Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Optimization Metaheuristic for Robust Multi-Agent Pathfinding

Jan Podlucký

Supervisor: Ing. David Zahrádka
January 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Podlucký Jan**	Personal ID number:	**503173**
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:

Optimization Metaheuristic for Robust Multi-Agent Pathfinding

Bachelor’s thesis title in Czech:

Optimalizační metaheuristika pro robustní multiagentní plánování

Guidelines:

1. Familiarize yourself with the problem of robust Multi-Agent Pathfinding (MAPF) and with methods to solve it.
2. Extend the Safe Interval Path Planning (SIPP) algorithm with the capability to find k-robust solutions.
3. Adapt the MAPF-LNS2 optimization metaheuristic to k-robust MAPF.
4. Design and implement various strategies for increasing robustness in context of practical applications of robotic fleets, e.g.: finding a (k+1)-robust solution given a k-robust solution, finding a k-robust solution given a (k+1)-robust solution, robustness increasing with time.
5. Experimentally verify the functionality of the developed methods and compare their properties with each other and with selected standard k-robust MAPF algorithms.

Bibliography / sources:

[1] H. Ma and S. Koenig, “AI buzzwords explained: Multi-agent path finding (MAPF),” AI Matters, vol. 3, no. 3, pp. 15–19, Oct. 2017.
[2] M. Phillips and M. Likhachev, “SIPP: Safe interval path planning for dynamic environments,” in 2011 IEEE International Conference on Robotics and Automation, May 2011, pp. 5628–5635.
[3] D. Atzmon, R. Stern, A. Felner, G. Wagner, R. Bartak, and N.-F. Zhou, “Robust Multi-Agent Path Finding,” Proceedings of the International Symposium on Combinatorial Search, vol. 9, no. 1, pp. 2–9, Sep. 2021.
[4] J. Li, Z. Chen, D. Harabor, P. J. Stuckey, and S. Koenig, “MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search,” Proceedings of the AAAI Conference on Artificial Intelligence, vol. 36, no. 9, pp. 10 256–10 265, Jun. 2022.

Name and workplace of bachelor’s thesis supervisor:

Ing. David Zahrádka   Intelligent and Mobile Robotics CIIRC

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

Date of bachelor’s thesis assignment: **22.09.2023**   Deadline for bachelor thesis submission: **09.01.2024**

Assignment valid until: **16.02.2025**

Ing. David Zahrádka	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

Acknowledgements

I would like to thank my thesis supervisor, Ing. David Zahrádka, for his guidance and support in working on this project. I would also like to thank RNDr. Miroslav Kulich, Ph.D., for his consultations.

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, January 8, 2024

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 8. ledna 2024

Abstract

This thesis focuses its attention on the problem of path planning for multiple robots, also referred to as agents. This is a problem of finding collision-free paths for multiple agents from each agent's start location to its goal location. This work focusses mainly on two MAPF algorithms, the MAPF-LNS2 algorithm and the MAPF-SIPP algorithm, which are being upgraded to solve a k-robust path planning problem. In the k-robust path planning, each agent can be delayed up to k time steps without colliding with any other agent. Furthermore, this thesis explores different strategies to find k-robust paths. The results show that the proposed k-robust MAPF-LNS2 method outperforms the standard k-robust MAPF-SIPP algorithm in terms of the success rate in finding the solutions and also in the sum of costs of the solution.

Keywords: MAPF-LNS2, MAPF-SIPP, K-robustness, Path planning

Supervisor: Ing. David Zahrádka
Intelligent and Mobile Robotics CIIRC

Abstrakt

Tato práce se zaměřuje na problém plánování trasy pro více robotů, v této práci nazývaných agenty. Jedná se o úlohu nalezení bezkolizních tras pro více agentů z jejich startovních míst do jejich cílových lokalit. Tato práce se soustředí především na dva multiagentní algoritmy, a to algoritmus MAPF-LNS2 a algoritmus MAPF-SIPP, které jsou vylepšeny tak, aby řešily problém plánování k-robustní cesty. V k-robustním plánování trasy může být jakýkoliv z agentů zpožděn až o k kroků bez toho, aby kolidoval s jakýmkoli jiným agentem. Tato práce dále zkoumá různé strategie pro nalezení k-robustních cest. Výsledky ukazují, že navržená k-robustní metoda MAPF-LNS2 předčí standardní k-robustní algoritmus MAPF-SIPP, jak co se týče úspěšnosti nalezení řešení, tak i ceny nalezených cest.

Klíčová slova: MAPF-LNS2, MAPF-SIPP, K-robustnost, Plánování trasy

Překlad názvu: Optimalizační metaheuristika pro robustní multiagentní plánování

Contents

Figures

Tables

Chapter 1
Introduction

As the inclination of the workforce towards manual labour experiences a gradual decline, the industry is putting a growing emphasis on automation. One of the many branches of automation are cooperative robots. These robots are used in many branches of industry, such as warehouse management [22] or, for example, airport surface operations [12]. Consequently, this thesis focuses its attention on the problem of path planning for multiple robots, also referred to as agents.

Multi-agent Path Finding (MAPF) is a problem of finding collision-free paths for multiple agents from each agent's start location to its goal location. The MAPF algorithm during the planning of the trajectories of the agents assumes that each agent will follow its plan perfectly. However, in the real world, agents may experience delays. They can be caused by many factors, such as a low battery of an agent or slippage of the robot wheel. Each of those unpredicted delays could cause a deviation from the plan and subsequent collision between agents.

Considering the fact that agents should not collide, we would need to stop all agents in the workspace and replan the trajectory of each of them, which would take some time depending on the number of agents used, the size of the workspace, and other factors. This approach is doable if there are not many agents and delays do not occur often. However, in large companies where there are hundreds of agents cooperating, this approach is inadmissible as the chances of at least one of them being delayed are much higher, and each interruption of the production costs a lot of money.

Therefore, we define a k-robust MAPF algorithm [2] as one that finds paths for agents in such a way that any number of them can be delayed by up to k time steps without resulting in collisions. The extension of MAPF algorithms to find k-robust paths is the main task of this thesis.

This thesis is focused mainly on two MAPF algorithms that are being upgraded to solve a k-robust path planning problem.

The first of the algorithms in this thesis is the Multi-agent Path Finding via Safe Interval Path Planning (MAPF-SIPP) which is here extended to find a k-robust path for each agent.

The second is MAPF-LNS2, which is a MAPF algorithm based on Large Neighbourhood Search (LNS) and its task is to find a k-robust set of paths for the specified MAPF problem, and additionally this thesis explores different strategies of finding k-robust paths.

A typical example of the problem addressed involves a given number of agents, their starting positions, their goal positions, the required robustness level k, and the task is to find a k-robust path for each agent using this specification. An example of a typical MAPF problem can be seen in Fig. 1.1.

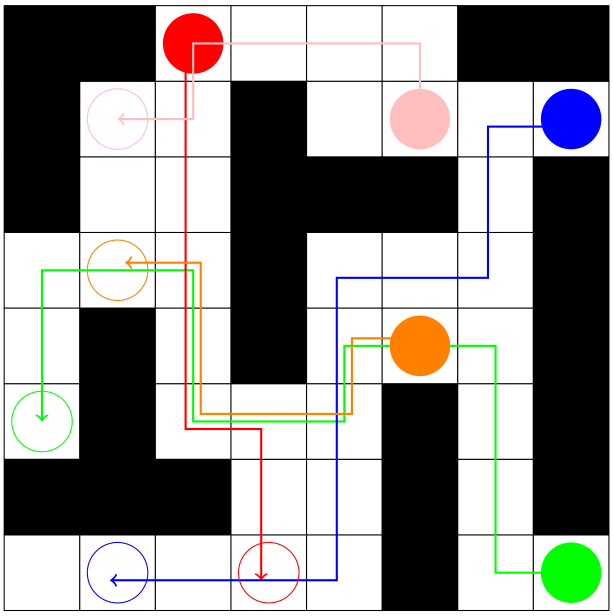


Figure 1.1: Example of map for MAPF

Chapter 2
Related works

The methods for solving Multi-Agent Path-finding (MAPF) are divided into optimal, which guarantee finding the optimal cost C^* solution, bounded suboptimal, which guarantee finding a solution with maximum cost of BC^*, where $B \geq 1$ is a given bound and sub-optimal, which do not guarantee any optimality of the solution. The comprehensive summary of the state-of-the-art MAPF methods can be seen in [19].

Among the state-of-the-art optimal methods, there is the Conflict Based Search (CBS) algorithm [15], based on iteratively resolving conflicts among agents until a collision-free solution is found, or the Increasing Cost Tree Search (ICTS) [16]. The next mentioned optimal method is the M* algorithm [21], which is a MAPF algorithm based on the A* algorithm.

Among suboptimal methods, there are many MAPF methods that use Prioritised Planning that are complete only on well-formed instances [23] such as Hierarchical Cooperative A* (HCA*) [18], or the Prioritised SIPP for Multi-Agent Path-Finding (MAPF-SIPP) [1], which operates similarly to HCA* with the difference that it uses the SIPP algorithm instead of A*. The subsequent suboptimal algorithm is MAPF-LNS [8], which combines a selected MAPF algorithm with the Large-Neighbourhood Search (LNS) [17]. This algorithm was then evolved into the incomplete algorithm MAPF-LNS2 [7], a fusion of Prioritised SIPPS, an algorithm created specifically for this purpose, and Large-Neighbourhood Search.

The example of a complete suboptimal Multi-Agent Path-finding (MAPF) algorithm can be the Push-and-Swap algorithm[10] or its modifications the Parallel Push-and-Swap[14].

Within suboptimal methods with guarantees, examples include the Enhanced CBS algorithm [4], or its modification the Explicit Estimation CBS [9].

The next modifications of the MAPF algorithm are those that expect possible delays among agents. Examples of such modifications include k-robust MAPF algorithms and p-robust MAPF algorithms. k-robust algorithms assume that each agent can be delayed by up to k time steps, and examples of these include k-robust CBS [2]. On the other hand, p-robust MAPF algorithms aim to reduce the probability of collision below a specified threshold $p \in (0, 1)$. Examples of these algorithms include p-robust CBS [3] and its modification, Greedy p-robust CBS [3].

Chapter 3
Problem Formulation

In this section, individual problems are formulated. In Section 3.1 the environment, shared between all problems, is defined, in Sections 3.2 and 3.3 the Single and Multiple Agent Path Finding is described, in Section 3.4 the k-robustness is defined and in the end in Section 3.5 the optimal path for single or multiple agents is defined.

3.1 Environment

The environment ENV is a graph $H(V, E)$ that can be represented as a grid divided into cells (vertices $v \in V$ of the graph), and neighbouring cells are connected in the graph by edges $e \in E$. As it is working with a discrete time, the time is divided into separate time steps. The cell is occupied in the time step t if there is an obstacle or is free (not occupied) otherwise.

There are two types of obstacles, static and dynamic. The static obstacle, which can be imagined as walls, occupies the cell for $t =< 0, \infty >$ and the dynamic obstacle (moving) occupies the cell for $t = (< t_1, t_2 > | t_1 \leq t_2)$ and then moves to another cell depending on the way this obstacle moves.

In each time step, an agent can move in one of the four cardinal directions *(up, down, right, left)* if the cell to which it moves is free or stays still in the current cell. The sequence of visited vertices v_t (cells on the map) in time t by agent a is called a path $p_a = \{v_1, v_2, \ldots, v_n\}$, where n is the length of the path.

3.2 Single Agent Path Finding

Given an environment ENV as previously described, the set of obstacles $O \subseteq ENV$, the agent a with its start node $s \in ENV \setminus O$ and the goal node $g \in ENV \setminus O$, the algorithm's task is to find an optimal path p^* for a from s to g. The optimal path is further described in Section 3.5.

In this thesis, two single agent path-finding algorithms are described: the A* algorithm and the Safe Interval Path Planning algorithm (SIPP).

3.3 MAPF

Given an environment ENV, the set of obstacles $O \subseteq ENV$, the set of agents A, the set of start nodes $S \subseteq ENV \setminus O$ and the set of goal nodes $G \subseteq ENV \setminus O$. The task of the multi-agent path finding algorithm (MAPF) is to find a set of paths P, which consists of one path p_a for each agent $a \in A$ from its start $s_a \in S$ to its goal $g_a \in G$.

In this thesis, two MAPF algorithms are described: Prioritised Planning [5] and MAPF through a Large Neighbourhood Search [8],[7].

3.4 K-robustness

The k-robustness [2] of an algorithm is important to prevent collisions due to unexpected delays of agents.

In the single agent path-finding algorithm, we say that the path of an agent is k-robust if the agent can be delayed by up to k time steps relative to the original plan and does not collide with any obstacle.

In the MAPF algorithm, a k-robust set of paths P is defined as a set of paths where each agent following path $p_i \in P$ can be delayed up to k time steps compared to the original plan and none of the agents collides with either some of the obstacles or another agent.

The number of k-collision of a path $p \in P$ is defined as the maximal number of collisions that can happen to an agent $a \in A$ while following its path p_a if any agent in an environment can be delayed by k time steps.

A path p is more robust than a path p', if the path p has a higher k-robustness than path p' or has the same k-robustness and also has a lower number of $k = (k + 1)$-collisions.

A set of paths P is more robust than a set of paths P' if all the paths $p_i \in P$ have a higher k-robustness than the lowest robust path $p' \in P'$ or have the same k-robustness as p', but have a lower sum of $k + 1$-collisions of all the paths $p_i \in P$ than a set of paths P'.

3.5 Optimal path

An optimal path is a path that minimises a given criterion. An example of optimisation criterion can be a length of the path, a number of k-collisions, or, for example, a sum of costs of all planned agents in the MAPF problem.

3.5.1 Single agent optimal path

If no k-robustness is required and agent collisions with obstacles are not acceptable, the optimal path could be defined as the path with the lowest number of the four direction moves because these agent moves use more fuel than the waiting action.

Given that in this work, all moves are considered to have the same cost, in this study we define the optimal path for single agent path finding algorithms as the path that takes the least amount of time steps and does not have any collisions. This number of time steps is called the cost of the path.

If k-robustness is required, the optimal path for a single agent is defined as the path with the lowest cost, which is k-robust.

3.5.2 Multiple agents optimal path

In the 0-robust Multi-Agent Path Finding, there are two main ways to define an optimal set of paths P using the Flowtime or makespan [11].

The first way is to say that the optimal P for multiple agents is the one that minimises the total sum of costs of each agent's path, which is also known as the Flowtime criterion.

The second way is to find the cost c_h of the path $p_h \in P$ with the highest cost and say that the optimal P is the one with the lowest c_h. This cost of the most expensive path is called the makespan of P.

If k-robustness is required, we again define the optimal set of paths P as either the one with the lowest makespan or the one with the lowest sum of costs, both fulfilling that all moves in P are k-robust.

Chapter 4

Method

This chapter describing the algorithms used is divided into two primary sections. Section 4.1 focuses on single-agent, while Section 4.2 focuses on multi-agent path finding.

4.1 Single-agent path planning

4.1.1 A* algorithm

This Section describes the first of the single-agent path planning algorithms used in this thesis, the A* algorithm [6], which finds a path for one agent from its start to its goal location, as described in the problem formulation section.

The algorithm (1) describes the A* path-finding algorithm.

First, the open list O_l is initialised with the starting vertex s and the closed list C_l is initialised as an empty list [lines 1-2].

Afterwards, the algorithm iterates until it finds the shortest path from s to the goal g, or until O_l becomes empty [line 3].

In each iteration, a vertex with the lowest value of function $f(n)$ (described below) is taken from o_l, marked as a current node c_n and moved to C_l [lines 4-6].

Then if c_n is the goal g, the path from g to s is reconstructed using the parent node of each node. This path is then returned in reverse order as the desired path from s to g [lines 7-8].

If c_n is not g, then each neighbour n of c_n is checked if it is not in c_l [line 10]. If it is, n is skipped because it is already an explored node.

If n has not been explored yet, its temporary score $g(c_n)_t = g(c_n) + cost(n)$ is calculated [line 12], where $cost(n)$ is the cost to get to the neighbour node n and $g(c_n)$ is the sum of costs of an agent from its s to the current location c_n.

If the neighbour n is not in o_l or $g(c_n)_t < g(n)$, then c_n is set as the neighbour's parent, $g(n) = g(c_n)_t$ and $f(n) = g(n) + h(n)$ [lines 14-16], where $h(n)$ is the heuristic function described below. Then n is added to o_l if it is not there yet [lines 17-18].

Then the main loop is run again until $o_l = \emptyset$. If the open list becomes empty without finding a solution, the algorithm finishes without a solution and reports a failure.

Algorithm 1 A* Algorithm

Input: map, start, goal
Output: solution or error

```
1: Initialise open list with start
2: Initialise closed list as empty
3: while open list is not empty do
4:     $c_n \leftarrow$ node in open list with lowest $f$ score
5:     Remove $c_n$ from open list
6:     Add $c_n$ to closed list
7:     if $c_n$ is goal then
8:         return path to $c_n$
9:     for all $n \in$ neighbor nodes of $c_n$ do
10:        if $n$ in closed list then
11:            continue
12:        $g(c_n)_t \leftarrow g(c_n) + cost(n)$
13:        if $n$ not in open list or $g(c_n)_t < g(n)$ then
14:            parent$(n) \leftarrow c_n$
15:            $g(n) \leftarrow g(c_n)_t$
16:            $f(n) \leftarrow g(n) + h(n)$
17:            if $n$ not in open list then
18:                Add $n$ to open list
19: return path_not_found
```

A* algorithm heuristics

This algorithm uses the so-called heuristic function $f(n) = g(n) + h(n)$, to choose the best vertex for expansion (Algorithm 1 [line 4]), which is an estimate of the cost of the path from start to goal through the vertex n to find the shortest p. $g(n)$ is the sum of costs from s to the current location of an agent, and $h(n)$ is the heuristic function, which is an estimate of the remaining cost from the current location n to the goal.

There are many ways to calculate $h(n)$, but it should never overestimate the real cost of reaching the goal from the current position, to find the shortest path.

In this work $h(n)$ is calculated as a Euclidean distance, shown in Fig. 4.1a and is calculated as $h(n) = \sqrt{(g_x - n_x)^2 + (g_y - n_y)^2}$, where g_x, g_y are the x and y coordinates of the goal and n_x, n_y are the current x and y coordinates.

Another commonly used heuristic function is the Manhattan distance, shown in Fig. 4.1b and is calculated on a grid base as
$h(n) = |g_x - n_x| + |g_y - n_y|$.

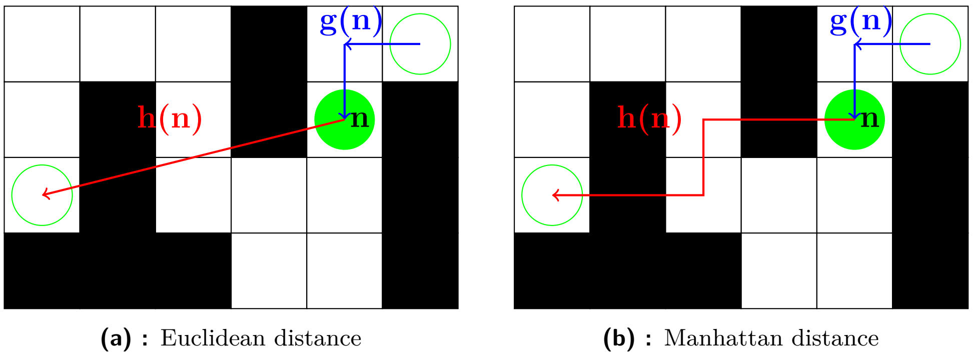


Figure 4.1: Comparison of Euclidean and Manhattan distances.

4.1.2 SIPP

The second single agent path-finding algorithm in this work is Safe Interval Path Planning (SIPP)[13], which is an algorithm used to find a trajectory for an agent in an environment with dynamic obstacles. It does so by constructing the so-called safe and collision intervals for each vertex.

Safe and collision intervals

The safe interval for each cell (x, y) in environment is defined as the continuous time interval (t_1, t_2) in which the cell is not occupied and the collision interval as the continuous time interval (t_1, t_2) in which the cell is occupied.

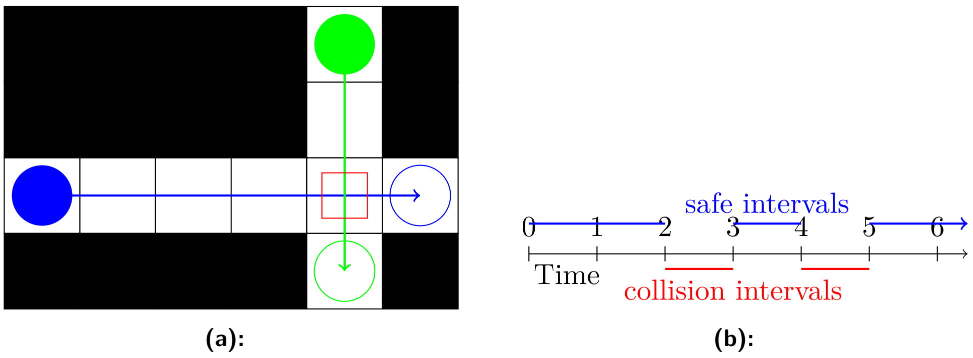


Figure 4.2: Safe and collision intervals for highlighted cell

The example of safe and collision intervals for one cell in a specific environment with 2 agents can be seen in Fig. 4.2. As seen in Fig. 4.2a, the highlighted cell is unoccupied until $t = 2$ (safe interval), then it is occupied for one time step by the green agent (collision interval), then it is again unoccupied until $t = 4$, then occupied by the blue agent for one time step and then unoccupied for the rest of the time.

SIPP algorithm

As seen in Algorithm 2, the SIPP search algorithm is the extension of the A* algorithm [13]. The biggest difference is in how the algorithm finds the neighbour nodes of the current node. The function for obtaining the neighbours of the vertices is described in Algorithm 4 [line 7].

In this algorithm, each vertex v has its list of safe intervals $sis(v) = \{si_0, si_1, \ldots, si_{n-1}\}$ sorted by time, where n is the number of safe intervals. A state S is defined as a tuple of the location on the map (vertex) $S.v$ and the safe interval index $S.id$, where $S.id$ is the order of the safe interval $si(S)$ in $sis(S.v)$. Each S also has its values $f(S) = g(S) + h(S)$ and $g(S)$ as in the A* algorithm. As a heuristic function $h(s)$, the Euclidean distance is used again as in A*.

First, open list o_l and closed list c_l are defined as empty lists. The start state S_s with $g(S_s) = 0$ and $f(S_s) = h(S_s)$, where $h(S_s)$ is the Euclidean distance from start to goal, is then pushed to o_l as a tuple of the start vertex and the index of the safe interval $id(S_s) = 0$.

SIPP then starts to iterate through all states s in o_l and in each iteration takes the one with the lowest $f(s)$ and marks it as a current state c_s. Then if $c_s.v$ is the goal vertex and the end of the safe interval of c_s is equal to ∞, the SIPP returns the path reconstructed by the function $GetPath$, which iterates through the parent states of each state starting from the goal state, until it reaches the start state. In each iteration, it adds the vertex $S.v$ to the path as its t -th element, where t is the time step in which the agent was in the state S. This process can be seen in detail in 3.

If c_s is not the goal, the neighbouring states finding function $GetNeighbors$ described in Algorithm 4 is run.

First, the algorithm to obtain the set of all neighbouring states N finds a set of all possible moves M from the current state c_s to one of the neighbouring vertices. Then iterates through M and for each move $m \in M$ calculates the earliest arrival time s_t to the neighbouring node as $s_t = g(c_s) + cost(m)$ where $g(c_s)$ is the g score of the state s and $cost(m)$ is the cost of m. Subsequently, it computes the latest arrival time e_t to the neighbouring node as the sum of $cost(m)$ and the end time of the safe interval in state c_s in which the agent is currently. In this algorithm, the cost of the move $cost(m) == 1$ as it is worked with agents that perform one move per second.

Subsequently, the algorithm finds all safe intervals $sis = \{si_0, si_1, \ldots, si_{n-1}\}$ for the neighbouring vertex n where the agent is after move m. For each $si_i \in sis$ check if the end of si_i is lower than s_t and the start of si_i is higher than e_t [line 8]. If not, this safe interval is skipped because it cannot be reached from c_s, and the algorithm continues with the next si_i. Otherwise, the time t is calculated as the earliest possible arrival time to the neighbour n from c_s. If t is valid, which means $t \geq s_t$ and $t \leq e_t$, vertex n is pushed with its safe interval number i and arrival time t to all neighbours found N. After all M and their sis are searched, the function returns all neighbours N found.

c_s is then pushed into the closed list, so this state will not be searched again.

This step can be performed without losing any solution, because the states of o_l are taken ascending by their value of f, where $f(c_s) = g(c_s) + h(c_s)$ so this state could not be reached earlier during its safe interval, because the value h is always the same for c_s and the value g is equal to the number of time steps.

The algorithm then iterates through each previously found neighbour state $n \in N$, unless $n \in c_l$, then it continues with the next neighbour $n \in N$ found. For every $n \notin c_l$ the temporary g score $g(n)_{temp} = g(c_s) + cost$ is then calculated, where $g(c_s)$ is the g score of the current node and the cost is the number of time steps needed to reach n from c_s. For example, if the safe interval of state n starts at time $t = 2$ but the agent is in the current state c_s in $t = 0$, it will need to wait one time step at the current state and then at $t = 1$ it will move to n, so the cost will be equal 2.

After $g(n)_{temp}$ is computed, there are two ways of what happens next. If the state n has not been discovered yet, it is added to o_l with $g(n) = g(n)_{temp}$ and $f(n) = g(n)_{temp} + h(n)$. If it has been and if $g(n)_{temp}$ is less than the stored value $g(n)_{old}$ of node n, then $g(n) = g(n)_{temp}$ and $f(n) = f(n)_{old} - g(n)_{old} + g(n)_{temp}$. This state is then again pushed into o_l and the SIPP starts to iterate through o_l again.

Algorithm 2 SIPP

```
 1: initialise $o_l$ and $c_l$
 2: add startState to $o_l$
 3: while $o_l$ is not empty do
 4:     current ← node in $o_l$ with lowest f score
 5:     if current.isSolution() then
 6:         return path
 7:     $N$ ← getNeighbors(current)
 8:     $c_l$.insert(current)
 9:     for each $n$ in $N$ do
10:         if $n$ not in $c_l$ then
11:             $g_{temp} = g(\text{current}) + n.\text{cost}$
12:             if not yet discovered node then
13:                 $f(n) = g_{temp} + h(n)$
14:                 $g(n) = g_{temp}$
15:                 $o_l$.increase($n$)
16:             else
17:                 if $g_{temp} < g(n).\text{old}$ then
18:                     $g(n) = g_{temp}$
19:                     $f(n) -= g(n).\text{old} - g_{temp}$
20:                     $o_l$.increase(n)
21: return false
```

Algorithm 3 GetPath

```
1:  while $curr.parent \neq$ nullptr do
2:      $prev \leftarrow curr.parent$
3:      $t \leftarrow prev.timestep + 1$
4:      while $t < curr.timestep$ do
5:          $p[t] \leftarrow prev.location$
6:          $t \leftarrow t + 1$
7:      $p[curr.timestep] \leftarrow curr.location$
8:      $curr \leftarrow prev$
9:  $p[0] \leftarrow curr.location$
10: return $p$
```

Algorithm 4 getNeighbors($s, neighbors$)

```
1:  $motions \leftarrow$ allNeighbors($s$.state)
2:  for each $m$ in $motions$ do
3:      $start\_t \leftarrow g(s) + m.cost$
4:      $end\_t \leftarrow$ safeIntervals($s$.state)[$s$.interval].end + $m.cost$
5:      $sis \leftarrow$ safeIntervals($m$.state)
6:      for $i \leftarrow 0$ to size($sis$) - 1 do
7:          $si \leftarrow sis[i]$
8:          if $si$.start $> end\_t$ or $si$.end $< start\_t$ then
9:              continue
10:         $t \leftarrow$ Earliest possible arrival time to $m.state$ in $si$
11:         if $t$ is valid then
12:             $neighbors$.emplace_back($m.state$ with $si$ and $t$)
13: return $neighbors$
```

4.2 Multi-Agent Path Finding

4.2.1 Prioritised Planning

Prioritised Planning (PP)[5] is a Multi-Agent Path Finding (MAPF) method that solves the problem by decomposing it into a sequence of simpler, prioritised subproblems, each of which is solved independently.

Each of the agents is assigned a priority based on certain criteria. The criterion could be, for example, their importance. In algorithms mentioned in this thesis, a random priority is used, as none of the agents is considered more important than the others. Subsequently, all agent paths are planned sequentially by a selected single agent path-finding algorithm, which takes into account paths planned for previous agents to avoid collisions.

The major benefit of PP lies in its efficiency, effectively coordinating multiple agents by decomposing the complex planning problem into sequential prioritised subproblems, thus reducing the runtime of the algorithm.

4.2.2 MAPF-SIPP

The first of the MAPF algorithms in this thesis is Multi-Agent Path Finding via Safe Interval Path Planning (MAPF-SIPP)[1] which is a MAPF method based on Prioritised Planning using SIPP as its single agent solver.

As seen in Algorithm (1), $CollInts = \{cints_0, cints_1, \dots cints_n\}$ is created as a list of collision intervals $cints_i = \{cint_0, cint_1, \dots cint_m\}$ for each vertex $v_i \in V$ of the map, where n is the total number of vertices and m is the number of collision intervals of vertex v_i.

Then, all static obstacles of the vertex $v_i \in V$ are added to $CollInts$ so that $cints_i = \{(0, \infty)\}$. This prevents any agent from visiting this vertex at any time.

A random order of agents is chosen in which their trajectory will be planned. The algorithm then starts iterating through all agents a_i, and in each iteration finds the path using the SIPP algorithm for agent a_i.

If SIPP returns a valid path p, MAPF-SIPP adds the path to the set of paths found and updates $CollInts$. $CollInts$ is updated by adding an additional interval (t_1, t_2) to each $cints_v \subset CollInts$, where (t_1, t_2) is the time interval when the vertex v was visited by the path p and $cints_v$ is the list of collision intervals for vertex v. If SIPP does not find a valid path, MAPF-SIPP returns that it was not able to plan a path for all the agents.

After successful planning of the paths for each of the agents, the algorithm returns the set of paths found.

The most significant advantage of this algorithm is its computation time efficiency. As it is planning the path for each agent separately, the time requirement is linear with increasing number of agents.

This algorithm is complete only on well-formed instances [23], which means that on instances that are not well-formed, it is not guaranteed to find a solution even if one exists.

For example, in Fig. 4.3 it can be seen that this problem has an easy solution. All the blue agent needs to do is wait until the green agent leaves the goal of the blue agent and then move one step toward the goal. MAPF-SIPP will not find a solution for this instance if the blue agent has higher priority. The green agent will not be able to reach the goal because the blue agent, which was planned earlier, will block its path from the start to the goal.

This can be prevented by using another order in which agents will be planned, but it adds a time complexity as every time the algorithm does not find a solution, the agents order needs to be changed.

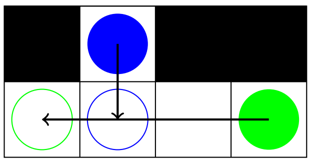


Figure 4.3: MAPF-SIPP drawback

Algorithm 5 MAPF-SIPP algorithm

Input: Agents, map
Output: Solution or Error

```
1: $CollInts \leftarrow$ static obstacles
2: for all $agent \in agents$ do
3:     $path \leftarrow sipp.\text{search}(agent, CollInts)$
4:     if $path$ then
5:         $AllPaths \leftarrow AllPaths \cup path$
6:         update $CollInts$ by path
7:     else
8:         return $Error$
9: return $AllPaths$
```

4.2.3 K-robust MAPF-SIPP

In order to find a set of k-robust solutions, where each agent can be delayed by up to k time steps, MAPF-SIPP must be modified.

The biggest difference from the original MAPF-SIPP is how the algorithm updates its list of collision intervals $CollInts$ [Algorithm 5, line 6].

In the original MAPF-SIPP algorithm, the set of all collision intervals $CollInts$ was updated after finding the path p by adding an additional interval (t_1, t_2) to each $cints_v$, where (t_1, t_2) was the time interval when the vertex v was visited by the path p.

In the k-robust MAPF-SIPP the $CollInts$ is updated by adding an interval $(t_1 - k, t_2 + k)$ to each $cints_v$ instead of adding the interval (t_1, t_2). By this approach, each vertex v of the environment has at least k time-step gap between being visited by different agents, which means that any of the agents can be delayed up to k time steps without colliding.

Except for updating the collision intervals, the k-robust MAPF-SIPP works the same way as the original one.

In Fig. 4.4 an example of a 1-robust plan can be seen. The green, higher-priority agent, which was planned earlier than the blue one, is heading left, and the blue agent heads down. The green agent currently occupies the cell (vertex) in which it is located, but it also virtually occupies the cell, where it was one time step earlier (right) and the cell in which it will be in the next time step (left). In the next time step, the green agent moves left and will occupy the cell that is one move left of the cell that is currently occupying and will virtually occupy the cells that are one move left from the cells that is currently virtually occupying. The agents can virtually occupy the same cell in the same time step, but cannot be in the cell that is virtually occupied by any other agent. Virtually occupying the cell, in which the agent will be one time step later, secures that none of the other agents can be there in the current time step and this agent will not k-collide with them.

The lower priority blue agent cannot move down in the next time step because it would virtually collide with the green agent, which means that the plan would not be 1-robust. This means that the blue agent needs to wait 3 time steps in its current location, until the cell where it wants to move becomes unoccupied.

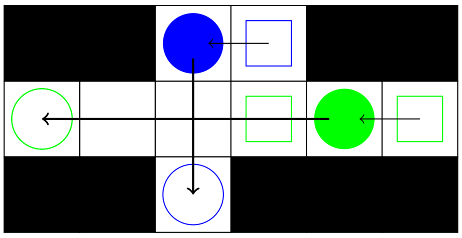


Figure 4.4: Example of 1-robust plan

4.2.4 MAPF-LNS2

MAPF-LNS2 [7] is an algorithm that combines some MAPF algorithm with a Large Neighbourhood Search (LNS) [17] to provide the best solution to the MAPF problem. In this work, PP using the SIPPS algorithm is used as the MAPF algorithm.

First, MAPF-LNS2 finds an initial solution (set of paths) that can contain collisions using the chosen MAPF algorithm (PP using SIPPS) and then tries to obtain a collision-free set of paths from the initial solution using LNS2, which is described below. Then, if LNS2 finds a collision-free solution, the total sum of costs of this set of paths is optimised using LNS [8].

In the following two sections, the LNS2 and then LNS methods will be described in the order in which MAPF-LNS2 uses them.

LNS2

The LNS is a metaheuristic optimisation technique that takes a given solution and tries to improve it. In this case, the solution is a set of paths. The algorithm tries to improve the solution by repeating a cycle of two phases called *destroy* and *repair*. In the *destroy* phase, the algorithm takes the given paths for some number of agents, which are named *neighbourhood* in this algorithm, and deletes them. The *neighbourhood* is chosen by the *destroy* operators, which are described in the Neighbourhood selection section. After destroying the paths of the agents chosen previously, the *repair* phase starts. The chosen MAPF algorithm in this phase replans the deleted paths of the agents in the *neighbourhood* and adds them to the unchanged paths of the agents that were not in the selected *neighbourhood*.

If the solution after this process is better (has fewer collisions), the LNS2 accepts those changes. Otherwise, the changes are rejected, and the paths of all agents remain unchanged in this cycle.

The LNS2 then continues with the next *destroy* and then *repair* cycle. This process is repeated until it is interrupted by the chosen stopping criteria (time limit, no collision, etc.).

Neighbourhood selection. In order for LNS2 to work efficiently, properly chosen *neighbourhood* selection methods are essential. In this algorithm, there are three approaches and a combination of them to find the best *neighbourhood*.

The best *neighbourhood* A_s with a size of N agents is searched using the plan P and the collision graph $G_c = (V_c, E_c)$, where V_c are agents of P and E_c are tuples of agents whose trajectories collide with each other.

Collision-Based Neighbourhood. The first way to create A_s is to take a random vertex $v \in V_c$ and find a largest connected graph $G'_c = (V'_c, E'_c)$ containing v and $G'_c \subseteq G_c$.

If $|V'_c| \leq N$ we add all agents from V'_c to A_s and then continue to add other random agents from P to A_s until $|A_s| = N$ using the random walk strategy.

If $|V'_c| > N$ we add N random vertices from V'_c to A_s.

Failure-Based Neighbourhoods. The second *neighbourhood* search algorithm is based on the idea of why some agents cannot find a non-collision way from start to goal. The two main reasons may be either that their way to the goal is blocked by some other agents or that the agent does not have enough time to move from the starting node before other agents visit the starting node, because all the neighbouring nodes of the starting node are occupied by some other agents.

To solve these two problems, the agent a that has the most collisions in its plan is chosen and added to A_s. Then a list of agents A^s and a list of agents A^g are created. All agents who visit the goal node of a are added to A^g and all agents who visit the start node of a are added to A^s.

There are three possible options:

If $|A^s \cup A^g| = 0$ there is no collision path existing and the algorithm just returns A_s.

The next option is $|A^s \cup A^g| \leq N$, then $|A^s \cup A^g|$ is added to A_s and random agents whose goal nodes are visited by some agent of A_s are added until $|A_s| = N$ or there are no more agents that visited the goal node of some agent of A_s.

The last option is $|A^s \cup A^g| > N$. If this happens, the agents are added to A_s following the rules. If $|A^s| = 0$ $N - 1$ random agents are added from A^g to A_s. Otherwise, if $|A^g| > N - 2$ the agent from A^s that visits the start node of a as the first is added to A_s and then $N - 2$ random agents is added from A^g to A_s. Otherwise, all agents are added from A^g to A_s, and then $N - 1 - |A^g|$ agents from A^s that visit the start node of a as the first are added to A_s.

Random Neighbourhoods. The third approach to creating *neighbourhoods* is to choose agents randomly. Each agent of V_c has a proportional chance counted as 1+ the number of agents with whom they collide, to be added to A_s.

ALNS2. Adaptive LNS2 (ALNS2) is a combination of the previous three *neighbourhood* selection methods $M = \{m_1, m_2, m_3\}$. In each iteration, the ALNS2 chooses some of the methods m_i based on their weight function value w_i of the success rate in reducing the number of colliding pairs of agents in the plan P. Each of the methods has a probability equal to $w_i / \sum_j w_j$ to be chosen as the *neighbourhood* selection method. The value w_i is initially set at 1 and then calculated as $w_i = \lambda \max\{0, cp_{before} - cp_{after}\} + w_i(1 - \lambda)$, where cp_{before} and cp_{after} are the number of colliding pairs of agents in the plan P before and after iteration and $\lambda \in (0, 1)$ is a parameter that determines how fast the ALNS reacts to the success of the chosen algorithm. In this algorithm $\lambda = 0.1$.

LNS

The LNS[8] operates similarly to LNS2, with the difference that LNS focusses optimising the sum of costs of a collision-free solution, while LNS2 is designed to minimise the number of colliding pairs. The next difference is in the used *neighbourhood* selection methods.

Neighbourhood selection. In the LNS method, there are again three *neighbourhood* selection methods as in the LNS2. The difference is in the way these methods find *neighbourhood* A_s.

Agent-Based Neighbourhood. In the first method, an agent a' that is not in *tabu_list* and whose path is the most delayed compared to the shortest path possible for this agent is chosen and inserted in A_s. This agent is then added to *tabu_list*. From a randomly selected vertex p_t visited by the agent a' path $P_{a'} = p_0, p_1, \dots p_n$ in the timestep t a *restricted random walk* is performed. This walk identifies all the vertices v along the path that are part of a shorter path than the current path of the agent a'. When searching for v, no consideration is given to the paths of other agents. All agents with potential collisions in v in time $t + m$, where m is the number of moves performed from vertex p_t are then being added to A_s until $|A_s| \neq N$. If not all agents with potential collisions are added and $|A_s| < N$, another agent is selected from those added to A_s, and a *restricted random walk* is performed for some random point on their path. After adding all agents to *tabu_list*, *tabu_list* is cleared.

Map Based Neighbourhood. The second method used to select the right *neighbourhood* focusses on finding intersecting vertices, as rearranging the order in which agents visit this vertex could improve the solution. Initially, all vertices visited more than twice are identified, and from this set, a random vertex is selected and placed into the queue.

Subsequently, a vertex x is dequeued. A random time step t is chosen within the time frame when agents visit the vertex x. Subsequently, agents are added to A_s by iteratively exploring which agents visit the vertex x within a certain number of time steps before or after the time step t until

N agents are collected in A_s or all time steps were explored. Following this, all neighbouring vertices of x are pushed into the queue if not previously explored. If $|A_s| < N$, the algorithm initiates a new iteration using the next vertex x dequeued from the queue.

Random Neighbourhood. In the third method, N random agents are pushed into A_s

ALNS. Adaptive LNS is a combination of three previously mentioned *neighbourhood* selection methods and operates similarly to ALNS2, with the difference that it does not consider the number of collisions but focusses on improving the sum of costs. Like in ALNS2, at the beginning, each *neighbourhood* selection method m_i, is assigned a weight w_i, initially set to 1. The weight is then calculated as $w_i = \lambda \max\{0, soc_{before} - soc_{after}\} + w_i(1-\lambda)$, where soc_{before} and soc_{after} are the sum of costs before and after re-planning the *neighbourhood* chosen by the method m_i. The probability of selecting a specific method m_i with weight w_i is again equal to $w_i / \sum_j w_j$.

SIPPS

Safe Interval Path Planning with Soft Constraints (SIPPS) [7] is a modified SIPP algorithm. The main difference between them is that if SIPPS does not find a path without collisions, it returns a path with the lowest number of collisions found. In the SIPP algorithm, the collisions are inadmissible, which means that if the algorithm does not find a path with no collisions, it returns no path.

In MAPF-LNS2, a prioritised SIPPS is used as an initial solver and also as a repairing algorithm.

This algorithm uses a data structure where each node n consists of vertex $v(n)$, the index $id(n)$ of its safe interval in the table of all safe intervals $T[v][id]$ and its safe interval $si(n) =< low, high >$, where the low value of $si(n)$ is also called the earliest arrival time of n and $si(n) \subseteq T[v(n)][id(n)]$.

Each node also has its $g(n), h(n)$ and $f(n)$ score, which are standard heuristic function elements ($h(n)$ is again the Euclidean distance), but also has its $c(n)$ score, which is a number of collisions made to get to this node from the start node. Each node also has its parent node and the *is_goal* flag, which is set by default to false.

The process of finding the path by SIPPS can be seen in Algorithm 6 and is described below.

Initially, a safe interval table $T[v][id]$ is created as an array of all safe intervals for each vertex v. This ensures that the path planned by this algorithm will not collide with any obstacle or will have the lowest possible number of collisions if a no-collision path does not exist. *Open_list* and *Closed_list* are initialised. Subsequently, $g(s_n), h(s_n), f(s_n)$and $c(s_n)$ of the start node s_n is computed and s_n is added to *Open_list* [lines 1-4].

The nodes in *Open_list* are sorted ascending by its value $c(n)$ in order to find a path with the lowest possible number of collisions. If more than one node has the same number of collisions, those same $c(n)$ value nodes are then sorted ascending by their $f(n)$.

The algorithm is executed by iterating through the $Open_list$ until a solution is found or the $Open_list$ becomes empty [line 5].

In each iteration, the first node of *Open_list* (the one with the lowest $c(n)$) is taken and marked as the node currently being explored c_n [line 6].

First, the flag *is_goal*, which is set by default to false, is checked for true. If it is, c_n is the goal, and the path from start to goal is reconstructed by the function $UpdatePath$, that iterates through all the nodes visited by the path and adds their location to the path [lines 7-8].

If *is_goal* is false, it is checked whether this node is the goal node. If so, the future collision score c_f for this node is calculated as a number of collisions with obstacles while standing at this node [lines 9-10]. If $c_f = 0$ the path is returned by $UpdatePath$ function because it is the path with the lowest number of collisions so there is no need to keep finding any other path [lines 11-12].

If there are future collisions at this node, the $c(c_n)$ score for this node is updated as $c(c_n) = c(c_n) + c_f$ and the *is_goal* flag of c_n is set to true because it is the goal node, but the path to this node might not be the lowest collision path. This node is then checked by $DominanceCheck$ function. If the output of this function is true, c_n is pushed to *Open_list* [lines 13-16].

Then, if $is\_goal = false$, c_n is added to *closed_list* [lines 17-18].

All the neighbours of c_n are then found by Get_Neighbours(n) function which is described below and a new node is made from each of them. Each of the new nodes is then checked by $DominanceCheck$ function and eventually pushed to *Open_list* [lines 19-25].

Getting Neighbours. Initially Get_Neighbours(n) algorithm finds a list of all reachable vertex v index id pairs from node n, where id is the index in the safe interval table of the reachable safe interval from node n. Then a list of successors is initialised as an empty list [lines 1-2].

The algorithm then iterates through all pairs found. In each iteration, it first finds a safe interval $< t_1, t_2 >$ in the table T of all safe intervals [line 4]. Then it finds the earliest arrival time $\in < t_1, t_2 >$ to the neighbouring node and marks it as t_1. If t_1 exists, the algorithm finds the earliest collision-free arrival time t_1' otherwise it starts the next iteration [lines 5-7].

In the next step, the algorithm selects one of three ways based on the values of variables t_1 and t_1', determining the following outcomes in each case. If $t_1 = t_1'$ the success set $Succ$ increases with vertex v, an interval $< t_1, t_2 >$ and a flag $WithColl$ set to false [line 10]. If t_1' does not exist, the vertex v, an interval $< t_1, t_2 >$ is pushed again into $Succ$, but this time the $WithColl$ flag is set to true because there is always a collision during this interval [line 12]. Otherwise, the interval needs to be separated into two intervals $< t_1, t_1' >$ and $< t_1', t_2 >$, where the first interval has a collision and the second does not. These two intervals are then pushed with the $WithColl$ flag set as mentioned above [lines 14-15].

Algorithm 6 SIPPS Algorithm

```
1: create safe interval table
2: initialize Open_list and Closed_list
3: compute $g$-, $h$-, $f$-, and $c$-values of $Start\_node$
4: $Open\_list \leftarrow \{Start\_node\}$
5: while $Open\_list$ is not empty do
6:     $current \leftarrow$ lowest_c_value($Open\_list$)
7:     if $current$.is_goal then
8:         return UpdatePath($current$)
9:     if $current == goal$ then
10:        $c_f \leftarrow$ GetFutureCol($current$)
11:        if $c_f = 0$ then
12:            return UpdatePath($current$)
13:        $c(current) \leftarrow c(current) + c_f$
14:        $current.is\_goal \leftarrow$ True
15:        if Dominance_check($current$) then
16:            $Open\_list \leftarrow Open\_list \cup current$
17:    if not $current$.is_goal then
18:        $closed\_list$.append($current$)
19:    $Neighbours \leftarrow$ Get_Neighbours($current$)
20:    for all $Neighbour \in Neighbours$ do
21:        $NewNode$ = CreateNode($Neighbour$)
22:        compute $g, h, f,$ of $NewNode$
23:        $c(NewNode) = c(current) + Neighbour$ with collision?
24:        if Dominance_check($NewNode$) then
25:            $Open\_list \leftarrow Open\_list \cup NewNode$
26: return -1
```

The list of all successors *Succ* is then returned after iterating through all neighbours of n [line 16].

Checking the dominance of the node. In order for SIPPS to work efficiently, it is essential to check if node n is not dominated by any other node in *Open_list* before pushing it there.

Initially, the algorithm finds a set of all nodes found as a conjunction of *Open_list* and *Closed_list*. Then it finds a set of nodes identical to n, which means a set of nodes with the same vertex, the same safe interval id, and the same Boolean value of *is_goal* [lines 1-2].

Then iterates through all the similar nodes s_n found. In each iteration, it first checks if the number of collisions $c(s_n)$ is lower than or equal to $c(n)$ and the earliest arrival time at s_n is lower than or equal to the earliest arrival time to n [line 4]. If so, the algorithm returns false because the old node s_n dominates the node n. Then it checks if the node n dominates some of the nodes of s_n, which means that $c(s_n) \geq c(n)$ and the earliest arrival time to s_n is $\geq$ then to n [line 6]. If both are true, the algorithm removes s_n

Algorithm 7 Get_Neighbours(n)

```
1: $N_n \leftarrow$ all reachable $(v, id)$ pairs from $n$
2: $Succ = \emptyset$
3: for all $(v, id) \in N_n$ do
4:     $< t_1, t_2 >= T[v][id]$
5:     $t_1 \leftarrow$ earliest arrival time
6:     if $t_1$ not exist then
7:         continue
8:     $t'_1 \leftarrow$ earliest no collision arrival time
9:     if $t_1 = t'_1$ then
10:        $Succ \leftarrow Succ \cup \{v, t_1, t_2, WithColl = False\}$
11:    else if $t'_1$ not exist then
12:        $Succ \leftarrow Succ \cup \{v, t_1, t_2, WithColl = True\}$
13:    else
14:        $Succ \leftarrow Succ \cup \{v, t_1, t'_1, WithColl = True\}$
15:        $Succ \leftarrow Succ \cup \{v, t'_1, t_2, WithColl = False\}$
16: return $Succ$
```

from *Open_list* and *Closed_list* and returns true. Finally, the algorithm determines whether the safe intervals of n and s_n overlap [line 9]. If so, the algorithm must create two disjoint intervals from these two overlapping intervals. This is done by comparing the start of the safe interval of node n with the start of the safe interval of node s_n [line 10]. If $si(n).low < si(s_n).low$ the $si(n).high$ is set to the value of $si(s_n).low$ otherwise, the $si(n).low$ is set to the value of $si(s_n).high$. After iterating through all similar nodes without returning, the algorithm returns true.

Algorithm 8 Dominance_check(n)

```
1: $All\_nodes = Open\_list \cup Closed\_list$
2: $N \leftarrow$ GetIdenticalNodes($n, All\_nodes$)
3: for all $s_n \in N$ do
4:     if $c(s_n) \leq c(n)$ & $si(s_n).low \leq si(n).low$ then
5:         return $False$
6:     else if $c(s_n) \geq c(n)$ & $si(s_n).low \geq si(n).low$ then
7:         delete $s_n$ from $Open\_list$ and $Closed\_list$
8:         return $True$
9:     else if $si(n).low < si(s_n).high$ & $si(s_n).low < si(n).high$ then
10:        if $si(n).low < si(s_n).low$ then
11:            $si(n).high = si(s_n).low$
12:        else
13:            $si(n).low = si(s_n).high$
14: return $True$
```

Algorithm 9 UpdatePath($curr$)

```
1: while $curr.parent \neq$ nullptr do
2:     $prev \leftarrow curr.parent$
3:     $t \leftarrow prev.timestep + 1$
4:     while $t < curr.timestep$ do
5:         $path[t] \leftarrow prev.location$
6:         $t \leftarrow t + 1$
7:     $path[curr.timestep] \leftarrow curr.location$
8:     $curr \leftarrow prev$
9: $path[0] \leftarrow curr.location$
10: return $path$
```

4.2.5 K-robust MAPF-LNS2

As this work focusses on extending existing MAPF algorithms into k-robust MAPF methods, this section introduces a second k-robust algorithm, namely the k-robust MAPF-LNS2.

The main idea of this algorithm, which involves planning the initial paths for all agents and then iteratively improving them using k-robust LNS, remains the same. The main difference is that the original purpose of LNS was to reduce the number of collisions and the sum of the agents' costs to a minimum. In K-robust MAPF-LNS2, the purpose of LNS is to iteratively improve the k-robustness of the paths, not the number of collisions (0-robustness) as in the original algorithm. The LNS in the k-robust algorithm also reduces the sum of costs, but the priority there is to find a k-robust set of paths. The K-robust MAPF-LNS2 algorithm was created as an extension of the MAPF-LNS2 algorithm. The biggest difference is that this algorithm does not use LNS2, but only uses a modified LNS.

K-robust LNS

K-robust LNS works in the same way as the original LNS, which means that k-robust LNS improves the initial solution as mentioned above. The improvement is again caused by repeating a cycle of two phases (*destroy*, *repair*).

In the *destroy* phase, again the *destroy* operators (described below) choose a set of paths called *neighbourhood* from the plan P and destroy these paths. These paths are then re-planned by k-robust SIPPS. The new paths are then accepted if the new plan P' is more robust than P or is the same robust as P and also has a better sum of costs of all $p \in P'$ than the previous plan.

In the *repair* phase, the paths chosen by the *destroy* operators are again re-planned as in the original LNS, but this time the k-robust SIPPS (described below) is used as the re-planning algorithm.

K-robust SIPPS

As mentioned above, k-robust SIPPS is a modified SIPPS algorithm used in the *k*-robust MAPF-LNS2 algorithm to plan the initial solution and replan the *neighbourhood* chosen by the *destroy* operators.

Algorithm 10 K-robust SIPPS Algorithm

```
1:  create safe interval table
2:  initialize Open_list and Closed_list
3:  compute $g$-, $h$-, $f$-, and $c$-values of $Start\_node$
4:  $Open\_list \leftarrow \{Start\_node\}$
5:  while $Open\_list$ is not empty do
6:      $current \leftarrow$ lowest_c_value($Open\_list$)
7:      if $current$.is_goal then
8:          return UpdatePath($current$)
9:      if $current == goal$ then
10:         $c_f \leftarrow$ GetFutureColVector($current$)
11:         if $c_f == 0$ then
12:             return UpdatePath($current$)
13:         $c(current) \leftarrow c(current) + c_f$
14:         $current.is\_goal \leftarrow$ True
15:         if Dominance_check($current$) then
16:             $Open\_list \leftarrow Open\_list \cup current$
17:     if not $current$.is_goal then
18:         $closed\_list$.append($current$)
19:     $Neighbours \leftarrow$ GetKrobustNeighbours($current$)
20:     for all $Neighbour \in Neighbours$ do
21:         $NewNode$ = CreateNode($Neighbour$)
22:         compute $g, h, f$, of $NewNode$
23:         $c(neighbour)[0] \leftarrow Neighbour$ with collision?
24:         for $i \leftarrow 1$ to $k$ do
25:             if $Neighbour.ColId == i$ then
26:                 $c(neighbour)[i] \leftarrow 1$
27:             else
28:                 $c(neighbour)[i] \leftarrow 0$
29:         $c(NewNode) = c(current) + c(neighbour)$
30:         if Dominance_check($NewNode$) then
31:             $Open\_list \leftarrow Open\_list \cup NewNode$
32: return -1
```

The biggest difference between the original SIPPS and the k-robust one is that in the SIPPS algorithm, each node n of a path p has a value $c(n)$, which is the number of collisions made to reach n. In the modified algorithm, $c'(n)$ is not a number, but is a vector $c'(n) = [c_o, c_1, \dots c_k]$, where k is the desired robustness of P and $c_i | i \in < 0, k >$ is the number of $k = i$ k-collisions made to reach n.

The k-robust SIPPS pseudo-algorithm can be seen in Algorithm 10. As can be seen, the only difference between this algorithm and the original is how it works with the value c of the nodes.

The first difference is that if the node currently being explored by the algorithm is the goal [lines 9-12], the algorithm needs to find not only a number of future collisions but also a number of k-collisions with agents that were in this vertex earlier.

The next difference is in the way the algorithm finds the neighbours and then works with them [lines 19-28]. The algorithm for finding neighbours is described below.

After finding the neighbours, the algorithm computes a $g(), h(), f()$ score of the neighbour and its collision vector [lines 22-27] and adds it to the collision vector of the current node. The rest of the algorithm remains the same as in the original SIPPS.

Getting k-robust Neighbours. In order for k-robust SIPPS to find a k-robust set of paths P, the function to obtain neighbours of a node needed to be upgraded. The pseudo-algorithm of the function can be seen in algorithms 11 and 12.

As can be seen in algorithm 11 in lines 10, 12, 14, the biggest difference is that the function to obtain neighbours uses another function 12, which instead of returning a node with safe interval (t_{start}, t_{end}) as in the standard SIPPS divides this interval into smaller intervals and adds to them an additional penalty according to the k-robustness of the interval.

Algorithm 11 GetK-robustNeighbours(n)

```
1: $N_n \leftarrow$ all reachable $(v, id)$ pairs from $n$
2: $Succ = \emptyset$
3: for all $(v, id) \in N_n$ do
4:     $< t_1, t_2 >= T[v][id]$
5:     $t_{earl} \leftarrow$ earliest arrival time
6:     if $t_{earl}$ not exist then
7:         continue
8:     $t'_1 \leftarrow$ earliest no collision arrival time
9:     if $t_{earl} == t'_1$ then
10:        $Succ \leftarrow Succ \cup GetSucc(t_1, t_2, t'_1, t_{earl}, v, 0)$
11:    else if $t'_1$ not exist then
12:        $Succ \leftarrow Succ \cup GetSucc(t_1, t_2, t'_1, t_{earl}, v, 1)$
13:    else
14:        $Succ \leftarrow Succ \cup GetSucc(t_1, t_2, t'_1, t_{earl}, v, 2)$
15: return $Succ$
```

The example of dividing the interval can be seen in Fig. 4.5.

As can be seen in the figure, the neighbouring vertex v has two safe intervals $((0, 5)$ and $(6, 10))$ that are reachable from the current node without any collision. The standard SIPPS would make two neighbouring nodes from these

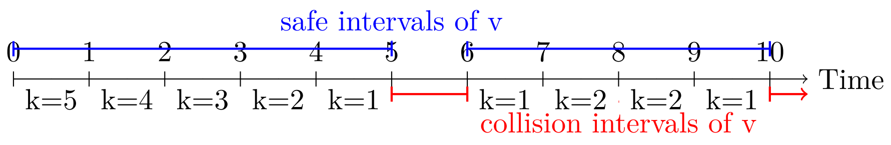


Figure 4.5: Safe and collision intervals

two intervals and return them. In the k-robust SIPPS these two intervals need to be divided according to the k-robustness of the algorithm.

For example, if $k = 2$, the k-robust SIPPS would divide these two intervals into 6 and add the following penalty to them. The first would be $(0, 3)$ and have a collision index $ColId = -1$ as this interval does not have any $k \leq 2$ collision. The next intervals would be $(3, 4), (7, 9)$ and their collision index $ColId = 2$, which means that the agent in this interval would collide if it was delayed by 2 time steps or the agent who visits v in time step 5 was delayed by 2 time steps. The last intervals would be $(4, 5), (6, 7), (9, 10)$ and their collision index $ColId = 1$ as they could collide if some agent was delayed by 1 time step.

All these intervals would create a new node with its interval and its $CollId$.

K-robust Repair operators

As mentioned above, as *repair* operators in k-robust MAPF-LNS2, k-robust SIPPS is used. However, there are 3 modifications of the k-robust SIPPS tested as *repair* operators. These are:

- Standard
- Reversed
- Binary

In the Standard repair method, the nodes in $OpenList$ are sorted by their collision vectors. This means that they are sorted by the number of 0-collisions, then if the number is equal by 1-collisions and so on up to k-collisions. As the last sorting parameters, the values f and then g are used.

In the Reversed method, the nodes in $OpenList$ are initially again sorted by the number of 0-collisions, but then by the reversed collision vector, meaning that after 0-collisions the next sorting parameter is the number of k-collisions, then $(k - 1)$-collisions up to 1-collisions. As the last sorting parameters, the values f and then g are used again.

In the last *repair* method (Binary), the nodes are sorted in the same way as in the Standard method, but this time the vector of collisions is not a vector of numbers but a vector of Booleans. This means that if the node has any i-collisions, its collision vector will have number 1 (true) as its i-th element or 0 (false) otherwise.

K-robust Neighbourhood selection

As in the original (0-robust) LNS, so in the k-robust LNS the significance of choosing the right *neighbourhood* persists in the context of finding the desired solution in the shortest run-time. To find the ideal *neighbourhood*, three new *neighbourhood* selection algorithms were created from the original Agent-Based Neighbourhood selection method. The remaining two methods (Map-Based Neighbourhood and Random Neighbourhood) remained unchanged.

In the initial Agent-Based Neighbourhood selection method, agents for *neighbourhood* A_s were selected by first including the most delayed agent a' in A_s. Subsequently, additional agents were incorporated into A_s using a *restricted random walk* strategy starting from the random vertex visited by a'.

In the k-robust LNS, the choice of the most delayed agent was replaced by three other methods:

- Random
- Most serious
- Least serious

In the Random method, an agent is randomly selected from agents with a k-collision, where $k \in < 0, \text{desired robustness} >$. In the Most Serious method, the selection is made based on an agent with the highest number of most serious collisions, while in the Least Serious method, the choice is an agent with the highest number of least serious collisions. As the most serious k-collisions, those with the lowest k are considered, while the least serious are those with the highest k.

K-robust ALNS

The k-robust ALNS is divided into two parts: the first part focusses on eliminating k-collisions, while the second part operates as a standard ALNS, focussing on minimising the total sum of costs.

The biggest difference from the original ALNS is that adaptive k-robust LNS is a combination of the five previously mentioned *neighbourhood* selection methods with the Standard and Binary *repair* method creating the total number of ten $destroy - repair$ pairs. In the standard ALNS there was just one *repair* operator and therefore ALNS only choose the *destroy* operator.

At first, each $destroy - repair$ pair i, is assigned a weight w_i, initially set to 1, which is then calculated as $w_i = \lambda \max\{0, \text{improvement}\} + w_i(1-\lambda)$, where *improvement* is calculated as the number of reduced k-collisions divided by the size of *neighbourhood*, where k is the index of the lowest k-robustness improved in this iteration. The probability of selecting a specific pair i with weight w_i is again equal to $w_i / \sum_j w_j$.

After reducing all k-collisions, the weight w_i is set again to 1 and is then calculated as in the standard ALNS.

Algorithm 12 GetSucc($t_1, t_2, t'_1, t_{earl}, v, type$)

```
for $i \leftarrow 1$ to $k+1$ do
    $pen_s \leftarrow i$
    $pen_e \leftarrow i$
    if $t_1 = 0$ then
        $pen_s \leftarrow 0$
    if $t_1 + i - 1 \geq t_{earl}$ then
        if type $= 0$ then
            $Succ \leftarrow Succ \cup \{v, t_1 + i - 1, \min(t_1 + i, t_2), false, pen_s\}$
        else if type $= 1$ then
            $Succ \leftarrow Succ \cup \{v, t_1 + i - 1, \min(t_1 + i, t_2), true, pen_s\}$
        else if type $= 2$ then
            if $t_1 + i - 1 \geq t'_1$ then
                $Succ \leftarrow Succ \cup \{v, t_1 + i - 1, \min(t_1 + i, t_2), false, pen_s\}$
            else
                $Succ \leftarrow Succ \cup \{v, t_1 + i - 1, \min(t_1 + i, t_2), true, pen_s\}$
    if $2 \times i > t_2 - t_1$ then
        break
    if $t_2 - i \geq t_{earl}$ and $t_2 \neq \infty$ then
        if type $= 0$ then
            $Succ \leftarrow Succ \cup \{v, t_2 - i, t_2 - i + 1, false, pen_e\}$
        else if type $= 1$ then
            $Succ \leftarrow Succ \cup \{v, t_2 - i, t_2 - i + 1, true, pen_e\}$
        else if type $= 2$ then
            if $t_2 - i \geq t'_1$ then
                $Succ \leftarrow Succ \cup \{v, t_2 - i, t_2 - i + 1, false, pen_e\}$
            else
                $Succ \leftarrow Succ \cup \{v, t_2 - i, t_2 - i + 1, true, pen_e\}$
    if $2 \times i = t_2 - t_1$ then
        break
if $\max(t_{earl}, t_1 + k) \leq t_2 - k$ then
    if type $= 0$ then
        $Succ \leftarrow Succ \cup \{v, \max(t_{earl}, t_1 + k), t_2 - k, false, 0\}$
    else if type $= 1$ then
        $Succ \leftarrow Succ \cup \{v, \max(t_{earl}, t_1 + k), t_2 - k, true, 0\}$
    else if type $= 2$ then
        if $t'_1 \geq \max(t_{earl}, t_1 + k)$ then
            $Succ \leftarrow Succ \cup \{v, \max(t_{earl}, t_1 + k), t'_1, true, 0\}$
        if $t_2 - k \geq \max(\max(t_{earl}, t_1 + k), t'_1)$ then
            $Succ \leftarrow Succ \cup \{v, \max(\max(t_{earl}, t_1 + k), t'_1), t_2 - k, false, 0\}$
return $Succ$
```

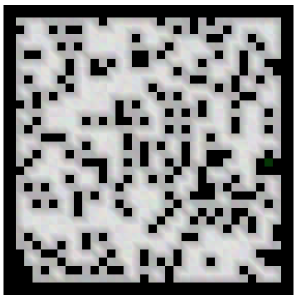
Chapter 5

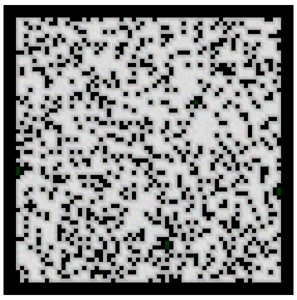
Experimental Results

The experimental results section is divided into 4 main subsections; in the first, the environment where all our algorithms are tested will be described, in the second one the experimental results of the k-robust MAPF-LNS2 algorithm will be presented, and in the third one the experimental results of the k-robust MAPF-SIPP will be presented. In the last section, the k-robust MAPF algorithms will be compared.

5.1 Environment

All algorithms used in this work were tested on MAPF benchmark website maps [20], specifically on the random-32-32-20 map (Fig. 5.1a), the random-64-64-20 map (Fig. 5.1b), and the warehouse-10-20-10-2-1 map (Fig. 5.1c). For each map, 10 random scenarios were chosen from the MAPF benchmark website.


(a) : Random-32


(b) : Random-64

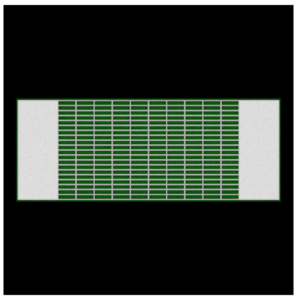

(c) : Warehouse

Figure 5.1: Maps of the used environments

5.2 K-robust MAPF-LNS2

In the initial part of the k-robust MAPF-LNS2 experimental section, all user-configurable parameters, including the map, number of agents, etc., will be presented. Subsequently, experiments will be conducted to determine the optimal size of *neighbourhood*, the most effective *repair* and *destroy* operators, and the best initial planning robustness l.

5.2.1 Algorithm Configuration Parameters

In this section, all the parameters that could or must be set in the k-robust MAPF-LNS2 algorithm are described.

Parameter	**Usage**	**Example**
Map	`-m [ -map ]`	`-m "random-32.map"`
Instance	`-a [ -agents ]`	`-a "random-32.scen"`
Num of Agents	`-k`	`-k 50`
Time Limit [s]	`-t [ -cutoffTime ]`	`-t 5`
Output Paths File	`-outputPaths`	`-outputPaths=paths.txt`
Max Iterations	`-maxIterations`	`-maxIterations=1000`
Statistics File	`-stats`	`-stats=statistics.csv`
Destroy Type	`-destroy_type`	`-destroy_type=1`
Repair Type	`-repair_type`	`-repair_type=0`
K-robustness	`-k_robust`	`-k_robust=6`
Initial K-robustness	`-initial_k_robust`	`-initial_k_robust=0`
Neighbourhood Size	`-neighborSize`	`-neighborSize=5`

Table 5.1: K-robust MAPF-LNS2 input parameters

The parameters required for the successful execution of the algorithm can be seen in Table 5.1. An example of correct usage is provided for each of the listed parameters.

As a map parameter, a file with the ".map" extension is requested containing the map on which the algorithm will be tested. The instance is an optional argument requiring a file with the ".scen" extension, which should contain a list of start and goal positions for agents. If this parameter is omitted, the algorithm automatically generates starting and target positions for the agents. Other parameters include the number of agents, the time limit in seconds, the maximum number of iterations of the algorithm, the robustness of the initial solution, the desired robustness, the size of the *neighbourhood*, a file for the resulting paths of agents, and a file for iteration statistics.

The final parameters correspond to the types of *destroy* and *repair* operators, each identified by a number from 0 to 2, as detailed in Table 5.2. For the execution of the Adaptive *destroy* – *repair* combination, the corresponding identifier is 5.

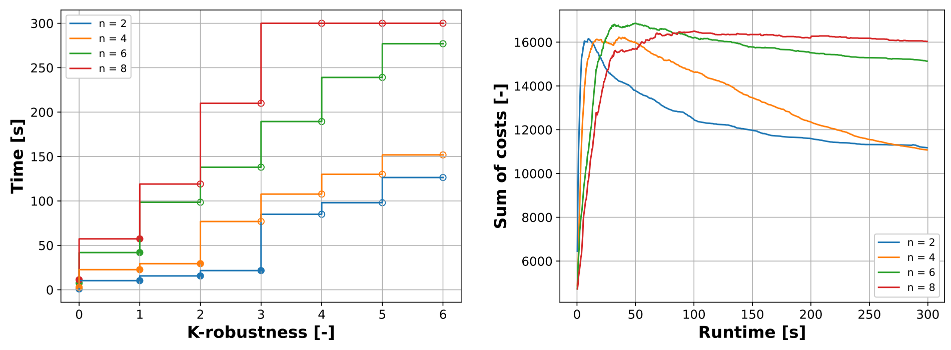
	0	1	2	5
Destroy	Random	Most serious	Least serious	Adaptive
Repair	Standard	Reversed	Binary	

Table 5.2: Destroy and Repair type numbers

5.2.2 Neighbourhood size selecting

The first of the experiments compares the influence of a size of *neighbourhood* on the time needed to reduce the number of k-collisions and the total sum of costs.

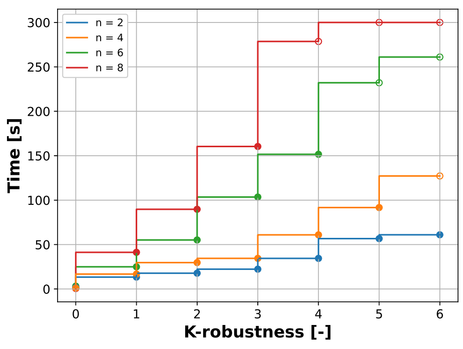
On each of the three chosen maps, four *neighbourhood* sizes $N = \{2, 4, 6, 8\}$ were tested with the *Standard repair* and *Most serious destroy* operators. On the random-32 map, the number of agents was set to 150, on the random-64 map it was set to 270, and on the warehouse map it was set to 260. The run-time of the experiments was set to 300 s, except for one experiment where it was set to 1800 s.



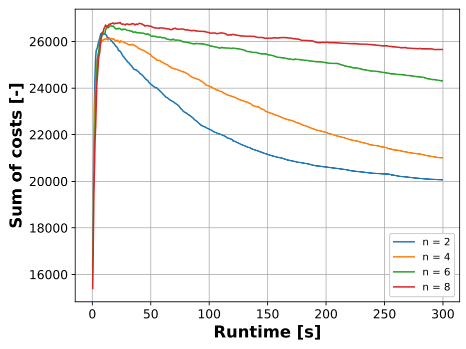
(a) : Robustness **(b) :** Sum of costs

Figure 5.2: Comparing neighbourhood sizes on random-32 map

For each of the selected maps, two types of graphs are presented. The first type, illustrated in Figs. 5.2a, 5.3a, and 5.4a, compares the average time needed for the algorithm to find the $k = \{0, \ldots, 6\}$-robust solution. If, for some experiments, the k'-robust solution is not found for $k' = \{0 \leq k' \leq k\}$, it is considered that all solutions with $k \geq k'$ were found in T_{max}, where T_{max} represents the run-time of the experiment. The full circles on the graph denote the robustness values achieved by each instance, while the empty circles represent values that were not reached in at least one of the experiments for the given N and were penalised at least once by penalty T_{max}. The second type, illustrated in Figs. 5.2b, 5.3b, and 5.4b, compares the average sum of costs of the agents in the chosen instances. It can be seen that the sum of costs increases rapidly as the algorithm finds a more robust solution at first, and then when it finds a k-robust solution, the sum of costs only decreases.

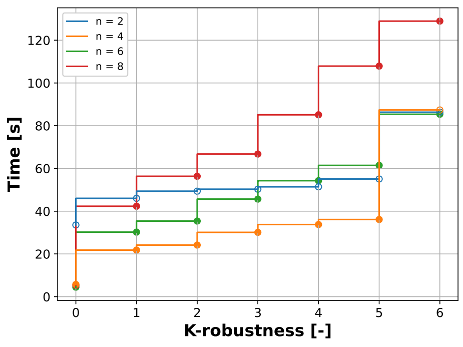


(a) : Robustness

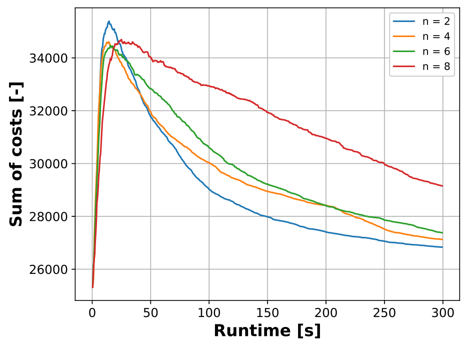


(b) : Sum of costs

Figure 5.3: Comparing neighbourhood sizes on random-64 map



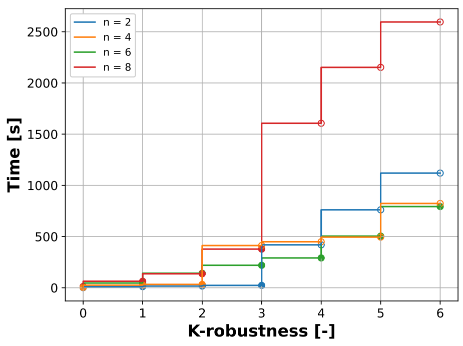
(a) : Robustness



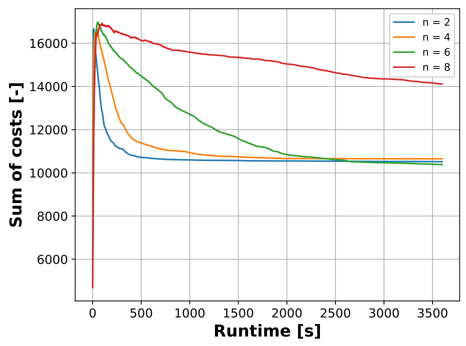
(b) : Sum of costs

Figure 5.4: Comparing neighbourhood sizes on warehouse map

Although a larger N in the LNS allows for more possible combinations in repairs, and therefore should find better solutions with better robustness and cost [8], Fig. 5.5 indicates that with increasing size, the time required to find individual k-robustness increases significantly.



(a) : Robustness



(b) : Sum of costs

Figure 5.5: Comparing neighbourhood sizes on random-32 with $T_{max} = 3600s$

The total number of iterations, accepted iterations and the success rate of accepting iterations for each N on each map can be seen in Table 5.3. The success rate (SR) is considered as the number of improving (accepted) iterations compared to all performed until reaching 90% of accepted iterations. This approach prevents penalising the success rate of methods that quickly find the optimal solution and subsequently have no more improving iterations. It can be seen that the larger *neighbourhoods* performs much fewer iterations and therefore finds the k-robust solution much slower than the smaller N. As can be seen, the dependence of the size of N on the number of iterations is not directly proportional, as one might expect. Instead, doubling the size of N results in approximately three times fewer iterations. Therefore, a *neighbourhood* of size $N > 6$, in these maps and instances, is not suitable for k-robust MAPF-LNS2.

N	Iterations	Accepted	SR
2	57607.3	1068.1	5.8 %
4	20296.2	689.3	5.1 %
6	9600.4	275.4	5.0 %
8	6442.6	150.4	4.2 %

(a) : Random-32 iterations

N	Iterations	Accepted	SR
2	27243.5	1574.9	9.7 %
4	9956.0	860.3	11.1 %
6	4933.2	323.7	9.2 %
8	3410.6	203.2	9.1 %

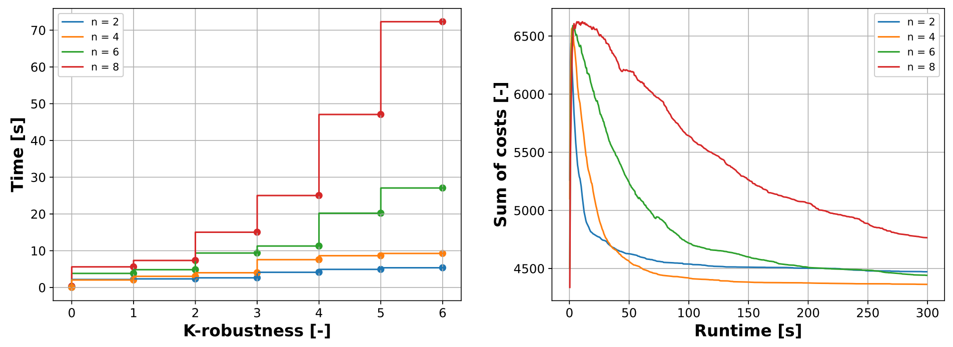
(b) : Random-64 iterations

N	Iterations	Accepted	SR
2	12884.5	1097.5	17.1 %
4	5827.0	817.5	21.0 %
6	3590.8	590.5	20.6 %
8	1861.4	321.6	21.0 %

(c) : Warehouse iterations

Table 5.3: Neighbourhood iterations statistics

As can be seen in Fig. 5.6, where the number of agents on map random-32 was reduced to 100, in simpler instances, a *neighbourhood* of size 4 and 6 finds a better solution in terms of the total sum of costs, but still finds the k-robust solution slower than $N = 2$, which is the main goal of this thesis.



(a) : Robustness **(b) :** Sum of costs

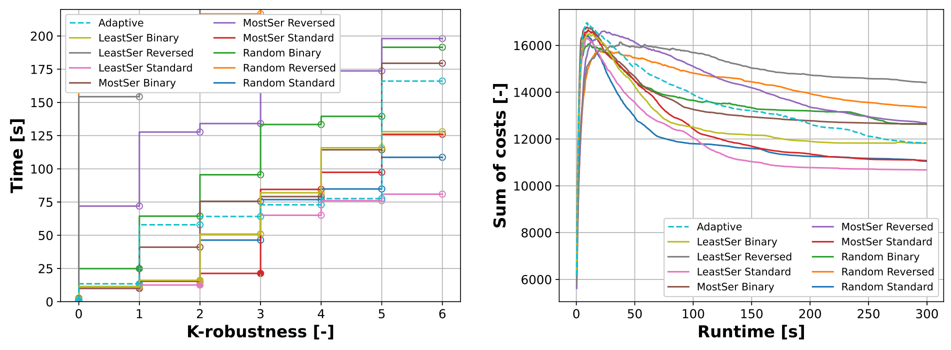
Figure 5.6: Comparing neighbourhood sizes on random-32 map with 100 agents

The *neighbourhood* of size 2 was the best on both "Random" maps, both in terms of finding a k-robust solution quickly and optimising the sum of costs. On the "Warehouse" map, again, $N = 2$ found solutions with a lower sum of costs, but in one case it did not even find a 0-robust solution. Therefore, it was penalised for all $k' \in \{0, \dots, 6\}$, making it slower to find k-robust solutions than $N = 4$. However, for further experiments, $N = 2$ is still chosen because it finds k-robust solutions faster and with a better sum of costs. If the algorithm does not find a solution in a minority of cases, a larger N size can be considered for these specific cases. However, in most cases, $N = 2$ proves to be the best.

5.2.3 Testing operators

The second of the experiments tested and compared the new *repair* and *destroy* operators. The 9 combinations of the 3 new *repair* and 3 new *destroy* operators were tested on the same instances and maps as in 5.2.2. The number of agents and the run-time also remained unchanged. The size of *neighboourhood* was set at 2 agents as it was previously shown to be the best.

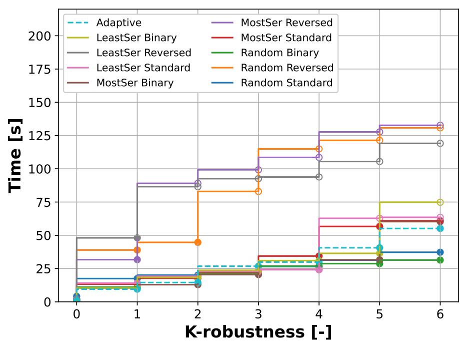
Figs. 5.7a, 5.8a, and 5.9a again show the k-robustness of each pair *destroy*−*repair* while Figs. 5.7b, 5.8b, and 5.9b again show the sum of costs of each pair.



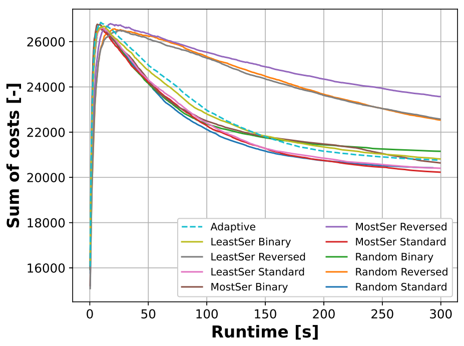
(a) : Robustness

(b) : Sum of costs

Figure 5.7: Comparing operators on random-32 map

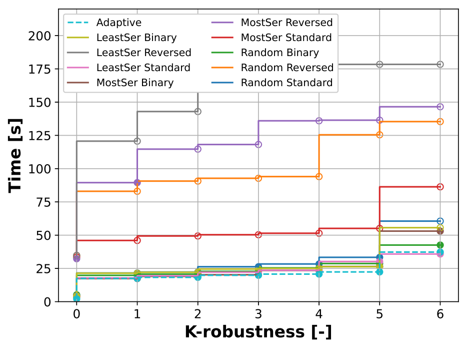


(a) : Robustness

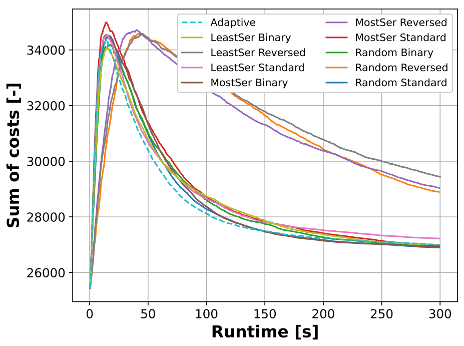


(b) : Sum of costs

Figure 5.8: Comparing operators on random-64 map



(a) : Robustness



(b) : Sum of costs

Figure 5.9: Comparing operators on warehouse map

In Figs. 5.7a, 5.8a, and 5.9a, it can be seen that three of the pairs perform significantly worse in finding the k-robust set of paths and also in finding the best cost solution (Figs. 5.7b, 5.8b, 5.9b), specifically the pairs with the "Reversed" *repair* operator. The average number of iterations performed by each pair *destroy* – *repair* can be seen in Table 5.4. It can be seen that the execution time of each *destroy* method is similar, considering that the number of iterations is usually comparable among them. It is also evident that the "Binary" *repair* method is slightly slower than the "Standard" method, and the execution of the "Reverse" method takes more than three times longer than the "Standard" method.

The performance of the remaining six pairs is highly dependent on the chosen map. Therefore, we present the "Adaptive" *destroy* – *repair* operator as a combination of the six best *destroy* – *repair* pairs using the Adaptive LNS method.

	Stand	Reverse	Binary
Random	57808.6	13993.4	43716.9
MostSer	57855.2	16025.6	41470.9
LeastSer	63229.6	13158.7	52056.1

(a) : Random-32 iterations

	Stand	Reverse	Binary
Random	26556.8	8657.1	26565.9
MostSer	27181.7	7484.2	25545.0
LeastSer	25516.9	8245.0	25082.1

(b) : Random-64 iterations

	Stand	Reverse	Binary
Random	14238.4	2873.9	13613.4
MostSer	12823.5	3019.6	13850.6
LeastSer	13261.1	2738.1	13276.6

(c) : Warehouse iterations

Table 5.4: Operators iterations statistics

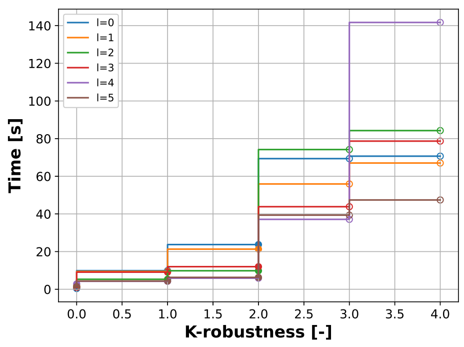
It can be seen that the "Adaptive" operator is never the best or the worst when compared to the other six best $destroy - repair$ combinations in terms of the speed of finding the k-robust solution. Instead, it consistently ranks around the middle in comparison with them. Therefore, it is an ideal method to reliably search for the k-robust set of paths. Even in the search for solutions with the lowest cost, the "Adaptive" operator is a good choice as it again performs similarly to the best $destroy - repair$ combinations.

5.2.4 Initial solution robustness

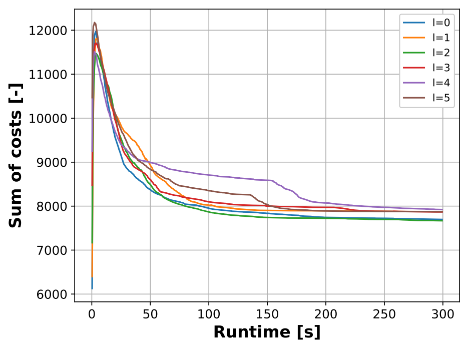
In this part of the work, the influence of the initial robustness l of the initial solution on the speed of finding the desired k-robustness was tested. Initially, k-robust MAPF-LNS2 attempted to find a l-robust set of paths using l-robust Prioritised SIPPS. Subsequently, regardless of whether it succeeded in finding the l-robust solution, this set of paths was used as the initial solution to find the k-robust set of paths.

The experiment was performed for $k \in \{4, 6\}$ and $l \in (0, k+1)$ with the same number of agents as in the previous tests, on the same maps and instances. The size of $neighbourhood$ was again chosen as 2 agents.

The influence of the initial l-robustness chosen on the time needed to find a k-robust solution can be seen in Figs. 5.10a, 5.12a and 5.14a, where the desired k-robustness is 4 and in Figs. 5.11a, 5.13a and 5.15a, where it is 6. The influence of the chosen l on the total sum of costs is shown in Figs. 5.10b, 5.11b, 5.12b, 5.13b, 5.14b, 5.15b.

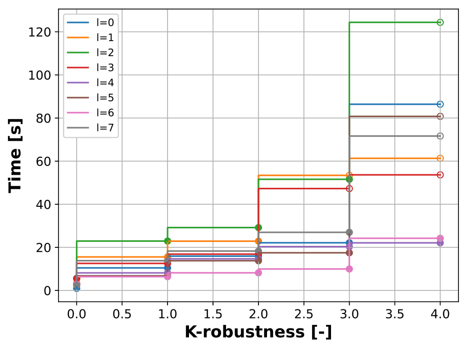


(a) : Robustness

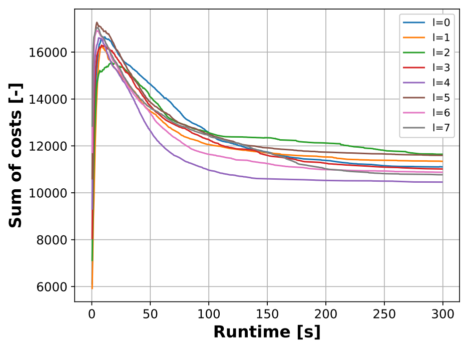


(b) : Sum of costs

Figure 5.10: Initial robustness testing on random-32 map with k=4

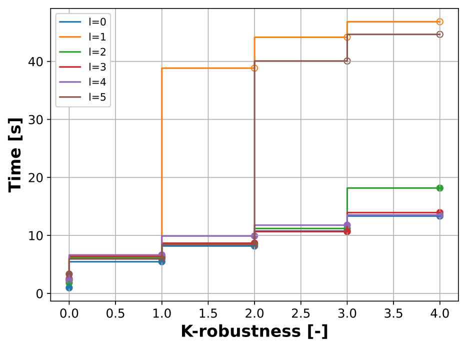


(a) : Robustness

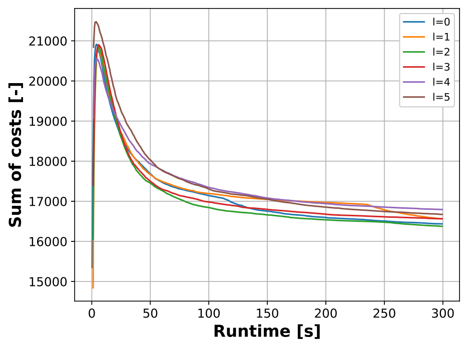


(b) : Sum of costs

Figure 5.11: Initial robustness testing on random-32 map with k=6

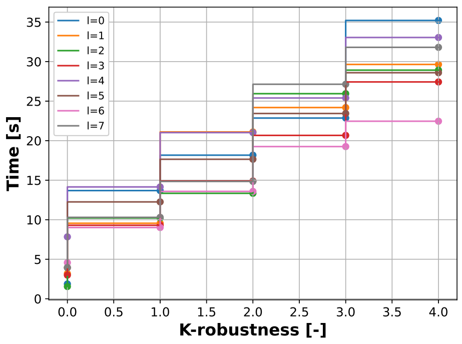


(a) : Robustness

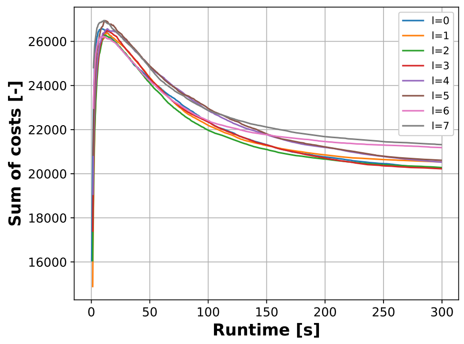


(b) : Sum of costs

Figure 5.12: Initial robustness testing on random-64 map with k=4

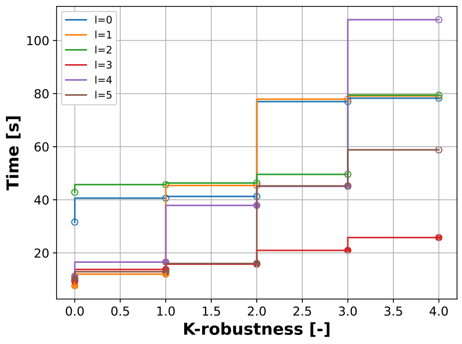


(a) : Robustness

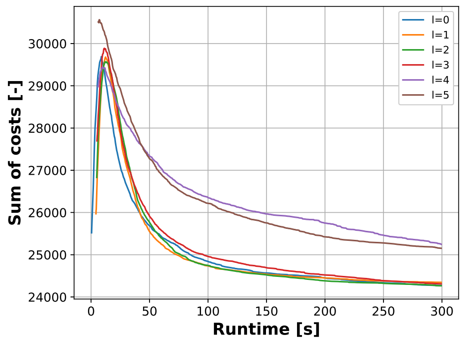


(b) : Sum of costs

Figure 5.13: Initial robustness testing on random-64 map with k=6

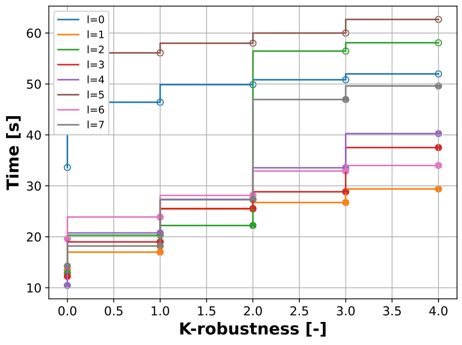


(a) : Robustness

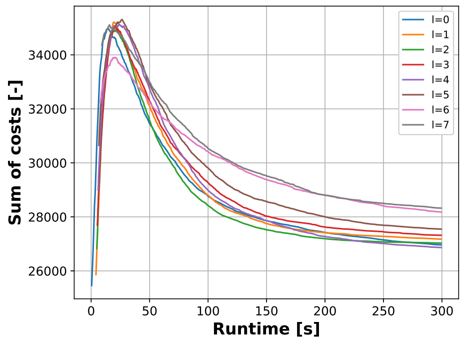


(b) : Sum of costs

Figure 5.14: Initial robustness testing on warehouse map with k=4



(a) : Robustness



(b) : Sum of costs

Figure 5.15: Initial robustness testing on warehouse map with k=6

As seen in the Figs. mentioned above, there is no clear dependence of the initial l-robustness on the time needed to find the k-robust solution. In some cases, such as in Figs. 5.10a, 5.11a, it appears that higher values of l find the resulting k faster, but in other cases, high values of l search for

k-robust solutions more slowly. Regarding the sum of costs, it is evident from the graphs that experiments with $l \geq k$ result in solutions with higher costs compared to lower l values.

Therefore, it cannot be conclusively stated with which initial l-robustness parameter it is best to run the algorithm to find the desired k-robust solution most quickly. For this reason, we will use the value $l = 3$, which never performs the worst in terms of the speed of finding k-robust solutions and is also not the worst in finding solutions with the lowest cost.

5.2.5 Robustness - cost

In the Table 5.5, the first column shows the average cost sum for each map for $k \in (0, 6)$. The second column shows the increment in the cost sum for a given k compared to $k = 0$. On the random-32 map, the number of agents was set to 150, on the random-64 map it was set to 270, and on the warehouse map it was set to 260. The run-time of the experiments was set to 300 s, and the Adaptive operator was used.

It can be seen that with increasing k-robustness the total sum of costs also increases. It can be seen that on the random-32 map it is much harder to find k-robust solutions, as there are many agents on a relatively small map, than on the other two maps, and therefore the increase of the sum of costs is much higher, meaning that this map with the chosen number of agents would probably need more run time to get better results. On the random-64 and warehouse map, it can be seen that the increase in the sum of costs from the 0-robust solution to, for example, the 1-robust solution is maximal 5%, which is not much considering the fact that it can prevent some collisions.

K	SoC	Incr.
0	3703.8	1
1	4204.7	1.14
2	5154.2	1.39
3	6475.8	1.75
4	8104.7	2.19
5	9472.0	2.56
6	11841.3	3.20

(a) : Random-32

K	SoC	Incr.
0	12209.1	1
1	12790.2	1.05
2	13546.8	1.11
3	14998.3	1.23
4	16920.2	1.39
5	18459.9	1.51
6	20762.7	1.70

(b) : Random-64

K	SoC	Incr.
0	21423.3	1
1	22030.4	1.03
2	22618.5	1.06
3	23285.2	1.08
4	24290.9	1.13
5	25642.4	1.20
6	27001.5	1.26

(c) : Warehouse

Table 5.5: Increasing robustness sum of costs

5.3 K-robust MAPF-SIPP

5.3.1 Parameters

Parameter	**Usage**	**Example**
Map	`-i`	`-i "random-32.map"`
Instance	`-j`	`-j "random-32.scen"`
Num of Agents	`-a`	`-a 50`
Statistics File	`-o`	`-o "stats.yaml"`
K-robustness	`-g`	`-g 6`
Blocking limit	`-b`	`-b 50`

Table 5.6: K-robust MAPF-SIPP input parameters

The parameters required for the algorithm and their correct use can be seen in Table 5.6.

As a map parameter, a file with the ".map" extension containing the map is required. Additionally, an instance parameter necessitates a file with the ".scen" extension, which includes a set of start and goal positions for agents. The following essential parameters include the number of agents, the desired k-robustness, and the address of the statistics file with the ".yaml" extension. The blocking limit parameter is the initial number of time steps in which no agent can move to the starting location of other agents. This parameter was added to k-robust MAPF-SIPP to increase the success rate of the algorithm by preventing agents from getting stuck in their starting location without possible movement.

5.3.2 Success rate

In Tables 5.7 and 5.8 can be seen the success rate of the MAPF-SIPP algorithm with the blocking limit set to 50. It can be seen that on the random-32 map the k-robust MAPF-SIPP algorithm found a 6-robust solution on at least 90% instances only for a maximum of 40 agents, while on the random-64 map it was able to find at least 90% of 6-robust solutions for up to 180 agents.

Agents:	10	20	30	40	50	60	70	80	90	100	110	120	130	140
k=1	1.0	1.0	1.0	1.0	0.9	0.8	0.7	0.7	0.7	0.7	0.6	0.4	0.3	0.4
k=2	1.0	1.0	1.0	1.0	0.9	0.8	0.7	0.7	0.7	0.7	0.6	0.5	0.4	0.2
k=3	1.0	1.0	0.9	0.9	0.9	0.8	0.7	0.7	0.7	0.7	0.5	0.5	0.3	0.1
k=4	1.0	1.0	0.9	0.9	0.9	0.8	0.7	0.7	0.7	0.7	0.4	0.4	0.2	0.0
k=5	1.0	1.0	0.9	0.9	0.7	0.7	0.7	0.7	0.7	0.5	0.4	0.1	0.2	0.1
k=6	1.0	1.0	0.9	0.9	0.7	0.7	0.7	0.6	0.5	0.3	0.1	0.0	0.0	0.0

Table 5.7: K-robust MAPF-SIPP success rate on random-32 map

Agents:	100	110	120	130	140	150	160	170	180	190	200	210	220	230	240
k=1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.9	0.8
k=2	1.0	1.0	0.9	0.9	0.9	0.9	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.8
k=3	1.0	1.0	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.8	0.8	0.8
k=4	1.0	1.0	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.8	0.7	0.8	0.9
k=5	1.0	1.0	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.7
k=6	1.0	1.0	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.8	0.8	0.8	0.8	0.8	0.8

Table 5.8: K-robust MAPF-SIPP success rate on random-64 map

5.4 Comparison of k-robust MAPF algorithms

In this section, both presented k-robust algorithms will be compared.

To evaluate algorithmic performance based on the overall cost effectiveness of their solutions, only the instances in which k-robust MAPF-SIPP successfully found a solution were chosen, k-robust MAPF-LNS2 found the desired k-robust solution in these instances every time. The time limit for the k-robust MAPF-LNS2 algorithm was set at 15 s and the k-robust MAPF-SIPP block limit was set at 50. In Tables 5.9, 5.10, and 5.11 the comparison of the total sum of costs of the two algorithms on the same instances can be seen. In Figs 5.9c, 5.10c, and 5.11c the increment of the k-robust MAPF-SIPP solution sum of costs can be seen compared to the solution found by k-robust MAPF-LNS2

Agents:	10	20	30	40
k=4	225.3	456.3	715.4	1015.4
k=6	229.5	471.7	758.9	1125.2

(a) : MAPF-LNS2

Agents:	10	20	30	40
k=4	239.7	500.3	842.4	1251.3
k=6	247.4	538.6	921.1	1404.2

(b) : MAPF-SIPP

Agents:	10	20	30	40
k=4	1.06	1.10	1.18	1.23
k=6	1.08	1.14	1.21	1.25

(c) : Increment

Table 5.9: Comparing algorithms sum of costs on random-32 map

Agents:	110	130	150	170
k=4	5445.1	6544.4	7733.4	9037.3
k=6	5865.9	7228.6	8800.1	10739.0

(a) : MAPF-LNS2

Agents:	110	130	150	170
k=4	6101.7	7432.4	8899.9	10453.9
k=6	6741.1	8314.6	10087.2	11999.6

(b) : MAPF-SIPP

Agents:	110	130	150	170
k=4	1.12	1.14	1.15	1.16
k=6	1.15	1.15	1.15	1.12

(c) : Increment

Table 5.10: Comparing algorithms sum of costs on random-64 map

Agents:	110	130	150	170
k=4	9771.2	11281.6	13274.8	15074.1
k=6	10036.0	11669.6	14009.2	16174.0

(a) : MAPF-LNS2

Agents:	110	130	150	170
k=4	10257.6	11929.7	13852.8	15702.9
k=6	10470.1	12222.3	14224.4	16197.7

(b) : MAPF-SIPP

Agents:	110	130	150	170
k=4	1.05	1.06	1.04	1.04
k=6	1.04	1.05	1.02	1.00

(c) : Increment

Table 5.11: Comparing algorithms sum of costs on warehouse map

It can be observed that the k-robust MAPF-LNS2 algorithm consistently found a less expensive solution in all cases. However, the solutions found by MAPF-SIPP are often not significantly more expensive than those found by MAPF-LNS2. For example, on the warehouse map, the solution found by k-robust MAPF-SIPP is a maximum of 1.06 times more expensive than the solution found by k-robust MAPF-LNS2. However, the solutions found for 40 agents on the random-32 map are up to 1.25 times worse, which is significant.

The next main advantage of k-robust MAPF-LNS2 over k-robust MAPF-SIPP is that if MAPF-LNS2 fails to find a k-robust solution, it returns at least the highest found i-robust solution, where $i < k$. In general, k-robust MAPF-LNS2 outperformed k-robust MAPF-SIPP in all experiments and all metrics.

Chapter 6

Conclusion

This work focused on planning a k-robust set of paths for multiple robots, allowing each of them to be delayed by up to k time steps without colliding. Its objective was to enhance two existing multiagent algorithms to search for k-robust paths. Specifically, a k-robust MAPF-SIPP algorithm and a k-robust MAPF-LNS2 algorithm were implemented. Various strategies were tested to find k-robust paths using k-robust MAPF-LNS2, such as the size of *neighbourhood* or the influence of l-robustness of the initial solution on the speed of finding the desired k-robust solution. In addition, new *destroy* and *repair* operators were designed and tested for use in k-robust MAPF-LNS2 to efficiently find a k-robust set of paths.

The k-robust MAPF-LNS2 algorithm outperforms the standard k-robust SIPP algorithm in terms of the success rate in finding the solutions and also in the sum of costs of the solution.

Furthermore, it would be interesting to design and test adaptive sizes of *neighbourhood* that could find k-robust solutions more quickly and with a lower sum of costs.

Bibliography

[1] Zain Alabedeen Ali and Konstantin Yakovlev. Prioritized SIPP for Multi-agent Path Finding with Kinematic Constraints. In Andrey Ronzhin, Gerhard Rigoll, and Roman Meshcheryakov, editors, *Interactive Collaborative Robotics*, Lecture Notes in Computer Science, pages 1–13, Cham, 2021. Springer International Publishing.

[2] Dor Atzmon, Ariel Felner, Roni Stern, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. k-Robust Multi-Agent Path Finding. *Proceedings of the International Symposium on Combinatorial Search*, 8(1):157–158, September 2021.

[3] Dor Atzmon, Roni Stern, Ariel Felner, Nathan R. Sturtevant, and Sven Koenig. Probabilistic Robust Multi-Agent Path Finding. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30:29–37, June 2020.

[4] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. *Proceedings of the International Symposium on Combinatorial Search*, 5(1):19–27, 2014.

[5] Michael Erdmann and Tomás Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2(1):477–521, November 1987.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[7] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter Stuckey, and Sven Koenig. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36:10256–10265, June 2022.

[8] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. Anytime Multi-Agent Path Finding via Large Neighborhood Search. volume 4, pages 4127–4135, August 2021.

[9] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14):12353–12362, May 2021.

[10] Ryan Luna and Kostas E. Bekris. Efficient and complete centralized multi-robot path planning. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3268–3275, September 2011.

[11] Hang Ma and Sven Koenig. AI buzzwords explained: multi-agent path finding (MAPF). *AI Matters*, 3(3):15–19, 2017.

[12] Robert Morris, Corina S Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, TK Satish Kumar, and Sven Koenig. Planning, scheduling and monitoring for airport surface operations. In *AAAI Workshop: Planning for Hybrid Systems*, pages 608–614, 2016.

[13] Mike Phillips and Maxim Likhachev. SIPP: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*, pages 5628–5635, May 2011.

[14] Qandeel Sajid, Ryan Luna, and Kostas Bekris. Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives. *Proceedings of the International Symposium on Combinatorial Search*, 3(1):88–96, 2012.

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

[16] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, February 2013.

[17] Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, Lecture Notes in Computer Science, pages 417–431, Berlin, Heidelberg, 1998. Springer.

[18] David Silver. Cooperative Pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 1(1):117–122, 2005.

[19] Roni Stern. Multi-Agent Path Finding – An Overview. In Gennady S. Osipov, Aleksandr I. Panov, and Konstantin S. Yakovlev, editors, *Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures*, Lecture Notes in Computer Science, pages 96–115. Springer International Publishing, Cham, 2019.

[20] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Kumar, Roman Barták, and Eli Boyarski. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *Proceedings of the International Symposium on Combinatorial Search*, 10(1):151–158, 2019.

[21] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24, February 2015.

[22] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*, 29(1):9–9, March 2008.

[23] Michal Čáp, Jiří Vokřínek, and Alexander Kleiner. Complete Decentralized Method for On-Line Multi-Robot Trajectory Planning in Well-formed Infrastructures. *Proceedings of the International Conference on Automated Planning and Scheduling*, 25:324–332, April 2015.

Appendix A
List of Abbreviations

Abbreviation	**Meaning**
ALNS	Adaptive Large Neighbourhood search
CBS	Conflict Based Search
HCA*	Hierarchical Cooperative A*
LNS	Large Neighbourhood Search
MAPF	Multi-Agent Path Finding
MAPF-LNS2	Multi-Agent Path Finding via Large Neighbourhood Search
MAPF-SIPP	Multi-Agent Path Finding via Safe Interval Path Planning
PP	Prioritised Planning
SIPP	Safe Interval Path Planning
SIPPS	Safe Interval Path Planning with Soft constraints
SR	Success rate