.5)$ and is removed from the sequence. From sequence $[O_{3,1}, O_{2,2}, O_{3,2}]$ operation $O_{3,1}$ is selected, it belongs to cell $(1, 0)$. This cell has no operations scheduled there, so it can be allocated to time interval $(R_{3,1}, C_{3,1}) = (1.5, 2)$. Next up is operation $O_{2,2}$, it belongs to machine $(1, 1)$. The last scheduled operation on this machine is $O_{2,1}$, in order to find whether $O_{2,2}$ can be allocated, we have to look at operation $O_{2+1,1}$ which is already scheduled and the start time is $R_{3,1} = 1.5$, therefore it can be scheduled to interval $(R_{2,2}, C_{2,2}) = (1.5, 2.5)$. The remaining operation is $O_{3,2}$ that is scheduled to $(R_{2,2}, C_{2,2}) = (2.5, 3)$. After scheduling, the last operation sequence is empty, so the decoding can stop, and the schedule is feasible. The final plan is shown in Table 3.4.

agent	plan				
A_1	cells	(1,2)	(1,1)	(0,1)	(0,1)
	actions	up	up	-	-
A_2	cells	(0,1)	(0,1)	(1,1)	(2,1)
	actions	wait	left	left	-

Table 3.4: Plan of example problem.

Example 3.7. Consider two agents, their paths shown in Figure 3.7b and chromosome $[1, 1, 2, 2, 1, 2]$. The chromosome is translated into the sequence of operations $[O_{2,1}, O_{3,1}, O_{2,2}, O_{3,2}, O_{4,1}, O_{4,2}]$. After allocating the first operations of each job, in the first three iterations operations $O_{2,1}, O_{3,1}, O_{2,2}$ are scheduled successfully and are removed from the sequence. When scheduling operation $O_{3,2}$, the last operation scheduled on machine $(1, 1)$ is $O_{3,1}$, so we must look ahead at operation $O_{4,1}$ which is not yet scheduled, therefore k is increased to $k \leftarrow 2$. Next up is the operation $O_{4,1}$, which belongs to machine $(1, 0)$, this machine has not been visited, so the operation can be scheduled right after the preceding operation $O_{3,1}$. The operation is removed from the sequence and k is restored to 1. The sequence is now $[O_{3,2}, O_{4,2}]$ and both these operations can be successfully scheduled and the resulting solution is feasible. The resulting plan is shown in Table 3.5.

agent	plan					
A_1	cells	(0,0)	(0,1)	(1,1)	(1,0)	(1,0)
	actions	down	right	up	-	-
A_2	cells	(0,2)	(0,2)	(0,1)	(1,1)	(1,2)
	actions	wait	up	right	down	-

Table 3.5: Plan of example problem.

Example 3.8. Consider two agents, their paths shown in Figure 3.7c and a chromosome $[1, 2, 1, 2, 1, 2]$. The chromosome is translated to the sequence $[O_{2,1}, O_{2,2}, O_{3,1}, O_{3,2}, O_{4,1}, O_{4,2}]$. Operations $O_{1,1}, O_{1,2}, O_{2,1}, O_{2,2}$ are all allocated successfully. Then the operation $O_{3,1}$ is scheduled unsuccessfully because it belongs to machine $(1, 1)$. The last scheduled operation on this machine is the operation $O_{2,2}$ and the following operation $O_{3,2}$ has not been scheduled. Similarly the operation $O_{3,2}$ cannot be scheduled because $O_{3,1}$ has not been allocated yet. The same applies to operations $O_{4,1}, O_{4,2}$. In the end, the sequence of operations is $[O_{3,1}, O_{3,2}, O_{4,1}, O_{4,2}]$ and k is equal to 4. All remaining operations in the sequence were scheduled unsuccessfully, and therefore decoding terminates as a failure. The plan corresponding to the partial schedule is shown in Figure 3.6.

agent	plan		
A_1	cells	(0,0)	(0,1)
	actions	down	-
A_2	cells	(1,2)	(1,1)
	actions	up	-

Table 3.6: Plan of example problem.

Example 3.9. Consider two agents, their paths shown in Figure 3.7d and a chromosome $[2, 1, 1]$. The chromosome is translated to sequence $[O_{2,2}, O_{2,1}, O_{3,1}]$. After scheduling the start operations of both jobs, operation $O_{2,2}$ is scheduled successfully on machine $(1, 1)$. Next up is the operation $O_{2,1}$, which belongs to machine $(1, 1)$, last scheduled operation on this machine is the operation $O_{2,2}$, however, this operation is the last operation of the job J_2, therefore, the decoding ends as a failure.

Figures 3.7a, 3.7b and 3.7c show three typical situations, where two agents share some part of their path. In the first situation, agents share only one cell in their path. In the second situation, agents share two or more consecutive cells in their paths and they visit these cells in the same order; contrarily, in the third situation they visit the cells in the reverse order.

Only two possible collision-less outcomes exist, either the agent A_1 or the agent A_2 must visit each cell of the shared section of their paths earlier than the other one. This approach performs well in the first two situations, that is, to change the order in which they visit the shared section of their paths, it is enough to change only the order of operations corresponding to the first cell of that section. However, this approach performs very poorly in the third situation. This is because in order to produce a feasible schedule that changes the order of the two agents, it is necessary to change the order of all operations corresponding to the cells present in the shared section of their paths. Naturally, this problem increases as the shared section grows larger.

3.4.4 The modified preference list-based representation

The main flaw of the approach described in Subsection 3.4.2 was that it often produced schedules that correspond to MAPF plans with swapping conflicts. Therefore, the motivation for proposing another approach was to design an operator that would not produce solutions containing swapping conflicts and thereby reduce the search space. The strategy is to propose an operator that can generate schedules in a more informed way.

Preference list-based representation

The proposed representation is inspired and based on *preference list-based representation* originally proposed in [7]. The chromosome of preference list-based representation is formed by a list of operation sequences, each sequence corresponds to one machine and determines in what order it is preferred to schedule the operations on that specific machine [6].

Example 3.10. Consider the schedule in Figure 2.2. This schedule could be decoded from sequences $[O_{2,2}, O_{1,4}, O_{2,1}]$, $[O_{1,2}, O_{1,3}, O_{2,4}, O_{3,1}]$ and $[O_{1,1}, O_{2,3}, O_{3,4}]$ representing preference lists of machines M_1, M_2, M_3.

When decoding a chromosome, we first take a set of all operations that are, based on the precedence constraints of each job, next to be scheduled. From this set, we select the operations with the highest relative preference, i.e., the operation that is present in the preference list right after the last operation scheduled on the corresponding machine, and allocate those operations. If there are no operations that satisfy these rules, the operations with the second highest preference are selected. This may continue until there is an operation that can be scheduled. The whole process is repeated until all operations are allocated.

Recognizing unfeasible chromosomes

In order to make the chromosome more intuitive and predictable, I added a rule to decoding that the preference list not only defines the preference of operations, but completely determines the order of operations on the corresponding machine. This decoding creates unambiguity, that is, a solution decoded from one chromosome can be represented only by the same chromosome. The disadvantage of this rule is that some chromosomes cannot be decoded into any schedule.

Definition 3.11. Let us define a function *preference number* as:

$$\rho : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \tag{3.10}$$

that assigns an order of operation $O_{i,j}$ in which it appears in the preference list corresponding to machine $M_{\lambda(i,j)}$.

If we assume that preference list completely determines the order of operations on the corresponding machines, then the preference number $\rho(i,j)$ of the operation $O_{i,j}$ determines in what order the operation appears in machine $M_{\lambda(i,j)}$.

Example 3.12. 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_{ij} occurs in the preference list, therefore $\rho(1,1) = 2$, $\rho(1,2) = 3$, $\rho(1,3) = 1$.

An advantage of preference list-based representation (assuming preference lists completely determine the order of operations) is that it is possible to identify some chromosomes that would be decoded as an unfeasible solution. The following list contains necessary conditions:

B1. The first operation O_{1j} of the job J_j must be the first item in the preference list of the corresponding machine $M_{\lambda(i,j)}$, i.e., $\rho(1,j) = 1$.

B2. 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.

B3. For operations O_{ij} and O_{kj} of the job J_j on the same machine, where O_{ij} precedes O_{kj} $(i < k)$, preference number $\rho(i,j)$ must be less that preference number $\rho(k,j)$.

B4. For operations O_{ij}, O_{kl} 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:

$$\rho(i,j) < \rho(k,l) \iff \rho(i+1,j) < \rho(k+1,l) \tag{3.11}$$

B5. For operations O_{ij}, O_{kl} 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:

$$\rho(i,j) < \rho(k,l) \iff \rho(i+1,j) < \rho(k-1,l) \tag{3.12}$$

Constraint B1 corresponds to the condition that each agent starts in a unique starting position. Constraint B2 corresponds to the condition that each agent stays in its goal position until the end of the simulation, this constraint can be omitted for the MAPF models with disappear-at-target behaviour. Constraint B3 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. And they traverse from one cell to another in the same direction. The violation of Constraint B4 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 Constraint B5 corresponds to two agents swapping their locations over the same edge, leading to either a swapping conflict or a vertex conflict (depends on whether the swap takes place in one time step or more).

Figures 3.8 and 3.9 show a useful visual representation, using which it is possible to detect violations of Constraints B1-B5. 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 in the horizontal axis, while the vertical one shows an order of operations on those machines determined by the adequate preference list. Operations at the bottom are the most preferable. To visually verify that Conditions B4 and B5 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. Ff they do, either Condition B4 and B5 is not satisfied. If we label each operation as O_{ij}, it is even possible to verify Constraints B1-B3. To completely check a chromosome, we must examine the path of each agent.

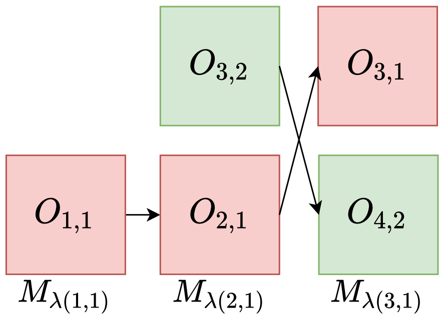


Figure 3.8: Preference list lined up in order of agent's A_1 path.

Example 3.13. Consider the sequence of machines corresponding to the path of the agent A_1 shown in Figure 3.8. We can see that 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 $M_{\lambda(3,1)}$, which 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 B4.

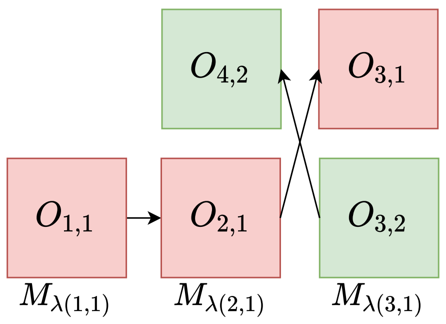


Figure 3.9: Preference list lined up in order of agent's A_1 path.

Example 3.14. Consider the sequence of machines corresponding to the path of the agent A_1 shown in Figure 3.9. We can see that 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 $M_{\lambda(3,1)}$, which also 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 B5.

Cycle conflict

Definition 3.15. The *dependency operation* of a machine M is the operation $O_{k+1,l}$, where $O_{k,l}$ is the last scheduled operation on the machine $M = M_{\lambda(k,l)}$.

Dependency operation was used in Algorithm 4, which allowed to schedule the operation $O_{i,j}$ only if the operation $O_{k+1,l}$, where $O_{k,l}$ is the last operation on the machine $M_{\lambda(i,j)} = M_{\lambda(k,l)}$, was already scheduled. Using the new definition, operation $O_{i,j}$ could be scheduled, only if the dependency operation of machine $M_{\lambda(i,j)}$ had been already scheduled.

Cycle conflict is a situation where four or more agents mutually exchange its position in one time step. Because modern solvers, such as CBS or ECBS can generate plans with cycle conflicts, I will assume that the cycle conflict is allowed. As mentioned in Algorithm 3 the process of applying the `schedule` function on the operation O_{ij} is that we determine the dependency operation of the machine $M_{\lambda(i,j)}$ and the operation O_{ij} may be scheduled as soon as the dependency operation is scheduled. However, in the situation of cycle conflict, there will be operations situated in the cycle, which cannot be scheduled this way. Therefore, it is necessary to use a different procedure, which will be described later in Algorithm 6. Example 3.16 shows a motivation for modification of decoding procedure.

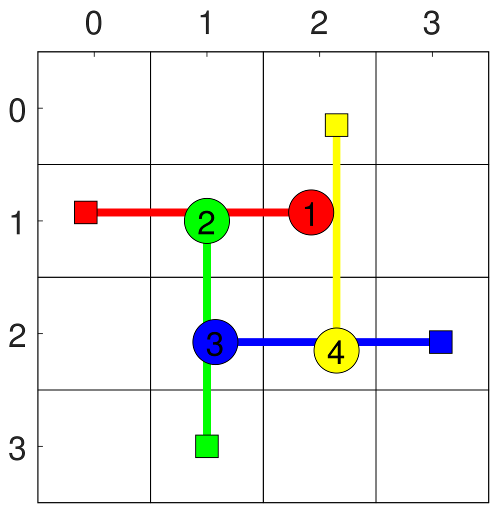


Figure 3.10: Paths of four agents in Example 3.16.

Example 3.16. Consider the chromosome described in Table 3.7. 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 the cycle conflict. After scheduling the first operation of each job, the dependency operation of the machine $(2, 2)$ is the operation $O_{2,4}$, dependency operation of the machine $(1, 2)$ is $O_{2,3}$, the dependency operation of the machine $(1, 1)$ is $O_{2,2}$ and the dependency of the machine $(2, 1)$ is $O_{2,1}$. 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 operation O_{ij} located in the cycle to the time interval $(T_{\text{min}}, T_{\text{min}} + p_{ij})$ and then continue with regular scheduling. The paths of the agents are shown in Figure 3.10 and a part of the schedule is shown in Figure 3.11.

machine	operation order	machine	operation order
$(0, 1)$	$[O_{3,1}]$	$(1, 1)$	$[O_{2,1}, O_{1,2}]$
$(2, 0)$	$[O_{3,4}]$	$(2, 1)$	$[O_{1,1}, O_{2,4}]$
$(1, 3)$	$[O_{3,2}]$	$(1, 2)$	$[O_{1,3}, O_{2,2}]$
$(3, 2)$	$[O_{3,3}]$	$(2, 2)$	$[O_{1,4}, O_{3,2}]$

Table 3.7: Preference lists from Example 3.16.

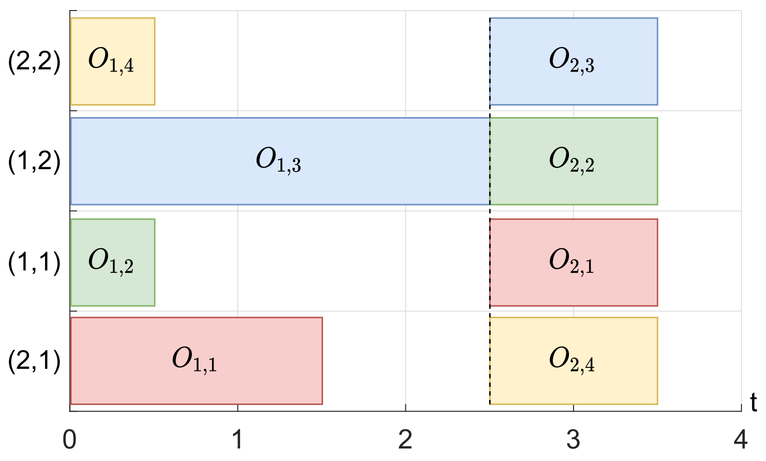


Figure 3.11: Part of the schedule in Example 3.16.

The purpose of Algorithm 5 is to recognise a cycle and determine the minimum start time.

Algorithm 5 Cycle detection

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

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_{ij}, 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 5 is the operation O_{ij} that is the next operation that should be scheduled according to the precedence constraints of the job J_j. Algorithm 5 gradually checks whether the operation O_{ij} 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_{ij} is either the input operation or the dependency operation of some machine, therefore it is implicitly checked that the operation O_{ij} 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_{ij} is redefined to be the dependency operation of the machine that contains the current operation O_{ij}. 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.

Decoding

As stated earlier, we assume that the preference lists completely determine the order in which operations appear on the adequate machine. The decoding Algorithm 6 uses functions `schedule` as defined in Algorithm 3 and function `forceSchedule`, which schedules the input operation O_{ij} to time interval $(T, T + p_{ij})$, where T is the second input argument.

Algorithm 6 attempts to schedule the operations of one job one by one. It is checked whether the operation satisfies the constraints of the corresponding preference list (Line 7) and attempted to schedule (Line 9). If the operation is scheduled successfully, we move on to the next operation of the job (Lines 8-12); otherwise, it is checked whether the reason for unsuccessful applying the schedule function was not caused by the fact that this operation is located in a cycle conflict (Line 14), if it is, we schedule the operation into calculated time interval (Line 16) and the attention moves to the next operation of the job (Lines 15-18). In the event that the operation cannot be scheduled, we move on to another job. Chromosomes that do not produce any schedule are detected by recognising that the decoding cannot add any other operation (Lines 23-25).

Example 3.17. Consider a problem with two agents; their paths are shown in Figure 3.12. The preference list corresponding to machine $(1, 1)$ is $[O_{2,2}, O_{2,1}]$ and to machine $(2, 1)$ is $[O_{5,2}, O_{3,1}]$, other preference lists contain only one operation. We start by scheduling operations of the job J_1, we can schedule operation $O_{1,1}$ but the operation $O_{2,1}$ cannot be scheduled because it is not its turn according to the preference list of $M_{\lambda(2,1)}$, the operation $O_{2,2}$ has to

Algorithm 6 Decoding procedure

```
do
    counters $\leftarrow \{0, \dots, 0\}$
    scheduleChanged $\leftarrow$ false
    for $j \in \{1, \dots, n\}$ do
        $i \leftarrow$ counters$[j]$
        $O_{kl} \leftarrow$ nextOperation$(M_{\lambda(i,j)})$
        if $O_{ij} \neq O_{kl}$ then continue
        else
            success $\leftarrow$ schedule$(O_{ij})$
            if success then
                scheduleChanged $\leftarrow$ true
                counters$[j] \leftarrow$ counters$[j] + 1$
            else
                cycleDetected, $T_{\min} \leftarrow$ detectCycle$(O_{ij})$
                if cycleDetected then
                    forceSchedule$(O_{ij}, T_{\min})$
                    scheduleChanged $\leftarrow$ true
                    counters$[j] \leftarrow$ counters$[j] + 1$
                end if
            end if
        end if
    end for
    if scheduleChanged = false then
        terminate decoding                    ▷ decoding not successful
    end if
while counters $\neq \{\mu(1), \dots, \mu(n)\}$
```

be allocated first, therefore we switch from scheduling the job J_1 to scheduling operations of job J_2, because all operations of the job J_2 occur first in each preference list all, they can all be scheduled on by one. We can then return to scheduling the job J_1 and allocate the remaining operations.

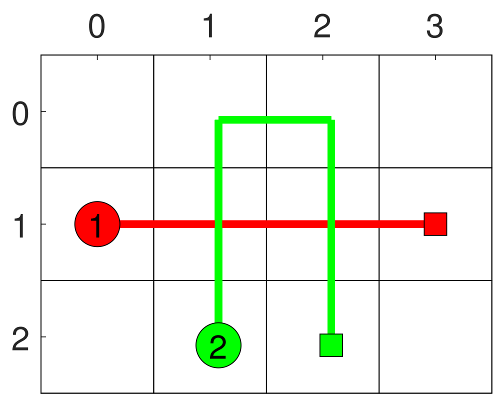


Figure 3.12: Paths of two agents from Example 3.17.

3.5 The final approach

With the modified preference list-based representation, we are now capable of representing any feasible MAPF plan. To construct the functional VNS method, two more things are left to acquire: an initial feasible solution and neighbourhood structures that can be used for shake and local search functions. This section presents a way to obtain a feasible initial solution from a collision-free plan and offers a random operator that can be used as a neighbourhood structure.

3.5.1 The initial solution

In order to apply VNS, it is first necessary to obtain an initial feasible solution. Apparently the most convenient way is to reuse the original collision-free plan that does not contain any unnecessary delays. Let us assume that we have the original plan as a sequence of time steps, where each time step contains information about the current position of each agent in the appropriate time step. First, create a list of jobs $J_1, \dots, J_n$ corresponding to 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. For each agent, we apply these two steps:

1. If the machine corresponding to the current position is not in the list of machines, then add it to the list.
2. If the current time step is not equal to 0 and the position of the agent in the previous time step is not equal to the current position of the agent. Then append a new operation (assigned to the current machine) to the list of operations of the job corresponding to the agent.

After the chromosome is obtained, 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 a processing requirement equal to 1. Afterwards, the decoding procedure can be applied, which produces the initial feasible schedule.

3.5.2 The proposed 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 Constraints B1-B5. 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 B1 and B2, when scheduling operation O_{ij}, which is the first operation of the job J_j, i.e., $i = 1$, then this operation must have $\rho(i, j) = 1$, if it is the last operation, i.e., $i = \mu(j)$, then $\rho(i, j)$ has to be equal to the length of the corresponding preference list. If in the current machine there is a operation O_{1l} that is the first operation of the job J_l, then $\rho(1, l)$ is the lower constraint. If there is a operation $O_{\mu(l),l}$ that is the last of the job J_l, then $\rho(\mu(l), l)$ serves as the upper constraint.

When determining the preference number $\rho(i, j)$ for the operation O_{ij} on preference list that contains another operation of the same job O_{kj}, where O_{ij} precedes O_{kj}, i.e., $i > k$, then $\rho(i, j)$ acts as a upper constraint. Note that if $i < k$ we do not take $\rho(i, j)$ as a lower constraint, because of this we may temporarily reorder the preference list so that it does not satisfy the constraint, however, by consistently applying the rule for $i < k$ for all operations of job J_j, then we end up with the correct preference list. This process is applied to satisfy Constraint B3.

In order to satisfy Constraint B4, 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 a 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 limit (when $\rho(k-1, l) < \rho(i-1, l)$) or as a upper limit (when $\rho(k - 1, l) > \rho(i - 1, l)$).

In order to satisfy Constraint B5, 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 a 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 limit (when $\rho(k+1, l) < \rho(i-1, l)$) or as a upper limit (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 these constraints, in that case we have to perform *backtracking*, that is, the process of iterating over a selected job in descending order until we reach a situation where it is possible to change the order so that we can avoid again breaking the constraints.

Algorithm 7 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($O_{ij}, \rho$)
10:     end if
11: end for
```

First, a random job J_j (corresponding to the agent A_j) is chosen (Line 1), it is then iterated over the operations of the job, starting from the operation $O_{2,j}$. The appropriate preference list for the first operations does not have to be modified, because the first operation should always have the preference number $\rho(i, j) = 1$. For each operation, we determine a list of preference numbers for the operation O_{ij} that would meet the constraints described at the beginning of this subsection (Line 4). If the constraints allow us to select a new preference number $\rho(i, j)$, we randomly select from those permissible preference numbers and reorder operations on the machine according to the new preference number (Lines 8-9); otherwise, it is backtracked until we reach the point where it is possible to choose a different order in the preference list, which would not cause conflict of constraints.

Example 3.18. Consider applying the proposed operator to a problem with at least three agents. In this example, it is not necessary to label concrete operations, 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 3.13. 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 3.14. 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.

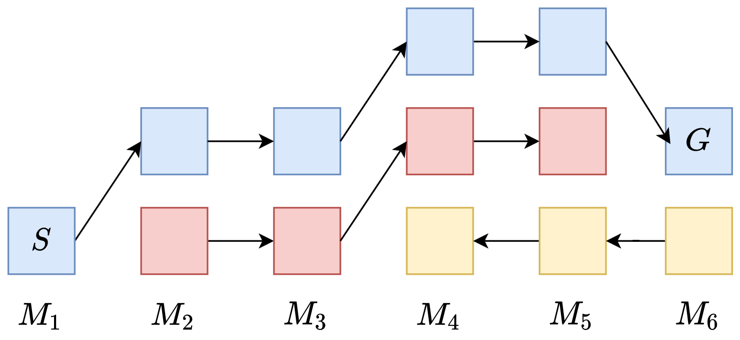


Figure 3.13: The original path of the blue agent.

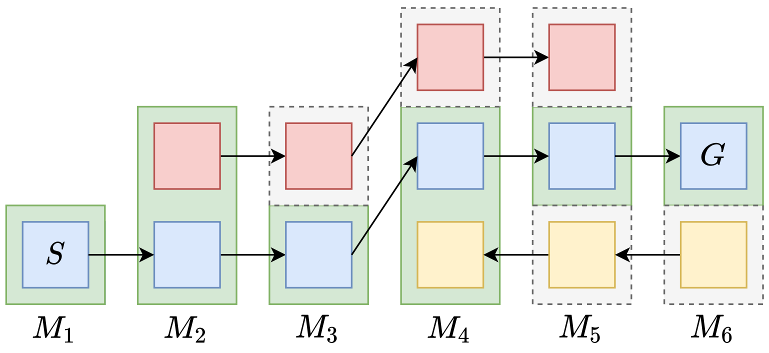
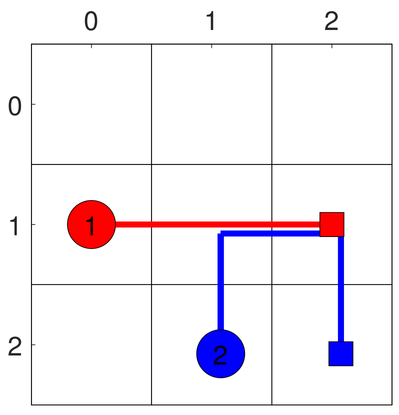


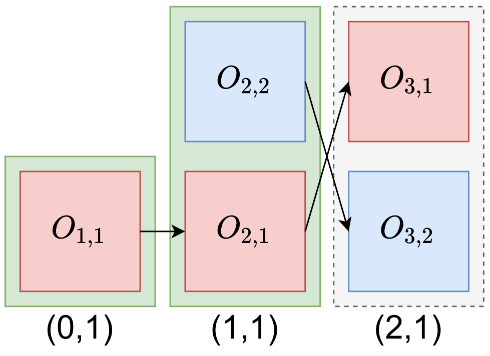
Figure 3.14: The path of the blue agent after applying the operator.

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

Example 3.19. Assume the situation shown in Figure 3.15. 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 B4. 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 B2 would occur.



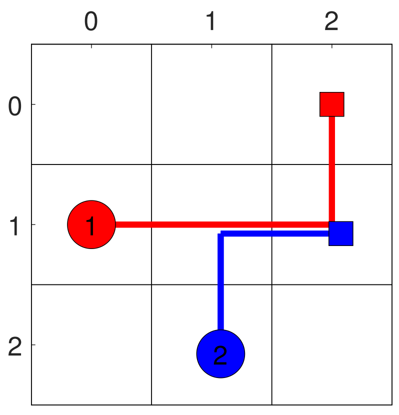
(a) : Paths of both agents.



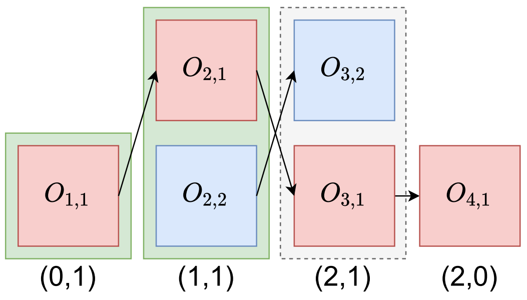
(b) : Path of the agent A_1.

Figure 3.15: Situation addressed in Example 3.19.

Example 3.20. Consider the situation shown in Figure 3.16. 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 B4, 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 B2.



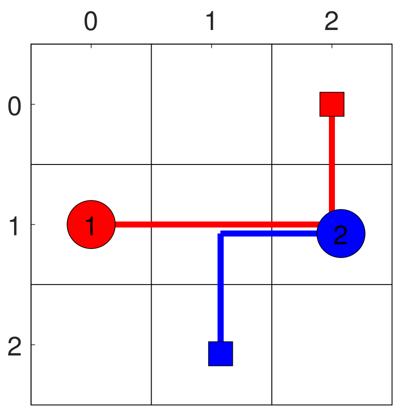
(a) : Paths of both agents.



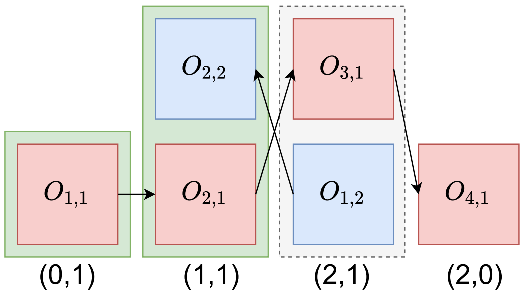
(b) : Path of the agent A_1.

Figure 3.16: Situation addressed in Example 3.20.

Example 3.21. Assume the situation shown in Figure 3.17. When the algorithm reaches the third operation of the job J_1, it should exchange the operations $O_{3,1}$ and $O_{1,2}$ on machine $(2, 1)$ to preserve the relations set in the previous machine $(1, 1)$ by Constraint B5. However, this is not possible, because if the operations were exchanged, the first operation of the job J_2 would not be the first in the preference list, which is a violation of Constraint B2.



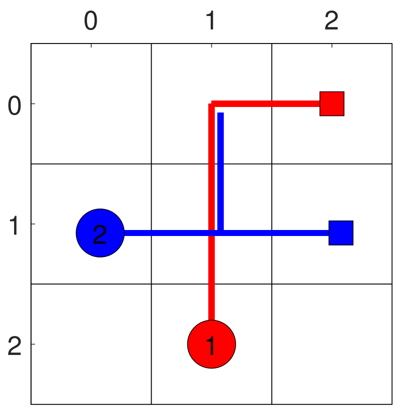
(a) : Paths of both agents.



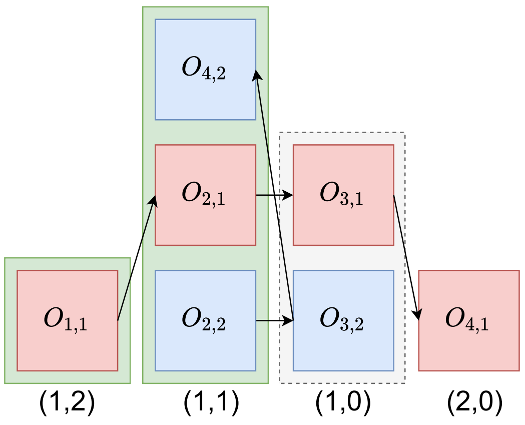
(b) : Path of the agent A_1.

Figure 3.17: Situation addressed in Example 3.21.

Example 3.22. Consider the situation shown in Figure 3.18. In order to satisfy Conditions B4 and B5, preference number $\rho(3,1)$ should be both lower and higher than the preference number $\rho(3,2)$, which is certainly not possible to satisfy.



(a) : Paths of both agents.



(b) : Path of the agent A_1.

Figure 3.18: Situation addressed in Example 3.22.

Example 3.23. Consider the situation shown in Figure 3.18. When the algorithm reaches the fifth operation of the agent A_1, it should place the operation $O_{5,1}$ below the operation $O_{5,2}$ to satisfy Constraint B4, but it should also place the operation above the operation $O_{2,1}$ to preserve the precedence of operations (Constraint B3). These conditions contradict each other; therefore, it is necessary to backtrack.

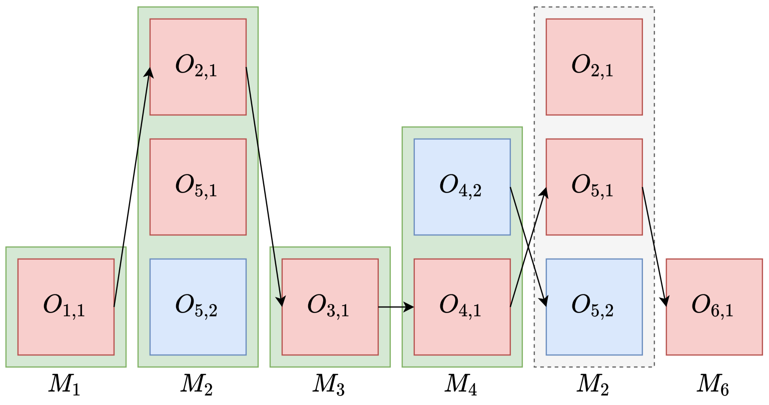


Figure 3.19: Situation addressed in Example 3.23.

3.5.3 Shake and local search

After the proposal of the random operator, it is now possible to form shake and local search functions that are the basis of the general VNS algorithm. Implementation of VNS proposed in [14] used functions `insert` and `exchange` for the local search. Shake functions were constructed as a permutations with repetitions of `insert` and `exchange` neighbourhood structures.

Because I proposed only one neighbourhood function, the design of modified VNS will be very simple. The proposed operator will be used in local search and the shake function is going to be formed by application of N proposed operators in a row. Appropriate value of N will be examined later by experiments.

Chapter 4

Experimental results

In this chapter, I present some experimental results. There were two sets of experiments carried out. The experiments aimed to experimentally verify the functionality of the VNS algorithm on JSS and MAPF problems. All experiments were performed on a computer with a Intel(R) Core(TM) i5-4210U CPU with 4GB RAM and all algorithms were implemented in C++.

4.1 Job Shop Scheduling

This section examines the functionality and properties of the VNS method used for JSS proposed in [14]. Unfortunately, the authors did not mention the number of iterations used nor the computational time of their experiments. Therefore, it is not possible to completely verify the correctness of my implementation.

Experiments were carried out on six problems from the JSPLIB dataset [17]. The parameters of the tested problems are shown in Table 4.1.

problem	abz07	abz08	abz09	la29	orb03	orb08
#machines	20	20	20	20	10	10
#jobs	15	15	15	10	10	10

Table 4.1: Size of tested problems.

As a cost function, makespan M was selected. The quality of solution was evaluated using relative error RE, defined as:

$$\mathrm{RE} = \frac{M - M_{\mathrm{opt}}}{M_{\mathrm{opt}}} \tag{4.1}$$

where M_{opt} indicates the optimal makespan, in case the optimum is known, or the lower bound of the optimum, in cases where the optimum of the problem is not known. Each problem was solved by VNS ten times, and I recorded the average relative error (denoted ARE) and the least relative error of all runs (denoted LRE). Each time VNS was run with a total of 10^6 iterations and relative errors after 10^2, 10^3, 10^4, 10^5 and 10^6 iterations were recorded. Table 4.2 shows the measured ARE and LRE, the reference ARE from [14] and the computation time.

The number of iterations was limited to 10^6 because larger numbers of iterations were very time-consuming. From the results, it is clear that the computational time grows approximately linearly with the number of iterations. Taking into account the quality of the solution found, the reference experiments from [14] showed better results. However, we can see that the improvement of the solutions, found by VNS, from 10^5 to 10^6 iterations is not negligible. Therefore, we can deduce that the quality of the solution could improve with an increase above 10^6 iterations and given enough time, the measured results could match the reference values.

4.2 Multi-Agent Path Finding

To verify the functionality of the experiments related to MAPF, maps from MAPF Benchmark Sets [16] were used.

From the two proposed representations of Multi-Agent Path Finding as the JSS problem, the one based on the preference list-based representation was selected, as it performed significantly better during testing in the early stage of development. The first experiment aimed to identify the appropriate shake function. The second experiment tested the quality of solutions found by VNS depending on the number of iterations. The third experiment examined the evolution of cost with increasing the number of iterations of VNS.

On each map, ten instances, that is, the start and goal positions of all agents, were randomly generated. Then each instance was solved using ECBS with the requirement that the makespan of the obtained plan will be equal at most to 105% of the makespan of the optimal solution. For this purpose, I used Keisuke Okumura's [12] implementation of ECBS.

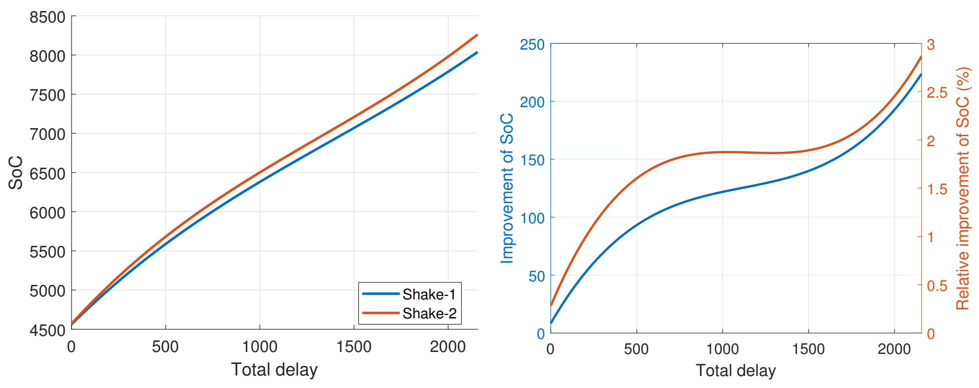
Model problem that we aim to solve is a situation in which some of the agents are delayed at their start position. To examine the effect of the total delay, that is, a sum of delays of all agents, experiments with the following parameters were carried out:

- There were a total of 100 agents.
- K random integers, which represent delays in time steps, are selected from range $\{1, \dots, 10\}$ and assigned to randomly selected agents (each agent can be selected more than once).
- For all $K \in \{0, 10, \dots, 390, 400\}$, delays are assigned to the appropriate agents. And VNS with the sum of costs as the objective function is run.

In each run, I recorded the achieved sum of costs and the computation time.

4.2.1 Appropriate shake function

Structure of the proposed VNS was outlined in Subsection 3.5.3. It was stated that shake function consists of N sequential applications of the proposed random operator described in Algorithm 7, and it is necessary to determine the value of N. I tested two variants of the shake function. The first variant tested (`Shake-1`) was simply Algorithm 7. The second variant of the shake function (`Shake-2`) applied Algorithm 7 twice in a row. This experiment was carried out on the map `random-64-64-20`, the quality of the solution was examined by measuring the sum of costs.



(a) : SoC of solutions found by VNS with `Shake-1` and `Shake-2` functions.

(b) : Improvement of SoC found using `Shake-1` relative to SoC found using `Shake-2`.

Figure 4.1: `Shake-1` and `Shake-2` comparison on map `room-64-64-16`.

Measured data were fitted using a polynomial, the results of the experiment are shown in Figures 4.2, 4.3 and 4.4. As you can see, `Shake-1` yielded slightly better results, so the next experiments were carried out using this VNS setup.

4.2.2 Variable Neighbourhood Search functionality

The functionality of Variable Neighbourhood Search was tested on three maps: `random-64-64-20`, `room-64-64-16` and `warehouse-10-20-10-2-1`. For reference solutions I used plans generated by an algorithm based on Action Dependency Graph (ADG) [9] implemented by Josef Weis [18]. This algorithm constructs a graph that contains the action-precedence relations and ensures that actions, which may cause a collision, are executed in the same order as they were planned in the original solution. ECBS plans trajectories that may contain cycle conflicts, the proposed algorithm can deal with cycle conflicts. However, ADG cannot repair such plans, therefore the success rate of ADG on each tested map is shown in Table 4.3.

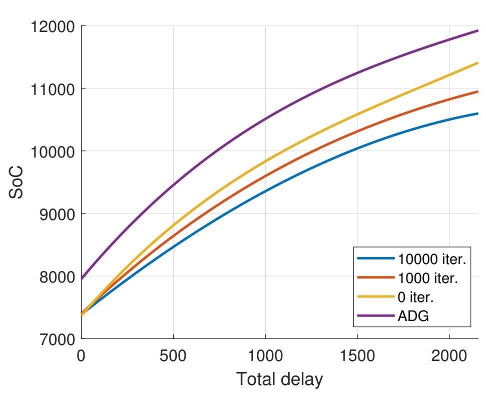
map	success rate (%)
`room-64-64-16`	70
`random-64-64-16`	100
`warehouse-10-20-10-2-1`	100

Table 4.3: Success rate of ADG on different maps.

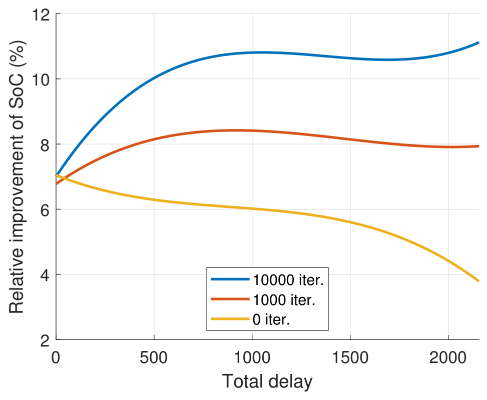
VNS algorithm was run with a total of 10000 iterations, and the SoC of each run was recorded at 0, 1000, 10000 iterations. The solution at 0 iterations is equivalent to the initial feasible solution described in Subsection 3.5.1. The measured data for both VNS and ADG were fitted using a polynomial and the results are shown in Figures 4.2, 4.3 and 4.4, the figures show the absolute SoC values and the improvement relative to the solution found by ADG defined as:

$$\mathrm{RI} = \frac{f_{\mathrm{ADG}} - f_{\mathrm{VNS}}}{f_{\mathrm{ADG}}} \tag{4.2}$$

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, I calculated the average time of each experiment. The average times are shown in Table 4.4.

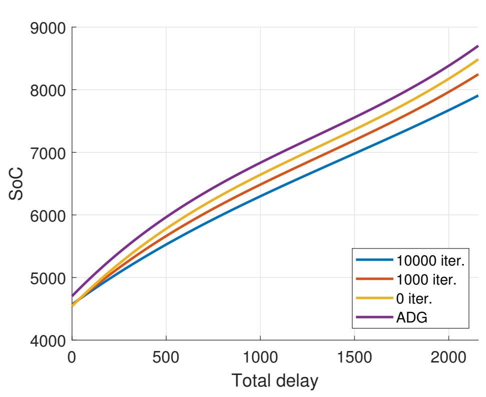


(a) : SoC for various number of iterations of VNS and SoC found by ADG.

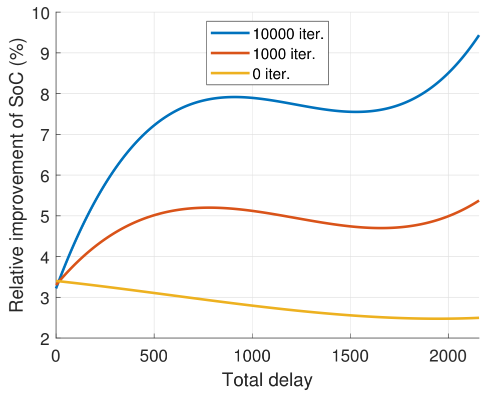


(b) : Improvement of SoC after applying VNS relative to solutions found by ADG.

Figure 4.2: Quality of solutions found by VNS and ADG on map `room-64-64-16`.

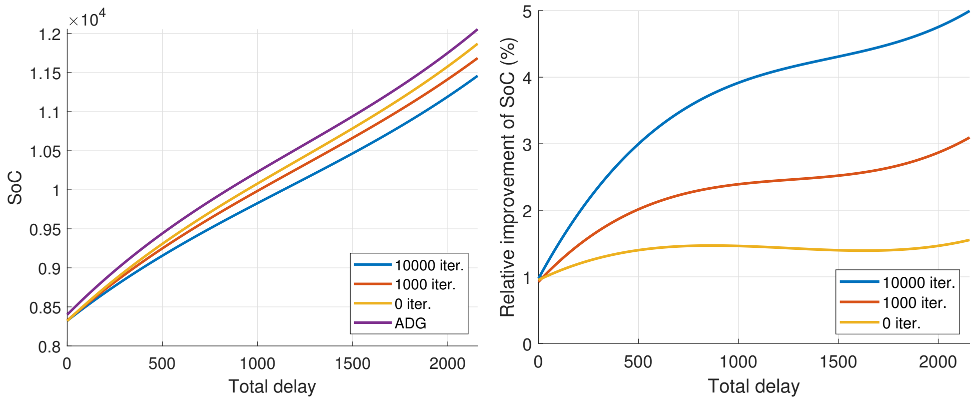


(a) : SoC for various number of iterations of VNS and SoC found by ADG.



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

Figure 4.3: Quality of solutions found by VNS and ADG on map `random-64-64-20`.



(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 4.4: Quality of solutions found by VNS and ADG on map `warehouse-10-20-10-2-1`.

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 4.4: Measured CPU time of running VNS on maps `room-64-64-20`, `random-64-64-16` and `warehouse-10-20-10-2-1`.

From the results, it is clear that VNS worked best compared to ADG on the map `room-64-64-16` both from the perspective of success rate and relative improvement of a solution. On the other two maps VNS could still find consistently better solutions, but the relative improvement did not reach the improvement measured on the first mentioned map.

To evaluate the usability of the VNS algorithm, the computation time should be taken into account. The instances plans on maps `random-64-64-20` and `warehouse-10-20-10-2-1` were all calculated using ECBS in less than a second. However, the time required to solve problems on map `room-64-64-16` 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-64-64-16`.

4.2.3 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-64-64-16` 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 4.5, the graph shows the evolution of SoC and the improvement relative to the initial solution. It is clear that most of the improvement was made during the beginning of the search - cca 94% of the total improvement was made in the first 10000 iterations.

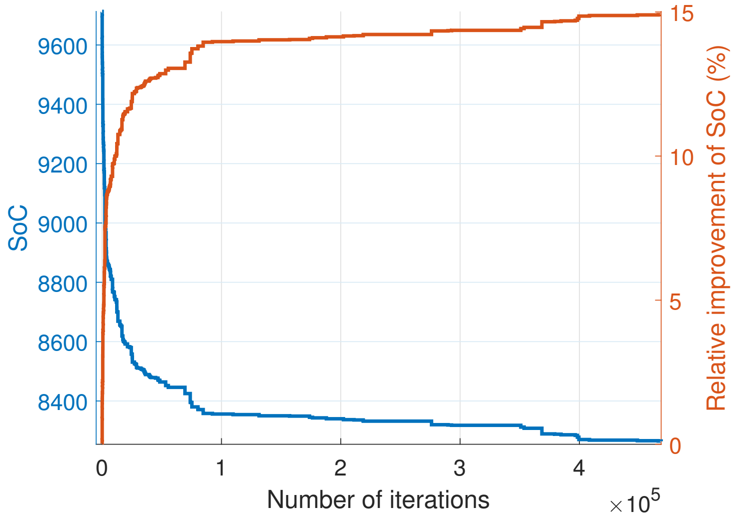


Figure 4.5: Evolution of solution with increasing iterations.

problem	*opt*	iter.	ARE (%)	LRE (%)	ref. ARE (%)	time (ms)
abz07	656	10^2	20.79	14.63	2.01	17
		10^3	17.46	13.26		174
		10^4	12.27	9.29		1714
		10^5	8.23	6.55		16678
		10^6	5.64	3.35		167565
abz08	665	10^2	24.96	22.70	1.99	16
		10^3	17.46	13.26		159
		10^4	14.75	12.18		1623
		10^5	11.24	7.96		16700
		10^6	7.36	6.16		166636
abz09	679	10^2	24.84	21.79	2.17	11
		10^3	19.26	15.61		109
		10^4	14.74	12.37		1067
		10^5	10.73	8.54		10606
		10^6	7.18	5.59		110435
la29	1152	10^2	22.11	17.88	1.80	13
		10^3	15.97	12.76		131
		10^4	11.60	9.20		1305
		10^5	9.57	7.20		12910
		10^6	6.93	3.29		123204
orb03	1005	10^2	22.82	18.90	2.60	4.02
		10^3	15.91	10.84		41
		10^4	10.00	4.47		397
		10^5	7.66	4.47		3970
		10^6	6.01	1.69		39186
orb08	899	10^2	20.51	15.46	1.63	8
		10^3	13.17	8.67		78
		10^4	7.75	2.55		781
		10^5	3.73	1.55		7792
		10^6	2.59	0.00		77658

Table 4.2: Results of experiments on six problems from JSPLIB.

Chapter 5

Conclusion and future work

5.1 Conclusion

In this thesis, I presented a procedure for translating a job shop schedule to MAPF plan and vice versa. Additional rules to the basic job shop model were added to recognise an unfeasible MAPF plan from a job shop schedule. Two modified representations of JSS were proposed and their pros and cons were presented. A simple variant of VNS was formed using the proposed neighbourhood function. The functionality of the proposed variant of the VNS algorithm was experimentally tested on MAPF Benchmark Sets [16] and was compared to algorithm that uses Action Dependency Graph. Experiments showed that the proposed algorithm consistently performed better than ADG.

5.2 Future work

I see room for improvement in the optimisation method used - the proposed variant of VNS has a very simple structure and uses the same function for both the local search and the shake function. Improvement could be obtained by creating new neighbourhood functions, for example, one that would make smaller changes to the plan that the proposed operator.

Alternatively, completely different optimisation method could be used - VNS does not use information that are available, such as the initial delay of agents, time that agent must spend waiting for other agents to pass, etc.. Using a method such as GRASP [2] and the information mentioned, it may be possible to explore the search space more efficiently and therefore reduce the time needed to find a sufficient solution.

The proposed variant of VNS was compared to ADG, which generates only robust MAPF plans. It would be interesting to explore the possibilities of representing robust plans as job shop schedules and then compare these two approaches.

Appendix A

Acronyms

ADG	Action Dependency Graph
CBS	Conflict Based Search
ECBS	Enhanced Conflict Based Search
JSS	Job Shop Scheduling
MAPF	Multi-Agent Path Finding
VNS	Variable Neighbourhood Search

Appendix B

Bibliography

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

[2] S. Binato, W. Hery, D. Loewenstern, and M. G. Resende. A GRASP for Job Shop Scheduling. In *Essays and surveys in metaheuristics*, pages 59–79. Springer, 2002.

[3] P. Brucker. *Scheduling algorithms / Peter Brucker.* Springer-Verlag, Berlin ; Heidelberg ; New York, 1st ed. 1995. edition, 1995.

[4] R. Bürgy and K. Bülbül. The Job Shop Scheduling problem with convex costs. *European Journal of Operational Research*, 268(1):82–100, 2018.

[5] J. Błażewicz, W. Domschke, and E. Pesch. The Job Shop Scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93(1):1–33, 1996.

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

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

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

[9] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian. Persistent and robust execution of mapf schedules in warehouses. *IEEE Robotics and Automation Letters*, 4(2):1125–1131, 2019.

[10] J. K. Lenstra and D. B. Shmoys. Elements of scheduling, 2020.

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

[12] K. Okumura, Y. Tamura, and X. Défago. Iterative refinement for real-time multi-robot path planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9690–9697, 2021.

[13] A. Z. Sevkli and F. E. Sevilgen. Variable Neighborhood Search for the orienteering problem. In *ISCIS*, 2006.

[14] M. Sevkli and M. E. Aydin. A Variable Neighbourhood Search algorithm for Job Shop Scheduling problems. In J. Gottlieb and G. R. Raidl, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 261–271, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

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

[16] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks, 2019.

[17] tamy0612. JSPLIB. `https://github.com/tamy0612/JSPLIB`, 2014.

[18] J. Weis. Robust Plan Execution in Multi-Agent Systems. Bachelor thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, [cit. 2022-05-20]. SUPERVISOR: RNDr. Miroslav Kulich, Ph.D.

Appendix C

Attached files

Folder	Description
`thesis`	Contains this thesis.
`JSS`	Implementation of VNS for JSS.
`MAPF3`	Implementation of VNS for MAPF.
`common`	C++ codes shared between folders `JSS` and `MAPF3`.
`EXPERIMENTS`	Contains measured data and scripts for data visualisation.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kubišta Daniel**	Personal ID number:	**492344**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Cybernetics and Robotics**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Plan Repair for a Team of Mobile Agents

Bachelor’s thesis title in Czech:

Oprava plánu pro team mobilních agentů

Guidelines:

Multi-Agent Path Finding aims to find the optimal collision-free trajectory for a team of mobile agents (robots) from their starting positions to given destinations. However, unexpected events may occur during plan execution that prevent the plan from being realized as designed. It is thus necessary to monitor whether such events have occurred and subsequently correct the plan. Plan correction on fixed paths can be viewed as a variant of the Job Shop Scheduling Problem.
The student's task will be:
1) Get acquainted with the methods of solving the Job Shop Scheduling Problem (JSS).
2) Implement selected metaheuristic for JSS (eg [2]).
3) Experimentally verify the functionality and properties of the implemented algorithm on the JSPLIB dataset (https://github.com/tamy0612/JSPLIB) and document the results.
4) Define the problem of plan repair as the JSS and modify the JSS solver for plan repair.
5) Verify experimentally properties of the proposed plan repair solver and document the results.

Bibliography / sources:

[1] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham and N. Ayanian, "Persistent and Robust Execution of MAPF Schedules in Warehouses," in IEEE Robotics and Automation Letters, vol. 4, no. 2, pp. 1125-1131, April 2019, doi: 10.1109/LRA.2019.2894217.
[2] Sevkli M., Aydin M.E. (2006) A Variable Neighbourhood Search Algorithm for Job Shop Scheduling Problems. In: Gottlieb J., Raidl G.R. (eds) Evolutionary Computation in Combinatorial Optimization. EvoCOP 2006. Lecture Notes in Computer Science, vol 3906. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11730095_22
[3] Hansen, P., Mladenović, N., Todosijević, R. et al. Variable neighborhood search: basics and variants. EURO J Comput Optim 5, 423–454 (2017). https://doi.org/10.1007/s13675-016-0075-x

Name and workplace of bachelor’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D.   Intelligent and Mobile Robotics CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **16.01.2022**   Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature