



UNIVERSITÀ DI PARMA

UNIVERSITÀ DEGLI STUDI DI PARMA

Dottorato di Ricerca in Tecnologie dell'Informazione

CICLO XXXVII

**Complete and Locally Optimal Algorithms for Multi-Agent
Path Finding on Graphs**

Coordinatore:

Chiar.mo Prof. Marco Locatelli

Tutor:

Chiar.mo Prof. Luca Consolini

Dottorando: *Irene Saccani*

Anni Accademici 2021/2022 - 2023/2024

Ai miei gatti, Pepe e Nyx

Summary

Introduction	1
1 Spectral Clustering approach for MAPF on Connected Graphs	15
1.1 Spectral Clustering and Bi-connected Components	16
1.1.1 Spectral Clustering	16
1.1.2 Bi-connected Components Graph	20
1.2 Combining Spectral decomposition, Push-and-Swap and BiBOX . .	22
1.3 Push-and-Swap modification	23
1.4 Experimental Results	26
1.4.1 Spectral Clustering and Bi-connected Components	26
1.4.2 Comparing BIBOX with Chi-Push-and-Swap and ChiBOX .	31
2 Feasibility and Algorithms for MAPF on Strongly Connected Digraphs	35
2.1 Problem Definition	36
2.2 Solving MAPF on undirected graphs	39
2.2.1 Convert MAPF into ts-PMT	39
2.2.2 Summary of algorithm presented in [1] to solve MAPF on graphs	42
2.3 Strongly connected digraphs	42
2.4 Feasibility of MAPF on strongly connected digraphs	46
2.4.1 Necessary and Sufficient condition for feasibility	49
2.5 Path planner for strongly connected digraph	51

2.5.1	Basic plans for <i>CONVERT-PATH</i> algorithm	52
2.5.2	Plans for movements through biconnected components . . .	56
2.5.3	<i>CONVERT-PATH</i> algorithm	66
2.6	Upper bound for solutions length and time complexity	68
2.7	Experimental Results	70
2.7.1	Random graphs	70
2.7.2	Graph based on <i>movingai</i> MAPF benchmarks	77
2.7.3	Real-life applications examples	78
3	Dynamic Programming based Local Search	83
3.1	Problem definition	84
3.2	Measuring the distance between plans	87
3.2.1	Path distances	88
3.2.2	Agent Distances	91
3.2.3	Optimization MAPF problem in a neighborhood of a reference plan	94
3.3	Iterative local optimization for FSA	95
3.3.1	Equivalence classes associated to the previously defined distances	97
3.3.2	Neighborhoods	99
3.3.3	Iterative Neighborhood Search	104
3.3.4	Dynamic Programming Algorithm with Dominance	106
3.3.5	Complexity results	110
3.4	Experimental results	117
3.4.1	Directed Graphs	117
3.4.2	Undirected Graphs	135
	Conclusion and Future Research	141
	Bibliography	145
	Ringraziamenti	153

List of Figures

1	Graph of a warehouse with 397 nodes.	1
2	Graph of a warehouse with 3399 nodes.	2
3	Example of a MAPF problem instance on a directed graph.	3
4	Vertex collision	3
5	Swapping collision	3
6	An illustration of the ear decomposition with the initial cycle C and three ears H_1 , H_2 , and H_3	6
1.1	3D representation of data extracted from a 30x30 grid graph.	17
1.2	3D representation and clustering of a 30x30 grid graph.	18
1.3	Cluster Graph a 30x30 grid graph.	19
1.4	Bi-connected components of a 30x30 grid graph.	20
1.5	Bi-connected component graphs of graphs in figure 1.4.	21
1.6	Random connected graphs with 100 vertices.	27
1.7	Random tree of bi-connected components with 100 vertices.	27
1.8	Connectivity with Agglomerative Clustering.	28
1.9	Connectivity with K-means.	29
1.10	Connectivity with Agglomerative Clustering.	30
1.11	Connectivity with K-means.	30
1.12	A graph used for the experiment. The graph contains 4 clusters: A, B, C, D.	31
1.13	A graph divided into 7 loops distinguished by letters A-G, which is to be solved by the original BiBOX algorithm.	32

1.14	The same graph as in figure 1.13, but after spectral decomposition, it is divided into four clusters A, B, C, and D, on which four parallel instances of ChiBOX, each dealing with one loop, could be run.	32
1.15	A graph decomposed into three components, with vertex 5 belonging to two components at once.	33
2.1	Undirected graph and corresponding biconnected component tree. A, B, and C are the trans-shipment vertices.	40
2.2	Example of strongly connected digraph: the corresponding underlying graph and biconnected component tree are shown in Figure 2.1.	43
2.3	Digraph with an open ear decomposition.	44
2.4	A digraph D and its underlying graph $\mathcal{G}(D)$	48
2.5	Example of 3-CYCLE ROTATION.	52
2.6	Example of BRING HOLE FROM v TO w	54
2.7	An example of situation of Entry Lemma 5.1. Pebble p has to move on w without altering the final location of the other pebbles. h_1 and h_2 are, respectively, the destination and transport holes. The solution plan is $f_{vw} = r_2^{C_1} r_3^{C_2} r_2^{C_3} (v \rightarrow y) r_3^{C_3} r_2^{C_2} r_2^{C_1}$	57
2.8	An example of the situation described in Attached-edge Lemma 5.3. Pebble p has to move on w without altering the final location of the other pebbles. The solution plan is $f_{vw} = r_{d(u,y)} (y \rightarrow v) r_k (v \rightarrow y) r_{d(y,w)}$	60
2.9	An example of the situation described in Stay in Lemma 5.4. Pebble p has to move on w without altering the final location of the other pebbles. h_1 and h_2 are, respectively, the destination and transport holes. The solution plan is $f_{vw} = r_2^{C_3} r_2^{C_2} r_2^{C_1} (x \rightarrow a_2) r_3^{C_1} r_3^{C_2} r_3^{C_3}$	61
2.10	An example of the situation described in Stay in Lemma 5.4 in which the initial position of the pebble $v = \mathcal{A}(p)$ and the target $w = \mathcal{A}(h_1)$ belong to the same cycle C_1 . In this case we move p on another cycle and then we proceed as in Figure 2.9.	64
2.11	From tree to strongly connected digraph.	71

2.12	Number of steps per number of agents of every instance in random graphs with 20, 60 and 100 nodes. The median of the number of steps of instances for each number of agents is shown in red.	72
2.13	Running time per number of agents on random graphs with 20, 60 and 100 nodes. The median of the running times of instances for each number of agents is shown in red.	73
2.14	Number of steps per number of agents of every instance in random graphs with 20, 60 and 100 nodes. The median of the number of steps of instances for each number of agents is shown in red.	74
2.15	Running time per number of agents of every instance in random graphs with 20, 60 and 100 nodes. The median of the running times of instances for each number of agents is shown in red.	75
2.16	Number of moves per number of agents on random graphs with 20 and 60 nodes for a high number of agents. The median of the number of steps of instances for each number of agents is shown in red. . . .	76
2.17	Running time per number of agents on random graphs with 20 and 60 nodes for a high number of agents. The median of the running time of instances for each number of agents is shown in red.	76
2.18	Strongly connected digraph obtained from room graph of <i>movingai</i> MAPF benchmark	77
2.19	Number of moves and running time per number of agents on graph in Figure 2.18. The median of the number of steps and running time of instances for each number of agents is shown in red.	78
2.20	Graph representing the warehouse.	79
2.21	Number of moves and computation times per number of agents in the warehouse of Figure 2.20, with medians highlighted in red	80
2.22	Number of moves and computation times per number of agents in the graphs with same structure as Figure 2.20.	80
3.1	Plan f_0 on graph with 10 nodes.	90
3.2	Plan g on graph with 10 nodes.	91

3.3	Plan h on graph with 10 nodes.	94
3.4	Average Reduction Ratio per $ V $ for 5 agents.	118
3.5	Average Reduction Ratio per $ V $ for 10 agents.	118
3.6	Average Reduction Ratio per $ V $ for 15 agents.	119
3.7	Average Running Time per $ V $ for 5 agents.	119
3.8	Average Running Time per $ V $ for 10 agents.	119
3.9	Average Running Time per $ V $ for 15 agents.	119
3.10	Average Reduction Ratio per $ P $ for 20 nodes.	120
3.11	Average Reduction Ratio per $ P $ for 50 nodes.	120
3.12	Average Reduction Ratio per $ P $ for 100 nodes.	120
3.13	Average Running Time per $ P $ for 20 nodes.	120
3.14	Average Running Time per $ P $ for 50 nodes.	121
3.15	Average Running Time per $ P $ for 100 nodes.	121
3.16	Average Reduction Ratio per $ V $ for 5 agents.	122
3.17	Average Reduction Ratio per $ V $ for 10 agents.	122
3.18	Average Reduction Ratio per $ V $ for 15 agents.	122
3.19	Average Running Time per $ V $ for 5 agents.	122
3.20	Average Running Time per $ V $ for 10 agents.	123
3.21	Average Running Time per $ V $ for 15 agents.	123
3.22	Average Reduction Ratio per $ P $ for 20 nodes.	124
3.23	Average Reduction Ratio per $ P $ for 50 nodes.	124
3.24	Average Reduction Ratio per $ P $ for 100 nodes.	124
3.25	Average Running Time per $ P $ for 20 nodes.	124
3.26	Average Running Time per $ P $ for 50 nodes.	125
3.27	Average Running Time per $ P $ for 100 nodes.	125
3.28	Average Reduction Ratio per $ V $ for 5 agents.	126
3.29	Average Reduction Ratio per $ V $ for 10 agents.	126
3.30	Average Reduction Ratio per $ V $ for 15 agents.	126
3.31	Average Running Time per $ V $ for 5 agents.	126
3.32	Average Running Time per $ V $ for 10 agents.	127
3.33	Average Running Time per $ V $ for 15 agents.	127

3.34	Average Reduction Ratio per $ P $ for 20 nodes.	128
3.35	Average Reduction Ratio per $ P $ for 50 nodes.	128
3.36	Average Reduction Ratio per $ P $ for 100 nodes.	128
3.37	Average Running Time per $ P $ for 20 nodes.	128
3.38	Average Running Time per $ P $ for 50 nodes.	129
3.39	Average Running Time per $ P $ for 100 nodes.	129
3.40	Average Reduction Ratio per $ V $ for 5 agents.	130
3.41	Average Reduction Ratio per $ V $ for 10 agents.	130
3.42	Average Reduction Ratio per $ V $ for 15 agents.	130
3.43	Average Running Time per $ V $ for 5 agents.	130
3.44	Average Running Time per $ V $ for 10 agents.	131
3.45	Average Running Time per $ V $ for 15 agents.	131
3.46	Average Reduction Ratio per $ P $ for 20 nodes.	132
3.47	Average Reduction Ratio per $ P $ for 50 nodes.	132
3.48	Average Reduction Ratio per $ P $ for 100 nodes.	132
3.49	Average Running Time per $ P $ for 20 nodes.	132
3.50	Average Running Time per $ P $ for 50 nodes.	133
3.51	Average Running Time per $ P $ for 100 nodes.	133
3.52	Maze-32-32-2 from MAPF benchmark suite.	135
3.53	Average Reduction Ratio per $ P $	136
3.54	Average Running Time per $ P $	136
3.55	Empty 4x4 graph.	137
3.56	Bottleneck graph.	138
3.57	Empty-8-8 graph from MAPF benchmark suite.	139

List of Tables

3.1	Configurations of plan f_0 , the k -th column contains the nodes occupied by the agents at step k	90
3.2	Configurations of plan g	91
3.3	Initial plan f_0	93
3.4	Second plan h	94
3.5	Results of tests on graph 3.55 with 14 agents. The best result for each instance is highlighted in red	138
3.6	Results of tests on graph 3.55 with 14 agents. The best result for each instance is highlighted in red	139
3.7	Results of tests on graph 3.57 with 14 agents. The best result for each instance is highlighted in red	140

Introduction

Graphs serve as an abstraction for a wide range of scenarios, from warehouse logistics to traffic optimization and even internet topology. Among these applications, one particularly useful case is the management of fleets of Automated Guided Vehicles (AGVs) within automated warehouses. To achieve this, the layout of the warehouse is represented as a graph, where each node corresponds to a location that an AGV needs to reach, and edges between nodes represent the possible routes the AGVs can take to move between these locations. Figures 1 and 2 illustrate two examples of graphs derived from real-world warehouse environments.

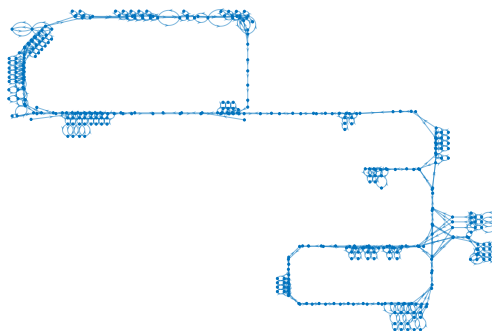


Figure 1: Graph of a warehouse with 397 nodes.

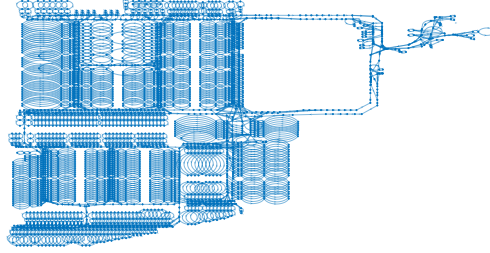


Figure 2: Graph of a warehouse with 3399 nodes.

Once the warehouse is mapped as a graph, AGVs can be treated as agents navigating through the graph. To efficiently manage the fleet, we want to make the AGVs travel in the fastest way when working inside the warehouse. For a single AGV, this problem can be approached as a Shortest Path Finding problem, where the agent needs to travel from a source node to a target node in the quickest way possible. This can be easily solved using well-known search algorithms like **Dijkstra's Algorithm**, which provides an optimal solution in polynomial time.

Managing multiple AGVs, however, is more complex. A simple approach is to treat each AGV independently. In this case, each AGV follows the shortest path from its position to the assigned target node, avoiding collisions with other AGVs when they cross paths. While this is a simple and fast method to manage the problem, it carries the risk of creating **deadlock** situations. In a deadlock, the AGVs block each other and cannot move, requiring manual intervention to resolve the issue.

Our approach proposes treating the entire fleet as multiple agents moving inside a graph. To avoid deadlock situations and make sure that the AGVs always reach their target, if that is possible, we focus on complete algorithms. Managing multiple agents is far more challenging than solving the path for a single AGV. While finding the optimal path for one agent can be done efficiently, this is not the case when dealing with multiple agents. This problem is known as Multi-Agent Path Finding (MAPF) problem.

We consider a graph (directed or undirected) $G = (V, E)$, where V represents the set of vertices (or nodes) and E represents the set of edges. A set of agents (or pebbles) P

are initially positioned on distinct nodes, and they can move to adjacent, unoccupied nodes.

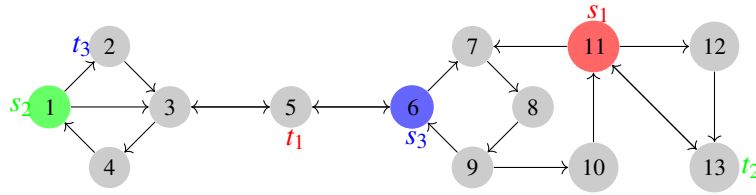


Figure 3: Example of a MAPF problem instance on a directed graph.

The goal of the Multi-Agent Path Finding (MAPF) problem [2] is to compute a sequence of movements that guides all agents to their designated target nodes while avoiding collisions. We address two types of possible collisions. The first is a vertex collision (Figure 4), which occurs when two or more agents are scheduled to occupy the same node at the same time. The second is a swapping collision (Figure 5), which happens when two agents attempt to traverse the same edge in opposite directions at the same time.



Figure 4: Vertex collision

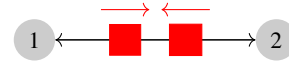


Figure 5: Swapping collision

There are several ways to define the optimality of a solution to the MAPF problem, and the literature generally focuses on three main objective functions:

- **Makespan:** The total time required for all agents to reach their target nodes, determined by the agent that takes the longest to reach its destination.
- **Sum-of-cost:** The sum of the individual times taken by each agent to reach its target node.

- **Fuel:** The total number of movements (between two different nodes) made by all agents.

As noted in [3], finding an optimal solution for a MAPF problem, regardless of the objective function, is NP-hard. This holds whether agents can move simultaneously or only one at a time. Due to the NP-hardness, it is impossible to compute an optimal solution in polynomial time with respect to the number of nodes or agents.

The primary motivation of this thesis is to address the MAPF problem using sub-optimal algorithms that can produce solutions in polynomial time. Although these solutions are not optimal, they aim to achieve high quality, which is crucial for practical, real-world applications.

State of the Art

MAPF on Undirected Graphs

MAPF problems are more extensively studied on undirected graphs. In this type of graphs, the edges connecting two different nodes can be traversed in both directions. An optimal solution can be found using a **Dynamic Programming** algorithm or algorithms such as **A*** or the **Dijkstra** algorithm, searching for a path in a graph in which each node represents a different configuration of all the agents. These algorithms can be highly computationally expensive, as finding optimal solutions for MAPF problems is NP-hard.

Another example of an optimal algorithm for MAPF on undirected graphs is **Conflict Based Search** (CBS) [4]. CBS is a two-level algorithm that takes a different approach compared to those mentioned above. At the lower level, the algorithm searches for the paths of individual agents, while at the higher level, it performs a search on a tree based on conflicts (Conflict Tree). First, the algorithm finds the shortest paths for all agents and calculates the solution cost, creating the root node of the conflict tree. At each iteration, the higher level selects the node with the lowest cost from the tree and checks for the first conflict between two agents. If no conflict exists, a solution is found. If a conflict is detected, the higher level generates two new nodes (one for

each agent involved in the conflict) and assigns a new constraint to each. The lower level then searches for a new path for the agent, ensuring it satisfies all the constraints imposed by the higher level. While CBS often explores fewer states than the other approaches, it is not practical for real-world applications due to its exponential time complexity.

As finding optimal solutions for MAPF problems is NP-hard, we focus on finding sub-optimal solutions. Among sub-optimal solvers, we can distinguish search-based and rule-based approaches.

Search-based sub-optimal solvers aim to provide a high-quality solution but are not complete (i.e., they are not always able to return a feasible solution). Two search-based sub-optimal algorithms that are bounded-sub-optimal variants of CBS are **Enhanced CBS** (ECBS) [5] and **Explicit Estimation CBS** (EECBS) [6]. The first algorithm uses *focal search*, an alternative approach to obtain bounded sub-optimal search algorithms based on A*. The second one employs online learning to obtain inadmissible estimates of solution cost and applies Explicit Estimation Search [7] to choose which node to expand next.

Another notable example of a search-based sub-optimal algorithm is Hierarchical Cooperative A* (HCA*) [8]. In this algorithm, time is divided into slots, meaning the time intervals during which agents can occupy nodes are constrained. The path of each agent is planned sequentially, following a predefined order. Once a path is found, it is stored in a matrix called the reservation table, which records the time slots during which each node is occupied. When searching for a path for the next agent, the algorithm consults the reservation table to determine whether the node it is exploring is free or occupied by other agents.

While search-based sub-optimal solvers reach solutions that have a good quality but are not complete, rule-based approaches favor completeness at low computational cost over solution quality. These algorithms, instead of searching for a solution, follow a set of movement rules specific for different scenarios that take each agent to its target.

Two major existing rule-based algorithms for MAPF are Push-and-Swap [9] and Bi-BOX [10, 11] (which is effective for biconnected graphs).

BiBOX The BiBOX algorithm assumes input MAPF where the underlying graph $G = (V, E)$ is bi-connected and there are exactly $|V| - 2$ agents (for fewer agents either fake agents can be used to fill up the graph or the algorithm can be slightly modified to deal with more vacant vertices). The rules of the BiBOX algorithms via which it moves agents into their goal positions significantly depend on the properties of bi-connected graphs.

A graph $G = (V, E)$ is bi-connected if, for any two distinct vertices $u, v \in V$, there exist two disjoint paths that connect u and v (alternatively, we can say that u, v belongs to a cycle in G). For graphs fulfilling this definition, it holds that a so-called *ear decomposition* can be applied to them. That is, each bi-connected graph $G = (V, E)$ can be constructed starting with an initial cycle by adding so-called *ears* to the currently constructed graph. Assume to have $G_i = (V_i, E_i)$ at hand, a graph at the i -th step of the ear decomposition, adding an ear is represented by introducing a path H_i consisting of fresh internal vertices, say $u, u_1, u_2, \dots, u_{n_i}, v$ where $n_i \in \mathbb{N}_0$ and $u, v \in V_i$, that is we have $G_{i+1} = (V_{i+1}, E_{i+1})$, where $V_{i+1} = V_i \cup \{u_1, u_2, \dots, u_{n_i}\}$ and $E_{i+1} = E_i \cup \{\{u, u_1\}, \{u_1, u_2\}, \dots, \{u_{n_i}, v\}\}$. In Figure 6 it is shown a bi-connected graph in which we can find an initial cycle C and three ears H_1, H_2, H_3 .

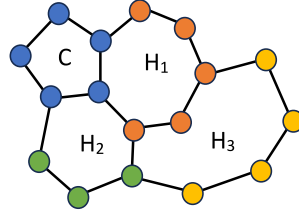


Figure 6: An illustration of the ear decomposition with the initial cycle C and three ears H_1, H_2 , and H_3 .

The BiBOX algorithm first determines an ear decomposition of the input graph G . Let us denote this ear decomposition $C, H_1, H_2, \dots, H_m$, where C is the initial cycle and H_i are the individual ears. The time complexity of determining the ear decomposition is $O(n^2)$ where $n = |V|$ [].

Then, the algorithm proceeds according to the ear decomposition, starting with the last ear H_m , into which agents are moved to their goal positions. Once all agents reach their goals in H_m , they will no longer move, and the last ear H_m can simply be ignored in the following stages of the BiBOX algorithm.

Moving agents into their goal positions within the ear H_m is done in a stack-like manner. Assume that $H_m = \{u, u_1, u_2, \dots, u_{n_m}, v\}$. We start with an agent whose goal is in u_{n_m} and move it u to u_1 and then rotate the ear once so that the agent appears in u_1 . Similarly, we continue with an agent whose goal is in u_{n_m-1} . Again, move the agent to u and rotate the ear once. We continue in the same way with the remaining agents whose goals are in H_m . If the agent we need to move into u already resides in H_m , we need to rotate it out of the ear and rotate the ear back to restore the situation. Eventually, we have all agents in their right positions in the ear.

Then the algorithm proceeds in the same way with the ears H_{m-1} , H_{m-2} , ... until the initial cycle C remains. For all the above processes, only one empty vertex is necessary. The second empty vertex will be used later.

At this stage, all agents with their goals in C can be regarded as a permutation that may differ from the goal permutation represented by agents' goal positions in C . BiBOX now utilizes two empty vertices to swap pairs of agents in C using a vertex from H_1 as a parking place. By being able to swap the agents, the goal permutation can be reached.

Push-and-Swap The Push-and-Swap algorithm is a method of solving MAPF in a sub-optimal yet efficient and complete manner. The algorithm consists of two basic primitives - one of them being Push and the other one being Swap. In Push, an agent moves along its shortest path to its goal location and, while doing so, pushes agents blocking its path out of their way, forcing them to clear a vertex. In some cases, though, a simple push may not be possible, and thus, the second primitive Swap must be employed. In the Swap primitive, two neighboring agents swap their positions. This requires the agents to move to a part of the graph where the swap is possible. A suitable part of a graph is a vertex with a degree of at least 3 and its neighbor. This vertex has to have at least two free neighboring vertices. If this is not fulfilled, the

algorithm may try to free the neighbors of agents occupying them. After the swap takes place, all agents are back to their original positions by reversing the sequence of moves before the swap has been made. If the swap is not possible to perform, the MAPF instance is deemed unsolvable for Push-and-Swap.

Push and Rotate Another important rule-based algorithm for MAPF on undirected graphs is *Push and Rotate* [12, 13], which solves every MAPF instance on graphs that contains at least two holes (i.e., unoccupied vertices). This algorithm can be seen as an extension of Push-and-Swap. The Push and Rotate consists of two different phases. In the first phase, it divides the graph into sub-graphs, assigns agents to them, and determines the priority in which the paths have to be planned. The sub-graphs are defined, ensuring that only agents assigned to the sub-graph can swap positions. In the second phase, the algorithm, as already seen in Push-and-Swap, tries to move the chosen agent along its shortest path. If it encounters other agents, then the action to be taken depends on the priority of the two agents. If the agent encountered has a lower priority, we can push this agent out of the way to an empty vertex. If this does not work, or if the agent has a higher priority, the algorithm tries to swap the positions of the agents. After the swap operation, the algorithm returns the higher-priority agent to its destination. In the Push and Rotate algorithm, we solve this problem by detecting whether there is a cycle of agents that want to move forward. If so, all these agents are advanced one step with the rotate operation.

Pebble Motion on Trees

Many of the rule-based algorithms deal with the special case of trees. In this case, the problem is called *pebble motion on a tree* (PMT). Among those who studied this problem, Auletta et al. [14] present an algorithm for the feasibility of PMT, from which a solution can be derived requiring $O(k^2(n-k))$ moves, where n is the number of nodes and k the number of pebbles.

Another easy algorithm that solves PMT in polynomial time is introduced by Khorshid et al. [15]. They present an algorithm for PMT (called **TASS**), easy to understand

and implement, which provides solutions of $O(n^4)$ moves. This algorithm explores the tree, starting from the leaves and working inward. When it finds a goal state that is not occupied by its target agent, it solves the motion planning problem of moving the agent to its goal node. The motion planning problem is solved by moving the agent along the path to the goal and swapping it with other agents when necessary. The algorithm explores in this manner all the trees, moving all the agents in their target nodes.

Recently, Ardizzoni et al. [Pub1] propose the **Leaves Procedure for PMT**, simple and very easy to implement, which finds solutions with a number of moves $O(knc + n^2)$, where c represents the maximum length of the corridors (paths whose internal nodes all have degree equal to two and whose end nodes have degree different from two) in the tree. This algorithm first moves the agents to a set of intermediate targets. To do this, the algorithm chooses an agent and moves it to a target leaf, solving a motion planning problem. Then, it removes the agent and the node from the tree and repeats the procedure with another agent. Lastly, each agent is taken from their intermediate target to their real target.

One of the best rule-based algorithms for MAPF in terms of length complexity is presented by Kornhauser. Indeed, in his Ph.D. thesis [16], he describes a procedure that finds solutions with $O(n^3)$ moves. However, this approach is very difficult to implement [17]. In all the problems considered thus far, we assume that each agent is assigned a specific target. However, there exists a variant known as the unlabeled problem, where each agent is allowed to reach any of the available targets, and it is only required that all targets are reached by exactly one agent. For instance, Le Bodic [18] addresses the unlabeled pebble motion problem on trees (UPMT) and introduces the first optimal algorithm, which is asymptotically as efficient as possible. The algorithm operates in time linear with respect to both the size of the input (the tree) and the size of the output (the optimal plan). Furthermore, it demonstrates that the length of the optimal solution is less than or equal to $k(n - 1)$.

MAPF on strongly connected directed graphs

A more unexplored MAPF problem is the one on strongly connected directed graphs. In directed graphs, the edges are ordered pairs, and the agents can travel on an edge only in the specified direction. Moreover, a graph is strongly connected if and only if it is possible to reach any node starting from any other node. Nebel [19] proves that finding a feasible solution of MAPF on a general directed graph (digraph) is NP-hard. However, in some special cases, this problem can be solved in polynomial time.

One relevant reference is the **diBOX**, introduced by Surynek [20], which solves MAPF on the specific class of biconnected digraphs, i.e., strongly connected digraphs where the undirected graphs obtained by ignoring the edge orientations have no cutting vertices. The proposed algorithm has polynomial complexity with respect to the number of nodes. This algorithm is the equivalent of BiBOX [10], already explained, for directed graphs.

Anytime MAPF algorithms

Generally, rule-based algorithms provide solutions that are much longer than the shortest one. This is a crucial limitation in industrial applications. For this reason, another class of algorithms, combining optimal and sub-optimal solvers, the class of anytime MAPF algorithms, is of particular relevance. Approaches in this class aim at first detecting quickly a feasible solution and then at improving it through suitable procedures. Among them, those based on Large Neighborhood Search (LNS) [21] are particularly well-known. LNS is a popular local search technique to improve the solution quality for combinatorial optimization problems. Starting from a given solution, it removes part of the solution, called a neighborhood, and treats the remaining part of the solution as fixed. Then, it repairs the solution and replaces the old solution if the repaired solution is better. This procedure is repeated until some stopping criterion is met.

For instance, MAPF-LNS [22] is an anytime MAPF framework that improves the quality of a solution obtained from a MAPF algorithm over time by re-planning sub-groups of agents, called neighborhoods, using LNS. In [22], the number of agents to

be re-planned is a predefined parameter N , and three different rules to select them are proposed: one based on agent paths, another based on graph topology, and, finally, a random one. In the first case, an agent is selected whose path is not optimal, and the algorithm performs a random walk to select a set of agents that are blocking each other. The first rule chooses agents that visit the same nodes with a degree greater than 2.

We also mention [23], where algorithm LNS2 is introduced. Such an algorithm starts from an infeasible solution and repeatedly replans subsets of agents to find plans with fewer conflicts, eventually converging to a conflict-free plan.

Outline of the Thesis and Contributions

In this thesis, we address the MAPF problem and try different approaches to find solutions that are both of high quality and have computational times that grow polynomially with the number of nodes and agents. Specifically, we focus on MAPF problems on both directed and undirected graphs, introducing novel approaches to find sub-optimal solutions using rule-based algorithms. Additionally, we describe an algorithm that uses dynamic programming to improve the quality of a given solution. The thesis is divided into three chapters.

The first chapter focuses on MAPF problems on undirected graphs and refers to the work in [Pub2]. We propose a new two-step algorithm based on two rule-based algorithms: BiBOX and Push-and-Swap. Previous studies suggest that these algorithms are effective for graphs densely populated with agents. On the other hand, they do not fully exploit the structure of the underlying graph. In this chapter, we aim to better utilize the graph's structure while maintaining the benefits of rule-based algorithms in dense scenarios. We propose hierarchical variants of BiBOX and Push-and-Swap that first decompose the input graph into highly connected components and then apply the specific rule-based algorithm to the components. This method provides two main advantages: first, the higher connectivity of the components enhances the efficiency of the MAPF algorithms, and second, the algorithms can run in parallel across individual components. To identify suitable components, we use Spectral Cluster-

ing [24], a numerical method that relies on eigenvalue computations from the graph's adjacency matrix. The advantage of numerical methods for graph clustering is that we can easily fine-tune the output clustering via the change of numeric parameters of the clustering algorithm to fit our needs. These parameters can be, for example, the number of clusters or the parameters used in the chosen clustering algorithm.

The second chapter examines MAPF problems on strongly connected digraphs, referencing works in [Pub3], [Pub4], and [Pub1]. We describe a procedure based on [25] that checks the feasibility of the problem in linear time with respect to the number of nodes. This approach can also be used to solve MAPF, but it tends to generate redundant solutions. We also present a necessary and sufficient condition for MAPF feasibility on strongly connected digraphs, adapted from [16] and [Pub1]. Finally, we introduce the diSC (digraph Strongly Connected) algorithm, which finds a solution for all admissible problems with at least two holes. This algorithm transforms the MAPF problem on a strongly connected digraph into a MAPF problem on a tree (PMT), which is then solved using the algorithm in [Pub1]. The solution is then converted into a solution on the digraph. This algorithm is proved to have a length complexity of $O(kn^2c)$.

The third chapter introduces new methods to shorten a given feasible sub-optimal solution for MAPF instance, based on [Pub5] and [Pub6]. We propose an iterative local search approach where neighborhoods are explored through dynamic programming. Starting from a feasible sub-optimal solution, we define a neighborhood and search for better solutions within it via dynamic programming. If a better solution is found, the process is repeated; otherwise, we stop, and the current solution is a local minimizer with respect to the given neighborhood. Given a distance parameter r and a function measuring the distance from the reference solution, the neighborhood is the set of solutions whose distance from the reference solution is not larger than r . In this chapter, we propose several different ways of measuring distances. These can be divided into two sets: the first one takes into account the distance between nodes on the graph, and the other is based on the set of agents that vary their path between two different solutions. Choosing different distances leads to different results. We also introduce an algorithm that alternates between these metrics during the iterative search.

The fact that our search is *local* is crucial for reducing time complexity. While dynamic programming could solve the MAPF problem globally, the state space grows exponentially with the number of agents, making it impractical for large instances. The introduction of a locality constraint allows solving the (local) problem through dynamic programming in a time that grows only polynomially with respect to the number of agents.

Chapter 1

Spectral Clustering approach for MAPF on Connected Graphs

In this chapter, we focus on Multi-Agent Path Finding problems on undirected graphs, and we will be referring to the work done in [Pub2]. As mentioned in the introduction, finding an optimal solution to this problem is NP-hard. Therefore, we focus on sub-optimal, rule-based algorithms. These algorithms are complete and tend to find solutions more quickly than search-based algorithms when managing a large number of agents.

In this chapter, we introduce two new algorithms based on existing rule-based algorithms that try to analyze and use the structure of the graphs. These algorithms combine rule-based algorithms for MAPF with spectral clustering, a numeric method that uses the calculation of eigenvalues of the graph's adjacency matrix to analyze its structure. The spectral clustering method is used to decompose the input graph into highly connected components that act as local sub-graphs for final agent reconfiguration. This introduces a hierarchical process of solving the MAPF problem.

The first new algorithm is a hierarchical version of BiBOX:

1. Spectral Clustering is used to partition the graph into clusters;
2. each cluster is further broken down into bi-connected components (though

16 Chapter 1. Spectral Clustering approach for MAPF on Connected Graphs

spectral clustering is fine-tuned so that this step is minimal, with clusters being nearly bi-connected);

3. agents are moved to their goal bi-connected components using a modified Push-and-Swap algorithm;
4. the final agent configuration within each bi-connected component is completed using BiBOX.

We refer to this algorithm as **Hierarchical BiBOX** or **ChiBOX** for short.

The second algorithm is a simple variation of the first. Instead of constructing the bi-connected components, we directly use Push-and-Swap to achieve the final configuration within each cluster. We will call this algorithm **Hierarchical Push-and-Swap** or **Chi-Push-and-Swap** for short.

1.1 Spectral Clustering and Bi-connected Components

This section explains the algorithm used to break the graph into components. The algorithm consists of two different steps. The first one uses Spectral Clustering to divide the graph's vertices into highly connected clusters. The second one takes those clusters and finds their bi-connected components.

1.1.1 Spectral Clustering

The Spectral Clustering algorithm uses the affinity matrix computed on the data to reduce the dimension of the space and then clusters the data using a chosen clustering algorithm [26]. In this chapter, we use the Spectral Clustering algorithm to identify highly connected sub-graphs of the initial graph. To do so, we use the adjacency matrix of the graph as the similarity matrix of the data [27].

Traslating the data The first step of the Spectral Clustering algorithm aims at performing a dimensionality reduction of the data. In our case, the first step uses the information on the graph inside the adjacency matrix to extract data on the structure

of the graph that will be usable by a clustering algorithm. To do so, we compute the normalized Laplacian matrix of the graph and its eigenvalues and eigenvectors. Then, the eigenvectors are sorted according to the eigenvalues in descending order. After finding the desired number of clusters k , we create the matrix U where the columns are the first k eigenvectors. The result will be a matrix $U \in \mathbb{R}^{n \times k}$ where n is the number of vertices on the graph and every vertex is described using k dimensions. In figure 1.1, we can see the vertices of a 30x30 grid graph described using the first 3 dimensions, which are the eigenvectors corresponding to the 3 largest eigenvalues.

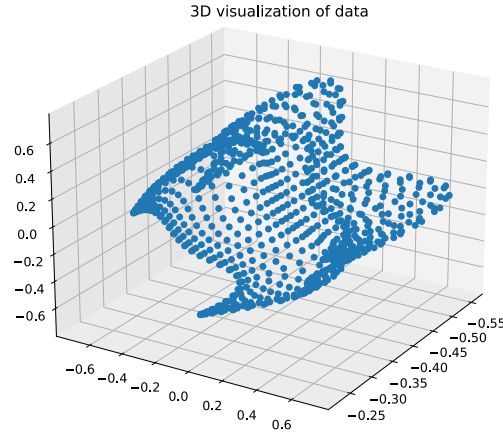


Figure 1.1: 3D representation of data extracted from a 30x30 grid graph.

Clustering and choosing k Having described the graph through n points in k dimensions, we can now use a clustering algorithm to group vertices into clusters. To do so, we can use different clustering algorithms, such as K-means or Hierarchical Clustering algorithms. As said in the last paragraph, to choose the eigenvectors and create the matrix U , it is necessary to choose the number of clusters k . We decided to use a heuristic called the Eigengap Heuristic to find a k that is close to the best number of clusters. Given the eigengap measures, which are the difference between two consecutive eigenvalues,

$$\delta_i = \lambda_i - \lambda_{i+1} \quad (1.1)$$

18 Chapter 1. Spectral Clustering approach for MAPF on Connected Graphs

k can be estimated as the argument of the highest eigengap:

$$k = \operatorname{argmax}(\delta_i) \quad (1.2)$$

In figure 1.2, the grid graph represented in figure 1.1 is divided into 7 clusters by the K-means algorithm. Algorithm 1 explains more clearly the steps followed to build the clusters.

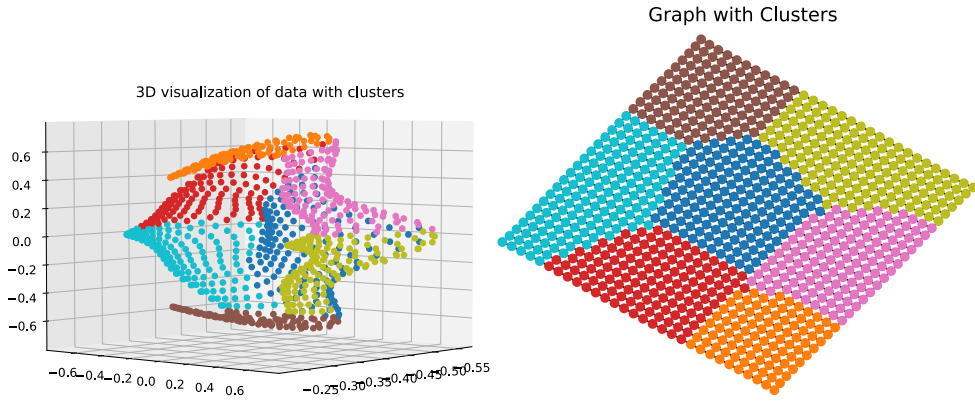


Figure 1.2: 3D representation and clustering of a 30x30 grid graph.

Building the Clusters Graph After finding the clusters on the graph, we can build a new graph where each vertex represents a cluster and each edge $e_{i,j}$ indicates if at least two vertices v_i and v_j , respectively, in the two different clusters i and j are connected.

In figure 1.3, the cluster graph is depicted, with colors matching clusters of figure 1.2.

Algorithm 1 Spectral Clustering

Input: A = adjacency matrix
 Output: clusters, c
 $L \leftarrow \text{NormalizedLaplacian}(A)$
 $\lambda \leftarrow \text{eigenvalues}(L)$
 $v \leftarrow \text{eigenvectors}(L)$
 $v, \lambda \leftarrow \text{sort}(v, \lambda)$
 $k \leftarrow \text{maxGap}(\lambda)$
 $U \leftarrow$ matrix where the columns are the first c eigenvectors
 $X \leftarrow$ normalized rows of U
 clusters $\leftarrow$ K-means(X, c)
return clusters, c

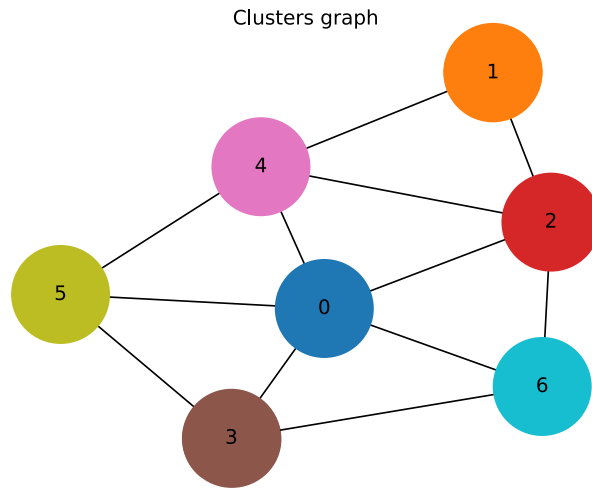


Figure 1.3: Cluster Graph a 30x30 grid graph.

Theorem 1.1. *The total time complexity of spectral clustering is $O(n^3) + O(cnT)$.*

Proof. The time complexity of finding the eigenvalues of the Laplacian is $O(n^3)$ where $n = |V|$. The time complexity of K-means is given by $O(cnT)$ where n is the

number of samples (again $n = |V|$ in our case), T is the number of iterations, and c is the number of clusters. ■

1.1.2 Bi-connected Components Graph

In this part of the algorithm, the Bi-connected Components Graph is built.

Searching Bi-connected components After dividing the graph into k clusters, each cluster is divided into its bi-connected components. To do so, we perform a depth-first search on the cluster. Figure 1.4 represents the different bi-connected components of the graph in figure 1.3.

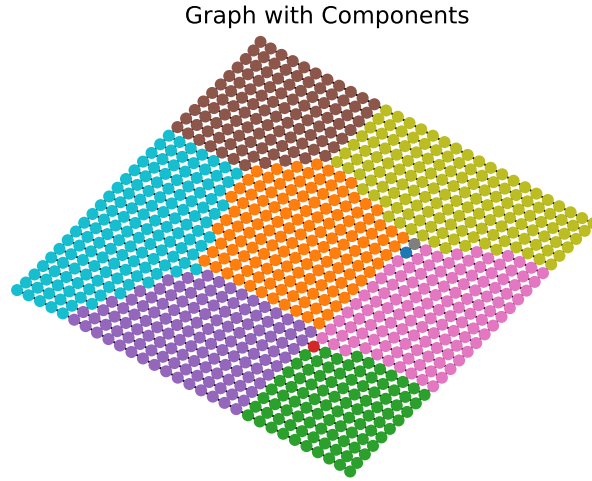


Figure 1.4: Bi-connected components of a 30x30 grid graph.

Building the Components Graph Having found the different bi-connected components, we can now build the graph, which we call the Components Graph, that is used in the first part of the final algorithm. Each vertex of the Components Graph represents a bi-connected component, and an edge (b_i, b_j) exists if and only if two vertices $v_i \in b_i$ and $v_j \in b_j$ of the original graph are adjacent, that is if it exists the edge (v_i, v_j) in the original graph.

Theorem 1.2. *The total time complexity of the building of the bi-connected component graph is $O(n^2)$.*

Proof. The search of bi-connected components on a cluster i is achieved using the Depth First Search algorithm that has the complexity of $O(n_i)$ where n_i is the number of vertices of the cluster, so the search of bi-connected components in all the clusters has the total complexity of $O(n)$, where $n = |V|$ the number of vertices in the original graph G . Searching the edges of the new graph has a complexity that is $O(n^2)$. Therefore, this part of the algorithm has a complexity that is $O(n^2)$. ■

In figure 1.5, we can see the bi-connected component graph of a 30x30 grid graph. The color of each vertex shows the cluster of each bi-connected component.

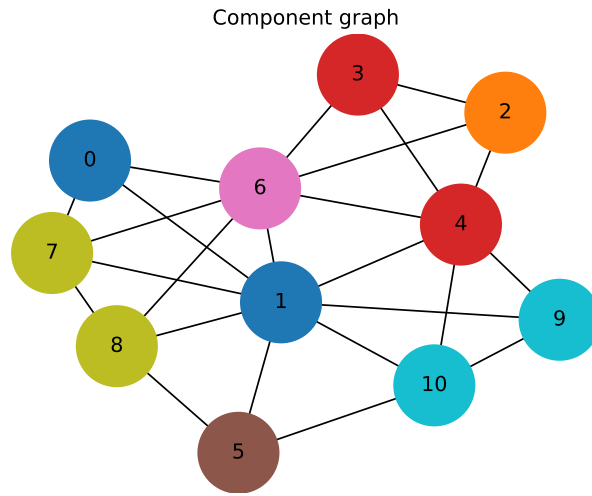


Figure 1.5: Bi-connected component graphs of graphs in figure 1.4.

The overall complexity of this part of the algorithm, in which we analyze and prepare the graph for the next part, is $O(n^3)$, as we know that the first part, as said in Proposition 1.1, has a complexity of $O(n^3)$ and the second part, as said in Proposition 1.2 has a complexity of $O(n^2)$.

1.2 Combining Spectral decomposition, Push-and-Swap and BiBOX

The output of Spectral Clustering consists of two parts. The first is a graph where each vertex represents one cluster - one bi-connected graph. This graph serves as input for the Push-and-Swap algorithm [9], which acts as a device to move agents between clusters. The second part of the output is a list of all decomposed clusters. These sub-graphs are bi-connected graphs connected to at least one other bi-connected sub-graph.

In Push-and-Swap, multiple agents can share a start vertex and a goal vertex. This is because all vertices represent sub-graphs between which agents move. A goal of an agent in Push-and-Swap is a particular vertex representing a sub-graph but without a concrete position in that sub-graph. The purpose of this algorithm is to get all agents into their goal sub-graphs.

This setting then serves as an input for the BiBOX algorithm [10]. Multiple instances of BiBOX have to be run for each of the n vertices resulting from Spectral Clustering. By decomposing the original graph into disjoint sub-graphs, it is possible to work on a reduced part of the problem at a time, and it is not necessary to consider all of the agents in the original graph.

By dividing a graph into several smaller disjoint instances, we reduce not only the graph itself but mainly the number of agents traversing this graph.

In a densely occupied graph, agents may traverse longer paths to avoid collisions with obstacles presented by other agents. Dividing that graph not only reduces staging necessary for planning paths for many agents at once but also reduces operations agents perform to get in their target loop, as this has already partially been performed by Push-and-Swap.

Disjoint BiBOX instances also offer the possibility of parallelization of this part of the algorithm, as no traversions of agents between these instances are further necessary.

One of the rules of BiBOX is that at least two vertices of its input graph have to be unoccupied [10]. For n decomposed bi-connected sub-graphs, this means that in total, there have to be $2n$ free vertices in the original graph. This algorithm is, therefore,

usable for $|V| - 2n$ agents, where $|V|$ is the number of vertices in the original graph, and n is the number of decomposed components.

If a sub-graph resulting from spectral decomposition contains only 1 or 2 vertices, this sub-graph is by definition [10] not bi-connected. In this setting, however, it can be treated as a trivial case if no agent has this component as its goal. In case of one or two vertices being an entire sub-graph, it is necessary to keep them free as entrance nodes for the Component-Push operation to be possible. In case of one vertex present, it doubles as both an entrance and an exit vertex.

The advantage of spectral clustering is that it has various numeric parameters through which fine-tune the output clustering for specific MAPF algorithms. Such fine-tuning is difficult with the standard graph search algorithms.

1.3 Push-and-Swap modification

To serve this setting more effectively, Push-and-Swap was modified in the following ways.

Firstly, individual vertices in the cluster graph represent bi-connected components - sub-graphs of the original graph G , not single vertices. As agents reach their goal vertex, they can be ignored as they move further into their respective sub-graph (cluster). This makes the resolve operation, in which agents that have previously reached their goal and were moved due to pushing and swapping other agents, had to be placed back at their goal positions unnecessary. As we can now ignore agents at their goal position during the whole course of the algorithm, agents at their goal will never be moved to a different sub-graph (cluster).

Different sub-graphs are connected by one vertex of each respective sub-graph. Let's call this vertex the entrance vertex. Entrance vertices have to be freed from agents at their goal sub-graphs for agents to pass between sub-graphs without a collision. Each sub-graph is a bi-connected graph and, as per BiBOX's precondition, has to contain at least two free vertices [10]. Therefore, maintaining the entrance vertex free from completed agents should be possible at all times if there are less than two agents present who do not have this component set as their goal component.

24 Chapter 1. Spectral Clustering approach for MAPF on Connected Graphs

Algorithm 2 Operation Component-Push

```
Input: a = agent number,  
C = component,  
p = agent position,  
u = used vertices  
Output: True if operation was successful  
neighbour  $\leftarrow$  get-neighbour(C, p, u, a.goal)  
u  $\leftarrow$  u  $\cup$  neighbour  
if occupying-agent(neighbour)  $\neq \emptyset$  then  
    if  $\neg$ component-push (occupying-agent(neighbour), C, neighbour, u) then  
        return False  
    end if  
end if  
assignment[a]  $\leftarrow$  neighbour  
path  $\leftarrow$  path  $\cup$  assignment  
return True
```

To achieve this, we propose a new function, Component-Push. In this function, all the agents in a respective sub-graph (cluster or bi-connected) are pushed further into that sub-graph so that an agent occupying the entrance vertex will free it while not moving away from its goal sub-graph. As this operation does not aim to move agents to their goal positions inside their sub-graph, a push operation is enough to clear the entrance vertex. All of the agents consider their neighboring vertex in the opposite direction than the entrance vertex as their goal. Therefore, in this operation, all of the agents present in the sub-graph move by one vertex. This way, they prepare their initial positions for the BiBOX algorithm, which commences after all of the agents have reached their goal sub-graphs.

Algorithm 2 describes the recursive operation Component-Push. The input of this function consists of an agent number belonging to agents to be pushed, the component - sub-graph where this is taking place, the position of an agent that is to be pushed,

Algorithm 3 Get neighbor

Input: g = goal vertex, C = component, p = agent position, u = used vertices

Output: Neighbor vertex if found, -1 otherwise

```

for  $e \in C.edges$  do
  if  $e.vertex_2 == p \wedge e.vertex_1 \notin u \wedge dist(p, g) \geq dist(e.vertex_1, g)$  then
    return  $e.vertex_1$ 
  end if
  if  $e.vertex_1 == p \wedge e.vertex_2 \notin u \wedge dist(p, g) \geq dist(e.vertex_2, g)$  then
    return  $e.vertex_2$ 
  end if
end for
return -1

```

and a list of vertices that have already been used to push.

Firstly, all neighbors of a 's vertex that fulfill requirements (they have not yet been used to push) are found. The current position of a is then added to u . Then, the push itself is performed. If the neighboring vertex is not free, it is to be freed with another Component-Push. If this sequence of actions is not possible to perform, *False* is returned. After freeing *neighbour*, the position of a is changed to it. List *assignment* stores information about the latest positions of all agents. Finally, the new assignment is added to *path*, which represents all paths of all agents from the beginning of the algorithm to the current step.

Agents in their starting arrangement are located in a component that may differ from their goal component. To move to a different component, they can also operate Component-Push, only they now have their goal position set as their entrance vertex. As if they were to push themselves further into the component, they push other agents only around that respective component, never between components.

This version can also possibly take advantage of these individual components when a graph without at least one vertex of degree of at least 3 is presented. It can perform the swap by temporarily using a vertex of a component with a free vertex.

Function *get-neighbour*($C, p, u, a.goal$) shown in line 1 of algorithm 2 is shown in algorithm 3. Its purpose is to find a neighboring vertex for the agent on position p in operation Component-Push, excluding vertices in u , which have already been used for the push. The neighboring vertex that is returned should have a lower or the same distance from a goal vertex g than position p . If no suitable vertex is found, -1 is returned, and the push cannot be performed.

Theorem 3.1. *The time complexity of the ChiBOX algorithm and the Chi-Push-and-Swap algorithm is $O(n^3)$.*

Proof. The time complexity of building clusters in the spectral clustering is dominated by the time complexity of the underlying rule-based MAPF algorithms, BiBOX or Push-and-Swap, that have the time complexity of $O(n^3)$ for $n = |V|$. ■

1.4 Experimental Results

In this section, we comment on preliminary experiments that we obtained with our implementation of ChiBOX and Chi-Push-and-Swap in Python.

1.4.1 Spectral Clustering and Bi-connected Components

The algorithm has been tested on two different types of graphs, and the average vertex connectivity on every bi-connected component has been evaluated.

Construction of graphs

The spectral clustering and the construction of the bi-connected component graph have been tested on two special kinds of generated graphs: random connected graphs and trees of bi-connected components.

Random Connected Graphs To create random connected graphs, an empty graph is created in *NetworkX*, and then random edges are added until the graph is no longer disconnected. In Figure 1.6, an example of this kind of graph is shown.

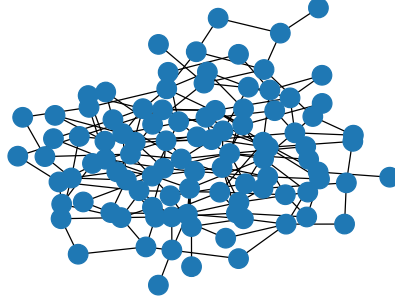


Figure 1.6: Random connected graphs with 100 vertices.

Trees of Bi-connected Components To create trees of biconnected components, we use a procedure also described in [Pub3] but for undirected graphs. First, we create a randomly connected undirected graph using *NetworkX* library. Then, we compute its spanning tree and transform each group of vertices with a father and more than one child into a biconnected component. In Figure 1.7 it is shown an example of a tree of bi-connected components with 100 vertices.

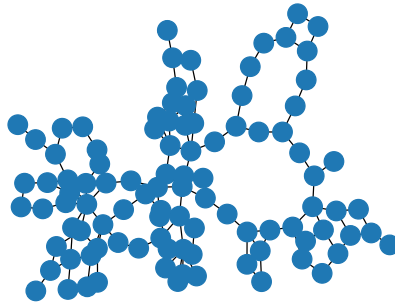


Figure 1.7: Random tree of bi-connected components with 100 vertices.

During tests, an upper and a lower bound on the number of clusters is imposed. The bounds are then adjusted if the resulting graph does not satisfy certain requirements, such as the connectivity of the cluster or the number of clusters.

Connectivity Results

For each kind of graph, we tested the algorithm using two different clustering algorithms inside Spectral Clustering to study how different types of clustering algorithms change the results of our algorithm. The first algorithm used is the K-means algorithm, which has the downside of relying too much on the initialization of the centroids. The second algorithm used is Agglomerative Clustering, a type of Hierarchical Clustering. To evaluate the performance of the algorithms, the average vertex connectivity of each biconnected component has been found, using the *NetworkX* function *average_node_connectivity()*, which finds the average of local vertex connectivity over all pairs of vertices. This is computed as the number of vertices that must be removed from the graph to disconnect a pair of vertices.

Random Connected Graph In figures 1.8 and 1.9, the mean connectivity for every number of vertices is shown using random connected graphs. We can see that the Agglomerative Cluster has better results in terms of the mean connectivity of the bi-connected components.

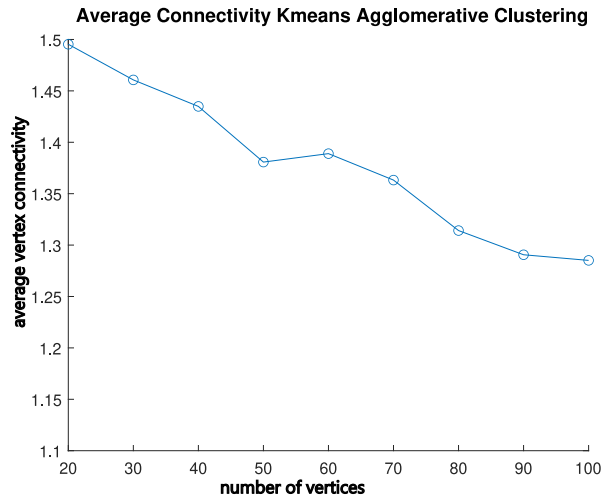


Figure 1.8: Connectivity with Agglomerative Clustering.

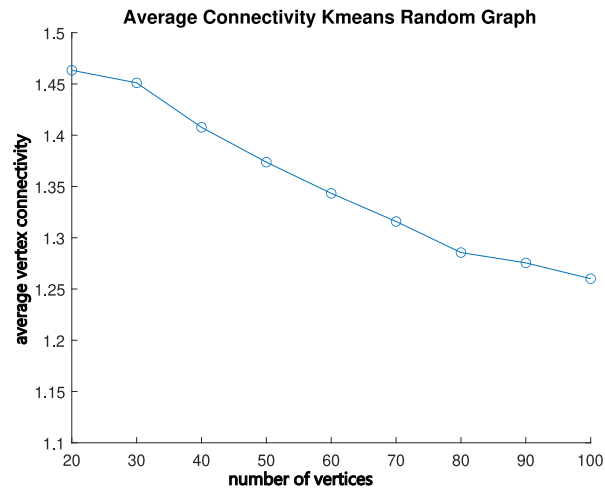


Figure 1.9: Connectivity with K-means.

Trees of Bi-connected Components Figures 1.10 and 1.11 represent the mean connectivity for every number of vertices of the bi-connected components on trees of bi-connected components. Even in this case, Agglomerative Clustering has better results than the K-means algorithm.

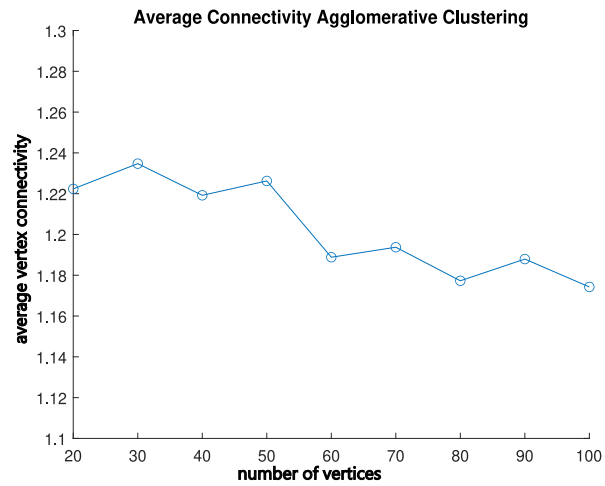


Figure 1.10: Connectivity with Agglomerative Clustering.

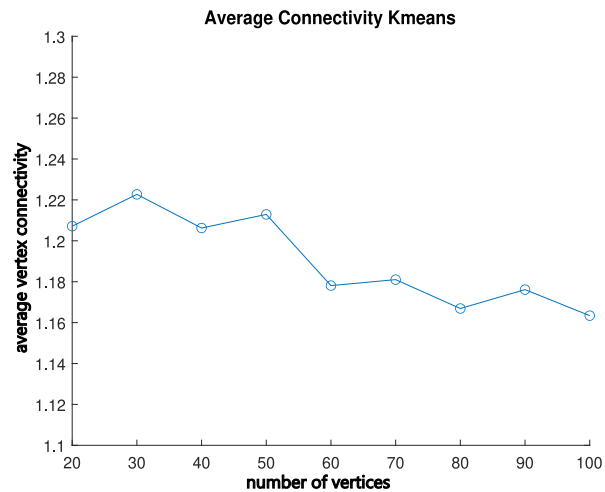


Figure 1.11: Connectivity with K-means.

1.4.2 Comparing BIBOX with Chi-Push-and-Swap and ChiBOX

Preliminary experiments have shown that the main advantage of our approach lies in the safe parallelization of BiBOX on individual sub-graphs. As these bi-connected sub-graphs are disjoint, agents in different sub-graphs can move in parallel and not wait after an agent in a different sub-graph has taken a step (although this is not possible in Chi-Push-and-Swap, where agents move sequentially). On a simple 12-vertex bi-connected graph decomposed into 4 clusters (4 bi-connected components), each having 3 vertices, 4 agents found paths using Chi-Push-and-Swap and then parallel ChiBOX in 13 steps, while in the original BiBOX, a sequential path was found with 15 steps. Therefore, Push-and-Swap and parallel ChiBOX are able to find a shorter path compared to BiBOX. This graph is shown in figure 1.12.

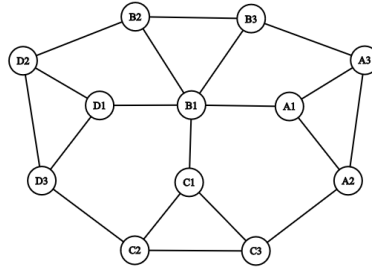


Figure 1.12: A graph used for the experiment. The graph contains 4 clusters: A, B, C, D.

As the same clusters may, in reality, be connected by more than one edge in different vertex, redundant edges are ignored in the input of Chi-Push-and-Swap. Preferable edges to preserve are those that connect higher-degree vertices in the original graph. Another advantage of this approach is the way loops are created. In the original graph, 7 loops were created for BiBOX, including the original cycle. However, due to the spectral decomposition, only 4 loops were necessary - one original cycle per bi-connected component. Therefore, this approach may aid in simplifying loop decompositions.

Another example of this transformation of a problem into several smaller sub-problems

32 Chapter 1. Spectral Clustering approach for MAPF on Connected Graphs

is shown in figures 1.13 and 1.14. A graph with 16 vertices in a grid pattern would have 7 loops created if using only BiBOX (figure 1.13 shows this case, each letter represents one loop), while when decomposing into 4 subgraphs, only the main loop is necessary for each one of them, making it possible to run 4 parallel instances of Chi-BOX, each on 1 loop only. This is shown in figure 1.14, where subgraphs are labeled A, B, C, and D.

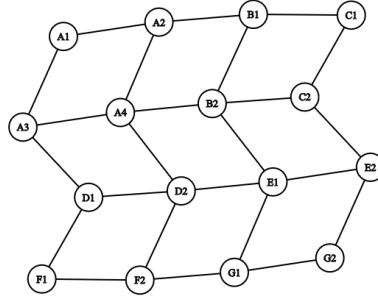


Figure 1.13: A graph divided into 7 loops distinguished by letters A-G, which is to be solved by the original BiBOX algorithm.

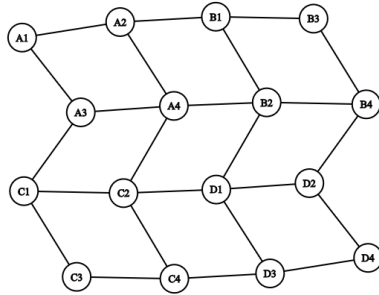


Figure 1.14: The same graph as in figure 1.13, but after spectral decomposition, it is divided into four clusters A, B, C, and D, on which four parallel instances of ChiBOX, each dealing with one loop, could be run.

Cases have been observed where the resulting decomposition contains one or more

subgraphs with only two vertices, with one of them being already a part of a different subgraph, while the other one is present only in that subgraph. If this was not modified, it would endanger the parallelization, as a vertex would be shared between parallel instances of Chi-BOX. Similarly, an agent in Chi-Push-and-Swap could potentially have more than one target component. However, it is possible to exclude the repeating vertex from the subgraph, which only contains two vertices, and thus create a subgraph containing only one vertex. An example of a decomposition like this is shown in figure 1.15. Its modification to be suitable for Chi-BOX and Chi-Push-and-Swap would divide two vertex components containing vertices 5 and 14 (shown in pink color), with vertex 5 already being a part of a different component. The new component would contain only vertex 14, and vertex 5 would be a part of only one larger component (shown in blue color).

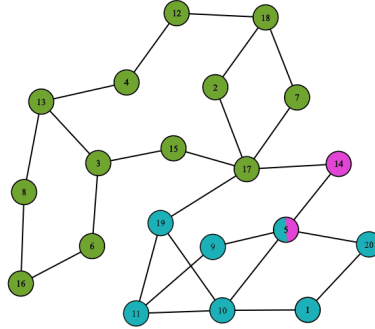


Figure 1.15: A graph decomposed into three components, with vertex 5 belonging to two components at once.

Chapter 2

Feasibility and Algorithms for MAPF on Strongly Connected Digraphs

The second chapter refers to the work in [Pub3], [Pub4] and [Pub1], and focuses on the MAPF problem on strongly connected directed graphs. In these graphs, it is possible to reach any node starting from any other node.

In this chapter, we formally define the problem and demonstrate that its feasibility can be determined in linear time. We then present two approaches for solving MAPF instances on strongly connected directed graphs. The first approach reduces the MAPF problem on strongly connected digraphs to the case of undirected graphs. While this strategy is easy to implement, it often results in redundant solutions. The second approach reduces MAPF on strongly connected digraphs to a different problem known as Pebble Motion on Trees (PMT). Specifically, we introduce an algorithm called diSC, which converts MAPF into a variant of PMT and then maps the obtained solution back onto the original graph. We will also prove that the length complexity of the diSC algorithm is linear with respect to the number of agents and polynomial with respect to the number of nodes. Finally, we present the experimental results of the diSC algorithm, tested on random graphs and on graphs representing warehouse

layouts.

It is important to note that this chapter addresses a subcategory of MAPF problems where it is assumed that agents move one at a time. Testing feasibility for these problems is equivalent to testing the feasibility of MAPF with forbidden *following conflicts*, which occur when one agent moves to a vertex that another agent occupied in the previous time step (see, e.g., [2]). In particular, any feasible solution to problems in this subcategory is also a feasible solution for MAPF. While such feasible solutions are typically far from optimal, they can be improved, for instance, by employing local search procedures that allow for the simultaneous movement of agents.

2.1 Problem Definition

Let $G = (V, E)$ be a digraph, with vertex set V and edge set E . We assign a unique label to each pebble and hole. Sets P and H contain the labels of the pebbles and, respectively, the holes. Each vertex of G is occupied by either a pebble or a hole so that $|V| = |P| + |H|$. A *configuration* is a function $\mathcal{A} : P \cup H \rightarrow V$ that assigns the occupied vertex to each pebble or hole. A configuration is *valid* if it is one-to-one (i.e., each vertex is occupied by only one pebble or hole). Set $\chi \subset \{P \cup H \rightarrow V\}$ represents all valid configurations. Given a configuration $\mathcal{A}$ and $u, v \in V$, we denote by $\mathcal{A}[u, v]$ the configuration obtained from $\mathcal{A}$ by exchanging the pebbles (or holes) placed at u and v :

$$\mathcal{A}[u, v](q) := \begin{cases} v, & \text{if } \mathcal{A}(q) = u; \\ u, & \text{if } \mathcal{A}(q) = v; \\ \mathcal{A}(q), & \text{otherwise.} \end{cases} \quad (2.1)$$

Function $\rho : \chi \times E \rightarrow \chi$ is a partially defined transition function such that $\rho(\mathcal{A}, u \rightarrow v)$ is defined if and only if v is empty (i.e., occupied by a hole). In this case, $\rho(\mathcal{A}, u \rightarrow v)$ is the configuration obtained by exchanging the pebble or the hole in u with the hole in v . Notation $\rho(\mathcal{A}, u \rightarrow v)!$ means that the function is defined. In other words $\rho(\mathcal{A}, u \rightarrow v)!$ if and only if $(u, v) \in E$ and $\mathcal{A}^{-1}(v) \in H$, and, if $\rho(\mathcal{A}, u \rightarrow v)!$, $\rho(\mathcal{A}, u \rightarrow v) = \mathcal{A}[u, v]$. Note that the hole in v moves along edge $u \rightarrow v$ in the reverse direction, while the pebble or hole on u moves to v .

We represent plans as ordered sequences of directed edges. It is convenient to view the elements of E as the symbols of a language. We denote by E^* the Kleene star of E , that is the set of ordered sequences of elements of E with arbitrary length, together with the empty string ε :

$$E^* = \bigcup_{i=1}^{\infty} E^i \cup \{\varepsilon\}.$$

We extend function $\rho : \mathcal{X} \times E \rightarrow \mathcal{X}$ to $\rho : \mathcal{X} \times E^* \rightarrow \mathcal{X}$, by setting $(\forall \mathcal{A} \in \mathcal{X}) \rho(\mathcal{A}, \varepsilon)!$ and $\rho(\mathcal{A}, \varepsilon) = \mathcal{A}$. Moreover, $(\forall s \in E^*, e \in E, \mathcal{A} \in \mathcal{X}) \rho(\mathcal{A}, se)!$ if and only if $\rho(\mathcal{A}, s)!$ and $\rho(\rho(\mathcal{A}, s), e)!$ and, if $\rho(\mathcal{A}, se)!$, $\rho(\mathcal{A}, se) = \rho(\rho(\mathcal{A}, s), e)$. A *move* is an element of E , and a *plan* is an element of E^* . Note that ε is the trivial plan that keeps all pebbles and holes in their positions. Note that a *move* corresponds to a movement action for a single agent, while all other agents execute a *waiting* action, i.e., they remain at their current node. This definition is consistent with the assumption that agents cannot move simultaneously. We define an equivalence relation $\sim$ on E^* , by setting, for $s, t \in E^*$, $s \sim t \leftrightarrow (\forall \mathcal{A} \in \mathcal{X} \text{ s.t. } \rho(\mathcal{A}, s)!, \rho(\mathcal{A}, t)!) \rho(\mathcal{A}, s) = \rho(\mathcal{A}, t)$. In other words, two plans are equivalent if they reconfigure pebbles and holes in the same way. Given a configuration $\mathcal{A}$ and a plan f such that $\rho(\mathcal{A}, f)!$, a plan f^{-1} is a *reverse* of f if $\mathcal{A} = \rho(\rho(\mathcal{A}, f), f^{-1})$ (i.e., f^{-1} moves each pebble and hole back to their initial positions). We can also write $ff^{-1} \sim \varepsilon$, so that f^{-1} behaves like a right-inverse. In Proposition 4.1 of Section 2.4, we will state that in the particular case of a strongly connected digraph, each plan has a reverse plan.

Our main problem is the Multi-Agent Path Finding (MAPF), which consists of finding a plan that re-positions all pebbles to assigned target vertices, avoiding collisions. For this problem, and the ones we will present later, the position of the holes is not relevant. Therefore, we introduce an equivalence relation $\sim$ between configurations

$$\mathcal{A}^1 \sim \mathcal{A}^2 \iff \forall p \in P \quad \mathcal{A}^1(p) = \mathcal{A}^2(p),$$

and we indicate with $\mathcal{A}^1$ the equivalence class of $\mathcal{A}^1$.

Definition 2.1.1. (MAPF problem). Let $G = (V, E)$ be a graph, P be a pebble set, $\mathcal{A}^s$ an initial valid configuration, $\mathcal{A}^t$ a final valid configuration. The MAPF instance $\langle G, \mathcal{A}^s, \mathcal{A}^t \rangle$ consists in finding a plan f such that $\mathcal{A}^t = \rho(\mathcal{A}^s, f)$.

We denote by $|f|$ the length of a plan f (i.e., the number of moves of f). If f is the solution to a MAPF instance, then $|f|$ represents the total number of actions, including waiting actions, made by each agent in such a way that each agent reaches its target, or, equivalently, the number of time steps needed to achieve the final configuration, which is the Makespan.

If G is an undirected tree, this problem is called *pebble motion on trees* (PMT). A variant of the PMT problem is *pebble motion on trees with trans-shipment vertices* (ts-PMT). This problem is a PMT on a tree in which the vertex set V is partitioned in trans-shipment and regular vertices.

Definition 2.1.2. A *trans-shipment vertex* is a vertex with a degree greater than one that cannot host a pebble: pebbles can cross this node but cannot stop there. More formally, given a trans-shipment vertex s ,

1. $\deg(s) \geq 2$;
2. $\rho(\mathcal{A}, (u \rightarrow s)(w \rightarrow v))!$ if and only if $w = s$, $(u, s), (s, v) \in E$, and $\mathcal{A}^{-1}(v) \in H$.
If $\rho(\mathcal{A}, (u \rightarrow s)(w \rightarrow v))!$, then $\rho(\mathcal{A}, (u \rightarrow s)(w \rightarrow v)) = \mathcal{A}[u, v]$.

The second property means that if a pebble is moved to a trans-shipment vertex, then it must be immediately moved to another node. We denote with V^t the set of all the trans-shipment vertices and $V^r = V \setminus V^t$ the set of regular vertices. Ardizzoni et al. [Pub1] presented an algorithm to solve ts-PMT in polynomial time.

Another variant of MAPF is the *motion planning problem*, which consists of finding a plan such that a single marked pebble reaches a desired target vertex, avoiding collisions with other pebbles that are moving *obstacles* [Pub1] [28, 29, 30, 31]. We recall its definition.

Definition 2.1.3. (Motion Planning problem). Let $G = (V, E)$ be a graph, P a set of pebbles, and $\mathcal{A}^s$ an initial valid configuration. Given a pebble $\bar{p}$ and a target node $t \in V$, find a plan f such that $t = \rho(\mathcal{A}^s, f)(\bar{p})$. We indicate such a plan with the notation $\mathcal{A}^s(\bar{p}) \Rightarrow t$.

Wu and Grumbach [30] discuss the feasibility of the motion planning problem on a strongly biconnected digraph and prove the following:

Theorem 1.1. (*Theorem 14 of [30]*) *Let D be a strongly biconnected digraph, P a set of pebbles, and H a set of holes. Then, any motion planning problem on D is feasible if and only if $|H| \geq 1$.*

Ardizzoni et al. [Pub1] provide an algorithm (CATERPILLAR) that finds solutions to this problem on a tree with $O(nc)$ moves.

2.2 Solving MAPF on undirected graphs

In this section, we recall the planning method for a connected undirected graph as introduced by Kroutiris et al. [1]. The main idea is to transform the graph $G = (V, E)$ into a tree $T := \mathcal{T}(G)$ and the MAPF problem into a ts -PMT problem. It is possible to prove that the MAPF problem is solvable on G if and only if the corresponding ts -PMT problem is solvable on T [25]. Moreover, the solution of MAPF can be obtained from the solution of the corresponding ts -PMT [1].

2.2.1 Convert MAPF into ts -PMT

Given a connected graph $G = (V, E)$, we construct the *biconnected component tree* $\mathcal{T}(G) = (V_T, E_T)$ as follows. We initialize $V_T = V$, $E_T = E$, and we convert each maximal non-trivial (i.e., with at least three vertices) biconnected component $S = (V_S, E_S) \subset G$ into a star subgraph. The nodes in V_S are the leaves of the star. The internal node of the star is a newly added trans-shipment vertex.

The conversion of S into a star involves the following steps:

1. add a trans-shipment vertex s ,
2. remove every edge $e \in E_S$,
3. add the edges $\{(u, s) | u \in V_T\}$.

Note that $V_T = V \cup V^t$, where V^t is the set of all trans-shipment vertices and V represents the set of regular vertices of tree T . Note that trans-shipment vertices of the biconnected component tree have the following properties:

- $|V^t| = K$, where K is the number of non-trivial biconnected component of G ;
- $\forall v \in V^t \deg(v) \geq 3$;
- $\forall v, w \in V^t d(v, w) > 1$, where $d(v, w)$ is the edge-distance between v and w .

The latter property, which means that two trans-shipment vertices cannot be adjacent, is introduced in the definition of *ts*-PMT in [Pub1].

Note that G and $\mathcal{T}(G)$ have a similar structure. Biconnected components of G correspond to star subgraphs in $\mathcal{T}(G)$, with trans-shipment vertices as internal nodes. Figure 2.1 shows an undirected graph and its corresponding biconnected component tree. Building $\mathcal{T}(G)$ from G takes a linear time with respect to $|V|$ [32].

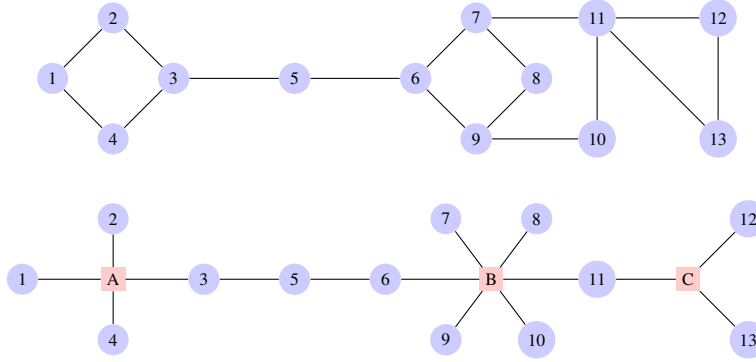


Figure 2.1: Undirected graph and corresponding biconnected component tree. A , B , and C are the trans-shipment vertices.

G and $\mathcal{T}(G)$ have the same number of pebbles but a different number of holes. Denoting with H_T the set of holes of the tree, it holds that $H_T = H \cup H^t$, where H^t is a new set of holes added for trans-shipment vertices ($|H_T| = |V_T|$).

Let $\mathcal{A} : P \cup H \rightarrow V$ be a configuration on G . We associate it to a configuration on $\mathcal{T}(G)$, $\mathcal{A}_T : P \cup H_T \rightarrow V_T$ such that $(\forall q \in P \cup H) \mathcal{A}_T(q) = \mathcal{A}(q) \subset V$ and $\mathcal{A}_T(H_T) = V_T$. Note that $\tilde{\mathcal{A}}_T|_{P \cup H} = \tilde{\mathcal{A}}$, since pebbles are on the same vertices. In this way, we associate every MAPF instance $\langle G, \tilde{\mathcal{A}}^s, \tilde{\mathcal{A}}^t \rangle$ to a *ts*-PMT instance $\langle \mathcal{T}(G), \tilde{\mathcal{A}}_T^s, \tilde{\mathcal{A}}_T^t \rangle$. Reference [25] proves the following important result.

Lemma 2.1. [25] Let $G = (V, E)$ be a connected undirected graph, which is not a cycle, and let $\mathcal{T}(G)$ be the corresponding biconnected component tree. Let $\mathcal{A}$ be an initial configuration on G and $\mathcal{A}_T$ the corresponding configuration on $\mathcal{T}(G)$. Let $a, b \in V$. Then, if $|H| \geq 2$, there is a plan f_{ab} such that $\mathcal{A}[a, b] = \rho(\mathcal{A}, f_{ab})$ if and only if there is a plan $f'_{a'b'}$ such that $\mathcal{A}_T[a', b'] = \rho(\mathcal{A}_T, f'_{a'b'})$.

As a consequence of this Lemma, it follows that:

Theorem 2.2. [25] Let G be a connected undirected graph which is not a cycle, with at least two holes. Then, MAPF on graph G is feasible if and only if *ts*-PMT on tree $\mathcal{T}(G)$ is feasible.

Since the feasibility of PMT on a tree $T = (V_T, E_T)$ is decidable in $O(|V_T|)$ time (see [14]), it follows that also MAPF on an undirected graph, which is not a cycle and with at least two holes, is decidable in linear time with respect to the number of nodes.

Moreover, any MAPF instance on a cycle is feasible if and only if, in the final configuration, the pebbles are in the same order as in the initial one [20].

In the case of graphs with only one hole, pebbles cannot move between different biconnected components (see [25]). Therefore, a MAPF problem on a graph with one hole is feasible only if the final position of each pebble is in the same connected component as the initial position. Moreover, the feasibility on a biconnected graph with one hole is decidable in linear time (see Remark 8.8 of [25]).

In conclusion, for *any* undirected graph $G = (V_G, E_G)$ it holds that:

Theorem 2.3. [25] The feasibility of a MAPF instance on G is decidable in $O(|V_G|)$ time.

2.2.2 Summary of algorithm presented in [1] to solve MAPF on graphs

The algorithm presented in [1] to solve the MAPF problem on a graph has the following structure:

1. Convert G into the biconnected component tree $\mathcal{T}(G)$ and convert MAPF into ts -PMT;
2. Solve ts -PMT problem on $\mathcal{T}(G)$;
3. Convert the solution of ts -PMT on $\mathcal{T}(G)$ into solutions of MAPF on G .

In particular, [1] presents a function *CONVERT-PATH* that converts the solutions of ts -PMT on $\mathcal{T}(G)$ into solutions of MAPF on G . When a pebble moves from v to u via a trans-shipment vertex s , this function first checks if there is a pebble-free path between v and u on G . If there is, this movement can be achieved. Otherwise, a more complex process is implied by the feasibility algorithm for graphs, presented more in detail in [15].

The algorithm that we will present in Section 2.5 to solve the MAPF problem on strongly connected digraphs has the same structure as the one just described for undirected graphs. The only difference consists in the *CONVERT-PATH* function: while the one presented in [1] converts a plan on a tree into a plan on the undirected graph, we will study a function that provides a plan on the strongly connected digraph.

2.3 Strongly connected digraphs

As said, we consider MAPF for *strongly connected digraphs*. In this section, we define this type of directed graph, and we go into the detail of their structure. This will be useful to present an algorithm to solve MAPF.

Definition 2.3.1. A digraph $D = (V, E)$ is *strongly connected* if for each $v, w \in V$, $v \neq w$, there exist a directed path from v to w , and a directed path from w to v in D .

Given a digraph D , we indicate with $\mathcal{G}(D)$ its *underlying graph*, that is, the undirected graph obtained by ignoring the orientations of the edges, and $\mathcal{T}(\mathcal{G}(D))$ the

corresponding biconnected component tree. Note that D is strongly connected only if $\mathcal{G}(D)$ is connected (see Figures 2.1 and 2.2).

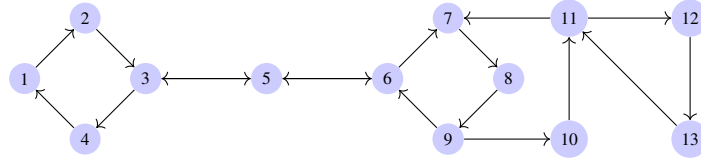


Figure 2.2: Example of strongly connected digraph: the corresponding underlying graph and biconnected component tree are shown in Figure 2.1.

An important property of strongly connected digraphs is that they can be decomposed into strongly biconnected components.

Definition 2.3.2. A digraph D is said to be strongly biconnected if D is strongly connected and $\mathcal{G}(D)$ is biconnected.

We recall that an undirected graph G is biconnected if it is connected and there are no cut vertices, i.e., the graph remains connected after removing any single vertex. The *partially-bidirectional cycle* is a simple example of a strongly biconnected digraph:

Definition 2.3.3. A digraph is a *partially-bidirectional cycle* if it consists of a simple cycle C , plus zero or more edges of the type (u, v) , where $(v, u) \in C$ (i.e., edges obtained by swapping the direction of an edge from C).

Reference [30] shows that strongly biconnected (respectively, strongly connected) digraphs have an open (respectively, closed) ear decompositions. We recall the definitions of open and closed ear decompositions. Given a graph $D = (V_D, E_D)$ and a sub-digraph $M = (V_M, E_M)$, a path π in D is a M -path if it is such that its start point and its endpoint are in V_M , no internal vertex is in V_M , and no edge of the path is in E_M . Moreover, a cycle C in D is a M -cycle if there is exactly one vertex of C in V_M .

Definition 2.3.4. Let $D = (V_D, E_D)$ be a digraph and $L = [L_0, L_1, \dots, L_r]$ an ordered sequence of sub-digraphs of D , where $L_i = (V_{L_i}, E_{L_i})$. We say that L is:

1. a *closed ear decomposition*, if:

- L_0 is a cycle,
 - for all $0 < i \leq r$, L_i is a D_i -path or a D_i -cycle, where $D_i = (V_{D_i}, E_{D_i})$ with $V_{D_i} = \bigcup_{0 \leq j < i} V_{L_j}$ and $E_{D_i} = \bigcup_{0 \leq j < i} E_{L_j}$,
 - $V_D = \bigcup_{0 \leq j \leq r} V_{L_j}$, $E_D = \bigcup_{0 \leq j \leq r} E_{L_j}$
2. an *open ear decomposition (oed)*, if it is a closed ear decomposition such that for all $0 < i \leq r$, L_i is a D_i -path, (i.e., it is not a D_i -cycle).

In Definition 2.3.4, each L_i is called an *ear*. In particular, L_0 is the basic cycle, and the other ears are derived ears. An ear is *trivial* if it has only one edge.

Definition 2.3.5. We say that an open ear decomposition of a strongly biconnected digraph is *regular (r-oed)* if the basic cycle L_0 has three or more vertices, and there exists a non-trivial derived ear with both ends attached to the basic cycle.

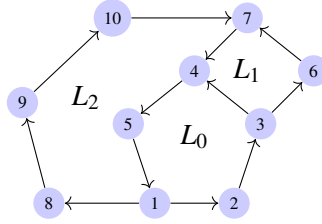


Figure 2.3: Digraph with an open ear decomposition.

Observation 2.3.1. Let $D = (V, E)$ be a digraph with an *oed* $L = [L_0, L_1, \dots, L_n]$. For each pair $v, w \in V$, there exists a sequence of cycles $C = [C_1, \dots, C_n]$ such that:

- $v \in V_{C_1}$ and $w \in V_{C_n}$;
- for all $j = 1, \dots, n-1$, $\exists a_j, b_j \in V_{C_j} \cap V_{C_{j+1}}$ such that $(a_j, b_j) \in E$.

Proof. Let $\pi = u_1 = v, u_2, \dots, u_{n-1}, u_n = w$ be a shortest path from v to w . Let L_i be an ear such that $v, u_2 \in V_{L_i}$. Let n_1 and m_1 be the startpoint and endpoint of L_i . Then, there exists a path π_1 from m_1 to n_1 and $C_1 = \pi_1 \cup L_i$ is the first cycle of the sequence. We initialize $C = [C_1]$ and we set $k = 1$. Now, if $n > 2$, for $j = 3, \dots, n$:

- if $u_j \in V_{C_k}$ we go to next iteration;
- otherwise, let π_{j-1} be a path from u_j to u_{j-2} , $C_{k+1} = (V_{C_{k+1}}, E_{C_{k+1}}) = \pi_{j-1} \cup (u_{j-2}, u_{j-1}) \cup (u_{j-1}, u_j)$ (note that $u_{j-2}, u_{j-1} \in V_{C_k} \cap V_{C_{k+1}}$); we add C_{k+1} to C and set $k = k + 1$, then we go to the next iteration.

□

Figure 2.3 shows a digraph with an *oed* $[L_0, L_1, L_2]$. The sequence of cycles associated to pair $v = 2, w = 10$ is $C = [C_0, C_2]$, where $C_0 = L_0$ and C_2 is the subgraph induced by $\{1, 8, 9, 10, 7, 4, 5\}$. Note that $(4, 5) \in C_0 \cap C_2$. The sequence associated to pair $v = 1, w = 6$ is simply $C = [C_1]$, where C_1 is the subgraph induced by $\{1, 2, 3, 6, 7, 4, 5\}$. In fact, nodes 1 and 6 belong to the same cycle.

We recall the following results that characterize strongly biconnected and strongly connected digraphs:

Theorem 3.1. *Let D be a non-trivial digraph.*

- *D is strongly biconnected if and only if D has an *oed*. Any cycle can be the starting point of an *oed* [33].*
- *D is strongly biconnected if and only if exactly one of the following holds [20]:*
 1. *D is a partially-bidirectional cycle;*
 2. *D has a *r-oed*.*

Theorem 3.2. [34] *Let D be a non-trivial digraph. D is strongly connected if and only if D has a closed ear decomposition.*

Observation 2.3.2. Roughly speaking, this last result means that a strongly connected digraph is composed of non-trivial, strongly biconnected components connected by corridors or articulation points. A *corridor* on a digraph $D = (V, E)$ is a sequence of adjacent vertices $u_1, \dots, u_n$ such that

- $(u_i, u_{i+1}), (u_{i+1}, u_i) \in E$ for each $i = 1, \dots, n - 1$;

- all the vertices of the sequence except u_1 and u_n (called *endpoints*) have exactly two neighbors.

For example, in Fig. 2.2, the subgraph induced by nodes 3,5 and 6 is a corridor. We indicate with $\mathcal{C}(D)$ the set of all corridors of D . Vertex $v \in V$ is an *articulation point* if its removal increases the number of connected components of the underlying graph $\mathcal{G}(D)$. In Fig. 2.2, nodes 3, 6, and 11 are articulation points. We define the subclass of corridors $\bar{\mathcal{C}}(D) \subset \mathcal{C}(D)$ that have only articulation points as endpoints.

Note that each star subgraph of $\mathcal{T}(\mathcal{G}(D))$ represents a biconnected component of $\mathcal{G}(D)$, which corresponds to a strongly biconnected component of D . Indeed, Theorem 9 of [30] defines a one-to-one correspondence between strongly biconnected components of D and biconnected components of $\mathcal{G}(D)$.

Finally, we recall the following definition about strongly biconnected digraphs adapted from [25], which will be useful in Section 2.5:

Definition 2.3.6. Let $B = (V, E)$ be a strongly biconnected digraph and $v \notin V$ be an external node. We consider a digraph $G = (V \cup \{v\}, \bar{E})$ with $E \subset \bar{E}$. We say that G is:

- a *strongly biconnected digraph with an entry-attached edge*, if there exists $z \in V$ such that $\bar{E} = \{(v, z)\} \cup E$;
- a *strongly biconnected digraph with an attached edge*, if there exists $z \in V$ such that $\bar{E} = \{(v, z), (z, v)\} \cup E$.

2.4 Feasibility of MAPF on strongly connected digraphs

In this section, we focus on the feasibility of the MAPF problem on strongly connected digraphs.

As shown in Proposition 2 and 3 of [35], in strongly connected digraphs, each move is reversible. From this, a more general result follows:

Proposition 4.1. In a strongly connected digraph, each plan has a reverse plan.

Proposition 4.1 leads to the following result about the feasibility of MAPF on digraphs:

Theorem 4.2. *Let $D = (V_D, E_D)$ be a strongly connected digraph. Then,*

1. *any MAPF instance on D is feasible if and only if it is feasible on the underlying graph $G = \mathcal{G}(D)$;*
2. *feasibility of any MAPF instance on D is decidable in linear time with respect to $|V_D|$.*

Proof. 1. The necessity is obvious. To prove sufficiency, let f' be a plan that solves a MAPF instance on $\mathcal{G}(D)$. Then, we can define a plan f on D in the following way. For each pebble move $u \rightarrow v$ in f' , if $(u, v) \in E_D$, we perform move $u \rightarrow v$ on D . Otherwise, since $(v, u) \in E_D$, we execute a reverse plan for $v \rightarrow u$, $(v \rightarrow u)^{-1}$, that exists by Proposition 4.1.

2. It follows from Theorem 2.3.

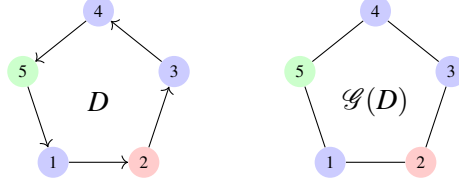
□

A direct consequence of Theorem 4.2 and Theorem 2.2 is the following important result:

Corollary 4.3. *Let D be a strongly connected digraph with at least two holes such that the corresponding underlying graph $\mathcal{G}(D)$ is not a cycle. Then, MAPF on D is feasible if and only if ts -PMT on tree $\mathcal{T}(\mathcal{G}(D))$ (i.e., the biconnected component tree of $\mathcal{G}(D)$) is feasible.*

The proof of Theorem 4.2 leverages the reversibility of each pebble motion in strongly connected digraphs. It presents a simple algorithm that reduces MAPF for strongly connected digraphs to the case of undirected graphs. However, this approach leads to redundant solutions since it does not exploit the directed graph structure. This fact is illustrated in Fig. 2.4, which shows a digraph D and its associated underlying graph $\mathcal{G}(D)$.

Example 2.4.1. A pebble p is placed at node 2, while all other nodes are free. We want to move p to 5. Plan $f' = (2 \rightarrow 1)(1 \rightarrow 5)$ is a solution of the corresponding problem on $\mathcal{G}(D)$. We convert this to a plan on D by applying the method in Theorem 4.2.

Figure 2.4: A digraph D and its underlying graph $\mathcal{G}(D)$.

Since $(2, 1) \notin E_D$, move $(2 \rightarrow 1)$ is converted into plan $(2, 3)(3, 4)(4, 5)(5, 1)$. Similarly, move $(1 \rightarrow 5)$ is converted into $(1, 2)(2, 3)(3, 4)(4, 5)$. This solution is longer than necessary, since the shorter plan $f = (2 \rightarrow 3)(3 \rightarrow 4)(4 \rightarrow 5)$ solves the overall problem.

Definition 2.4.1. We say that a MAPF solution algorithm has *length complexity* $O(n^k)$ if there exists a positive real constant $\gamma \in \mathbb{R}$ such that, for any MAPF instance on a graph with n nodes, the algorithm is able to provide a solution plan f of length $|f| \leq \gamma \cdot n^k$.

Given a strongly directed graph $D = (V, E)$, and denoting by $\mathcal{C}(D)$ the set of all the cycles contained in D , we define

$$N = \max_{e \in E} \min\{|C| : C \in \mathcal{C}(D), e \in E_C\}. \quad (2.2)$$

Proposition 4.4. The length complexity of the algorithm described in Theorem 4.2 is $O(n^3 N^2)$.

Proof. On undirected graphs, Kornhauser ([16]) proposes a MAPF algorithm that computes solutions of length γn^3 , where $\gamma \in \mathbb{R}$ is a positive real constant. In the worst case, each pebble move $u \rightarrow v$ on the undirected graph must be converted into a reverse plan (as in Example 2.4.1). A reverse plan consists of a complete rotation of a cycle, which takes $O(n_c^2)$ moves, where n_c is the number of nodes of the cycle that contains arc $u \rightarrow v$. Each edge $e_i \in E$, crossed by the pebbles in the solution, belongs to a collection of cycles $\{C_i^j\}_j$ of the graph, among which we can choose the one with minimum length $n_i \leq N$, denoted with $C_i = \arg \min\{|C| : C \in \mathcal{C}(D), e_i \in E_C\}$. Let

$[C_1, C_2, \dots, C_m]$ be the sequence of cycles of minimum length, chosen for each edge, and $(n_1, n_2, \dots, n_m)$ the corresponding number of nodes. Then, the overall solution length is bounded by

$$\gamma n^3 \cdot \sum_{i=1}^m O(n_i^2) \leq \gamma n^3 N^2.$$

□

2.4.1 Necessary and Sufficient condition for feasibility

Let us consider a strongly connected digraph $D = (V, E)$ and the corresponding bi-connected component tree $T = (V_T, E_T)$ with $V_T = V \cup V^t$. In this section, we want to derive a necessary and sufficient condition (n.s.c.) for the feasibility of MAPF on D from the n.s.c. of the ts -PMT on the corresponding tree T .

First of all, we remind the n.s.c for ts -PMT. To do that, on T , we define a vertex-distance $\tilde{d}$, which does not take into account trans-shipment vertices. Given $u, v \in V_T$, $\tilde{d}(u, v)$ counts how many regular vertices belong to the path π_{uv} :

$$\tilde{d}_T(u, v) := |\pi_{uv} \cap V|.$$

Consequently, we also define $\tilde{c}_1$ and $\tilde{c}_2$, which count corridor lengths according to the new definition of distance:

$$\tilde{c}_T^1 := \max\{\tilde{d}(a, b) : \pi_{ab} \in \mathcal{C}(T)\},$$

$$\tilde{c}_T^2 := \max\{\tilde{d}(a, b) : \pi_{ab} \in \bar{\mathcal{C}}(T)\}.$$

Moreover, we define $\tilde{c}_T := \max\{\tilde{c}_T^1, \tilde{c}_T^2 + 1\}$. In the same way we can define $\tilde{d}_D$, $\tilde{c}_D^1$, $\tilde{c}_D^2$ and $\tilde{c}_D$ on the digraph D .

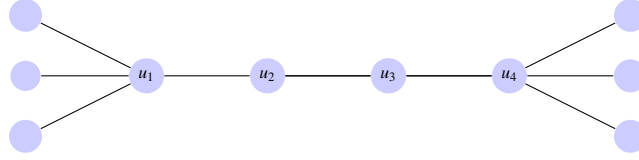
Kornhauser et al. [16] provided a n.s.c. for the feasibility of PMT on trees, from which Ardizzoni et al. [Pub1] derived the following n.s.c. for the feasibility of ts -PMT:

$$|H| \geq |V^t| + \tilde{c}^T, \quad (2.3)$$

where we recall that H denotes the set of holes. Since establishing the feasibility of MAPF on a digraph is equivalent to establishing the feasibility on the corresponding biconnected component tree (see Corollary 4.3), we can easily deduce that (2.3) is the n.s.c. for solvability of MAPF on a strongly connected digraph which is not a partially-bidirectional cycle and which has at least two holes. However, we want to translate (2.3) into a condition directly verifiable on the digraph.

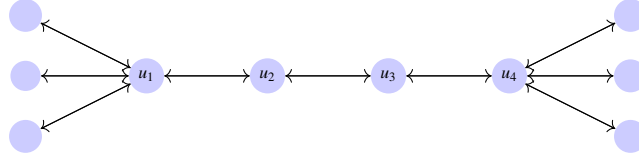
To do that, we have to find the relation between $\tilde{c}_T$ and $\tilde{c}_D$.

We consider a corridor $u_1, u_2, \dots, u_{n-1}, u_n$ on the tree.

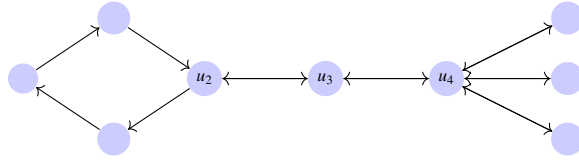


There are three possibilities:

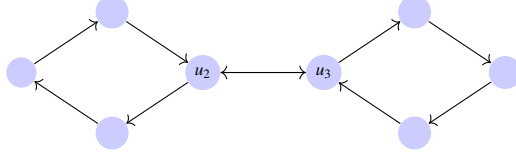
1. $u_1, u_n \notin V^t$: the corresponding corridor on D is exactly the same. Note that $\tilde{d}_T(u_1, u_n) = \tilde{d}_D(u_1, u_n)$.



2. $u_1 \in V^t, u_n \notin V^t$ (or vice versa): the corresponding corridor on D is $u_2, \dots, u_{n-1}, u_n$. Note that $\tilde{d}_T(u_1, u_n) = \tilde{d}_D(u_2, u_n)$.



3. $u_1, u_n \in V^t$: the corresponding corridor on D is $u_2, \dots, u_{n-1}$. Note that $\tilde{d}_T(u_1, u_n) = \tilde{d}_D(u_2, u_{n-1})$.



Since the vertex-length of the corridors on the digraph is equal to the vertex-length of the corresponding corridors on the tree, it follows that $\tilde{c}_D^1 = \tilde{c}_T^1$, $\tilde{c}_D^2 = \tilde{c}_T^2$ and $\tilde{c}_D = \tilde{c}_T$. Since the number of trans-shipment vertices is equal to the number of nontrivial biconnected components (i.e., $|V^t| = K$), condition (2.3) is equivalent to

$$|H| \geq K + \tilde{c}_D. \quad (2.4)$$

2.5 Path planner for strongly connected digraph

The first possible approach to solve MAPF on strongly connected digraphs is described in Theorem 4.2. This algorithm solves MAPF on the corresponding underlying graph and then converts the solution into a solution on the original digraph. However, as discussed in Section 2.4, in some cases, solutions found with this procedure may be too long.

To find shorter solutions in polynomial time, we present the diSC algorithm that better exploits the structure of the directed graph. In particular, we will exploit the fact that strongly connected digraphs can be decomposed into strongly biconnected components (see Observation 2.3.2).

In Section 2.4, we proved that MAPF on D is feasible if and only if ts -PMT on $\mathcal{T}(\mathcal{G}(D))$ is feasible. Therefore, as in the case of undirected graphs (see Section 2.2), we can reduce MAPF to ts -PMT, solve it, and then convert the solution on the tree into a solution on the original directed graph.

Thus, the diSC strategy is the same as presented in Section 2.2.2 to solve MAPF on undirected graphs. The main steps are the following:

1. Convert the digraph D into a tree $T = \mathcal{T}(\mathcal{G}(D))$ and consider the corresponding ts -PMT problem.

2. Solve the ts -PMT problem on tree T .
3. Convert the solution plan on T into a plan on D by a suitable algorithm *CONVERT-PATH*.

The novelty in the diSC algorithm lies in the definition of algorithm *CONVERT-PATH*, which allows converting a plan f' , that solves the ts -PMT, into a plan f , that solves the MAPF on the digraph. In Theorem 5.6, we will describe in detail algorithm *CONVERT-PATH*.

2.5.1 Basic plans for *CONVERT-PATH* algorithm

Algorithm *CONVERT-PATH* uses a number of basic plans that perform simple tasks.

k -CYCLE ROTATION

This plan rotates all pebbles on a cycle, moving each one by k positions. In particular, if $k = 1$, each pebble moves forward by one position. If $k = n$ (where n is the cycle length), it performs a *complete* rotation so that each agent returns to its initial position. We denote this plan with r_k^C .

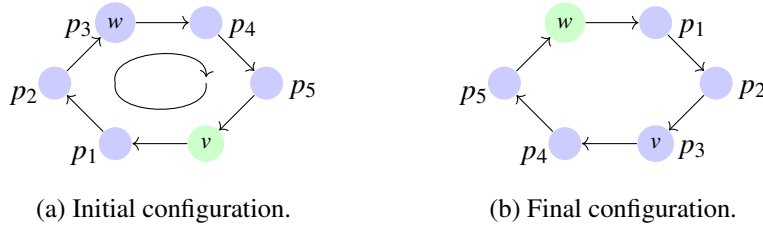


Figure 2.5: Example of 3-CYCLE ROTATION.

More formally, let $C = (V_C, E_C)$ be a cycle, with a set of pebbles P , and a set of holes H , with $|H| \geq 1$. Let $\mathcal{A}$ be an initial configuration, $v \in \mathcal{A}(H)$, and $w \in V_C$ be such that $(v, w) \in E_C$ (i.e., w is the successor of v on C).

Let $\pi = u_1 = w, \dots, u_n = v$ be the path on C from w to v . We define plan 1-CYCLE ROTATION as

$$r_1^C = (u_{n-1} \rightarrow u_n, \dots, u_1 \rightarrow u_2). \quad (2.5)$$

In other words, for each j from $n-1$ to 1, if there is a pebble on u_j , we move it on u_{j+1} . The new configuration $\bar{\mathcal{A}}$ is defined as follows:

$$\bar{\mathcal{A}}(q) := \begin{cases} u_{j+1}, & \text{if } \mathcal{A}(q) = u_j \quad j = 1, \dots, n-1; \\ w, & \text{if } \mathcal{A}(q) = v; \end{cases} \quad (2.6)$$

which means that all pebbles and holes on C change positions.

For $k \in \mathbb{N}$, a k -CYCLE ROTATION over C is obtained by performing k 1-CYCLE ROTATIONS over C . We denote the plan corresponding to a k -Cycle Rotation over C by r_k^C . Let $l_C = |V_C|$ be the length of cycle C . Plan $r_{l_C}^C$ is a *complete* rotation of C and brings all pebbles and holes back to their initial positions. In other words $r_{l_C}^C \sim \varepsilon$, where ε is the empty plan. Since $\varepsilon \sim r_{l_C}^C = r_k^C r_{l_C-k}^C$, it follows that complementary rotation $r_{l_C-k}^C$ is an inverse plan of r_k^C .

If $C = [C_1, \dots, C_m]$ is an ordered sequence of cycles, that are subgraphs of the same graph, and $k = (k_1, \dots, k_m) \in \mathbb{N}^m$, R_k^C denotes the plan obtained by concatenating a k_1 -Cycle Rotation over C_1 , a k_2 -Cycle Rotation over C_2 , and analogous rotations over the remaining cycles of C , namely:

$$R_k^C = r_{k_1}^{C_1} \dots r_{k_m}^{C_m}.$$

Set $s = (s_n, s_{n-1}, \dots, s_1) = (l_{C_n} - k_n, \dots, l_{C_1} - k_1)$ and $\hat{C} = [C_n, C_{n-1}, \dots, C_1]$. Then

$$R_k^C R_s^{\hat{C}} = r_{k_1}^{C_1} \dots r_{k_n}^{C_n} r_{s_n}^{C_n} \dots r_{s_1}^{C_1} \sim \varepsilon,$$

since $r_{k_n}^{C_n} r_{s_n}^{C_n} \sim \varepsilon$, and analogous reductions holds for the remaining terms. This implies that $R_s^{\hat{C}}$ is an inverse plan of R_k^C . We denote $R_s^{\hat{C}}$ by $(R_k^C)^+$. In other words, an inverse plan of R_k^C consists of a sequence of complementary rotations in inverse order.

BRING HOLE FROM v TO w

Holes are needed to execute any move or plan. For example, a k -CYCLE ROTATION is not possible unless there is at least one hole in the cycle. For this reason, it is useful

to define this plan, which allows moving a hole from one position of the graph to another. We denote these plans with $h_{v,w}$.

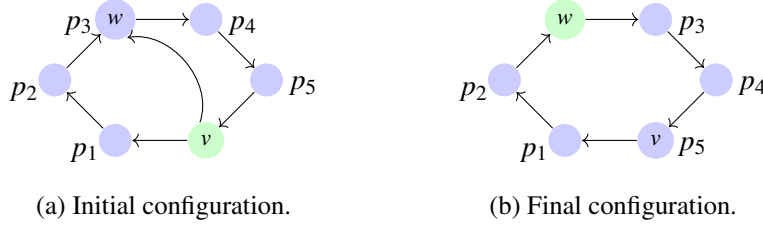


Figure 2.6: Example of BRING HOLE FROM v TO w .

Let $\mathcal{A}$ be an initial configuration, such that $v \in \mathcal{A}(H)$ (i.e., v is an unoccupied vertex). There are two cases:

1. v and w belong to the same cycle C . In this case, let $\pi = u_1 = w, \dots, u_n = v$ be the path on C from w to v . We define the plan BRING HOLE FROM v TO w as

$$h_{v,w} = (u_{n-1} \rightarrow u_n, \dots, u_1 \rightarrow u_2). \quad (2.7)$$

In other words, for each j from $n-1$ to 1 , if there is a pebble on u_j , we move it on u_{j+1} . The new configuration $\tilde{\mathcal{A}}$ is defined as follows:

$$\tilde{\mathcal{A}}(q) := \begin{cases} u_{j+1}, & \text{if } \mathcal{A}(q) = u_j \quad j = 1, \dots, n-1; \\ w, & \text{if } \mathcal{A}(q) = v; \\ \mathcal{A}(q), & \text{otherwise,} \end{cases} \quad (2.8)$$

which means that only pebbles and holes along path π change positions. Moreover, since the graph is strongly connected by Proposition 4.1, there exists a reverse plan $h_{v,w}^{-1}$, which returns pebbles and holes to their initial positions. In particular, in this case, the reverse plan consists of moving the pebbles along the cycle until the initial configuration is obtained again. We call BRING BACK HOLE FROM w TO v the plan $h_{v,w}^{-1}$.

2. Otherwise, by Observation 2.3.1, there exists a sequence of cycles $C = [C_{i_1}, C_{i_2}, \dots, C_{i_n}]$ such that $v \in C_{i_1}$, $w \in C_{i_n}$, and for all $j = 1, \dots, n$, $C_{i_j} \cap C_{i_{j+1}}$ contains at least one node a_{i_j} . Starting from C_{i_1} , we perform the following operations: we perform a $d(v, a_{i_1})$ -CYCLE ROTATION over cycle C_{i_1} ; for all $j = 2, \dots, n-1$ we perform a $d(a_{i_j}, a_{i_{j+1}})$ -CYCLE ROTATION over cycle C_{i_j} ; we perform a $d(a_{i_{n-1}}, y)$ -CYCLE ROTATION over cycle C_{i_n} . Setting $k = (d(w, a_{i_1}), d(a_{i_2}, a_{i_3}), \dots, d(a_{i_{n-2}}, a_{i_{n-1}}))$, this sequence of rotations corresponds to R_k^C . Now, the hole is in the same cycle as w , so we continue as in step 1. So, BRING HOLE FROM v TO w is defined as

$$h_{v,w} = R_k^C h_{a_{i_{n-1}},w}.$$

In this case, the reverse plan BRING BACK HOLE FROM w TO v is defined by

$$h_{v,w}^{-1} = h_{a_{i_{n-1}},w}^{-1} (R_k^C)^+.$$

where $s = (s_n, s_{n-1}, \dots, s_1) = (l_{C_n} - k_n, \dots, l_{C_1} - k_1)$ and $\hat{C} = [C_n, C_{n-1}, \dots, C_1]$.

Three more plans can be directly defined from BRING HOLE FROM v TO w :

1. **BRING BACK HOLE FROM w TO v** is a reverse of BRING HOLE FROM v TO w . Hence, it returns the hole to its initial position and moves all agents back to their initial positions. We denote these plan with $h_{v,w}^{-1}$.
2. **BRING HOLE FROM v TO A SUCCESSOR OF w** . Let $\mathcal{A}$ be an initial configuration, such that $v \in \mathcal{A}(H)$. Let $\pi = u_1 = w, \dots, u_n = v$ be a shortest path from w to v , where u_2 is the successor of w along π . Then, BRING HOLE FROM v TO A SUCCESSOR OF w ($h_{v,s(w)}$) is defined as BRING HOLE FROM v TO u_2 .
3. **BRING BACK HOLE FROM A SUCCESSOR OF w TO v** . Let $h_{v,s(w)}$ be a plan BRING HOLE FROM v TO w . We call BRING BACK HOLE FROM w TO v its reverse plan $h_{v,s(w)}^{-1}$.

Observation 2.5.1. Let $\mathcal{A}$ be an initial configuration, $h_{v,w}$ a plan BRING HOLE FROM v TO w , and $\bar{\mathcal{A}} = \rho(\mathcal{A}, h_{v,w})$ the corresponding final configuration. Given $a, b \in V$ with $b \neq v$, $p = \mathcal{A}^{-1}(a)$ and $q = \mathcal{A}^{-1}(b)$ are the pebbles or holes that occupy a, b . Then, $\bar{\mathcal{A}}[\bar{\mathcal{A}}(p), \bar{\mathcal{A}}(q)] = \overline{\mathcal{A}[a, b]}$. That is, the configuration obtained performing $h_{v,w}$ on $\mathcal{A}[a, b]$ is equal to the one obtained by exchanging $\bar{\mathcal{A}}(p)$ and $\bar{\mathcal{A}}(q)$ on $\bar{\mathcal{A}}$.

2.5.2 Plans for movements through biconnected components

As said, *CONVERT-PATH* algorithm converts a PMT solution into a MAPF solution. The most complex subtask is the conversion of a movement in a 'star' into a sequence of movements within the associated biconnected component. In particular, each pebble motion to a different spike of the same star requires crossing the star transshipment vertex. We need to convert this motion into a plan on the digraph that allows moving the pebble to the same final position without altering the positions of all other pebbles.

Observation 2.5.2. In Observation 2.3.2, we noted that a strongly connected digraph $D = (V, E)$ is composed of non-trivial biconnected components and corridors. Following the classification presented in [25] for undirected graphs, in strongly connected directed graphs, we defined three different motion tasks. Following [25], for each task, we present one or more Lemmas that describe a possible plan:

1. a pebble moves within the same strongly biconnected component: *Stay in Lemma 5.4*;
2. a pebble moves from a corridor to a strongly biconnected component (or vice versa): *Entry Lemma 5.1, Go out Lemma 5.2, Attached-Edge Lemma 5.3*;
3. a pebble moves from a strongly biconnected component to another one, connected by an articulation point: *Two Biconnected Components Lemma 5.5*.

All these Lemmas define plans that require two holes, with the following roles:

- the *destination* hole h_1 is in the final position of pebble p . In the final configuration, h_1 and p will be exchanged;
- the *transport* hole h_2 is used to move p to the position of h_1 . Indeed, each pebble motion requires the presence of a hole. Each plan, in the end, brings back h_2 to its initial position, together with all pebbles, with the exception of p .

Lemma 5.1. Entry. Let P be a set of pebbles and H , with $|H| \geq 2$, a set of holes on $G = (V \cup \{v\}, \bar{E})$, where G is a strongly biconnected digraph with an entry-attached edge (v, y) (see Definition 2.3.6). Let $\mathcal{A}$ be a configuration, $p \in P$ such that $\mathcal{A}(p) = v$, and $w \in \mathcal{A}(H)$. Let $\mathcal{A}[v, w]$ be the configuration defined in (2.1). Then, there exists a plan f_{vw} such that $\mathcal{A}[v, w] = \rho(\mathcal{A}, f_{vw})$, i.e., that moves p from v to w , without altering the locations of the other pebbles.

Figure 2.7 shows an example of this situation.

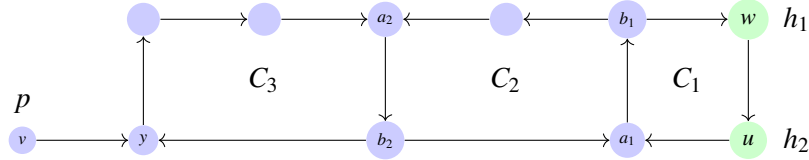


Figure 2.7: An example of situation of Entry Lemma 5.1. Pebble p has to move on w without altering the final location of the other pebbles. h_1 and h_2 are, respectively, the destination and transport holes. The solution plan is $f_{vw} = r_2^{C_1} r_3^{C_2} r_2^{C_3} (v \rightarrow y) r_3^{C_3} r_2^{C_2} r_2^{C_1}$.

Proof. Let $y \in V$ be such that $(v, y) \in E$. Let h_1 be the *destination* hole in w ($\mathcal{A}(h_1) = w$), and let h_2 be a *transport* hole in $V \setminus \{w\}$ (note that h_2 exists since we are assuming that $|H| \geq 2$). If G is a partially-bidirectional cycle, we set $n = 1$ and $C_1 = C_G$, where C_G is the directed cycle contained in G . We perform a $d(w, y)$ -CYCLE ROTATION over cycle C_G , where $d(w, y)$ is the distance between nodes w and y . In this way, hole h_1 moves to y . Next, pebble p moves on y with $v \rightarrow y$. Finally, we perform the complementary $L_{C_G} - d(y, w)$ -CYCLE ROTATION over C_G , in order to move p to w . Namely, the plan is $f_{vw} = r_{d(w, y)}^{C_G} (v \rightarrow y) (r_{d(w, y)}^{C_G})^+$, and the final configuration is $\mathcal{A}[v, w]$. Indeed, apart from p and h_1 , which are exchanged, all pebbles and holes are moved l_{C_G} times, which means that they complete a full revolution, returning to their initial positions. If G has a r -oed $L = [L_0, \dots, L_r]$, let u be a successor of w such that $\mathcal{A}(h_2) = u$. If it does not exist, we perform BRING HOLE FROM $\mathcal{A}(h_2)$ TO A SUCCESSOR OF w and set u as the successor of w , which corresponds to the

new position of h_2 . In the end, we will bring back h_2 to its initial position with BRING BACK HOLE FROM A SUCCESSOR OF w TO $\mathcal{A}(h_2)$. By Observation 2.3.1, there exists a sequence of cycles $C = [C_{i_1}, C_{i_2}, \dots, C_{i_n}]$ such that $v \in C_{i_1}$ and $u \in C_{i_n}$ and for all $j = 1, \dots, n$, $C_{i_j} \cap C_{i_{j+1}}$ has at least two nodes a_{i_j} and b_{i_j} with $(a_{i_j}, b_{i_j}) \in E$. Starting from C_{i_1} , we perform the following operations: we perform a $d(w, a_{i_1})$ -CYCLE ROTATION over cycle C_{i_1} ; for all $j = 2, \dots, n-1$ we perform a $d(a_{i_j}, a_{i_{j+1}})$ -CYCLE ROTATION over cycle C_{i_j} ; we perform a $d(a_{i_{n-1}}, y)$ -CYCLE ROTATION over cycle C_{i_n} . Setting $k = (d(w, a_{i_1}), d(a_{i_2}, a_{i_3}), \dots, d(a_{i_{n-1}}, y))$, this sequence of rotations corresponds to R_k^C . Then, we move p to y and perform the inverse sequence $(R_k^C)^+$. At the end of this plan, p is on w , and all other pebbles are in their initial positions. Hence, the overall plan that allows us to prove the thesis is $f_{vw} = h_{h_2s(w)} R_k^C (v \rightarrow y) (R_k^C)^+ h_{hs(w)}^{-1}$. $\square$

Lemma 5.2. Go out. Let P be a set of pebbles and H , with $|H| \geq 2$, a set of holes on $G = (V \cup \{w\}, \bar{E})$, where G is a strongly biconnected digraph with an attached edge (y, w) (see Definition 2.3.6). Let $\mathcal{A}$ be a configuration, $p \in P$ such that $\mathcal{A}(p) = v \in V$, and $w \in \mathcal{A}(H)$. Let $\mathcal{A}[v, w]$ be the configuration defined in (2.1). Then, there exists a plan f_{vw} such that $\mathcal{A}[v, w] = \rho(\mathcal{A}, f_{vw})$, i.e., that moves p from v to w , without altering the locations of the other pebbles.

Proof. Let h_1 be the destination hole in w ($\mathcal{A}(h_1) = w$), and let h_2 be a transport hole in $V \setminus \{w\}$. Without loss of generality, we can suppose that exists u a successor of v such that $\mathcal{A}(h_2) = u$. Otherwise, it would be enough to perform at the beginning BRING HOLE FROM $\mathcal{A}(h_2)$ TO A SUCCESSOR OF v and at the end, we would bring back h_2 to its initial position with BRING BACK HOLE FROM A SUCCESSOR OF v TO $\mathcal{A}(h_2)$. By Observation 2.3.1, there exists a sequence of cycles $C = [C_{i_1}, C_{i_2}, \dots, C_{i_m}]$ such that $v, u \in C_{i_1}$ and $y \in C_{i_m}$ and for all $j = 1, \dots, m$, $C_{i_j} \cap C_{i_{j+1}}$ has at least two nodes a_{i_j} and b_{i_j} with $(a_{i_j}, b_{i_j}) \in E$. Starting from C_{i_1} , we perform the following operations: we perform a $d(u, b_{i_1})$ -CYCLE ROTATION over cycle C_{i_1} ; for all $j = 2, \dots, m-1$ we perform a $d(b_{i_j}, b_{i_{j+1}})$ -CYCLE ROTATION over cycle C_{i_j} ; we perform a $d(b_{i_{m-1}}, y)$ -CYCLE ROTATION over cycle C_{i_m} . Setting $k = (d(u, b_{i_1}), d(b_{i_2}, b_{i_3}), \dots, d(b_{i_{m-1}}, y))$, this sequence of rotations corresponds to R_k^C . After these rotations, p is on $x \in V$, a

predecessor of y . Then, we move p to y and then w and perform the inverse sequence $(R_k^C)^+$. At the end of this plan, p is on w , and all other pebbles are in their initial positions. Hence, the overall plan that allows us to prove the thesis is $f_{vw} = R_k^C(x \rightarrow y)(y \rightarrow w)(R_k^C)^+ h_{hs(w)}^{-1}$.

□

Lemma 5.3. Attached-Edge. Let P be a set of pebbles and H a set of holes on $D = (V \cup \{v\}, \bar{E})$, a strongly biconnected digraph with an attached edge such that $|H| \geq 2$. Let $\mathcal{A}$ be a configuration, $p \in P$ such that $\mathcal{A}(p) = u$, and $w \in \mathcal{A}(H)$. Let $\mathcal{A}[u, w]$ be a final configuration defined as in (2.1), then there exists a plan f_{uw} such that $\mathcal{A}[u, w] = \rho(\mathcal{A}, f_{uw})$.

Figure 2.8 shows an example of this situation.

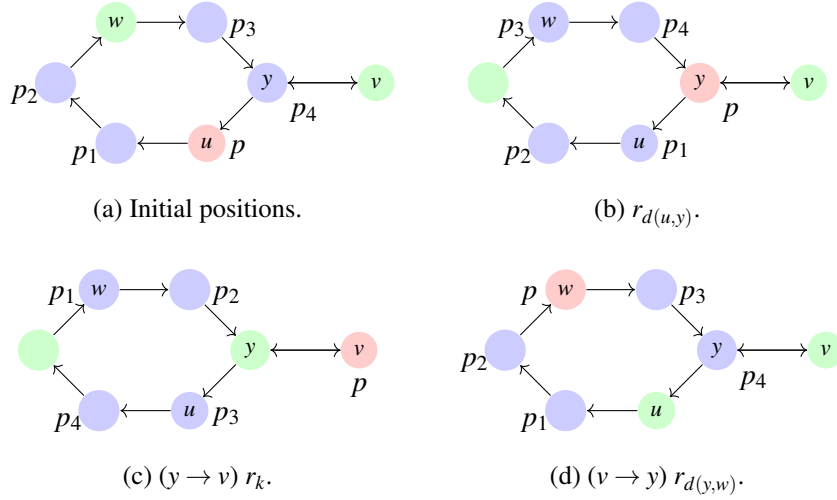


Figure 2.8: An example of the situation described in Attached-edge Lemma 5.3. Pebble p has to move on w without altering the final location of the other pebbles. The solution plan is $f_{vw} = r_{d(u,y)}(y \rightarrow v) r_k(v \rightarrow y) r_{d(y,w)}$.

Proof. If the strongly biconnected component of D has a r -oed, the proof follows

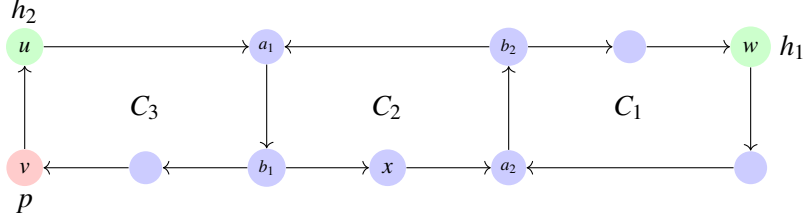
from Lemma 5.4. Otherwise, the strongly biconnected component is a partially-bidirectional cycle $G = (V, E)$. We consider the cycle C_G contained in G , which has length l_{C_G} . Let h_1 be the *destination* hole on w and h_2 a *transport* hole (which exists, since $|H| \geq 2$). Without loss of generality, suppose that $\mathcal{A}(h_2) = v$. Indeed, if this were not the case, we can BRING HOLE FROM $\mathcal{A}(h_2)$ TO v and finally bring it back to its initial position. In this case, the new initial configuration is $\bar{\mathcal{A}} = \rho(\mathcal{A}, h_{\mathcal{A}(h_2)v})$ and we have to consider $\bar{u} = \mathcal{A}(p)$ and $\bar{w} = \mathcal{A}(h_1)$. Let $y \in V$ be the cycle node that shares an arc with v , and consider the distances from u and w to y , $d_1 := d(u, y)$ and $d_2 := d(w, y)$. Performing a d_1 -CYCLE ROTATION over C_G , we move p from w to y . Then, we move p on v with $y \rightarrow v$. Now, let

$$k = \begin{cases} d_2 - d_1 & \text{if } d_2 \geq d_1, \\ l + d_2 - d_1 & \text{if } d_2 < d_1. \end{cases}$$

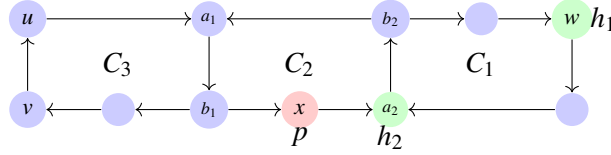
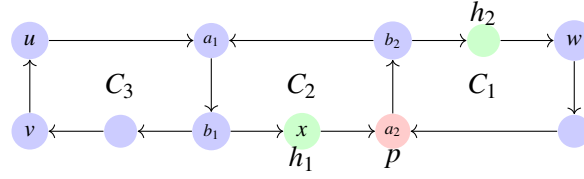
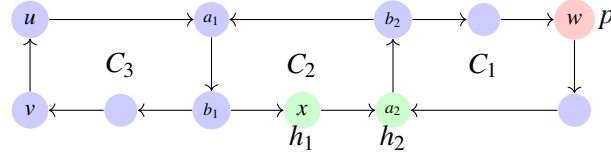
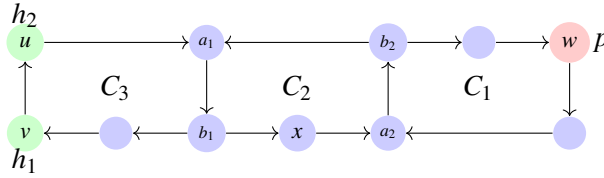
We perform a k -CYCLE ROTATION over C_G so that we move the hole h_1 from w to y . Next, we move p from v to y with $v \rightarrow y$. Finally, we perform a $d(y, w)$ -CYCLE ROTATION over C_G to move p on w . To conclude, $f_{uw} = r_{d(u,y)}^{C_G}(y \rightarrow v) r_k^{C_G}(v \rightarrow y) r_{d(y,w)}^{C_G}$. $\square$

Lemma 5.4. Stay in. Let P be a set of pebbles and H , with $|H| \geq 2$, be a set of holes on $D = (V, E)$, a strongly biconnected digraph with a *r-oed*. Let $\mathcal{A}$ be a configuration, $p \in P$ such that $\mathcal{A}(p) = v$, and $w \in \mathcal{A}(H)$. Let $\mathcal{A}[v, w]$ be a configuration defined as in (2.1), then there exists a plan f_{vw} such that $\mathcal{A}[v, w] = \rho(\mathcal{A}, f_{vw})$.

Figure 2.9 shows an example of this situation.



(a) Initial configuration.

(b) New configuration after performing $r_2^{C_3} r_2^{C_2}$ (c) New configuration after performing $r_2^{C_1} (x \rightarrow a_2)$ (d) New configuration after performing $r_3^{C_1}$ 

(e) Final configuration after performing $r_3^{C_2} r_3^{C_3}$. Pebble p reached the target w , and the other pebbles are in the same initial positions.

Figure 2.9: An example of the situation described in Stay in Lemma 5.4. Pebble p has to move on w without altering the final location of the other pebbles. h_1 and h_2 are, respectively, the destination and transport holes. The solution plan is $f_{vw} = r_2^{C_3} r_2^{C_2} r_2^{C_1} (x \rightarrow a_2) r_3^{C_1} r_3^{C_2} r_3^{C_3}$.

Proof. Let h_1 be the *destination* hole in w ($\mathcal{A}(h_1) = w$), and let h_2 be a *transport* hole in $V \setminus \{w\}$. The initial position of pebble p is v ($\mathcal{A}_p = v$). Without loss of generality, we can suppose that exists u is a successor of v such that $\mathcal{A}(h_2) = u$ (see Figure 2.9a). Otherwise, it would be enough to perform at the beginning BRING HOLE FROM $\mathcal{A}(h_2)$ TO A SUCCESSOR OF v and at the end, we would bring back h_2 to its initial position with BRING BACK HOLE FROM A SUCCESSOR OF v TO $\mathcal{A}(h_2)$.

There are two cases:

1. the initial position $\mathcal{A}(p) = v$ of the pebble and the target position $\mathcal{A}(h_1) = w$ does not belong to the same cycle. By Observation 2.3.1, there exists a sequence of cycles $C = [C_{i_1}, C_{i_2}, \dots, C_{i_m}]$ such that $v \in C_{i_1}$ and $w \in C_{i_m}$ and for all $j = 1, \dots, m$, $C_{i_j} \cap C_{i_{j+1}}$ has at least two nodes a_{i_j} and b_{i_j} with $(a_{i_j}, b_{i_j}) \in E$. In particular, choose $a_{i_{m-1}}$ such that his predecessor node x in $C_{i_{m-1}}$ does not belong to C_{i_m} .

Now, starting from C_{i_1} , we perform the following operations: we perform a $d(v, a_{i_1})$ -CYCLE ROTATION over cycle C_{i_1} ; for all $j = 2, \dots, m-3$ we perform a $d(a_{i_j}, a_{i_{j+1}})$ -CYCLE ROTATION over cycle C_{i_j} ; we perform a $d(a_{i_{m-2}}, x)$ -CYCLE ROTATION over cycle $C_{i_{m-1}}$ (Fig. 2.9b) and then a $d(w, a_{i_{m-1}})$ -CYCLE ROTATION over cycle C_{i_m} . Setting $k = (d(v, a_{i_1}), d(a_{i_2}, a_{i_3}), \dots, d(a_{i_{m-2}}, x), d(w, a_{i_{m-1}}))$, this sequence of rotations corresponds to R_k^C . Then, we move p to $a_{i_{m-1}}$ (Fig. 2.9c) and perform the inverse sequence $(R_k^C)^+$. At the end of this plan, p is on w , and all other pebbles are in their initial positions (Fig. 2.9e). Hence, the overall plan that allows us to prove the thesis is $f_{vw} = R_k^C(x \rightarrow a_{i_{m-1}})(R_k^C)^+$.

2. $\mathcal{A}(p) = v$ and $\mathcal{A}(h_1) = w$ belongs to the same cycle C_1 . In this case, the idea is to move the pebble to another cycle and then use the procedure described at point 1) (see Figure 2.10). Let C_2 a cycle such that $|C_1 \cap C_2| \geq 2$. We denote with a and b the first and the last nodes that belong to the intersection, with c the predecessor of b in $C_1 \cap C_2$ and with d the successor of b in C_1 . We perform a $d(v, c)$ -CYCLE ROTATION over cycle C_1 and then a 2-CYCLE ROTATION over cycle C_2 . Now, if the destination hole belongs to the intersection between the two cycles (i.e., $\mathcal{A}(h_1) \in C_1 \cap C_2$), we perform a $d(\mathcal{A}(h_1), d)$ -CYCLE RO-

TATION of C_1 to move h_1 out of C_2 . Thus, setting $k = (d(v, c), 2, d(\mathcal{A}(h_1), d))$ and $C = [C_1, C_2, C_1]$, so far we have performed R_k^C . Now pebble p is in $C_2 \setminus C_1$ and h_1 in $C_1 \setminus C_2$. Therefore, we can proceed as in point 1). Finally, we perform $(R_k^C)^+$.

□

The next lemma deals with the case of two biconnected components joined by an articulation point (e.g., like $\{6, 7, 8, 9, 10, 11\}$ and $\{11, 12, 13\}$ in Figure 2.2, where the articulation point is node 11).

Lemma 5.5. Two Biconnected Components. Let P be a set of pebbles and H , with $|H| \geq 2$, a set of holes on $D = (V, E)$, a strongly connected digraph composed of two biconnected components joined by an articulation point. Let $\mathcal{A}$ be a configuration, $p \in P$ be such that $\mathcal{A}(p) = a$, and $b \in \mathcal{A}(H)$. Let $\mathcal{A}[a, b]$ be a final configuration defined as in (2.1). Then, there exists a plan f_{ab} such that $\mathcal{A}[a, b] = \rho(\mathcal{A}, f_{ab})$.

Proof. Let $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$ be the two biconnected components and v be the articulation point, i.e., $V_1 \cup V_2 = V$, $E_1 \cup E_2 = E$, $V_1 \cap V_2 = \{v\}$, and $E_1 \cap E_2 = \emptyset$. Let h_1 be the *destination* hole on b , and let h_2 be a *transport* hole. We discuss different cases.

1. $a \in V_1 \setminus \{v\}$ and $b \in V_2 \setminus \{v\}$.

Without loss of generality, we assume that $\mathcal{A}(h_2) = v$. Indeed, if this is not the case, we can BRING HOLE FROM $\mathcal{A}(h_2)$ TO v , and the new initial configuration is $\tilde{\mathcal{A}} = \rho(\mathcal{A}, h_{\mathcal{A}(h_2)v})$. Note that $h_{\mathcal{A}(h_2)v}$ could change the position either of h_1 or of p , i.e., either $\tilde{a} = \tilde{\mathcal{A}}(p) \neq a$ or $\tilde{b} = \tilde{\mathcal{A}}(h_1) \neq b$. By the procedure described below, we will reach $\tilde{\mathcal{A}}[\tilde{a}, \tilde{b}]$, and at that point we will need to perform BRING BACK HOLE FROM v TO $\mathcal{A}(h_2)$ in order to obtain, by Observation 2.5.1, $\mathcal{A}[a, b] = \rho(\tilde{\mathcal{A}}[\tilde{a}, \tilde{b}], h_{v, \mathcal{A}(h_2)}^{-1})$.

Assuming $\mathcal{A}(h_2) = v$, let $y \in V_1$ be such that $(y, v) \in E_1$, i.e., y is a predecessor of v . Now, by *Entry Lemma 5.1*, there exists a plan g such that, setting

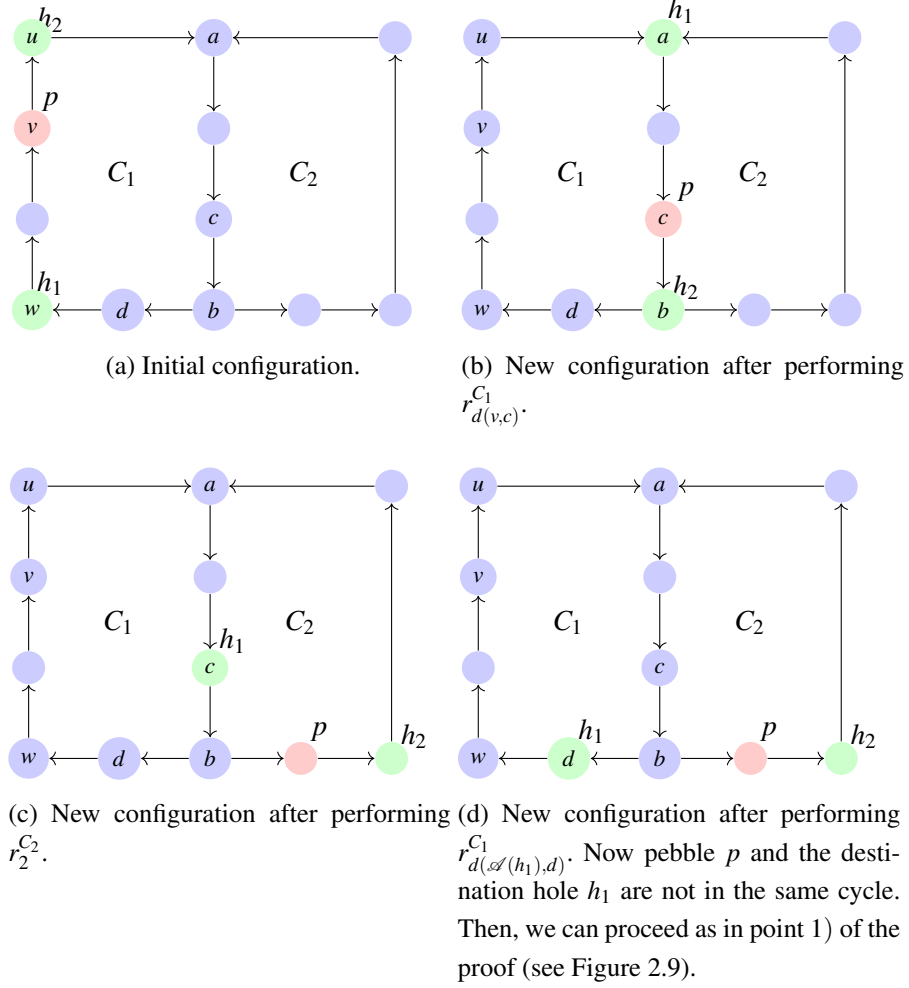


Figure 2.10: An example of the situation described in Stay in Lemma 5.4 in which the initial position of the pebble $v = \mathcal{A}(p)$ and the target $w = \mathcal{A}(h_1)$ belong to the same cycle C_1 . In this case we move p on another cycle and then we proceed as in Figure 2.9.

$\mathcal{A}^1 = \rho(\mathcal{A}, g)$, $\mathcal{A}^1(p) = b$ and, for all $q \in H \cup P \setminus \{p\}$ such that $\mathcal{A}^1(q) \in V_2$, $\mathcal{A}^1(q) = \mathcal{A}(q)$ holds. Plan g is defined as follows

$$g = h_{v,s(b)} R_k^C t_{a,v} (R_k^C)^+ h_{v,s(b)}^{-1},$$

where C is a sequence of cycles, k a vector, and $t_{a,v}$ is a plan which moves pebble p to the unoccupied vertex v . In particular, if B_1 :

- has a *r-oed*: $t_{a,v} = a \Rightarrow v$, (see Definition 2.1.3) which by Theorem 1.1 exists since in B_1 there is at least one hole (h_1);
- is a partially-bidirectional cycle: $t_{a,v} = r_{d(a,v)}^{C_{B_1}}$.

After performing g , both holes h_1 and h_2 are in B_1 (in particular, $\mathcal{A}^1(h_2) = v$). If B_1 has a *r-oed*, by Lemma 5.4 there exists a plan f so that $\mathcal{A}^2 = \rho(\mathcal{A}^1, f)$ is such that $\mathcal{A}^2(h_1) = a$ and for all $q \in P \cup H \setminus \{h_1\}$, $\mathcal{A}^2(q) \in V_1$, $\mathcal{A}^2(q) = \mathcal{A}(q)$. So, finally $\mathcal{A}^2 = \mathcal{A}[a, b]$. If B_1 is a partially-bidirectional cycle, performing $r_{d(v,a)}^{C_{B_1}}$ is sufficient to bring p on a and the other pebbles of B_1 on their initial positions.

2. Suppose that $a, b \in V_1$.

- B_1 has a *r-oed*. We can assume without loss of generality that $\mathcal{A}(h_2) \in V_1$ (if not, it would be enough to BRING HOLE FROM $\mathcal{A}(h_2)$ TO v and finally BRING BACK HOLE FROM v TO $\mathcal{A}(h_2)$). Since in B_1 there are two holes, by Lemma 5.4, we can move p to b without changing the final position of the other pebbles.
- B_1 is a cycle. We can assume without loss of generality that $\mathcal{A}(h_2) \in V_2 \setminus \{v\}$. Then, by point 1) of this proof, we can first move p from a to $\mathcal{A}(h_2)$ and then from $\mathcal{A}(h_2)$ to b , without changing the final position of the other pebbles.

□

2.5.3 CONVERT-PATH algorithm

The following theorem shows that each plan on the biconnected components tree can be converted into a plan on the original digraph. The proof uses the Lemmas defined above to explicitly show how to convert a plan f' on the tree into a plan f on the digraph. This allows explicitly defining *CONVERT-PATH* algorithm.

Theorem 5.6. *Let P be a set of pebbles and H a set of holes on a strongly connected digraph $D = (V, E)$, which is not a partially-bidirectional cycle, and let $T = (V_T, E_T)$ be the corresponding biconnected component tree. Let $\mathcal{A}$ be an initial configuration on D and $\tilde{\mathcal{A}}$ the corresponding configuration on T . Let $a, b \in V$ and $p \in P$ be a pebble on a . Then, if $|H| \geq 2$, there is a plan f_{ab} on D such that $\mathcal{A}[a, b] = \rho(\mathcal{A}, f_{ab})$ if and only if there is a plan f'_{ab} on T such that $\mathcal{A}_T[a, b] = \rho(\mathcal{A}_T, f'_{ab})$.*

Proof. Let f_{ab} be a plan on D . For every single move $m = u \rightarrow v$ in this plan, recalling Observation 2.3.2 and noting that u, v either belong to the same biconnected component or the same corridor, there are two cases. In the first case, there exists a non-trivial strongly biconnected component $B = (V_B, E_B)$ such that $u, v \in V_B$. Then, we define m' , a corresponding plan on T , as follows: let S be the star in T corresponding to B , and let s be the trans-shipment vertex of S ; then, $m' = (u \rightarrow s)(s \rightarrow v)$. In the second case, i.e., u, v belong to the same corridor, then $m' = m$. f'_{ab} is defined as the composition of all the moves m' just described.

Conversely, let f'_{ab} be a plan on T . Then, by Observation 2.5.2

1. If a and b are not in the same star on T , then they are not in the same strongly biconnected component on D . By Observation 2.5.2 f_{ab} will be composed by movements: from/to a corridor to/from a strongly biconnected component (f_{ab} exists by *Attached-Edge Lemma 5.3*); from a strongly biconnected component to another one, connected by an articulation point (f_{ab} exists by *Two Biconnected Components Lemma 5.5*); from a node to another one of the same corridor ($f'_{ab} = f_{ab}$).
2. If a and b belong to the same "star" on T , then a and b belong to the same

strongly biconnected component $B_1 = (W_1, F_1)$ on D . There are two possibilities:

- (a) B_1 has at least two holes. In this case, if B_1 has a *r-oed*, by Lemma 5.4 $f_{a,b}$ exists. If B_1 is a partially-bidirectional cycle, there are two cases:
 - there is another biconnected component $B_2 = (W_2, F_2)$ such that B_1 and B_2 are joined by an articulation point. In this case existence of $f_{a,b}$ follows from Lemma 5.5;
 - there is a node $v \in V \setminus W_1$ such that a node $w \in W_1$ with $(v, w), (w, v) \in E$ exists. Therefore, $G = (W_1 \cup \{v\}, F_1 \cup \{(v, w), (w, v)\})$ is a cycle with an attached edge, and existence of $f_{a,b}$ follows from Lemma 5.3;
- (b) B_1 has only one hole. In this case, let h_1 be the hole on b and h_2 a hole such that $\mathcal{A}(h_2) = w \notin W_1$ (which exists since $|H| \geq 2$) and $u \in W_1$ a node different from a and b (which exists since non-trivial biconnected components have at least three nodes). Let $h_{w,u}$ be the plan BRING HOLE FROM w TO u which moves the hole, and $\tilde{\mathcal{A}} = (\mathcal{A}, h_{w,u})$. Then:
 - a') if $\tilde{\mathcal{A}}(p), \tilde{\mathcal{A}}(h_1) \in W_1$, then we do perform BRING HOLE FROM w TO u , we replace a and b with $\tilde{\mathcal{A}}(p)$ and $\tilde{\mathcal{A}}(h_1)$, and by point a) we find the plan $f_{\tilde{\mathcal{A}}(p)\tilde{\mathcal{A}}(h_1)}$; finally we perform $h_{w,u}^{-1}$ and h_2 returns to its initial position.
 - b') if $\tilde{\mathcal{A}}(h_1) \in W_1$ but $\tilde{\mathcal{A}}(p) \notin W_1$, we do perform BRING HOLE FROM w TO u and we fall into the case where the start and final position of the pebble are not in the same biconnected component. Therefore, $f_{ab} = h_{w,u} f_{\tilde{\mathcal{A}}(p)\tilde{\mathcal{A}}(h_1)} h_{w,u}^{-1}$, where $f_{\tilde{\mathcal{A}}(p)\tilde{\mathcal{A}}(h_1)}$ exists by point 1).
 - c') if $\tilde{\mathcal{A}}(p) \in W_1$ but $\tilde{\mathcal{A}}(h_1) \notin W_1$, we do not perform BRING HOLE FROM w TO u . First, we move h_1 away from b by BRING HOLE FROM b TO u . Then, we perform BRING HOLE FROM w TO b . Then, we can proceed as in a') or b') with u replaced by b . Finally, we will need to perform BRING BACK HOLE FROM u TO b . In formulas, given $\tilde{\mathcal{A}} = \rho(\mathcal{A}, h_{b,u} h_{w,b})$ the final plan is $f_{ab} = h_{b,u} h_{w,b} f_{\tilde{\mathcal{A}}(p)\tilde{\mathcal{A}}(h_1)} h_{w,b}^{-1} h_{b,u}^{-1}$.

2.6 Upper bound for solutions length and time complexity

In this section, we provide an upper bound for the length of MAPF solutions found by the diSC Algorithm. For a MAPF instance on a graph G composed by K non-trivial biconnected components, this length depends on:

- A. the solution length of the ts -PMT problem on the associated biconnected components tree,
- B. the number of trans-shipment vertex crossings in the ts -PMT solution,
- C. the length complexity of the conversion of a movement through a trans-shipment vertex on the tree into a plan on the digraph,
- D. the length complexity of plan BRING HOLE FROM v TO w and the number of times this plan is used.

We can bound these quantities as follows.

A. In the literature, the best upper bound for PMT solutions is $O(|P|nc + n^2)$, where c is the maximum length of the corridors of the tree (it is proved by Ardizzoni et al. [Pub1]).

B. In the procedure described by Ardizzoni et al. [Pub1], any vertex is crossed at most $O(|P|c)$ times by the pebbles in the overall solution (see Proposition 5.3 of reference [Pub1]).

C. To calculate the cost of the conversion, first, we have to compute the length of a k -CYCLE ROTATION. Let C_i be a cycle with n_i nodes and $k \leq n_i$. Then a 1-CYCLE ROTATION takes $O(n_i)$, while a k -CYCLE ROTATION takes $O(k \cdot n_i)$ which, in the worst case, is $O(n_i^2)$. If $C = [C_1, \dots, C_m]$ is an ordered sequence of cycles with n nodes, and $k = (k_1, \dots, k_m) \in \mathbb{N}^m$, the length complexity of the plan R_k^C , obtained by concatenating a k_1 -CYCLE ROTATION over C_1 , a k_2 -CYCLE ROTATION over C_2 , and analogous

rotations over the remaining cycles of C are given by

$$O\left(\sum_{i=1}^m k_i \cdot n_i\right)$$

Therefore, the complexity of a *complete* k -CYCLE ROTATION of C is $O(n^2)$. Note that also the plan BRING HOLE FROM v TO w has quadratic length, since it is a k -CYCLE ROTATION.

The plans described in *Stay in Lemma 5.4*, *Entry Lemma 5.1*, *Go out Lemma 5.2*, *Attached-Edge Lemma 5.3* and *Two Biconnected Components Lemma 5.5* are all combination of plans of the type k -CYCLE ROTATION and BRING HOLE FROM v TO w . Therefore, on a biconnected component with n_j nodes, they have $O(n_j^2)$ length.

D. To calculate the total cost of the function *CONVERT-PATH*, in addition to the cost of the plans presented in the lemmas, we must also consider how many times we use function BRING HOLE FROM v TO w . Indeed, in the Proof of Theorem 5.6, when a pebble has to move within a biconnected component, but there are not enough holes to perform this movement, we use BRING HOLE FROM v TO w to add a hole. In the worst case, BRING HOLE FROM v TO w is used once for each traversal of a trans-shipment vertex. Since the complexity of this plan is $O(n^2)$ and from point B, we know that each vertex is crossed $O(|P|c)$ times, we conclude that the overall complexity due to BRING HOLE FROM v TO w is $O(|P|n^2c)$.

If graph G is composed of K non-trivial biconnected components, the cost of the conversion performed by the function *CONVERT-PATH* is

$$\underbrace{O(|P|n^2c)}_D + \sum_{i=1}^K \left(\underbrace{O(|P|c)}_B \cdot \underbrace{O(n_i^2)}_C \right) =$$

$$O(|P|n^2c) + O(|P|c) \cdot O(n^2 + K) = O(|P|n^2c).$$

where n_i is the number of nodes of the i -th biconnected component, and $\sum_{i=1}^K n_i \leq n + K$ since some nodes can belong to more than one biconnected component.

Since the length complexity of the PMT solution is $O(|P|nc + n^2)$ (see point A), the overall complexity is dominated by the conversion cost, and the following results

hold:

Theorem 6.1. *The length complexity of the diSC algorithm is $O(|P|n^2c)$.*

Note that with the diSC Algorithm, we find a better complexity result than the one obtained through the method of Theorem 4.2 (see Proposition 4.4).

Remark 6.2. Note that the choice between the diSC algorithm and the procedure described in Theorem 4.2 depends on the structure of the digraph and the number of pebbles. If cycles of the graph have a small length, the latter procedure might be more convenient. For example, if $N = 3$ the complexity is $O(9n^3) = O(n^3)$. Instead, if the corridors have a delimited length with respect to the dimensions of the graph (for example, $c = 6$), the diSC algorithm is usually preferable. In any case, we note that in the worst case, the diSC algorithm has a complexity of $O(n^4)$, while the other procedure has a complexity of $O(n^5)$ in the worst case.

2.7 Experimental Results

We implemented the diSC algorithm in Matlab. We tested it on random graphs as well as on real-life scenarios. We used a *Intel(R) Core(TM) i7-4510U CPU @ 2.60 GHz* processor with 16 GB of RAM. Note that we did not employ standard benchmark graphs used for MAPF (see, e.g., [2]), such as graphs from the MovingAI library. Indeed, while these graphs may vary in shape, they are usually grid-based, undirected, and biconnected graphs (see, e.g., DAO maps and Open $N \times N$ grids). Instead, we used one of the graphs present in the MovingAI library to define a random, directed, strongly connected graph.

2.7.1 Random graphs

We generated random graphs in two different ways.

Random graphs based on trees.

To generate graphs with multiple biconnected components, we used the following procedure, described also in [Pub3]. First, we create a random connected undirected

graph with function `networkx.connected_watts_strogatz_graph()`, of NetworkX library¹. Then, we construct a maximum spanning rooted tree. We process the tree nodes with a breadth-first order. Every node that has a number n of children higher than 1 is converted into a biconnected component, together with its children, with the following method. We substitute the parent node and its children with a directed cycle with a random number of nodes lower or equal to $n + 1$. Then, we add directed ears of random length (but sufficiently small, not to exceed the total number of $n + 1$ nodes assigned to the biconnected component) and random initial and final nodes until the number of nodes in the resulting biconnected component equals $n + 1$. After processing the tree, every remaining undirected edge $\{u, v\}$ is converted into two directed edges (u, v) , (v, u) .

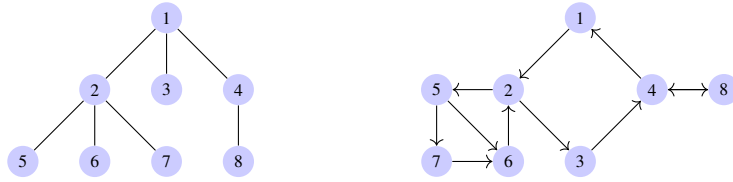


Figure 2.11: From tree to strongly connected digraph.

For every number of nodes, we generated a set of 100 graphs. We ran the algorithm varying both the number of nodes and the number of agents. For every generated graph (100 for every different number of nodes), we created MAPF problem instances with agents that vary from 1 to half of the number of nodes and random initial and final positions. For example, we generated 100 graphs with 60 nodes, and for every graph, we created one instance for every number of agents varying from 1 to 30, resulting in 3000 instances. For each obtained solution, we recorded the overall number of moves and the computation time.

¹<https://networkx.org/>

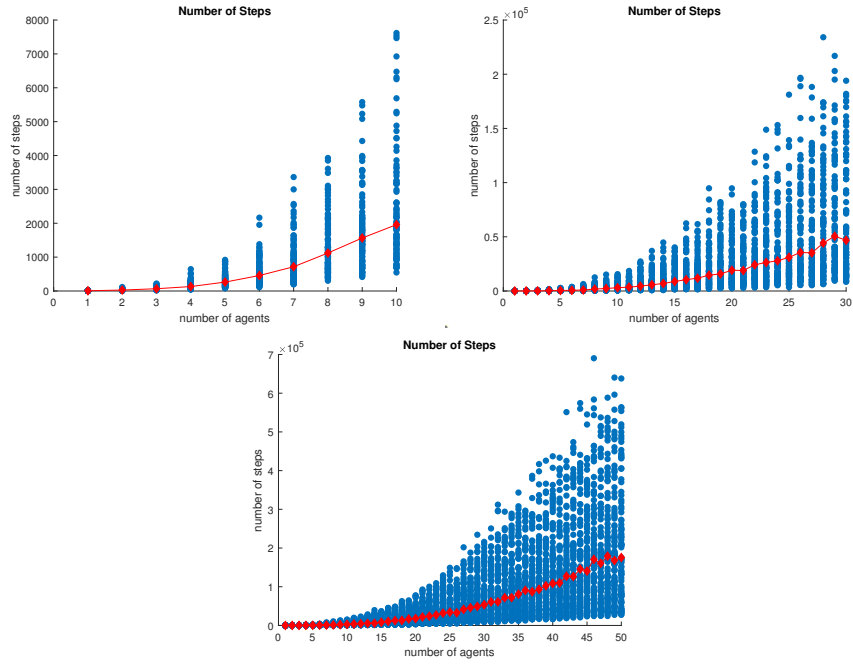


Figure 2.12: Number of steps per number of agents of every instance in random graphs with 20, 60, and 100 nodes. The median of the number of steps of instances for each number of agents is shown in red.

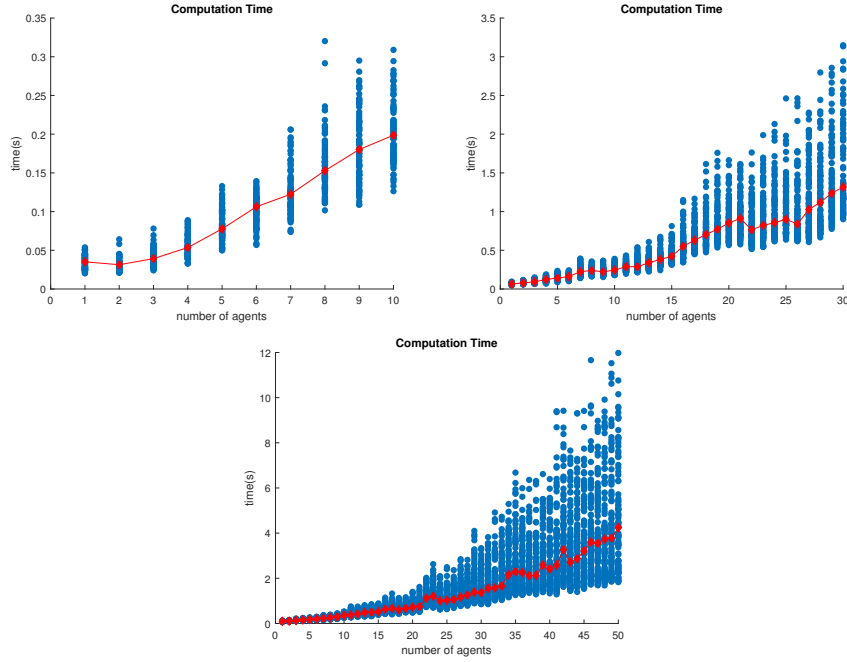


Figure 2.13: Running time per number of agents on random graphs with 20, 60, and 100 nodes. The median of the running times of instances for each number of agents is shown in red.

Fig. 2.12 shows the solution lengths of all the instances as a function of the number of agents on graphs with 20, 60, and 100 nodes. In red are shown the medians of the number of steps for every different number of agents. Roughly, the overall number of steps grows polynomially with respect to the number of agents, in line with Theorem 6.1. In Fig. 2.13, we can see the running times of the different instances with the corresponding medians shown in red. Similarly to the number of steps, the running time grows polynomially with respect to the number of agents.

Graphs generated by randomly added edges.

We also considered a second method for generating random graphs. We start from an empty graph and randomly add edges until the resulting graph is strongly connected.

This procedure gives us graphs that are more connected than the one generated before. With this procedure, we generated random graphs with 20, 60, and 100 nodes. As done before, for every number of agents and number of nodes, we created 100 MAPF problem instances. For each number of nodes, the number of agents varies from 1 to half of the number of nodes in the first test and from 1 to the highest number of agents possible for the second one. For each solution, we recorded the number of steps and the computation time.

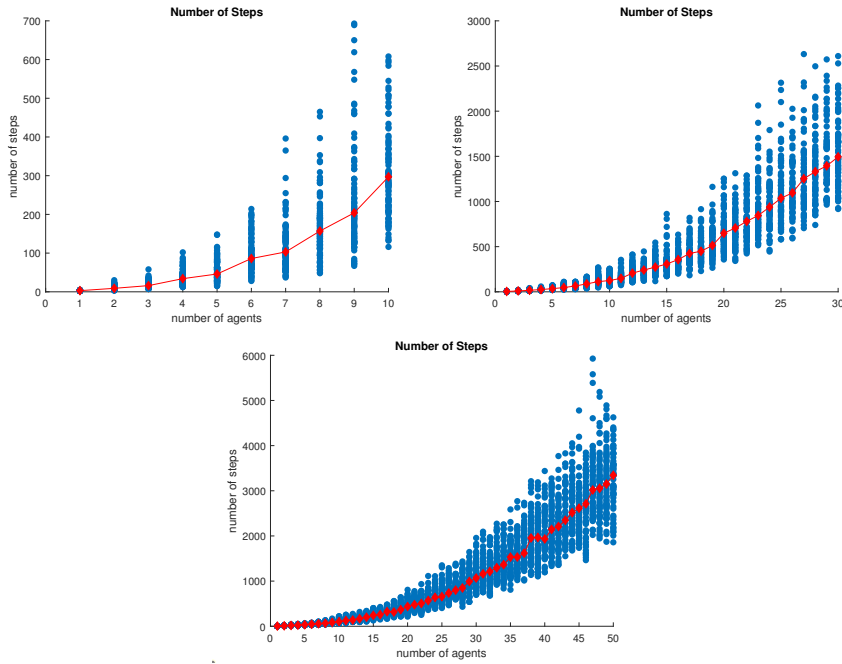


Figure 2.14: Number of steps per number of agents of every instance in random graphs with 20, 60, and 100 nodes. The median of the number of steps of instances for each number of agents is shown in red.

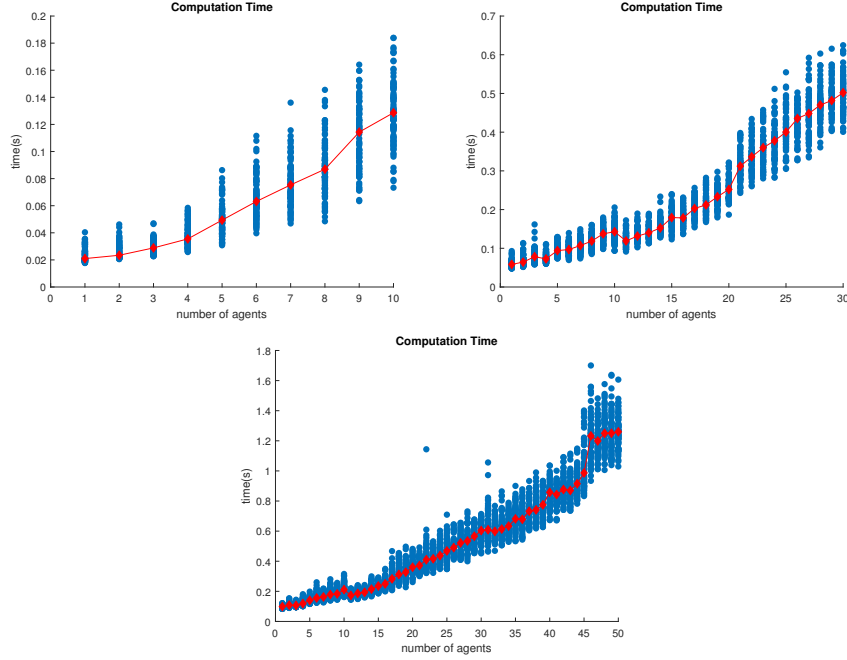


Figure 2.15: Running time per number of agents of every instance in random graphs with 20, 60, and 100 nodes. The median of the running times of instances for each number of agents is shown in red.

Fig. 2.14 shows the number of steps of the obtained solutions, with medians highlighted in red. Also, in this case, the number of steps appears to increase polynomially with respect to the number of agents. By comparing this picture and Fig. 2.12, we can see how, in the second case, the number of steps is much lower. This can be explained by looking at the structure of the graphs. In the first case, we have more corridors and more biconnected components, whereas in the second case, the graph is more connected, and the number of biconnected components is much lower. The lower number of biconnected components results in a lower number of steps in both the initial solution on the tree and the final solution on the strongly connected graph. Fig. 2.15 shows the running times of the solutions. The red line represents the medians of the

running times for each number of agents. Running times grow polynomially with respect to the number of agents.

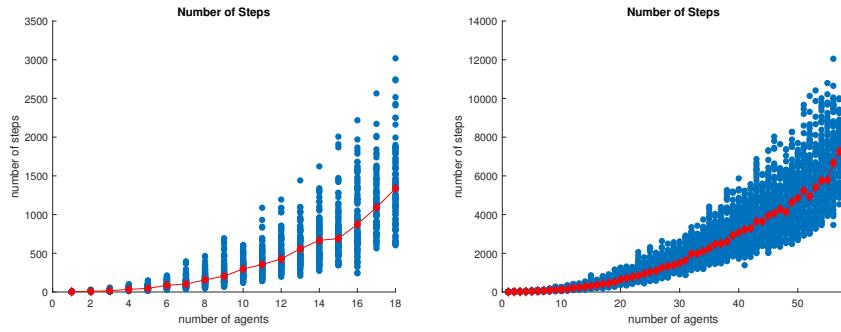


Figure 2.16: Number of moves per number of agents on random graphs with 20 and 60 nodes for a high number of agents. The median of the number of steps of instances for each number of agents is shown in red.

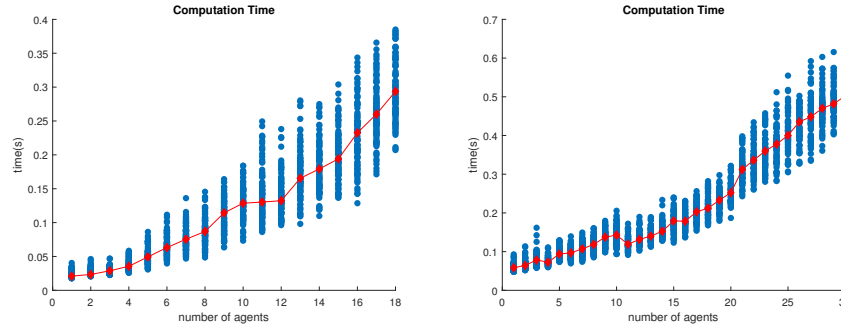


Figure 2.17: Running time per number of agents on random graphs with 20 and 60 nodes for a high number of agents. The median of the running time of instances for each number of agents is shown in red.

In Fig. 2.16, the number of steps of instances with a high number of agents is shown, with medians highlighted in red. In particular, the maximum considered number of

agents is $|P| = n - 2$, where n is the number of nodes. Note that some instances with many agents can be infeasible. We removed such instances. Note that the growth of the solution length with respect to the number of agents seems to be polynomial in all the considered ranges. In Fig. 2.17, we can see the running time of these instances, with the medians shown in red. Also, in this case, the running time seems to grow polynomially with respect to the number of agents.

2.7.2 Graph based on *movingai* MAPF benchmarks

We tested the algorithm on a benchmark for Multi-Agent Path Finding problems presented in [2].

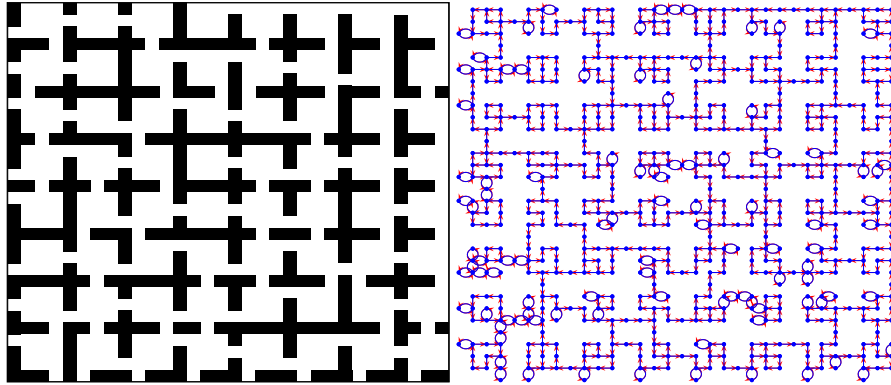


Figure 2.18: Strongly connected digraph obtained from room graph of *movingai* MAPF benchmark

Figure 2.18 shows a grid map composed of 64 rooms, each consisting of 3x3 cells and interconnected by doors. We defined an undirected graph, where each white cell corresponds to a node, and each edge links nodes of adjacent cells. Then, we generated a directed graph by transforming each edge of the undirected graph into two opposite edges of the directed graph. Then, we randomly removed some directed edges, still guaranteeing that the resulting graph is strongly connected. The result is a strongly connected directed graph with 682 nodes and 870 edges. We created 100 instances for

each number of agents from 5 to 35, with a step of 5. In these instances, we randomly selected the initial and final configurations of the agents.

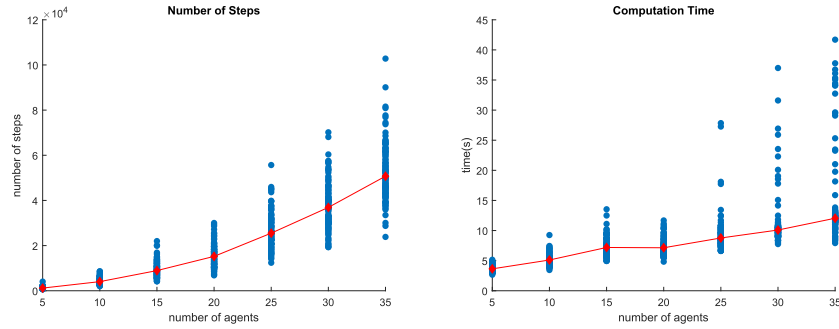


Figure 2.19: Number of moves and running time per number of agents on the graph in Figure 2.18. The median of the number of steps and running time of instances for each number of agents is shown in red.

Figure 2.19 shows the number of steps and the computation times wrt the number of agents, with medians highlighted in red. Even in this case, both the number of steps and the computation time appear to grow polynomially wrt the number of agents.

2.7.3 Real-life applications examples

As a second example, we considered a real-life warehouse layout provided by packaging company Ocme S.r.l., based in Parma, Italy. This layout is associated with a strongly connected digraph with 397 nodes (see Figure. 2.20). This graph consists of a biconnected component with 304 nodes and corridors connected to it. The corresponding biconnected component tree does not contain any corridors with endpoints that are only articulation points. Therefore, $\mathcal{C}$ is empty and $\bar{c}_T^2 = 0$. Additionally, since $\bar{c}_T^1 = 4$, it follows that $\bar{c}_T = 5$. We note that the necessary and sufficient condition (2.3) becomes $|H| \geq 6$, which is always satisfied in the simulations we performed.

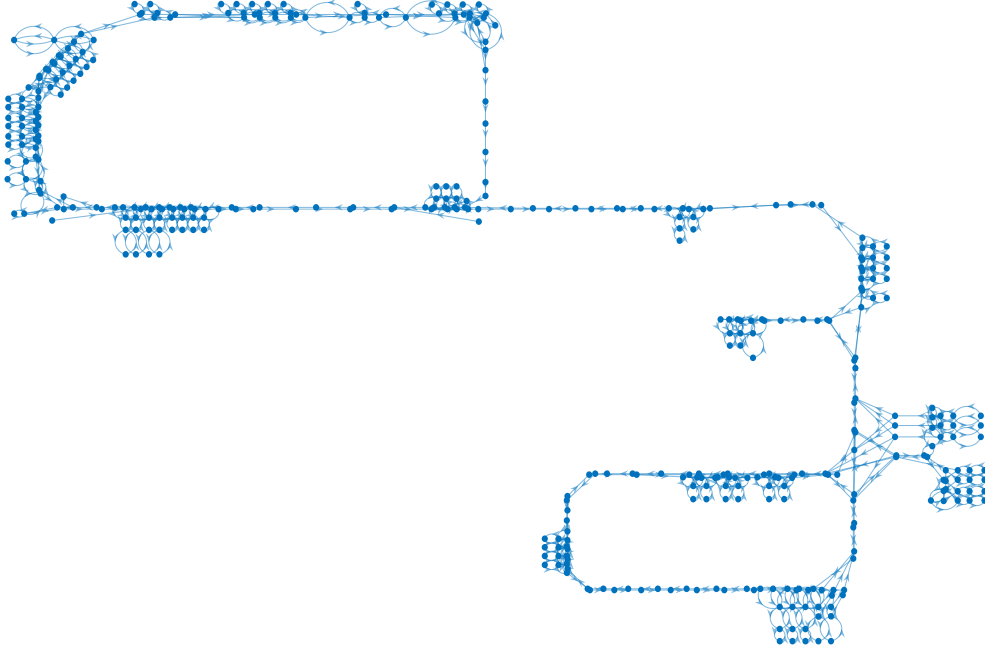


Figure 2.20: Graph representing the warehouse.

We ran the algorithm, varying the number of agents from 1 to 10. For each number of agents we generated 100 random MAPF instances, and we solved them with the diSC algorithm. Fig. 2.21 shows the number of steps and the running time with respect to the number of agents, with the corresponding medians highlighted in red. Both the number of steps and the running time seem to grow polynomially with respect to the number of agents.

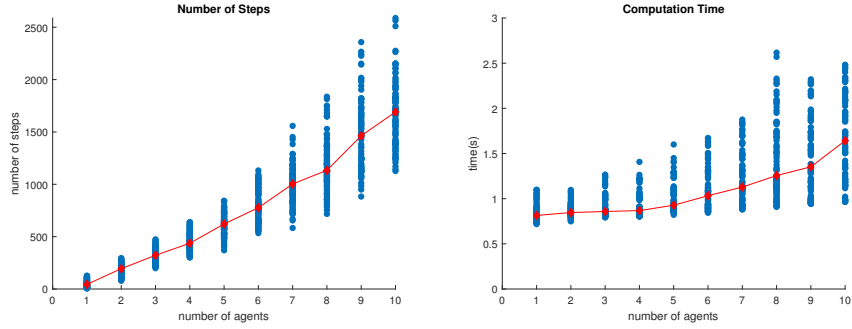


Figure 2.21: Number of moves and computation times per number of agents in the warehouse of Figure 2.20, with medians highlighted in red

Moreover, we generated graphs with the same structure as the warehouse of Figure 2.20. In particular, we created graphs with the same number of nodes and the same node degrees as the warehouse. To do so, we used the NetworkX² function *directed_configuration_model()* that creates a random graph with the specified in and out degrees.

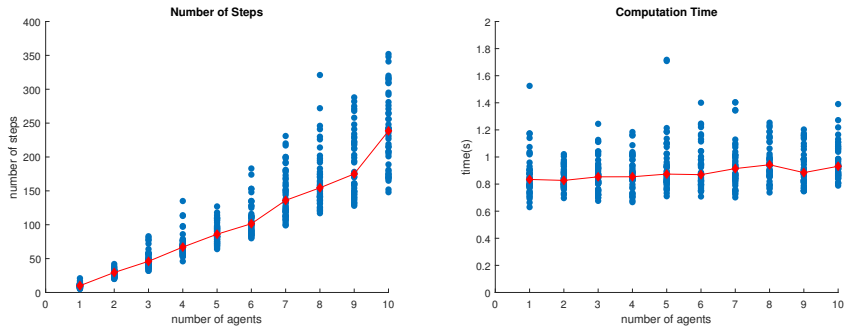


Figure 2.22: Number of moves and computation times per number of agents in the graphs with the same structure as Figure 2.20.

²<https://networkx.org/>

We generated 100 random instances for each number of agents, varying from 1 to 10 agents. Fig. 2.22 shows the number of steps and the running time, with the corresponding medians highlighted in red. Both seem to grow polynomially with respect to the number of agents.

As demonstrated by these tests, the diSC algorithm proves effective in managing warehouse agents, particularly in scenarios involving congestion and deadlock risk. The results of the simulations show that agent paths can be computed in a time below one second, even in cases with a growing number of agents. This indicates that the algorithm is suitable for real-time applications, enabling frequent path replanning. However, the solution provided by the diSC algorithm cannot be considered final, as it does not allow for simultaneous agent movements. Nevertheless, these solutions can serve as a starting point for local search algorithms that minimize agent movements and allow for simultaneous motion [Pub6] [22]. Furthermore, the algorithm does not take into account the physical dimensions of the agents, treating them as point-like entities. Therefore, it may be beneficial to complement it with other algorithms that consider the volume occupied by the agents, as well as additional constraints imposed by the warehouse structure (see, e.g., [Pub1]).

Chapter 3

Dynamic Programming based Local Search

In the first two chapters, we focused on sub-optimal rule-based methods for solving MAPF instances on both undirected and directed graphs. While these approaches offer good computational complexity, the solutions they produce are often far from optimal. This chapter aims to improve such feasible solutions with a procedure that has polynomial computational complexity with respect to both the numbers of nodes and agents. This chapter refers to the work in [Pub6] and [Pub5]. To improve the given feasible solutions, we introduce an iterative local search procedure for MAPF. Starting with a sub-optimal solution, we perform a local search within its neighborhood to find a shorter and more efficient solution. This procedure is repeated iteratively until no further improvements can be made. The resulting solution, while still sub-optimal, is generally of much higher quality than the initial one. We use dynamic programming for the local search procedure. The locality constraint is crucial in reducing the time complexity of the algorithm. Although the general MAPF problem could theoretically be solved using dynamic programming, the number of states to explore grows exponentially with the number of agents, making it impractical for larger problems. By introducing the locality constraint, we limit the space of the dynamic programming search, allowing us to solve the (local) dynamic programming

problem in a time that grows only polynomially with respect to the number of agents.

Notation

A directed graph is a pair $G = (V, E)$ where V is a set of nodes and $E \subset \{(x, y) \in V^2 \mid x \neq y\}$ is a set of directed arcs. A path p on G is a sequence of adjacent nodes of V (i.e., $p = \sigma_1 \cdots \sigma_m$, with $(\forall i \in \{1, \dots, m\}) (\sigma_i, \sigma_{i+1}) \in E$). An alphabet $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ is a set of symbols. A word is any finite sequence of symbols. The set of all words over Σ is Σ^* , which also contains the empty word ε . Given a word $w \in \Sigma^*$, $|w|$ denotes its length. Given $s, t \in \Sigma^*$, the word obtained by writing t after s is the concatenation of s and t , denoted by $st \in \Sigma^*$; we call t a suffix of st and s a prefix of st . Use prefixes and suffixes to represent portions of paths. A Finite State Automaton (FSA) is a quintuple $(\Sigma, X, x_0, \delta, F)$, where Σ is a finite set of symbols, X is a set of states, $x_0 \in X$ is the initial state and $\delta : X \times \Sigma \rightarrow X$ is the transition function. Function δ can be partially defined, that is, not defined for all elements of the domain. Given $x \in X$, $\sigma \in \Sigma$, if $\delta(x, \sigma)$ is defined, we use notation $\delta(x, \sigma)!$. Function δ can be extended to a function $\delta : X \times \Sigma^* \rightarrow X$ by setting $\delta(x, s\sigma) = \delta(\delta(x, s), \sigma)$ and $\delta(x, \varepsilon) = x$.

3.1 Problem definition

Let $G = (V, E)$ be a directed graph, with vertex set V and edge set E . We identify each agent with a unique label, and the ordered set P contains these labels. A *configuration* is a function $\mathcal{A} : P \rightarrow V$ that assigns the occupied vertex to each agent. A configuration is *valid* if it is injective (i.e., each vertex is occupied by at most one agent). Set $\mathcal{C} \subset \{P \rightarrow V\}$ represents all valid configurations. We represent a configuration $\mathcal{A} \in \mathcal{C}$ also by an ordered set of vertices $(v_1, v_2, \dots, v_r)$, where v_i , $i = 1, \dots, r$ is the vertex occupied by the i -th agent. Time is assumed to be discretized. At every time step, each agent occupies one vertex and executes a single action. There are two types of actions: *wait* and *move*. We denote the wait action by ι . An agent that executes this action remains in its current vertex for another time step. We denote a move action by $u \rightarrow v$. In this case, the agent moves from its current vertex u to an adjacent vertex

v (i.e., $(u, v) \in E$). Therefore, the set of all possible actions for a single agent is

$$\bar{E} = E \cup \{\iota\}.$$

Function $\rho : \mathcal{C} \times \bar{E} \rightarrow \mathcal{C}$ is a partially defined transition function such that $\mathcal{A}' = \rho(\mathcal{A}, u \rightarrow v)$ is the configuration obtained by moving an agent from u to v :

$$\mathcal{A}'(q) := \begin{cases} v, & \text{if } \mathcal{A}(q) = u; \\ \mathcal{A}(q), & \text{otherwise.} \end{cases} \quad (3.1)$$

Notation $\rho(\mathcal{A}, u \rightarrow v)!$ means that the function is well-defined. In other words $\rho(\mathcal{A}, u \rightarrow v)!$ if and only if $(u, v) \in E$ and $\mathcal{A}' \in \mathcal{C}$. Moreover, $(\forall \mathcal{A} \in \mathcal{C}) \rho(\mathcal{A}, \iota)!$ and $\rho(\mathcal{A}, \iota) = \mathcal{A}$. Since multiple agents can move at the same time step, an action of the whole fleet is an element $a = (a_1, \dots, a_{|P|})$ of $\mathcal{E} = \bar{E}^{|P|}$, where a_i is the action of agent i . We can extend function $\rho : \mathcal{C} \times \bar{E} \rightarrow \mathcal{C}$ to $\rho : \mathcal{C} \times \mathcal{E} \rightarrow \mathcal{C}$, by setting $\mathcal{A}' = \rho(\mathcal{A}, a)$ equal to the configuration obtained by moving agent i along edge a_i (or by not moving the agent if $a_i = \iota$). In this case, $(\forall a \in \mathcal{E}, \mathcal{A} \in \mathcal{C}) \rho(\mathcal{A}, a)!$ if and only if the following conditions hold:

1. $\mathcal{A}' \in \mathcal{C}$: two or more agents cannot occupy the same vertex at the same time step;
2. $\forall i = 1, \dots, |P|$, if $a_i = (u, v)$, then $\nexists j \in \{1, \dots, |P|\}$ such that $a_j = (v, u)$: two agents cannot swap locations in a single time step.

We represent plans as ordered sequences of actions. It is convenient to view the elements of $\mathcal{E}$ as the symbols of a language. We denote by $\mathcal{E}^*$ the Kleene star of $\mathcal{E}$, that is the set of ordered sequences of elements of $\mathcal{E}$ with arbitrary length, together with the empty string ε :

$$\mathcal{E}^* = \bigcup_{i=1}^{\infty} \mathcal{E}^i \cup \{\varepsilon\}.$$

We extend function $\rho : \mathcal{C} \times \mathcal{E} \rightarrow \mathcal{C}$ to $\rho : \mathcal{C} \times \mathcal{E}^* \rightarrow \mathcal{C}$ according to the following rules

1. $(\forall s \in \mathcal{E}^*, e \in \mathcal{E}, \mathcal{A} \in \mathcal{C}) \rho(\mathcal{A}, se)!$ if and only if $\rho(\mathcal{A}, s)!$ and $\rho(\rho(\mathcal{A}, s), e)!$

2. if $\rho(\mathcal{A}, se)!$, then $\rho(\mathcal{A}, se) = \rho(\rho(\mathcal{A}, s), e)$.

Note that ε is the trivial plan that keeps all agents at their positions. In some cases, it is convenient to represent the individual plan of each agent, defining a plan f as an element of $((\bar{E})^*)^{|P|}$. In this way, $f \in \mathcal{E}^*$ is a $|P|$ -tuple $f = (f^1, \dots, f^{|P|})$, where $f^i = \{a_i^1, \dots, a_i^{|f|}\}$ represents the ordered sequence of the actions of agent i . For all $i = 1, \dots, |P|$, we call f^i the *path plan of agent i* . We denote by $\mathcal{L}$ the set of plans such that $\rho(\mathcal{A}, f)$ is well-defined:

$$\mathcal{L} = \{f \in \mathcal{E}^* : \rho(\mathcal{A}, f)!\}.$$

The problem of detecting a feasible solution is the following:

Problem 1. (Feasibility MAPF problem). Given a digraph $G = (V, E)$, an agent set P , an initial valid configuration $\mathcal{A}$, and a final valid configuration $\mathcal{A}^t$, find a plan $f \in \mathcal{L}$ such that $\mathcal{A}^t = \rho(\mathcal{A}, f)$.

It is natural to represent the agents' behaviour by an FSA $(\mathcal{E}, \mathcal{C}, \mathcal{A}, \rho, \{\mathcal{A}^t\})$, where the alphabet $\mathcal{E}$ is the set of actions, the state set $\mathcal{C}$ consists in valid configurations, $\mathcal{A}$ is the initial state, ρ is the transition function, and $\mathcal{A}^t$ is the final state. In this way, Problem 1 corresponds to a reachability task for the FSA.

For a feasible plan f , we define $|f|$ as the length of plan f (i.e., the number of time steps needed for all agents to reach the final configuration through plan f). Furthermore, given $k \in \mathbb{N}$, we denote by f_k the k -th prefix of f (that is, the prefix of f of length k , made up of the first k actions of f). Note that $|f_k| = k$. Also, note that $|f|$ corresponds to the cost function usually called *Makespan*. Given a MAPF instance, with initial configuration $\mathcal{A}$ and final configuration $\mathcal{A}^t$, the *optimization MAPF problem* aims at finding a feasible plan f of minimal length $|f|$.

Problem 2. (Optimization MAPF problem). Let $\mathcal{A}$ and $\mathcal{A}^t$ be valid configurations on a digraph G , solve

$$\begin{aligned} \min & |f| \\ \text{s.t. } & \mathcal{A}^t = \rho(\mathcal{A}, f) \\ & f \in \mathcal{L}. \end{aligned} \tag{3.2}$$

Other cost functions have also been used in the literature. *Sum-of-costs*, for example, is the sum of the number of time steps that each agent employs to reach its target without leaving it again. Unfortunately, finding the optimal solution (i.e., the minimal *Makespan* or *sum-of-costs*) is NP-hard [3]. Intuitively, this happens since the set of possible plans grows exponentially with respect to the number of agents and nodes. In this chapter, we propose an approach to find a good quality sub-optimal solution in polynomial time. Roughly speaking, the method we propose is a local search outlined in what follows:

- Using a rule-based algorithm, we find a plan f that solves the feasibility MAPF problem.
- We define a neighborhood of plan f , whose size grows polynomially with respect to the number of agents and nodes. In this neighborhood, we look for a feasible solution $\hat{f}$, shorter than f .
- If we can find such a solution, we set $f = \hat{f}$ and reiterate the local search. Otherwise, we stop and return the current plan.

This algorithm runs in polynomial time with respect to the number of agents and nodes. Indeed, under mild assumptions, a rule-based algorithm provides a feasible solution in polynomial time, and this solution has polynomial length. At each iteration, the length is decreased by at least one so that the number of iterations is polynomially bounded. Finally, we will define neighborhoods with polynomial cardinality with respect to the number of agents and nodes. Thus, they can be explored in polynomial time. At the end of the procedure, we obtain, in polynomial time, a solution to the MAPF problem that is locally optimal, with respect to the employed neighborhood, but not necessarily globally optimal.

3.2 Measuring the distance between plans

The key point of this chapter is to define suitable neighborhoods of a reference plan f , whose cardinality grows polynomially with respect to the number of agents and

nodes. As a first step, we show possible ways to measure the distance between a generic plan g and a reference plan f . We introduce two families of distances:

- In the first family (path distances), we take into account the graph distance between the nodes occupied by the agents in f and g .
- In the second family (agent distances), we count the number of agents in f and g that have different individual plans.

In the following, we define these two families in more detail.

3.2.1 Path distances

In this section, we focus on distances based on the length of the shortest paths connecting the vertices of the graph.

Definition 3.2.1. Let $G = (V, E)$ be a digraph and P be a set of agents. We define the distance of vertex u from vertex v as the length of the shortest path on G from v to u :

$$\hat{d} : V \times V \rightarrow \mathbb{N} \quad \hat{d}(u, v) = \ell(\pi_{vu}), \quad (3.3)$$

where π_{vu} is the shortest path in G from v to u and $\ell(\pi_{vu})$ is the length of that path, defined as the number of edges of π_{vu} .

Note that $\hat{d}$ is not symmetrical, since π_{uv} and π_{vu} can be different. Next, we define the distance of configuration $\mathcal{A}^1$ from configuration $\mathcal{A}^2$ as the sum of the distances between the vertices that each agent occupies in the two configurations:

$$\bar{d} : \mathcal{C} \times \mathcal{C} \rightarrow \mathbb{N} \quad \bar{d}(\mathcal{A}^1, \mathcal{A}^2) = \sum_{p \in P} \hat{d}(\mathcal{A}^1(p), \mathcal{A}^2(p)).$$

Finally, we define the asymmetrical distance between two plans in $\mathcal{L}$. To this end, we associate to each plan f a function $\psi_f : \mathbb{N} \rightarrow \mathcal{C}$, so that $\psi_f(k)$ is the configuration corresponding to plan f at step k . If $k > |f|$, that is the step k is larger than the length of f , $\psi_f(k)$ is the last configuration:

$$\psi_f(k) := \begin{cases} \rho(\mathcal{A}, f_k), & k < |f|, \\ \rho(\mathcal{A}, f), & k \geq |f|. \end{cases}$$

Note that $\psi_f(k)$, the configuration at step k , depends on f_k , the k -th prefix of f . We define the distance of plan f from plan g with respect to the associated functions ψ_f, ψ_g . We consider the following distances:

1. ∞ -distance:

$$d_\infty(f, g) := \max_{1 \leq k \leq \min\{|f|, |g|\}} \bar{d}(\psi_f(k), \psi_g(k)); \quad (3.4)$$

2. 1-distance:

$$d_1(f, g) := \sum_{k=1}^{\min\{|f|, |g|\}} \bar{d}(\psi_f(k), \psi_g(k)); \quad (3.5)$$

3. max-min distance:

$$d_\infty^*(f, g) := \max_{k \in \mathbb{N}} \min_{h \in \mathbb{N}} \bar{d}(\psi_f(k), \psi_g(h)); \quad (3.6)$$

4. sum-min distance:

$$d_1^*(f, g) := \sum_{k=1}^{\min\{|f|, |g|\}} \min_{h \in \mathbb{N}} \bar{d}(\psi_f(k), \psi_g(h)). \quad (3.7)$$

Namely, the ∞ -distance of plans f and g corresponds to the maximum, with respect to time-step k , of the distance between the corresponding configurations at k . The 1-distance corresponds to the sum, with respect to time-step k , of the distances between the corresponding configurations at k . The max-min distance (respectively, the sum-min distance) corresponds to the maximum (respectively, the sum) with respect to k , of the distance of the configuration that plan f reaches at step k with respect to the set of all configurations encountered by plan g . It is easy to see that, for each couple of plans f, g , the 1-distance is the largest of the four, while the max-min distance is the smallest.

Example 3.2.1. Figure 3.1 shows a directed graph with 10 nodes. Tables 3.1 and 3.2 define two plans f_0 and g , depicted also in Figures 3.1 and 3.2. All configurations from $k = 0$ to $k = 2$ are the same in the two plans. All configurations in g , with the exception of the one for step $k = 3$, are also present in f_0 . At $k = 3$, plan g has a configuration $(7, 6, 5)$. The configuration in f_0 that is closer to $(7, 6, 5)$ is the one at

$k = 3$, that is $(7, 10, 5)$. Agent 2 is the only agent that occupies different nodes (6 and 10) in the two configurations. The shortest path between node 6 and node 10 has 3 edges. Hence, both the sum-min distance and the max-min distance between the two plans are 3. That is, $d_{\infty}^*(g, f_0) = 3$, $d_1^*(g, f_0) = 3$. Finally, the configurations of plan g at $k = 4$ and $k = 5$ have a distance from the corresponding configurations of plan f_0 both equal 8. Therefore, the ∞ -distance is equal to 8, and the 1-distance is equal to $3 + 8 + 8 = 19$. That is, $d_{\infty}(g, f_0) = 8$, $d_1(g, f_0) = 19$.

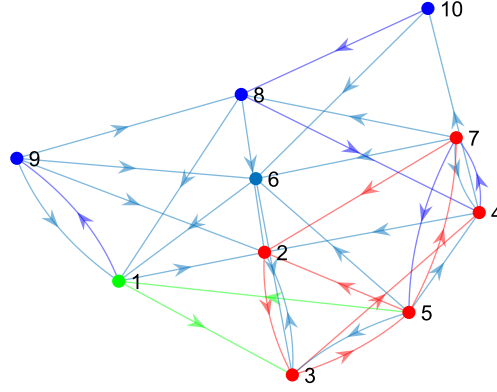
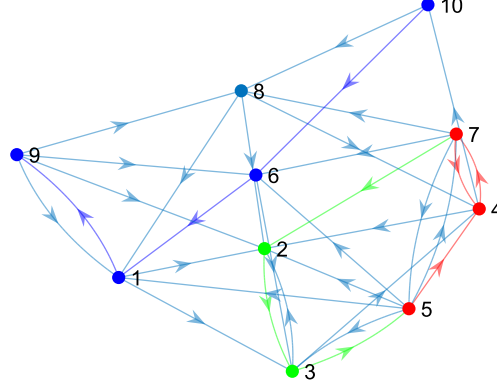


Figure 3.1: Plan f_0 on graph with 10 nodes.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	5	7	7	7	2	3	5	2	2	2	2	3	4	4
2	10	10	10	10	10	10	10	8	4	7	5	1	1	9
3	7	2	3	5	1	1	1	1	3	3	3	5	5	5

Table 3.1: Configurations of plan f_0 , the k -th column contains the nodes occupied by the agents at step k

Figure 3.2: Plan g on a graph with 10 nodes.

k	0	1	2	3	4	5
1	5	7	7	7	4	4
2	10	10	10	6	1	9
3	7	2	3	5	5	5

Table 3.2: Configurations of plan g .

3.2.2 Agent Distances

Alternatively, we can define the distance between two plans as the number of agents that occupy different positions in the two plans. First, we define the agent distance between two configurations. Let $U : \mathcal{C} \times \mathcal{C} \rightarrow \mathbb{P}(P)$ be a function that associates to each pair of configurations the subset of agents that are on different nodes in the two configurations:

$$U : \mathcal{C} \times \mathcal{C} \rightarrow \mathbb{P}(P) \quad U(\mathcal{A}_1, \mathcal{A}_2) := \left\{ p \in P : \mathcal{A}_1(p) \neq \mathcal{A}_2(p) \right\}, \quad (3.8)$$

The distance between two configurations $\mathcal{A}_1$ and $\mathcal{A}_2$ is the cardinality of $U(\mathcal{A}_1, \mathcal{A}_2)$:

$$\tilde{d}(\mathcal{A}_1, \mathcal{A}_2) := |U(\mathcal{A}_1, \mathcal{A}_2)|. \quad (3.9)$$

Similarly, we define a new function $\mathcal{U}$ which associates to each pair of plans f and g the subset of agents that have different positions in at least one time step $k \in \{1, \dots, \max(|f|, |g|)\}$

$$\mathcal{U} : \mathcal{L} \times \mathcal{L} \rightarrow \mathbb{P}(P) \quad \mathcal{U}(f, g) := \left\{ p \in P : \exists k \in \{1, \dots, \max(|f|, |g|)\}, \right. \\ \left. (\psi_f(k))(p) \neq (\psi_g(k))(p) \right\}. \quad (3.10)$$

Note that:

$$\mathcal{U}(f, g) = \bigcup_{k=1}^{\max(|f|, |g|)} U(\psi_f(k), \psi_g(k)), \quad (3.11)$$

or, equivalently:

$$\mathcal{U}(f, g) = \left\{ p \in P : f^p \neq g^p \right\}. \quad (3.12)$$

We define a new agent distance between two plans, called *u-agents distance*, as the cardinality of $\mathcal{U}(f, g)$:

$$d_U(f, g) := |\mathcal{U}(f, g)|. \quad (3.13)$$

Namely, this distance represents the number of agents that, in the two plans, occupy different positions in at least one time-step or, equivalently, the number of agents that have a different *path plan* in f and g .

Alternatively, we can define another agent distance d_m called *max-agents distance* between two plans f and g as the maximum of all the distances between the pairs of configurations crossed by the two plans as the time step varies from 1 to $\max(|f|, |g|)$

$$d_m(f, g) := \max_{k \in \{1, \dots, \max(|f|, |g|)\}} \tilde{d}(\psi_f(k), \psi_g(k)), \quad (3.14)$$

where $\tilde{d}(\psi_f(k), \psi_g(k))$ is the distance (3.9) between configurations. The following relationship exists between the two distances defined above.

Proposition 2.1. Let $f, g \in \mathcal{L}$ be two plans. It holds that:

$$d_m(f, g) \leq d_U(f, g).$$

Proof. From (3.11) it follows that:

$$\forall k \in \mathbb{N}, |U(\psi_f(k), \psi_g(k))| \leq |\mathcal{U}(f, g)|,$$

so that:

$$\forall k \in \mathbb{N}, \tilde{d}(\psi_f(k), \psi_g(k)) \leq d_U(f, g),$$

and, consequently:

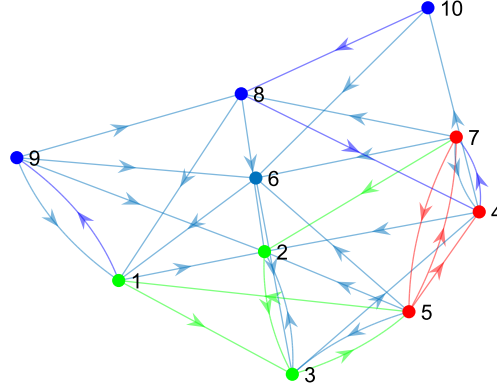
$$\max_{k \in \mathbb{N}} \tilde{d}(\psi_f(k), \psi_g(k)) \leq d_U(f, g).$$

□

Example 3.2.2. Consider plan f_0 in Table 3.3, and plan h in Table 3.4, depicted also in Figures 3.1 and 3.3. The u-agents distance and the max-agents distance between the two plans are 1 because only the first agent has different path plans in f_0 and h .

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	5	7	7	7	2	3	5	2	2	2	2	3	4	4
2	10	10	10	10	10	10	10	8	4	7	5	1	1	9
3	7	2	3	5	1	1	1	1	3	3	3	5	5	5

Table 3.3: Initial plan f_0 .

Figure 3.3: Plan h on a graph with 10 nodes.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	5	7	7	7	5	5	5	5	5	4	4	4	4	4
2	10	10	10	10	10	10	10	8	4	7	5	1	1	9
3	7	2	3	5	1	1	1	1	3	3	3	5	5	5

Table 3.4: Second plan h .

3.2.3 Optimization MAPF problem in a neighborhood of a reference plan

After having defined these distances, we introduce a variant of the optimization MAPF problem (3.2), the *optimization MAPF problem constrained to a given plan*. We consider a sub-optimal solution f_0 of a MAPF instance, and we want to find another solution for the same problem, which is not too far from f_0 and has better quality (i.e., shorter length). Let $\mathcal{A}$ and $\mathcal{A}^t$ be initial and final valid configurations on a digraph G . Let $f_0 \in \mathcal{L}$ be such that $\mathcal{A}^t = \rho(\mathcal{A}, f_0)$ (i.e., f_0 is a feasible solution of the MAPF instance). Set $r \in \mathbb{N}$ and let d be the distance between plans. The optimization MAPF problem, with *Makespan*, constrained to f_0 is:

$$\begin{aligned}
& \min |f| \\
& \text{s.t. } \mathcal{A}^t = \rho(\mathcal{A}, f) \\
& f \in \mathcal{L}, d(f, f_0) \leq r.
\end{aligned} \tag{3.15}$$

3.3 Iterative local optimization for FSA

We present the basic idea of our approach in a slightly more general setting. Given a FSA $(\Sigma, X, x_0, \delta, X_M)$, consider problem

$$\begin{aligned}
& \min_{s \in \Sigma^*} |s| \\
& \text{s.t. } \delta(x_0, s)! \text{ and } \delta(x_0, s) \in X_M.
\end{aligned} \tag{3.16}$$

In Problem (3.16), we want to find a string s of minimum length such that FSA reaches a state in X_M . Note that Problem (3.2) is a specific case of (3.16), with $\Sigma = \mathcal{C}$, $X = \mathcal{C}$, $x_0 = \mathcal{A}$, $X_M = \{\mathcal{A}^t\}$, $\delta = \rho$. Let $d : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}^+$ be a function such that

- 1) $d(s, s) = 0, s \in \Sigma^*$,
- 2) $d(s\sigma, t) \geq d(s, t), s, t \in \Sigma^*, \sigma \in \Sigma$,
- 3) $d(\varepsilon, s) = 0, s \in \Sigma^*$.

Roughly speaking, for $s, t \in \Sigma^*$, $d(s, t)$ represents the distance of s to t . In 1), we require that the distance of s to itself is 0. In 2), we require $d(s, t)$ to be not smaller than $d(r, t)$, where r is any prefix of s . In 3), we require the empty string ε to have 0 distance to any string. Note that all distances defined in Section 3.2 satisfy these properties. Let $s_0 \in \Sigma^*$ be a feasible solution of (3.16) (that is, $\delta(x_0, s_0) \in X_M$). Let $\mathcal{L}$ be the subset of Σ^* consisting of strings s such that $\delta(x_0, s)!$.

Consider the following problem.

$$\begin{aligned} \min_{s \in \mathcal{L}} |s| \\ \text{s.t. } \delta(x_0, s) \in X_M \end{aligned} \quad (3.17)$$

$$d(s, s_0) \leq r.$$

Note that (3.15) is a specific instance of (3.17) with $s_0 = f_0$. The overall idea for solving (3.16) consists of the following steps:

- Find a feasible solution s_0 of (3.16).
- Solve Problem (3.17). If the found solution s^* is such that $|s^*| < |s|$, set $s_0 = s^*$ and repeat this step.

We would like to solve Problem (3.17) by dynamic programming. One problem is that the distance constraint $d(s, s_0) \leq r$ depends on string s , and it is not a function of the state only (that is, of $\delta(x_0, s)$). To overcome this problem, we define an equivalence class $\sim$ on $\mathcal{L}$ such that, for any $s, t \in \mathcal{L}$ such that $s \sim t$, and any $\sigma \in \Sigma$

$$\begin{aligned} |s| &= |t|, \\ \delta(x_0, s) &= \delta(x_0, t), \\ d(s\sigma, s_0) &\leq r \Leftrightarrow d(t\sigma, s_0) \leq r. \end{aligned} \quad (3.18)$$

In other words, if $s \sim t$, then s and t have the same length and determine the transition of the FSA from the initial state x_0 to the same final state. Moreover, if $s\sigma$ satisfies the distance requirement, also $t\sigma$ satisfies it.

Then, we define another FSE $(\hat{X}, \mathcal{S}, [\varepsilon], \hat{\delta}, X_M/\sim)$. The state space $\hat{X}$ is defined recursively as follows:

- 1) $[\varepsilon] \in \hat{X}$
- 2) if $[s] \in \hat{X}$, $\delta(s, \sigma, s_0)!$, and $d(s\sigma, s_0) \leq r$ then $[s\sigma] \in \hat{X}$.

Note that condition 2) is well-posed. Indeed, from our assumptions on $\sim$, condition $d(s\sigma, s_0) \leq r$ does not change if we substitute s with any other member of equivalence class $[s]$.

We define transition function $\hat{\delta} : \hat{X} \times S \rightarrow \hat{X}$ by setting

- 1) $\hat{\delta}([s], \sigma) \neq \perp \Leftrightarrow d(s\sigma, s_0) \leq r$,
- 2) $\hat{\delta}([s], \sigma) = [s\sigma]$.

Then, we convert Problem (3.17) into the following

$$\begin{aligned} \min_{s \in \mathcal{L}} |s| \\ \text{s.t. } \hat{\delta}([\varepsilon], s) \in X_M / \sim. \end{aligned} \quad (3.19)$$

Note that in Problem (3.19), the state space is $\mathcal{L} / \sim$, the set of equivalence classes of $\mathcal{L}$ with respect to $\sim$. Problem (3.19) is a simple reachability problem for FSA and can be solved by dynamic programming. Indeed, we removed from the new state space $\hat{X}$ all the states that correspond to strings that violate the distance constraint. Moreover, $\hat{\delta}$ allows only those transitions that lead to states inside $\hat{X}$. The following statement follows by construction.

Proposition 3.1. String s is a solution of (3.17) if and only if it is a solution of (3.19).

3.3.1 Equivalence classes associated with the previously defined distances

We present the equivalence relations on $\mathcal{L}$ associated with the previously defined distances. These equivalence classes satisfy properties (3.18). For the ∞ -distance (3.4), the max-min distance (3.6), and max-agents distance (3.14), $f \sim g$ if and only if

1. $|f| = |g|$;
2. $\rho(\mathcal{A}, f) = \rho(\mathcal{A}, g)$.

We define an injective function $\alpha : \mathcal{L} / \sim \rightarrow \mathbb{N} \times \mathcal{C}$ as follows

$$\alpha([f]) = (|f|, \rho(\mathcal{A}, f)). \quad (3.20)$$

This function is well-defined because, if $f_1 \sim f_2$, then $|f_1| = |f_2|$ and $\rho(\mathcal{A}, f_1) = \rho(\mathcal{A}, f_2)$. Moreover, α is injective because, if f_1 and f_2 are such that $\alpha([f_1]) = \alpha([f_2])$, then $|f_1| = |f_2|$ and $\rho(\mathcal{A}, f_1) = \rho(\mathcal{A}, f_2)$, therefore, $[f_1] = [f_2]$. We will use

function α to represent equivalence classes $\mathcal{L}/\sim$ in the dynamic programming solution algorithm.

For the 1-distance (3.5) and the sum-min distance (3.7), $f \sim g$ if and only if

1. $|f| = |g|$;
2. $\rho(\mathcal{A}, f) = \rho(\mathcal{A}, g)$;
3. $d(f, f_0) = d(g, f_0)$;

where $d = d_1$ for the 1-distance and $d = d_1^*$ for the sum-min distance. Again, we define an injective function α , representing equivalence classes. Namely, $\alpha : \mathcal{L}/\sim \rightarrow \mathbb{N} \times \mathcal{C} \times \mathbb{N}$ is such that:

$$\alpha([f]) = (|f|, \rho(\mathcal{A}, f), d(f, f_0)). \quad (3.21)$$

We can show that this function is well-defined and injective with the same procedure used for (3.20).

For the u-agents distance, (3.13), $f \sim g$ if and only if

1. $|f| = |g|$;
2. $\rho(\mathcal{A}, f) = \rho(\mathcal{A}, g)$;
3. $\mathcal{U}(f, f_0) = \mathcal{U}(g, f_0)$;

where $\mathcal{U}(f, f_0)$ is the subset of agents defined in (3.10).

We represent the state with $\alpha : \mathcal{L}/\sim \rightarrow \mathbb{N} \times \mathcal{C} \times \mathbb{P}(P)$, defined as follows:

$$\alpha([f]) = (|f|, \rho(\mathcal{A}, f), \mathcal{U}(f, f_0)). \quad (3.22)$$

Again, we can show that this function is well-defined and injective, following the same procedure used for (3.20).

3.3.2 Neighborhoods

Given the distances defined in Section 3.2 and the definition of the equivalence classes in Section 3.3.1, we can define the neighborhood of a configuration and of an equivalence class. Moreover, we can upper estimate the cardinality of such neighborhoods. Such an estimate is needed to evaluate the time needed to explore the neighborhoods, an operation that is central to the approach proposed in this chapter. We define a neighborhood of a configuration as a ball centered in $\mathcal{A} \in \mathcal{C}$ of radius r

$$\mathcal{B}_r(\mathcal{A}) := \{\mathcal{A}^* \in \mathcal{C} : d(\mathcal{A}^*, \mathcal{A}) \leq r\}, \quad (3.23)$$

where $d(\mathcal{A}^*, \mathcal{A})$ can be either $\bar{d}(\mathcal{A}^*, \mathcal{A})$ or $\tilde{d}(\mathcal{A}^*, \mathcal{A})$. Given a distance d between plans and $r \in \mathbb{N}$, we define the neighborhood of an equivalence class $[f] \in \mathcal{L}$ as:

$$\mathcal{B}_r([f]) := \{[g], g \in \mathcal{L} : |g| \leq |f|, d(g, f) \leq r\}. \quad (3.24)$$

Paths Neighborhoods

Let us consider the distances defined in Section 3.2.1 and compute the upper bound for the cardinality of both neighborhoods defined in (3.23) and (3.24). First, we define a ball $\mathcal{B}_r(v)$ centered at $v \in V$ of radius $r \in \mathbb{N}$:

$$\mathcal{B}_r(v) := \{u \in V : \hat{d}(u, v) \leq r\}.$$

Note that, for simplicity, we use the same notation ($\mathcal{B}_r$) for balls of vertices, configurations, and plans. The meaning of set $\mathcal{B}_r$ depends on the argument (vertex, configuration, or plan). We denote by $\tilde{\mathcal{B}}_r(v)$ the border of the ball, obtained by replacing $\leq$ with $=$ in the definition. Let $\phi = \text{outdeg}(G)$ be the maximum out-degree of digraph $G = (V, E)$. The following proposition provides an upper bound on the cardinality of $\tilde{\mathcal{B}}_r(v)$.

Proposition 3.2. It holds that

$$|\tilde{\mathcal{B}}_r(v)| \leq \phi^r. \quad (3.25)$$

Proof. Let n_h be the number of nodes at a distance h from v . Note that $n_1 \leq \phi$, and $\forall h \geq 2$ $n_h \leq n_{h-1}(\phi - 1)$. By induction, $n_h \leq \phi(\phi - 1)^{h-1} \forall h \geq 1$. Therefore, an

upper-bound for the number of nodes on the border of the ball is

$$|\bar{\mathcal{B}}_r(v)| \leq \phi(\phi - 1)^{r-1} \leq \phi^r.$$

□

Next, we take into account the ball defined in (3.23), denoting with $\bar{\mathcal{B}}_r(\mathcal{A})$ its border. An upper bound for the cardinality of the ball is given by the following proposition.

Proposition 3.3. It holds that

$$|\mathcal{B}_r(\mathcal{A})| \leq \binom{k+r}{r} \phi^r. \quad (3.26)$$

where $k = |P|$ and $\phi = \text{outdeg}(G)$.

Proof. First of all, we find an upper bound for the number of configurations at a distance h from $\mathcal{A}$, i.e., an upper bound for the cardinality of the border of the ball centered in $\mathcal{A}$ of radius h :

$$\bar{\mathcal{B}}_h(\mathcal{A}) = \left\{ \mathcal{A}^* \in \mathcal{C} : \sum_{i=1}^{|P|} \hat{d}(\mathcal{A}(p_i), \mathcal{A}^*(p_i)) = h \right\}.$$

Let $(h_1, \dots, h_{|P|})$ be a $|P|$ -decomposition of h (i.e., $\sum_{i=1}^{|P|} h_i = h$). Taking into account Proposition 3.2, an upper bound for the number of configurations for which $\hat{d}(\mathcal{A}(p_i), \mathcal{A}^*(p_i)) = h_i, i = 1, \dots, |P|$, is

$$\prod_{i=1}^{|P|} |\bar{\mathcal{B}}_{h_i}(\mathcal{A}(p_i))| \leq \prod_{i=1}^{|P|} \phi^{h_i} = \phi^h.$$

By standard combinatorial arguments, the number of $|P|$ -decompositions of h is

$$\frac{(h + (|P| - 1))!}{h! (|P| - 1)!}.$$

Therefore, the cardinality of the border of the ball of radius h can be bounded from above by:

$$|\bar{\mathcal{B}}_h(\mathcal{A})| = \sum_{(h_1, \dots, h_{|P|}) : \sum_{i=1}^{|P|} h_i = h} \prod_{i=1}^{|P|} |\bar{\mathcal{B}}_{h_i}(\mathcal{A}(p_i))| \leq$$

$$\leq \frac{(h + (|P| - 1))!}{h! (|P| - 1)!} \phi^h,$$

and the total number of configurations in $\mathcal{B}_r(\mathcal{A})$ can be overestimated as follows:

$$\begin{aligned} |\mathcal{B}_r(\mathcal{A})| &= 1 + \sum_{h=1}^r |\tilde{\mathcal{B}}_h(\mathcal{A})| \leq 1 + \sum_{h=1}^r \frac{(h + |P| - 1)!}{h! (|P| - 1)!} \phi^h = \\ &= \sum_{h=0}^r \frac{(h + |P| - 1)!}{h! (|P| - 1)!} \phi^h \leq \binom{|P| + r}{r} \phi^r. \end{aligned}$$

□

From these bounds, we can compute an upper bound of the cardinality of $\mathcal{B}_r([f])$ for $f \in \mathcal{L}$ defined in (3.24). Here, we consider the max-min distance (3.6). Since the max-min distance is the smallest among the path distances, the upper bound stated in the proposition also applies to the neighborhood constructed using the other path distances and, in particular, the sum-min distance (3.7).

Proposition 3.4. It holds that:

$$|\mathcal{B}_r(f)| \leq |f|^2 \binom{k+r}{r} \phi^r (r+1), \quad (3.27)$$

where $k = |P|$ and $\phi = \text{outdeg}(G)$. Moreover, $\exists C = C(r) \in (0, 1]$ such that

$$|\mathcal{B}_r(f)| \leq |f|^2 C (k+r)^r \phi^r (r+1).$$

Proof. Let $f \in \mathcal{L}$ and $\hat{f} \in \mathcal{B}_r(f)$. Let $I = \{1, \dots, |\hat{f}|\}$, $J = \{1, \dots, |f|\}$, and $R = \{0, \dots, r\}$. Then:

$$\alpha(\mathcal{B}_r(f)) \subset \left(I \times \bigcup_{j \in J} \mathcal{B}_r(\psi_f(j)) \times R \right).$$

Indeed, it holds that $|\hat{f}| \leq |f|$ and, moreover, $d^*(\hat{f}, f) \leq r$ implies that

$$\max_{1 \leq i \leq |\hat{f}|} \min_{1 \leq j \leq |f|} \bar{d}(\psi_{\hat{f}}(i), \psi_f(j)) \leq r,$$

and, in particular, for $i = |\hat{f}|$ there exists $j \in J$ such that

$$\bar{d}(\rho(\mathcal{A}, \hat{f}), \psi_f(j)) \leq r,$$

which means that $\rho(\mathcal{A}, \hat{f}) \in \mathcal{B}_r(\psi_f(j))$. Therefore, recalling that α is injective,

$$|\mathcal{B}_r(f)| \leq |I \times \bigcup_{j \in J} \mathcal{B}_r(\psi_f(j)) \times R|.$$

Then, also in view of (3.26), we have that:

$$\begin{aligned} |\mathcal{B}_r(f)| &\leq |f| \left(\sum_{j=1}^{|f|} |\mathcal{B}_r(\psi_f(j))| \right) (r+1) \leq \\ &\leq |f|^2 \binom{k+r}{r} \phi^r (r+1). \end{aligned}$$

The last statement of the proposition follows from

$$\binom{k+r}{r} \leq \frac{(k+r)^r}{r!},$$

and setting $C := \frac{1}{r!}$. □

Note that given the last statement of the above proposition, we have that the neighborhood of f of radius r has a polynomial cardinality with respect to the number of nodes for each fixed integer value r .

Agents Neighborhoods

We consider the agent distances defined in Section 3.2.2 and compute upper bounds for the cardinality of the neighborhoods defined in (3.23) and (3.24).

First, we take into account the neighborhood of a configuration defined in (3.23) with the agent distance, defined in (3.9). The following result gives an upper bound on the cardinality of the corresponding ball.

Proposition 3.5. It holds that

$$|\mathcal{B}_r(\mathcal{A})| \leq \binom{k}{r} \frac{(n-k+r)!}{(n-k)!}, \quad (3.28)$$

where $k = |P|$ is the number of agents and $n = |V|$ the number of nodes.

Proof. From the definitions of the ball in (3.23) and of the distance in (3.9), it follows that each configuration $\mathcal{A}^*$ in $\mathcal{B}_r(\mathcal{A})$ has at most r agents occupying different positions compared to configuration $\mathcal{A}$. We denote with $P_r \subset P$ the subset of r agents that change their positions. The number of different choices of r agents corresponds to the number of r -combinations of a set of cardinality k . It is the binomial coefficient $\binom{k}{r}$. Since the remaining agents $P \setminus P_r$ are $k - r$, the available nodes that can be occupied by the agents in a given subset P_r are $n - k + r$. So, for a fixed subset P_r , the new positions can be chosen in $\frac{(n-k+r)!}{(n-k)!}$ different ways. Therefore, the total number of different configurations $\mathcal{A}^*$ in $\mathcal{B}_r(\mathcal{A})$ is

$$|\mathcal{B}_r(\mathcal{A})| \leq \binom{k}{r} \frac{(n-k+r)!}{(n-k)!}.$$

□

Then, we can compute an upper bound for the cardinality of $\mathcal{B}_r(f)$, with $f \in \mathcal{L}$, for the max-agents distance defined in (3.14).

Proposition 3.6. If we consider the max-agents distance (3.14), it holds that:

$$|\mathcal{B}_r(f)| \leq |f| \binom{k}{r} \frac{(n-k+r)!}{(n-k)!}. \quad (3.29)$$

Moreover, we have that, for fixed r , the neighborhood has a polynomial cardinality with respect to the number of nodes. In particular, $\exists C = C(r) \in (0, 1]$ such that

$$|\mathcal{B}_r(f)| \leq C |f| k^r (n-k+r)^r.$$

Proof. Let $f \in \mathcal{L}$, $\hat{f} \in \mathcal{B}_r(f)$, $I = \{1, \dots, |\hat{f}|\}$ and $J = \{1, \dots, |f|\}$. Let

$$\mathcal{P}_r = \{P_r : P_r \subset P, |P_r| = r\},$$

be the family of subset of P of cardinality equal to r . Then:

$$\alpha(\mathcal{B}_r(f)) \subset \left(I \times \bigcup_{j \in J} \mathcal{B}_r(\psi_f(j)) \times \mathcal{P}_r \right). \quad (3.30)$$

Indeed, it holds that $|\hat{f}| \leq |f|$ and $d_1^*(\hat{f}, f) \leq r$ implies that, for $j = |\hat{f}|$,

$$\hat{d}(\rho(\mathcal{A}, \hat{f}), \psi_f(|\hat{f}|)) = \hat{d}(\psi_f(|\hat{f}|), \psi_f(|\hat{f}|)) \leq d_1^*(\hat{f}, f) \leq r,$$

which means that $\rho(\mathcal{A}, \hat{f}) \in \mathcal{B}_r(\psi_f(|\hat{f}|))$. From (3.30), note that:

$$\forall \hat{f} \in B_r(f) \quad \alpha(\hat{f}) \in \{|\hat{f}|\} \times \mathcal{B}_r(\psi_f(|\hat{f}|)) \times \mathcal{P}_r.$$

Therefore, $|\alpha(\hat{f})| \leq |\mathcal{B}_r(\psi_f(j)) \times \mathcal{P}_r|$. We can also state that, for a fixed r , the subset of agents whose position varies in $\psi_f(j)$ is a subset of the agents i moved within path f^i . So, the choice of the subset is already accounted for in the binomial coefficient inside the cardinality of $\mathcal{B}_r(\psi_f(j))$.

Recalling that α is injective, it follows that:

$$|\mathcal{B}_r(f)| \leq |\alpha(\mathcal{B}_r(f))| \leq \left| \bigcup_{j \in J} \mathcal{B}_r(\psi_f(j)) \right| \leq \sum_{j=1}^{|f|} |\mathcal{B}_r(\psi_f(j))| \leq |f| \binom{k}{r} \frac{(n-k+r)!}{(n-k)!},$$

where the last inequality follows from Proposition 3.5.

The last statement of the proposition follows from

$$\binom{k}{r} \leq \frac{k^r}{r!}, \quad \frac{(n-k+r)!}{(n-k)!} = r! \binom{n-k+r}{r} \leq (n-k+r)^r,$$

after setting $C := \frac{1}{r!}$. □

We proved these results using the max-agents distance. However, from Proposition 2.1, it follows that the upper bounds computed are also valid for neighborhoods defined with the u-agents distance (3.13).

3.3.3 Iterative Neighborhood Search

We propose an iterative approach for finding a sub-optimal solution of Problem (3.19). The returned solution is locally optimal with respect to the neighborhood of a reference solution. At each iteration, we solve an instance of Problem (3.19). The algorithm takes as input a feasible solution f_0 that may be of poor quality. For instance,

we can obtain f_0 from a rule-based algorithm, such as diSC [Pub3]. We aim at improving f_0 , obtaining a shorter solution. To this end, we solve Problem (3.19) with a dynamic programming algorithm. For an assigned plan distance, we search in the neighborhood $\mathcal{B}_r(f_0)$ for plans shorter than f_0 through algorithm *Dynprog*, which we will describe below. If we cannot obtain a solution shorter than f_0 (that is, f_0 is locally optimal), we stop the algorithm. Otherwise, if we obtain an improved solution f^* , we redefine the reference solution as $f_0 = f^*$ and repeat the whole procedure. We iterate until we cannot shorten the current solution any further. This algorithm can be classified as a **Neighborhood Search** algorithm (see [36, 21]).

To define the neighborhood $\mathcal{B}_r(f_0)$, we can use any distance function among those presented in Section 3.2. In our numerical experiments, we used the sum-min distance and the u-agents distance defined in (3.13). Algorithm 4 presents the steps of the procedure just described.

Algorithm 4 Neighborhood Search

```

Input:  $f_0, \mathcal{A}, \mathcal{A}^t, r$ 
Output:  $f^*$ 
 $f^* \leftarrow f_0$ 
do
     $f_0 \leftarrow f^*$ 
     $f^* \leftarrow \text{DynProg}(f_0, \mathcal{B}_r(f_0), \mathcal{A}, \mathcal{A}^t)$ 
while  $|f^*| < |f_0|$ 
return  $f^*$ 

```

Also, we can alternate different types of neighborhoods. This may allow us to find solutions of better quality since we are exploring larger neighborhoods. For example, we propose Algorithm 5, that first defines $\mathcal{B}_r(f_0)$ with the sum-min distance and then defines it with the u-agents distance (3.13). In this case, the final solution is locally optimal with respect to both neighborhoods.

The input of this algorithm is made up of the initial plan f_0 , the initial and final configurations $\mathcal{A}$, and $\mathcal{A}^t$ and the radius r . The output is the locally optimal solution

Algorithm 5 Alternated Neighborhood Search

```

Input:  $f_0, \mathcal{A}, \mathcal{A}^t, r$ 
Output:  $f^*$ 
 $f^* \leftarrow f_0$ 
do
   $f_0 \leftarrow f^*$ 
   $f_1 \leftarrow \text{NeighSearch}(f_0, \mathcal{A}, \mathcal{A}^t, \mathcal{B}_r(f_0))$ , using u-agents distance
   $f^* \leftarrow \text{NeighSearch}(f_1, \mathcal{A}, \mathcal{A}^t, \mathcal{B}_r(f_1))$ , using sum-min distance
while  $|f^*| < |f_0|$ 
return  $f^*$ 

```

f^* .

3.3.4 Dynamic Programming Algorithm with Dominance

To search for the optimal solution to the Problem (3.19), we employ a *Dynamic Programming* (DP) algorithm. In generic DP problems, we are given a state space S where $A \subset S$ is the set of target states, a transition function $g : S \rightarrow P(S)$, where P is the power set of S , and an objective function $c : S \rightarrow \mathbb{R}$. Starting from an initial state $s_0 \in S$, we iteratively expand states through a suitably defined transition function g until we reach a state $s_t \in A$ with minimum objective function value.

States and transition function for Path Distances

When we employ the sum-min distance, the states represent the equivalence classes of relation $\sim$, defined in Section 3.3.1 for this distance. Namely, we use the corresponding injection $\alpha : \mathcal{L}/\sim \rightarrow \mathbb{N} \times \mathcal{C} \times \mathbb{N}$ to associate to each equivalence class $[f]$ a triple $(\beta, \gamma, \sigma) = \alpha([f])$, where β is the length of f , γ the configuration obtained by applying f to the initial state $\mathcal{A}$, and σ is the distance of f from reference plan f_0 . Namely, the state space is

$$\mathcal{S} := \alpha(\mathcal{L}/\sim) \subset \mathbb{N} \times \mathcal{C} \times \mathbb{N},$$

where α is defined in (3.21). Each state $s = (\beta, \gamma, \sigma) \in \mathcal{S}$ represents the equivalence class:

$$\alpha^{-1}(s) = \{f \in \mathcal{L} : \beta = |f|, \gamma = \rho(\mathcal{A}, f), \sigma = d_1^*(f, f_0)\}.$$

The initial state is $s_0 = \alpha(\varepsilon) = (0, \mathcal{A}, 0)$. We use a priority queue Q to store the states that have not been visited yet. At the beginning, $Q = \{s_0\}$. We define a partial ordering on $\mathcal{S}$ based on length. Namely, if $s_1 = (\beta_1, \gamma_1, \sigma_1), s_2 = (\beta_2, \gamma_2, \sigma_2), s_1 < s_2$ if $\beta_1 < \beta_2$. We order the elements of Q according to this ordering.

A state $s_1 = (\beta_1, \gamma_1, \sigma_1)$ *dominates* $s_2 = (\beta_2, \gamma_2, \sigma_2)$ if

$$\beta_1 \leq \beta_2, \gamma_1 = \gamma_2, \sigma_1 \leq \sigma_2.$$

In other words, s_1 dominates s_2 if the plans f_1, f_2 , corresponding to s_1 and s_2 , satisfy the following properties: Plan f_1 is not longer than f_2 , f_1 and f_2 lead to the same final configuration, and the distance of f_1 from the reference solution f_0 is not larger than the one of f_2 . If s_1 dominates s_2 , we can discard s_2 . In general, we remove from Q all dominated states.

We also define the following *transition function*, which allows us to (possibly) add new states to the priority queue:

$$\tilde{\rho} : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{S}$$

$$\tilde{\rho}((\beta, \gamma, \sigma), e) := (\beta + 1, \rho(\gamma, e), \sigma + \min_{k \in \mathbb{N}} \bar{d}_{\mathcal{A}}(\rho(\gamma, e), \psi_{f_0}(k))).$$

Applying this function to state $s = (\beta, \gamma, \sigma)$:

- adds 1 to length β ,
- updates the final configuration γ , applying function $\rho(\gamma, e)$ to γ , where e is the chosen set of edges;
- updates σ , adding the computed minimum distance between the updated final configuration $\rho(\gamma, e)$ and the reference plan f_0 .

We denote by

$$\Sigma := \left\{ \tilde{\rho}((\beta, \gamma, \sigma), e) : e \in \mathcal{E} \text{ and } \sigma + \min_{k \in \mathbb{N}} \bar{d}_{\mathcal{A}}(\rho(\gamma, e), \psi_{f_0}(k)) \leq r \right\} \subset \mathcal{S},$$

the set of states generated through the transition function $\tilde{\rho}$, applied to current state (β, γ, σ) . Moreover, we denote with

$$\Gamma := \alpha(\mathcal{B}_r(f_0)) \subset \mathcal{S},$$

the set of states that can be visited during a neighborhood search.

States and transition function for Agent Distances

Using the u-agents distance introduced in Section 3.2.2, the states represent the equivalence classes of the corresponding relation $\sim$, presented in (3.22). We use the associated injection $\alpha : \mathcal{L}/\sim \rightarrow \mathbb{N} \times \mathcal{C} \times \mathbb{N}$ to assign, to each equivalence class $\hat{f}$, a triple $(\beta, \gamma, \Omega) = \alpha(\hat{f})$. Compared to the previous section, the third element of the triple is Ω , which represents the subset of agents such that their path plans in a representative of $\hat{f}$ and f_0 are different.

Namely, the state space is

$$\mathcal{S} := \alpha(\mathcal{L}/\sim) \subset \mathbb{N} \times \mathcal{C} \times \mathbb{P}(P),$$

where α is defined in (3.22). Since α is injective, $\mathcal{S}$ and $\mathcal{L}/\sim$ are in one-to-one correspondence. Each state $s = (\beta, \gamma, \Omega) \in \mathcal{S}$, represents the equivalence class:

$$\alpha^{-1}(s) = \{f \in \mathcal{L} : \beta = |f|, \gamma = \rho(\mathcal{A}, f), \Omega = \mathcal{U}(f, f_0)\}.$$

We set the initial state as $s_0 = \alpha(\varepsilon) = (0, \mathcal{A}, \emptyset)$ and use a priority queue Q to store the states that will be explored. At the beginning, $Q = \{s_0\}$. We order the elements of Q following an ordering on $\mathcal{S}$ based on the length. Namely, if $s_1 = (\beta_1, \gamma_1, \Omega_1), s_2 = (\beta_2, \gamma_2, \Omega_2)$, $s_1 < s_2$ if $\beta_1 < \beta_2$.

A state $s_1 = (\beta_1, \gamma_1, \Omega_1)$ *dominates* $s_2 = (\beta_2, \gamma_2, \Omega_2)$ if

- $\beta_1 = \beta_2$,
- $\gamma_1 = \gamma_2$,
- $\Omega_1 \subseteq \Omega_2$.

In other words, all the representatives $f_1 \in \alpha^{-1}(s_1)$ and $f_2 \in \alpha^{-1}(s_2)$ satisfy the following properties:

- $|f_1| = |f_2|$: f_1 is not longer than f_2 ;
- $\rho(\mathcal{A}, f_1) = \rho(\mathcal{A}, f_2)$: f_1 leads to the same configuration as f_2 ;
- $\mathcal{U}(f_1, f_0) \subseteq \mathcal{U}(f_2, f_0)$: all the agents $p \in P$ such that $f_1^p \neq f_0^p$, are such that $f_2^p \neq f_0^p$.

We remove from Q all dominated states. Again, we define a *transition function* $\hat{\rho} : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{S}$, which allows us to (possibly) add new states to the priority queue:

$$\hat{\rho}((\beta, \gamma, \Omega), e) := (\beta + 1, \rho(\gamma, e), \Omega \cup \mathcal{U}(\rho(\gamma, e), \psi_{f_0}(\beta + 1))).$$

Applying this function to state $s = (\beta, \gamma, \Omega)$:

- adds 1 to length β ,
- updates configuration γ , applying $\rho(\gamma, e)$ to γ , where e is the chosen set of edges (note that the choice for e can be restricted to a subset of edges with cardinality not larger than r , and is, thus, polynomial for fixed r , due to the definition of the neighborhood);
- updates Ω , adding the agents that change their configuration from the reference plan in the step $\beta + 1$ if they are not already in Ω .

As in the previous section, we denote by Σ the set of new states which can be generated through the transition function and with Γ the set of states that can be visited during a neighborhood search.

Algorithm

Algorithm 6 describes the Dynamic Programming algorithm. The priority queue Q is a set of states sorted according to the length β of their representatives. The function $\text{insert}(Q, x)$ inserts a state x , maintaining the partial order of Q , in the sense that,

after the insertion of x , all the elements of the queue still respect the ordering previously defined. Function $\text{remove}(Q, x)$ removes x from Q . The head of the queue, that is the state with minimal β , is denoted by $Q[0]$. The algorithm explores the state space starting from the initial state s_0 . At each iteration, a state with minimum β is extracted from the queue. If the state extracted is a target state, that is, its second component (the configuration) is $\mathcal{A}^t$, the algorithm stops, and we return a representative of the optimal solution of Problem (3.19) (for a given equivalence class $\hat{f}$, the function $\text{repr}(\hat{f})$ returns a representative of the class). Otherwise, the algorithm employs function $\text{expand}(s, f_0, r)$, based on the transition function previously defined, to generate new states. If a new state is not dominated, then it is added to the queue Q . Moreover, we remove from Q all states dominated by one of the newly added states.

3.3.5 Complexity results

In this section, we provide a complexity analysis of the proposed algorithms.

Theorem 3.7. *Algorithms 4, 5 and 6 have polynomial time complexity with respect to the number of nodes of the graph, for a fixed value r , and assuming that each call of the procedure expand has unit cost.*

Proof. First, we consider the case with the sum-min distance. At each iteration of Algorithm 6, we remove one state s from set Q . Since $Q \subseteq \Gamma$, an upper bound for the number of iterations is $|\Gamma|$. If the state s is not a final state, then it is expanded and generates a set Σ of new states. For each newly generated state, we need to check whether it is dominated by or dominates states that are already in Q . A rough implementation of such check operation requires $O(|Q|)$ operations. Since the check must be repeated for all states in Σ , the overall number of operations in a single iteration is $O(|Q||\Sigma|)$. Since it also holds that $\Sigma \subseteq \Gamma$, recalling the bound on the number of iterations, we have that the overall number of operations of the algorithm is $O(|\Gamma|^3)$. Reminding that α is injective and using the result of Proposition 3.4,

$$|\Gamma| = |\mathcal{B}_r(f_0)| \leq |f_0|^2 C(k+r)^r \phi^r (r+1),$$

Algorithm 6 Dynamic Programming with Dominance

```

Input:  $f_0, r, \mathcal{A}, \mathcal{A}^t$ 
Output:  $f$ 
 $s_0 \leftarrow (0, \mathcal{A}, 0)$  or  $(0, \mathcal{A}, \emptyset)$ 
insert( $Q, s_0$ )
while  $Q \neq \emptyset$  do:
     $s = (\beta, \gamma, \sigma) \leftarrow Q[0]$ 
    if  $\gamma = \mathcal{A}_t$  then
         $f \leftarrow \text{repr}(\alpha^{-1}(s))$ 
         $Q \leftarrow \emptyset$ 
    else
         $\Sigma \leftarrow \text{expand}(s, f_0, r)$ 
        for  $s_k \in \Sigma$  do
            if  $s_k$  is not dominated in  $Q$  then
                insert( $Q, s_k$ )
                for  $s_i \in Q$  do
                    if  $s_k$  dominates  $s_i$  then
                        remove( $Q, s_i$ )
                    end if
                end for
            end if
        end for
    end if
end while

return  $f$ 

```

where $k = |P|$ and $\phi = \text{outdeg}(G)$. Therefore, the time complexity of Algorithm 6 is

$$O(|f_0|^6 C^3 (k+r)^{3r} \phi^{3r} (r+1)^3).$$

Algorithm 4 calls Algorithm 6 at most $|f_0|$ times (after each call, the length of the

plan is decreased by at least one), and so it has time complexity

$$O(|f_0|^7 C^3 (k+r)^{3r} \phi^{3r} (r+1)^3).$$

Polynomiality, for fixed r , follows from the observation that rule-based algorithms are able to return an initial plan f_0 with polynomial length.

For the case where the agent distance defined in (3.13) is used, a similar development, exploiting Proposition 3.6, leads to the complexity $O(|f_0|^3 C^3 k^{3r} (n-k+r)^{3r})$ for Algorithm 6, and $O(|f_0|^4 C^3 k^{3r} (n-k+r)^{3r})$ for Algorithm 4.

Finally, we observe that also Algorithm 5 has a polynomial time complexity, as it calls Algorithm 4 with the two different neighborhoods at most $|f_0|$ times. $\square$

Then, the polynomial time complexity of Algorithms 4, 5 and 6 is established if we are able to prove that the `expand` procedure has polynomial time complexity. This will be shown in what follows.

Procedure `expand`

As explained in the proof of Theorem 3.7, the `expand` function returns a number of states $\Sigma \subset \Gamma$ that grows polynomially with the number of agents. The set of states reachable from a state s_0 , without keeping into account the distance constraint, is:

$$\Sigma_0 = \{\tilde{p}(s_0, e) : e \in \mathcal{E}\},$$

whose cardinality, in the worst case, grows exponentially with respect to the number of agents. For instance, if the out-degree of all the nodes in γ is ϕ (the maximum out-degree of the graph), the number of configurations reachable from γ could be as large as $\phi^{|\mathcal{P}|}$ (recall that $\mathcal{E} = |\bar{E}|^{|\mathcal{P}|}$). This means that we cannot compute Σ by first computing Σ_0 and then discarding all its elements with a distance larger than r from f_0 , as the resulting time complexity would be exponential with respect to the number of agents. We can use different approaches, depending on the type of distance used, to overcome this problem.

U-Agents Distance In case the U-Agents Distance is employed, and the state is $s_0 = (\beta, \gamma, \Omega)$, the `expand` procedure is polynomial as the search for new configurations can be performed as follows:

- First, we choose the agents that will change their path with respect to the original path f_0 . In the worst case, Ω , that is the set of agents that have already changed their path, is empty, so we have to choose r agents. The number of different ways we can choose the agents is $\binom{|P|}{r}$.
- Then, we search for the new positions of those agents. As the agents that change their positions are r , in the worst case, the number of configurations reachable from γ is ϕ^r , where ϕ is the maximum out-degree of the graph.

With this implementation, an upper bound for the time complexity of the `expand` procedure is:

$$\binom{|P|}{r} \phi^r \leq \frac{|P|^r}{r!} \phi^r,$$

and is thus polynomial with respect to the number of agents (for fixed r).

Sum-Min Distance If we employ the sum-min distance, we introduce Algorithm 7 as a possible implementation of the procedure `expand`. The main idea is to update the position of one agent at a time, checking for each agent that the configurations we are building are inside the neighborhood. A configuration $\mathcal{A}$ of ordered agents in P can be seen as a string $a_{|P|} = n_1 n_2 \dots n_{|P|}$ such that $\mathcal{A}(q) = n_q$ for $q = 1, \dots, |P|$. Next, we can define a partial configuration with length $t \leq |P|$ as a string $a_t = n_1 n_2 \dots n_t$. Finally, for a given plan f_0 , we can define the distance between the partial configuration a_t and plan f_0 as follows:

$$d(a_t, f_0) = \sum_{i=1}^t \min_{k \in \{1, \dots, |f_0|\}} \hat{d}(n_i, (\psi_{f_0}(k))_i),$$

where $(\psi_{f_0}(k))_i$ is the i -th symbol of the string $\psi_{f_0}(k)$.

Algorithm 7 takes in input the original plan f_0 , the distance r , and the state $s_0 = (\beta, \gamma, \sigma)$ and returns a list of states Σ . The algorithm first initializes a list L_1 of strings

Algorithm 7 Expand for Path Distance

```

1: Input:  $f_0, r, s_0 = (\beta, \gamma, \sigma)$ 
2: Output:  $\Sigma$ 
3:  $L_1 \leftarrow \{\varepsilon\}$ 
4: for  $p \in P$  do
5:    $N \leftarrow adjacent\_nodes(\gamma(p))$ 
6:    $L_0 \leftarrow L_1$ 
7:    $L_1 \leftarrow \emptyset$ 
8:   for  $a_0 \in L_0$  do
9:     for  $n \in N$  do
10:       $a \leftarrow a_0n$ 
11:      if  $\sigma + d(a, f_0) \leq r \wedge isvalid(a)$  then
12:         $L_1 \leftarrow L_1 \cup \{a\}$ 
13:      end if
14:    end for
15:  end for
16: end for
17:  $\Sigma \leftarrow \emptyset$ 
18: for  $a \in L_1$  do
19:    $s \leftarrow (\beta + 1, a, \sigma + d(a, f_0))$ 
20:    $\Sigma \leftarrow \Sigma \cup \{s\}$ 
21: end for
22: return  $\Sigma$ 

```

with the empty string ε . Then, for each $p \in P$, we first generate the set N of nodes adjacent to node $\gamma(p)$ (procedure *adjacent_nodes*). Next, we copy L_1 into the list L_0 , and we set L_1 equal to the empty collection. Then, for each partial configuration $a_0 \in L_0$ and each node $n \in N$, we generate a new string a appending n at the end of a_0 . If the new partial configuration a does not lead to a violation of the distance constraint, i.e., the sum of σ and $d(a, f_0)$ does not exceed r , and, moreover, the partial configuration

a is valid (procedure *isvalid* checks that each node inside a is not repeated, i.e., that each node is occupied by at most one agent), we add a to L_1 . Once all agents have been processed and L_1 only contains full configurations, we generate a new state with the first component $\beta + 1$ for each member of L_1 , and we add it to Σ , which is then returned by the procedure.

Proposition 3.8. Let $\mathbb{A}_P$ be the set of all the partial configurations generated at line 10 of Algorithm 7. Then,

$$|\mathbb{A}_P| \leq \sum_{k=0}^{|P|-1} \phi^{r+1} \binom{k+r}{r} |f_0|.$$

Moreover, $\exists C = C(r) \in (0, 1]$ such that:

$$|\mathbb{A}_P| \leq C |P| \phi^r (|P| - 1 + r)^r.$$

Proof. For an ordered subset of agents $I = \{1, \dots, i\} \subseteq P$, we denote with $\mathbb{A}_I$ the set of partial configurations of length in $\{1, \dots, i\}$ generated up to iteration i at line 10 of Algorithm 7. We can prove that:

$$|\mathbb{A}_I| \leq \sum_{k=0}^{i-1} \phi^{r+1} \binom{k+r}{r} |f_0|. \quad (3.31)$$

This can be achieved by induction.

- *Base Case:* for $i = 1$, we have that the number of partial configurations $|\mathbb{A}_{\{1\}}|$ is simply the number of the nodes adjacent to the position of agent 1, which is not larger than ϕ , where ϕ is the maximum out-degree of the graph. Then:

$$i = 1 \implies |\mathbb{A}_1| \leq \phi \leq \phi^{r+1} |f_0| = \sum_{k=0}^{1-1} \phi^{r+1} \binom{k+r}{r} |f_0| = \phi^{r+1} \binom{r}{r} |f_0|.$$

- *Induction Step:* Now we assume that the statement is true for I and we prove it is also true for $I \cup \{i+1\}$. By the inductive assumption, it holds that:

$$|\mathbb{A}_I| \leq \sum_{k=0}^{i-1} \phi^{r+1} \binom{k+r}{r} |f_0|.$$

Then, we can compute the number of partial configurations explored for agent $i + 1$. Looking at Algorithm 7, all partial configurations that are in the list L_0 when we add a node adjacent to the $(i + 1)$ -th node of the current configuration γ , have a distance from f_0 lower than or equal to r . Let $f_0^I \in \mathcal{E}^*$ be the partial plan of agents $\{1, \dots, i\}$, i.e., plan f_0 restricted to the first i agents. Recalling the definition of the neighborhood of a configuration in (3.23), L_0 is a subset of the union of all the neighborhoods of the partial configurations for agents $\{1, \dots, i\}$ inside f_0^I . We have that:

$$|L_0| \leq \left| \bigcup_{j=1, \dots, |f_0^I|} \mathcal{B}_r(\psi_{f_0^I}(j)) \right| \leq \phi^r \binom{i+r}{r} |f_0|.$$

For each partial configuration inside L_0 , we add the new possible positions of agent $i + 1$ to create all the new partial configurations for agents $\{1, \dots, i + 1\}$. As the new possible positions of agent $i + 1$ are at most ϕ , the set $\mathbb{S}_{i+1}$ of new partial configurations is such that:

$$|\mathbb{S}_{i+1}| \leq \phi^{r+1} \binom{i+r}{r} |f_0|.$$

Therefore, at the end of step $i + 1$, the number of partial configurations explored is:

$$\begin{aligned} |\mathbb{A}_{I \cup \{i+1\}}| &= |\mathbb{A}_I| + |\mathbb{S}_{i+1}| \leq \sum_{k=0}^{i-1} \phi^{r+1} \binom{k+r}{r} |f_0| + \phi^{r+1} \binom{i+r}{r} |f_0| = \\ &= \sum_{k=0}^{i+1-1} \phi^{r+1} \binom{k+r}{r} |f_0|. \end{aligned} \quad (3.32)$$

The proposition follows directly from (3.31) when $I = P$. The last statement of the proposition follows from

$$\sum_{k=0}^{|P|-1} \phi^{r+1} \binom{k+r}{r} |f_0| \leq \sum_{k=0}^{|P|-1} \phi^{r+1} \frac{(k+r)^r}{r!} |f_0| \leq |P| \phi^{r+1} \frac{(|P|-1+r)^r}{r!},$$

and setting $C := \frac{1}{r!}$.

□

Proposition 3.9. Algorithm 7 has a polynomial time complexity with respect to the number of agents for fixed r .

Proof. The time complexity of Algorithm 7 follows from Proposition 3.8 and the observation that the most expensive operation at the i -th iteration, i.e., the computation, performed at line 11, of the distance d between each partial configuration with length i and plan f_0 , requires $i|f_0|$ operations. Indeed, for each agent $j \in \{1, \dots, i\}$ we need to compute the minimum distance between the j -th node of the partial configuration and node $(\psi_f(k))_j$, for $k \in \{1, \dots, |f_0|\}$. $\square$

3.4 Experimental results

In this section, we test the local search algorithms presented in the chapter. In particular, in the first part, we report the experiments performed on directed graphs with Algorithm 4, both with path distances and agent distances, and with the alternating Algorithm 5. In the second part, we present the tests performed on undirected graphs in the *movingai* MAPF benchmark suite [2] and the comparative experiment done in relation to the MAPF-LNS algorithm [22].

We coded Algorithm 4 in C++, and Algorithm 5 with *MATLAB*. We ran all tests on a *11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz* processor with a 16 GB RAM.

3.4.1 Directed Graphs

We generated random directed graphs with $|V|$ nodes ranging from 20 to 100 by 10 and $|E| = 4|V|$ edges. The graphs are generated by creating $|E|$ random ordered pair of nodes and using them to build a directed graph. Only strongly connected graphs are selected. The number of agents $|P|$ ranges from 2 to 18, while initial and final configurations are randomly generated. Initial solutions (if they exist) are generated through the diSC algorithm described in [Pub3]. Note that this algorithm is complete, i.e., it always returns a feasible solution in case one exists. However, the quality of such an initial solution is usually low (i.e., the length of the plan is usually long). For each set of experiments, we show the reduction ratio obtained after the application

of local search, compared to the initial solution, and the running time (in seconds). The reduction ratio is given by the ratio $\frac{|f^*|}{|f_0|}$, where f_0 and f^* are, respectively, the initial and the final solution. The graphs reported show the results wrt to $|V|$ for $|P| = \{5, 10, 15\}$ and wrt to $|P|$ for $|V| = \{20, 50, 100\}$.

Path Distance

In the first set of experiments, we tested Algorithm 4 with sum-min distance. As the time needed to explore the neighborhood increases exponentially with the radius r of the neighborhood, we set $r = 1$. This choice allows us to test multiple instances in a short time.

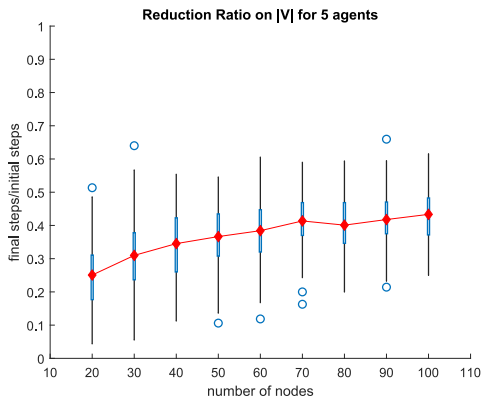


Figure 3.4: Average Reduction Ratio per $|V|$ for 5 agents.

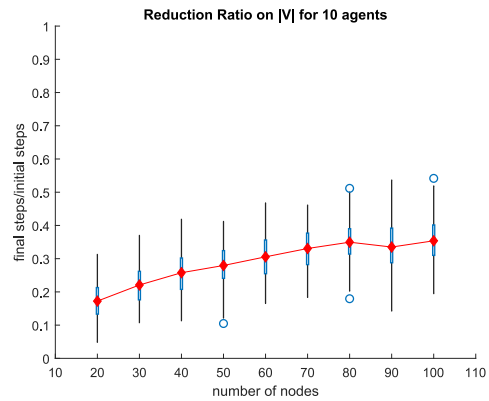


Figure 3.5: Average Reduction Ratio per $|V|$ for 10 agents.

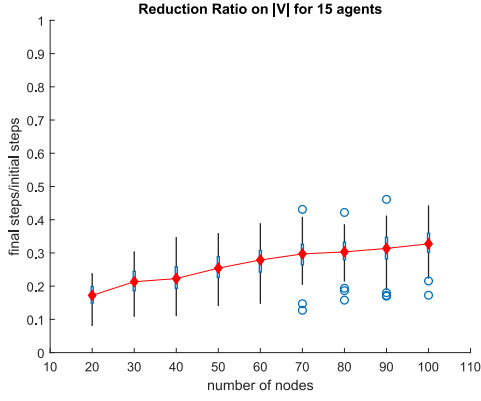


Figure 3.6: Average Reduction Ratio per $|V|$ for 15 agents.

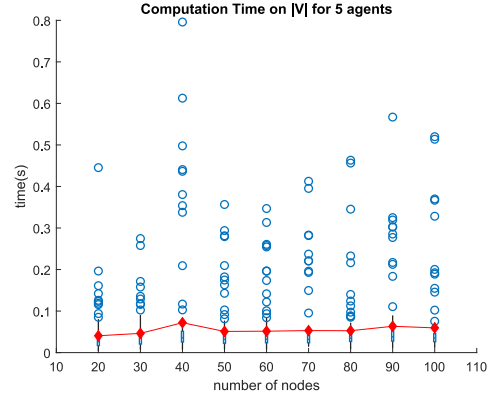


Figure 3.7: Average Running Time per $|V|$ for 5 agents.

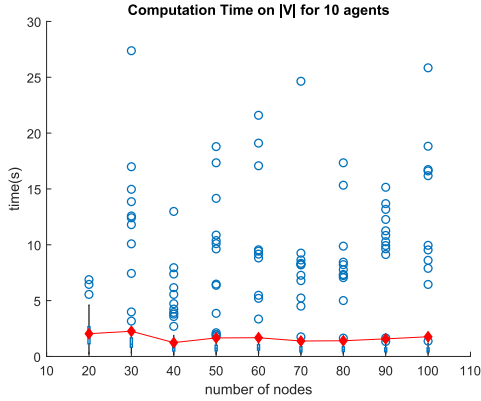


Figure 3.8: Average Running Time per $|V|$ for 10 agents.

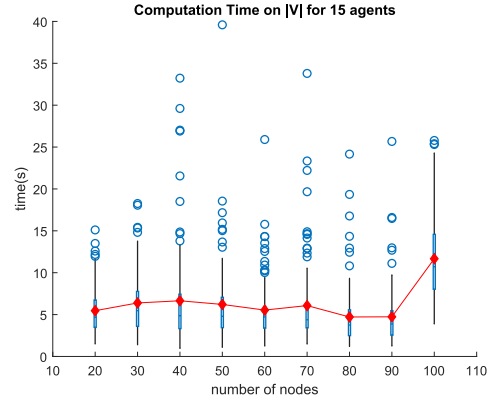


Figure 3.9: Average Running Time per $|V|$ for 15 agents.

Figures 3.4, 3.5, 3.6 and Figures 3.7, 3.8, 3.9 show the reduction ratio and the running time of Algorithm 4 wrt to the number of nodes $|V|$ for three different numbers $|P|$ of agents (namely, $|P|=5, 10, 15$), with the average highlighted in red. In Figures 3.4,

3.5 and 3.6, the reduction ratio slightly increases as the number of nodes grows for all the three different numbers of agents $|P|$. The computation time, instead, remains nearly constant.

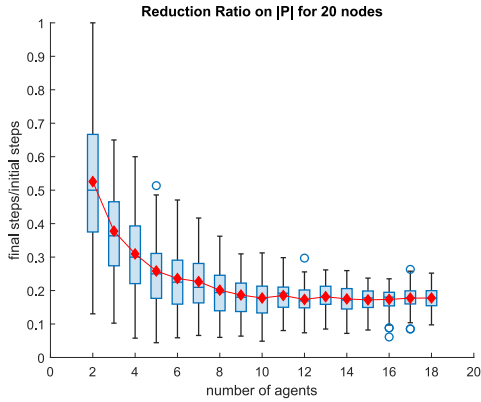


Figure 3.10: Average Reduction Ratio per $|P|$ for 20 nodes.

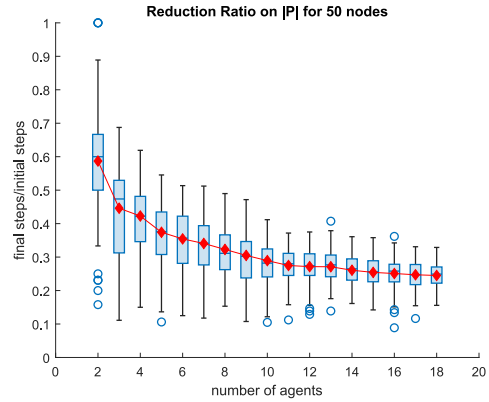


Figure 3.11: Average Reduction Ratio per $|P|$ for 50 nodes.

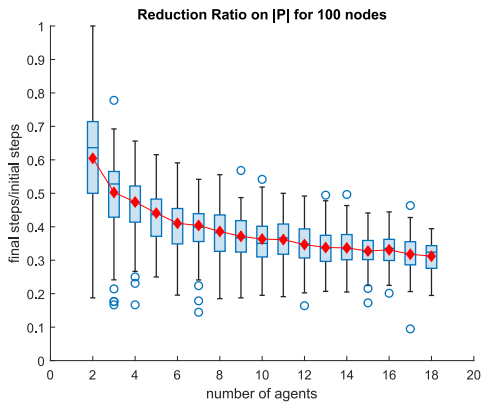


Figure 3.12: Average Reduction Ratio per $|P|$ for 100 nodes.

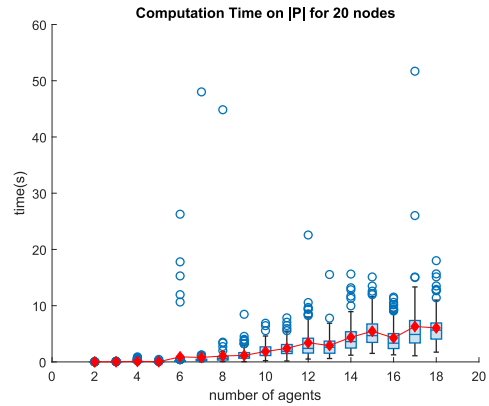


Figure 3.13: Average Running Time per $|P|$ for 20 nodes.

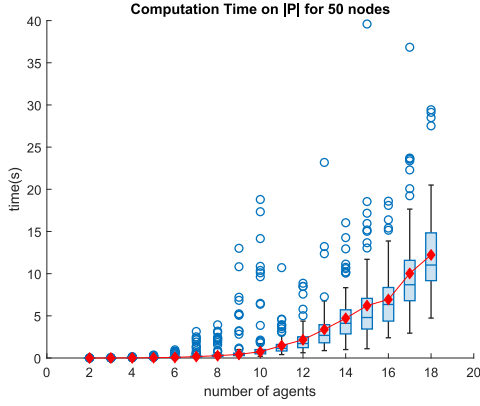


Figure 3.14: Average Running Time per $|P|$ for 50 nodes.

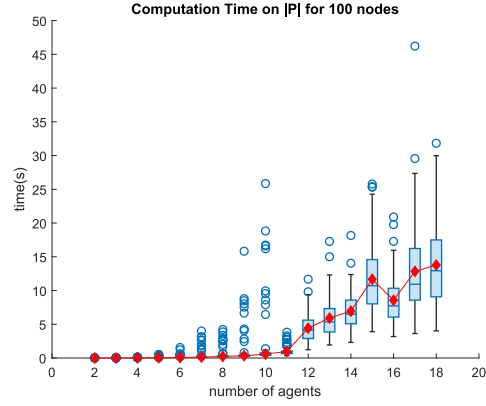


Figure 3.15: Average Running Time per $|P|$ for 100 nodes.

In Figures 3.10, 3.11, 3.12 and Figures 3.13, 3.14, 3.15 we report the reduction ratio and the running time (in seconds), respectively, wrt to $|P|$ for three values of $|V|$ (namely, $|V|=20, 50, 100$). The average is highlighted in red. In these figures, we can observe that the reduction ratio decreases while the computation time increases with the number of agents. The trends in the plots wrt both the number of nodes and the number of agents can be explained by stating that a larger improvement is more likely for more complex instances, i.e., those with a limited difference between the number of nodes and the number of agents, which makes the graph more crowded.

Agent Distance

In the second set of experiments, we tested Algorithm 4 with u-agents distance. In particular, we set the radius $r=1$ so that the path of at most one agent can be changed at a time during the procedure.

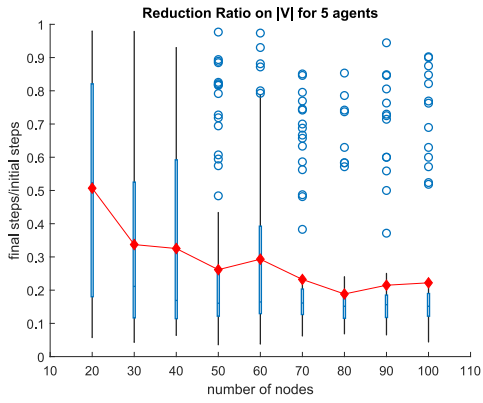


Figure 3.16: Average Reduction Ratio per $|V|$ for 5 agents.

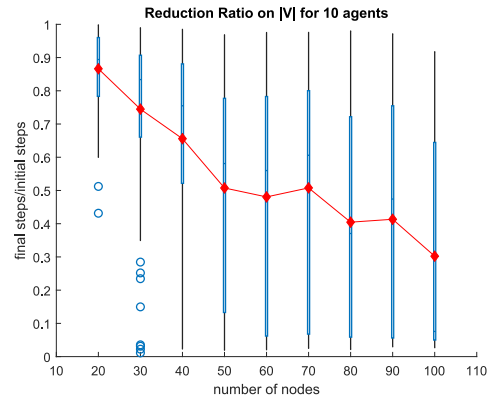


Figure 3.17: Average Reduction Ratio per $|V|$ for 10 agents.

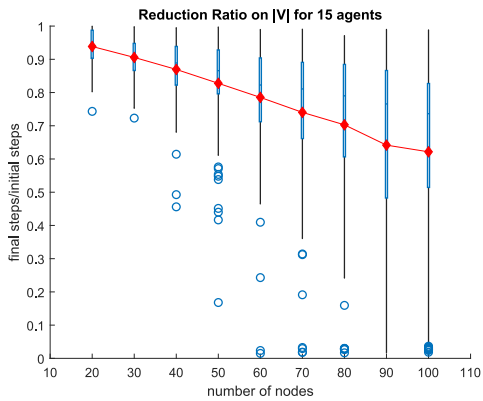


Figure 3.18: Average Reduction Ratio per $|V|$ for 15 agents.

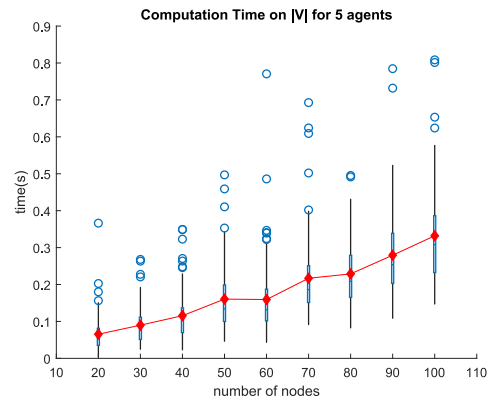


Figure 3.19: Average Running Time per $|V|$ for 5 agents.

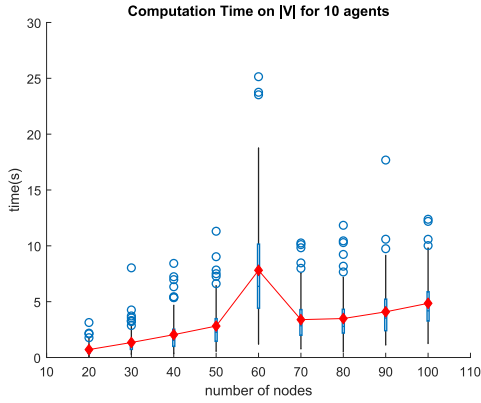


Figure 3.20: Average Running Time per $|V|$ for 10 agents.

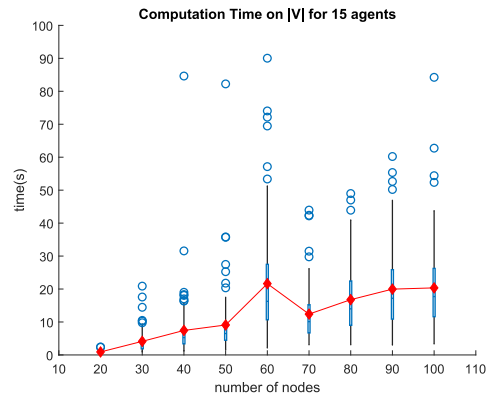


Figure 3.21: Average Running Time per $|V|$ for 15 agents.

In Figures 3.16, 3.17, 3.18 and 3.19, 3.20, 3.21 we report the reduction ratio and the running time (in seconds), respectively, wrt $|V|$ for three values of $|P|$. The average is highlighted in red. Comparing Figures 3.16, 3.17 and 3.18, we can observe that not only the reduction ratio decreases wrt the number of nodes, but also that it is much higher for a high number of agents $|P|$. Regarding the computation time, Figures 3.19, 3.20 and 3.21 show that it increases wrt to the number of nodes for all the number of agents tested.

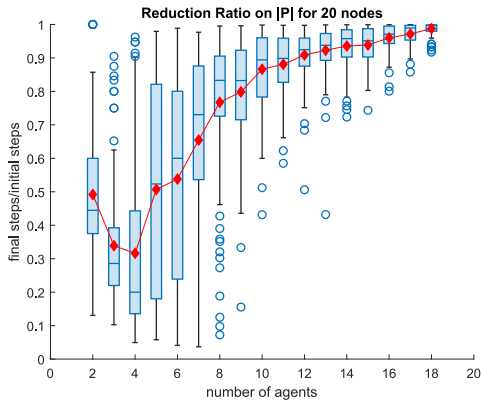


Figure 3.22: Average Reduction Ratio per $|P|$ for 20 nodes.

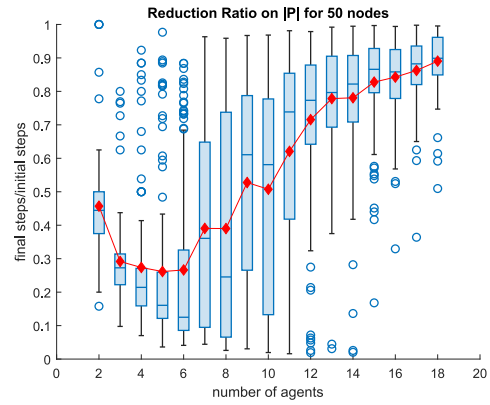


Figure 3.23: Average Reduction Ratio per $|P|$ for 50 nodes.

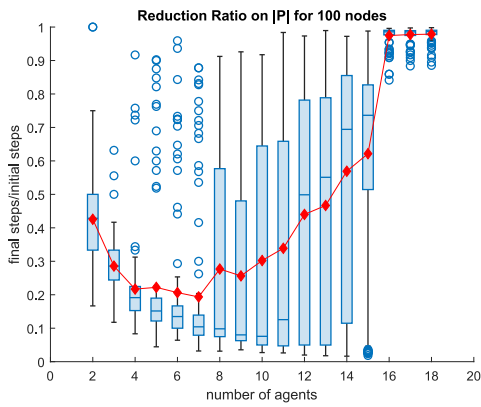


Figure 3.24: Average Reduction Ratio per $|P|$ for 100 nodes.

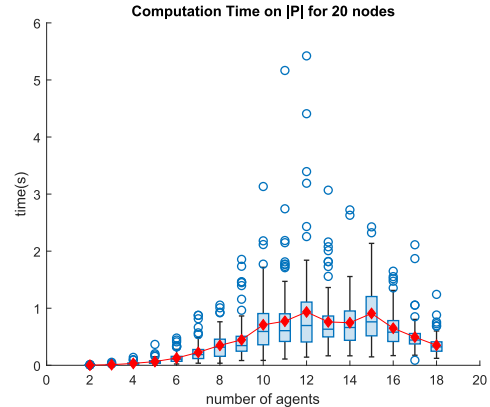


Figure 3.25: Average Running Time per $|P|$ for 20 nodes.

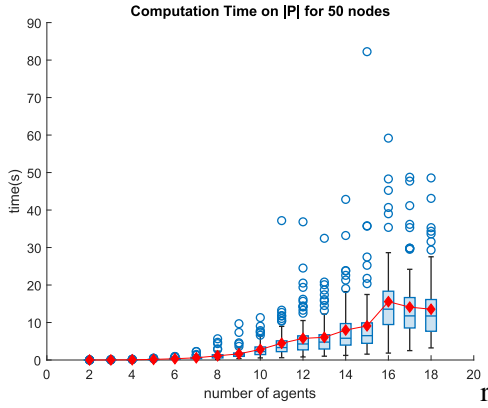


Figure 3.26: Average Running Time per $|P|$ for 50 nodes.

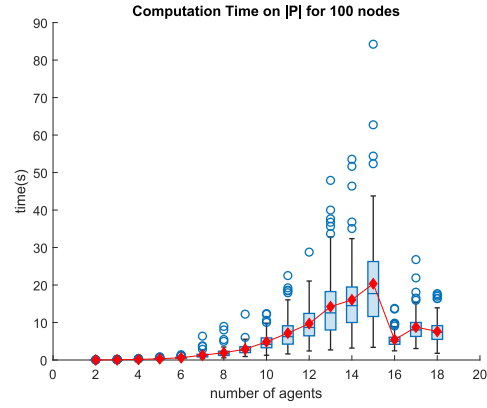


Figure 3.27: Average Running Time per $|P|$ for 100 nodes.

Figures 3.22, 3.23, 3.24 and 3.25, 3.26, 3.27 represent the ratio of decrease and computation times wrt $|P|$ for three values of $|V|$, with the average highlighted in red. We can observe that, in these images, the reduction ratio decreases only for the first few numbers of agents and then increases for higher $|P|$ values. The running time increases slowly until a certain $|P|$ value, and then it starts decreasing. This can be explained by looking at Figures 3.22, 3.23, 3.24, as the running time is lower if the algorithm is not really effective and so the reduction ratio is higher. Looking at the results displayed both Figures 3.16, 3.17, 3.18 and Figures 3.22, 3.23, 3.24, we can state that this algorithm is more effective when a small number of agents moves inside a large graph. It is also worthwhile to remark that Algorithm 4 tends to display opposite behaviors when run with the path and the agent distance.

Alternate Neighborhood Search

Finally, in the last set of experiments, we tested Algorithm 5.

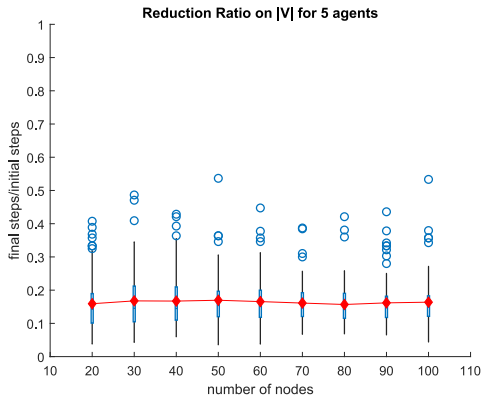


Figure 3.28: Average Reduction Ratio per $|V|$ for 5 agents.

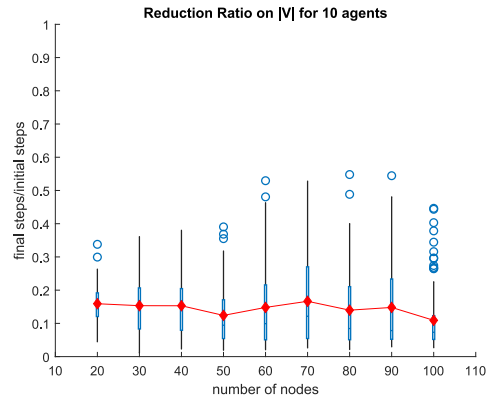


Figure 3.29: Average Reduction Ratio per $|V|$ for 10 agents.

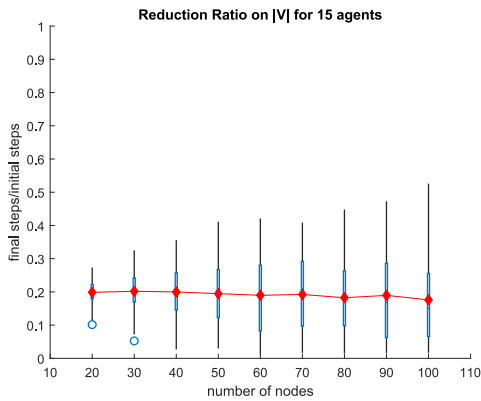


Figure 3.30: Average Reduction Ratio per $|V|$ for 15 agents.

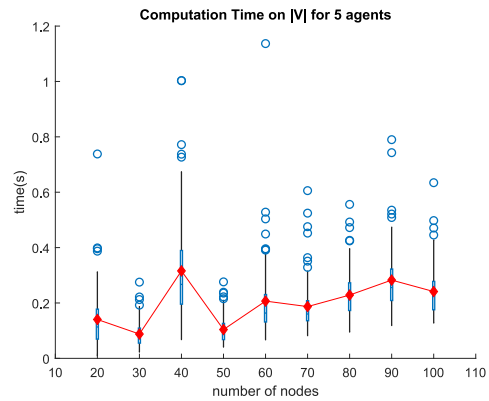


Figure 3.31: Average Running Time per $|V|$ for 5 agents.

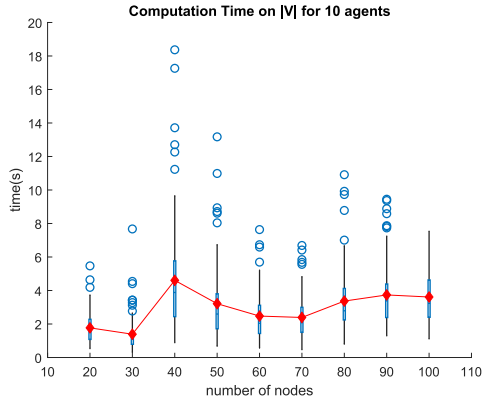


Figure 3.32: Average Running Time per $|V|$ for 10 agents.

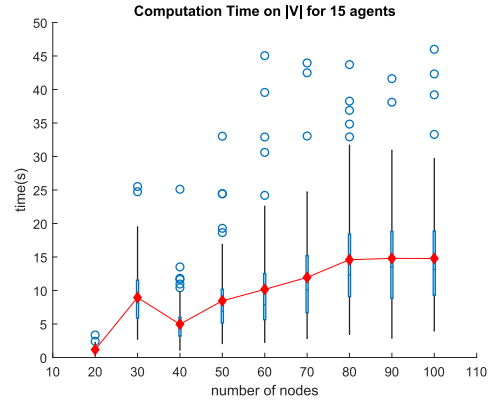


Figure 3.33: Average Running Time per $|V|$ for 15 agents.

In Figures 3.28, 3.29, 3.30 and 3.31, 3.32, 3.33 we report the reduction ratio and the running time (in seconds), respectively, wrt $|V|$ for three different values of $|P|$. The average is highlighted in red. The results show that the reduction ratio is nearly constant wrt every number of nodes tested, whereas the running time fluctuates and slightly increases.

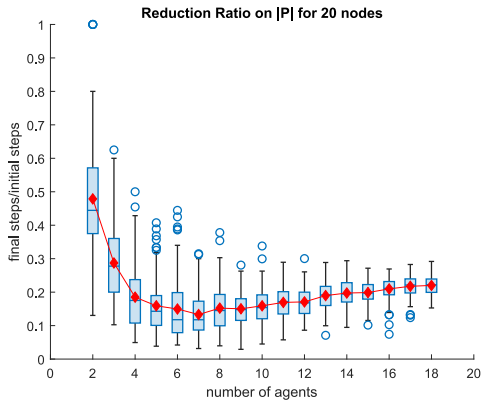


Figure 3.34: Average Reduction Ratio per $|P|$ for 20 nodes.

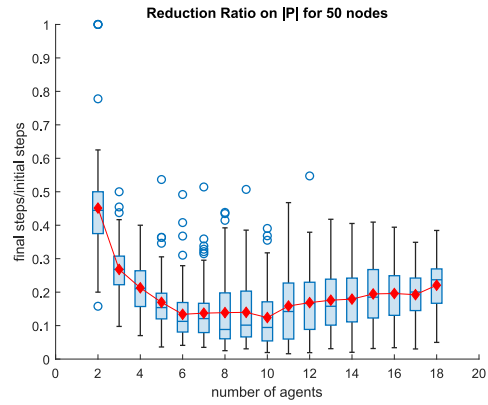


Figure 3.35: Average Reduction Ratio per $|P|$ for 50 nodes.

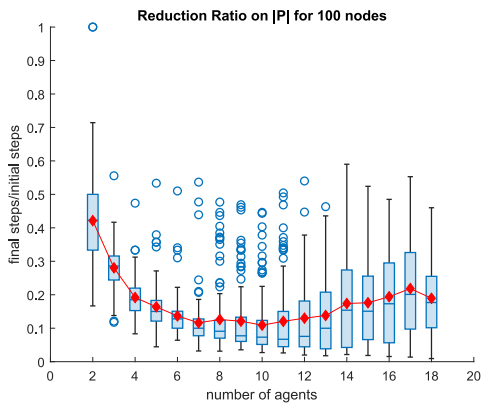


Figure 3.36: Average Reduction Ratio per $|P|$ for 100 nodes.

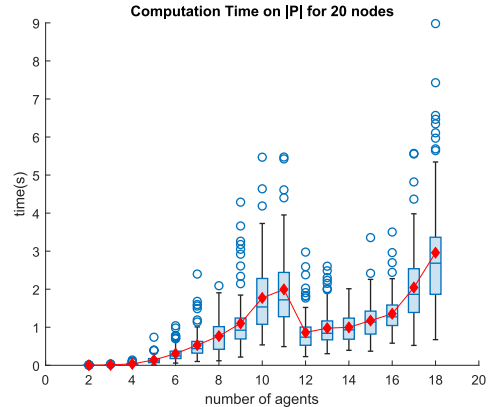


Figure 3.37: Average Running Time per $|P|$ for 20 nodes.

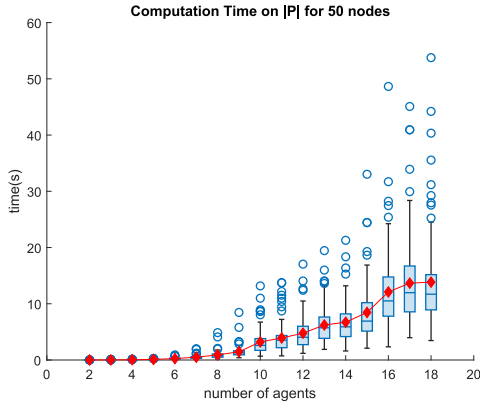


Figure 3.38: Average Running Time per $|P|$ for 50 nodes.

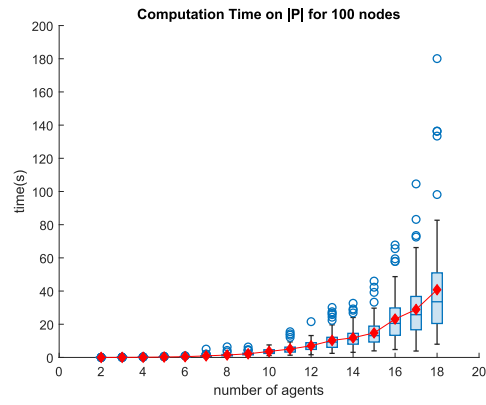


Figure 3.39: Average Running Time per $|P|$ for 100 nodes.

Figures 3.34, 3.35, 3.36 and 3.37, 3.38, 3.39 show the reduction ratio and computation times wrt $|P|$ for three different values of $|V|$, with the average highlighted in red. We can see that the reduction ratio presents the same behavior for all three tested values of $|V|$, decreasing for a lower number of agents and then slightly increasing. The running time, instead, increases with the number of agents. This algorithm performs well for nearly all the instances, with the worst results (in terms of reduction ratio) for a lower number of nodes. This can be explained by observing that the simpler instances have initial plans already close to the optimal solution in comparison to the others.

We can now compare the average results of all three sets of experiments.

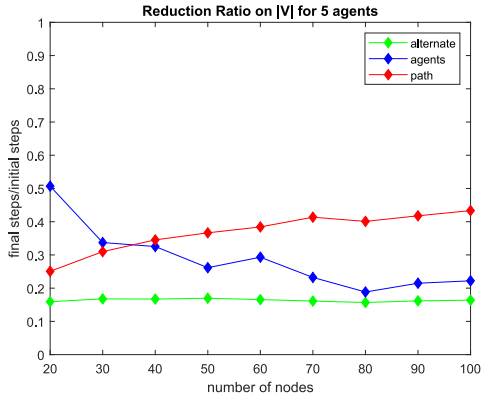


Figure 3.40: Average Reduction Ratio per $|V|$ for 5 agents.

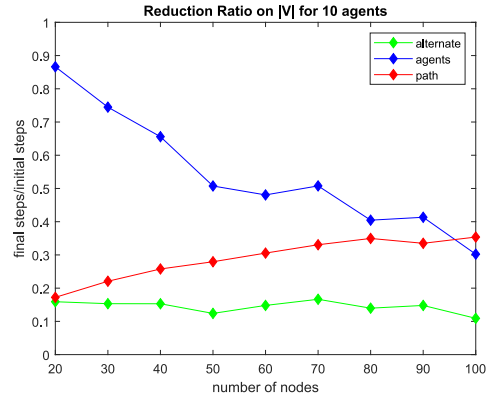


Figure 3.41: Average Reduction Ratio per $|V|$ for 10 agents.

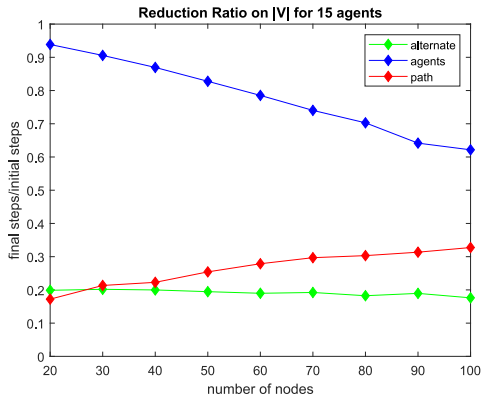


Figure 3.42: Average Reduction Ratio per $|V|$ for 15 agents.

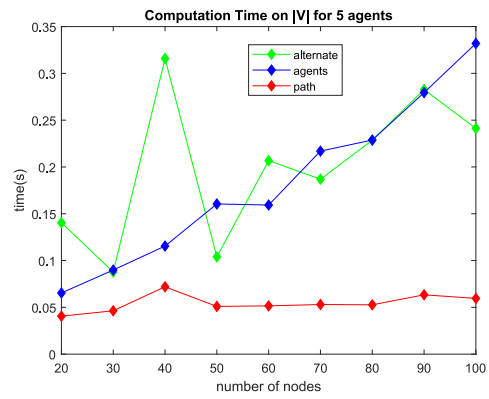


Figure 3.43: Average Running Time per $|V|$ for 5 agents.

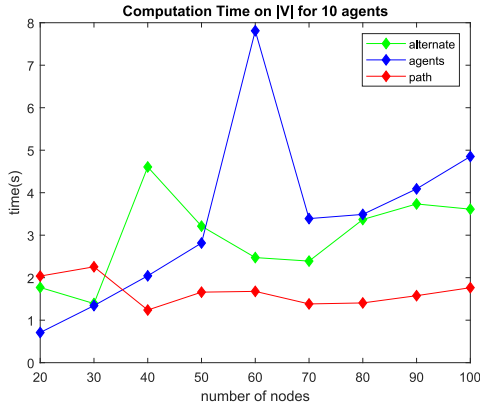


Figure 3.44: Average Running Time per $|V|$ for 10 agents.

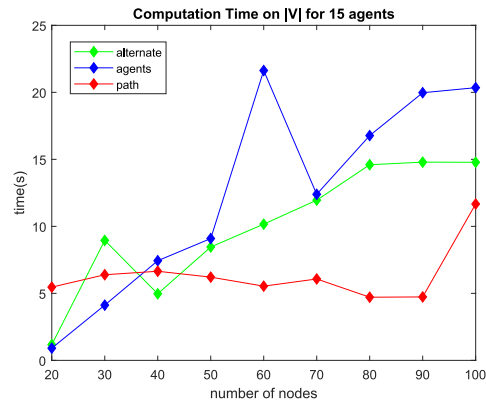


Figure 3.45: Average Running Time per $|V|$ for 15 agents.

In Figures 3.40, 3.41, 3.42 and 3.43, 3.44, 3.45 we can see the comparison between the reduction rates and the computation times of all the approaches wrt the number of nodes $|V|$. In these images, we can see that the Alternate Neighborhood Search always reaches the best plan in comparison to the other approaches. We can also notice that the tests performed using the u-agents distance and the path distance show very different behaviors based on the number of agents. In Figure 3.40, obtained using a low number of agents, using the u-agents distance has better results for $|V| > 30$. On the other hand, for a high number of agents, such as the results depicted in the third image where $|P| = 15$, the path distance approach reaches far better plans for all the number of nodes. This behavior is related to the fact that the u-agents distance approach is not as effective as the path distance approach in crowded graphs (i.e., where the number of agents is high compared to the number of nodes). Figures 3.43, 3.44 and 3.45 show that the algorithm that uses the path-distance is overall faster than the other two algorithms, whereas the other two algorithms have comparable running times that increase wrt the number of nodes.

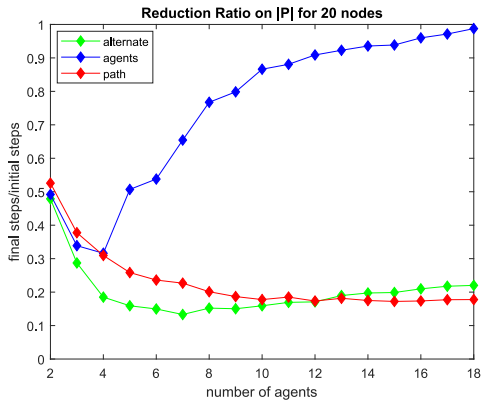


Figure 3.46: Average Reduction Ratio per $|P|$ for 20 nodes.

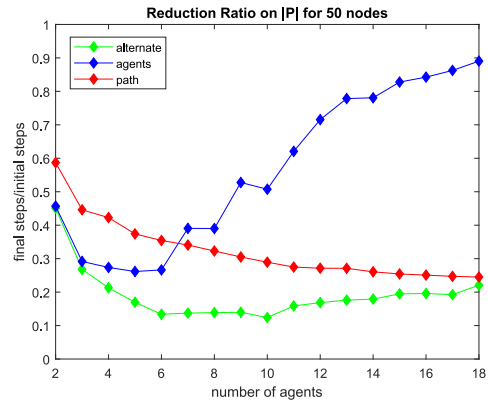


Figure 3.47: Average Reduction Ratio per $|P|$ for 50 nodes.

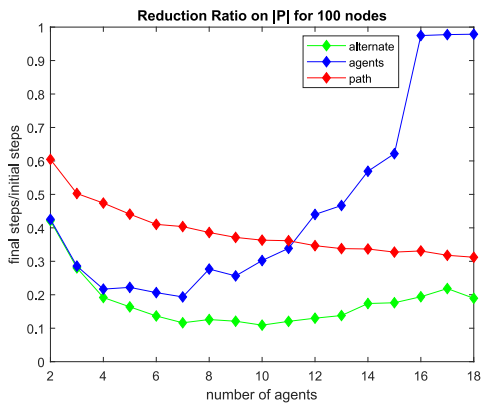


Figure 3.48: Average Reduction Ratio per $|P|$ for 100 nodes.

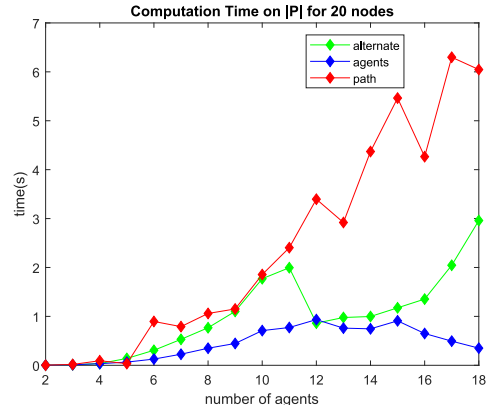


Figure 3.49: Average Running Time per $|P|$ for 20 nodes.

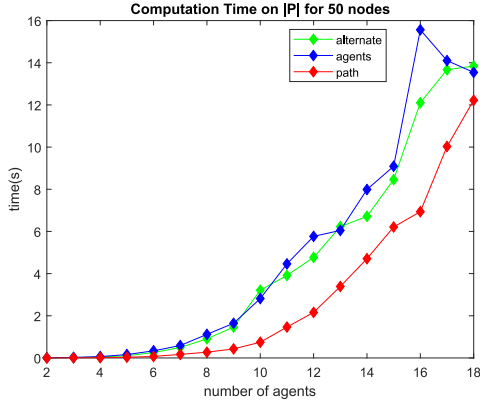


Figure 3.50: Average Running Time per $|P|$ for 50 nodes.

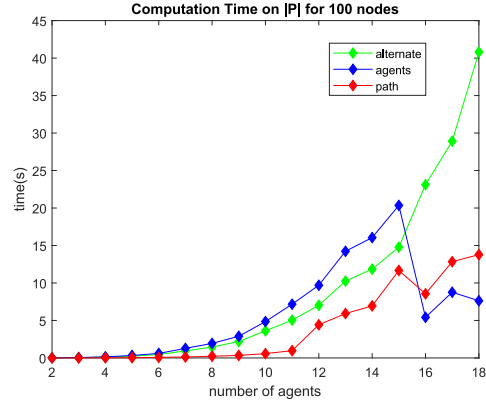


Figure 3.51: Average Running Time per $|P|$ for 100 nodes.

Figures 3.46, 3.47, 3.48 and 3.49, 3.50, 3.51 show the comparison between the reduction rates and the computation times of all the approaches wrt the number of agents $|P|$. As already observed for the results wrt $|V|$, we can see that the behavior of the three different sets of experiments changes in all three different images, depending on the number of agents $|P|$. In Figures 3.46, 3.47 and 3.48 the reduction rates of both Algorithm 4 with sum-min distance and Algorithm 5 decrease wrt the number of agents, whereas the reduction rate of Algorithm 4 with u-agents distance first slightly decreases and then increases, reaching values near 1. This increase is much faster in the first image, where $|V| = 20$, and slows down for higher numbers of nodes. In particular, in the last image where $|V| = 100$, for $|P| < 12$, the algorithm with u-agents distance presents a lower reduction ratio than the algorithm with sum-min distance. The running time appears to increase polynomially wrt $|P|$ for all the algorithms and all the three values of $|V|$. In the first image, we can see how the three sets of experiments present different behaviors. In this case, the running time of the algorithm that uses the sum-min distance is much higher than the others. This can be explained partly as this algorithm reaches better plans than the algorithm that uses the u-agents distance. While the results reached by the alternate neighborhood search are compa-

rable with the ones from the first algorithm, looking at the structure of Algorithm 5, we can see that the first distance used is the u-agents one. This reduces the running time, as Algorithm 4 with sum-min distance is slower if the initial plan f_0 is longer. In Algorithm 5, the longest plan used as the initial plan for Algorithm 4 with sum-min distance has already been reduced by Algorithm 4 with u-agents distance. This explains why, even if Algorithm 5 uses the algorithm with path distance, the running time is, in this case, lower.

3.4.2 Undirected Graphs

We tested the Alternate Neighborhood Search algorithm also with an undirected graph, called maze-32-32-2, from the MAPF benchmark suite from the Moving AI Lab [2]. We set the number of agents as equal to 10, 20, 30, and 40 and randomly generated 10 instances for each different number of agents.

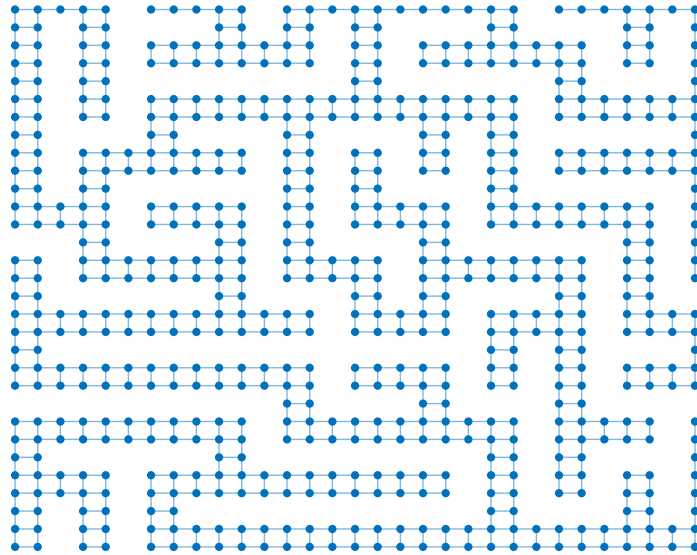


Figure 3.52: Maze-32-32-2 from MAPF benchmark suite.

We first obtained the initial solution using the **Parallel Push and Swap** algorithm [37]. We then run the ANS algorithm with radius 1 for both kinds of distances using the Sum of Costs as the objective.

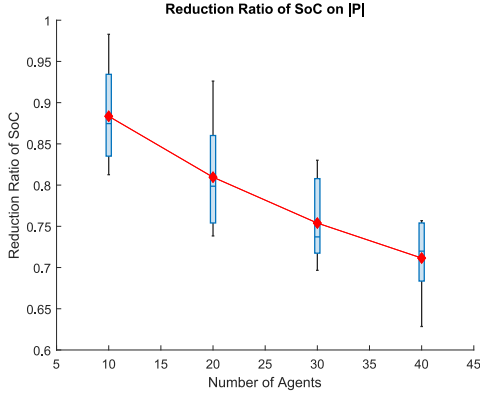


Figure 3.53: Average Reduction Ratio per $|P|$.

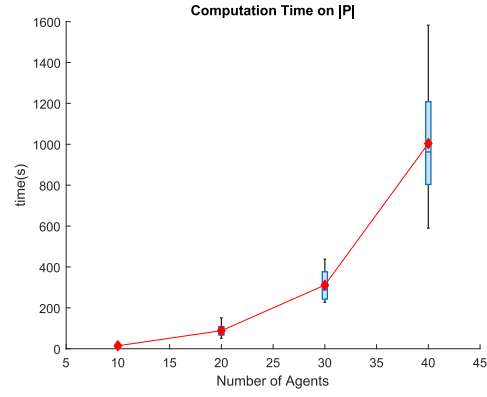


Figure 3.54: Average Running Time per $|P|$.

Figures 3.53 and 3.54 show the reduction ratio and computation times wrt $|P|$. We can see that the algorithm is more effective for higher numbers of agents, while the computation time increases polynomially wrt the number of agents.

Similarities and Differences with MAPF-LNS

As already explained in the introduction, a well-known algorithm based on Large Neighborhood Search is **MAPF-LNS** [22]. This algorithm iteratively improves the quality of a solution obtained from a MAPF algorithm by re-planning subgroups of agents. Its approach is similar to ours if we only look at the agent distances. However, the highlight of the MAPF-LNS algorithm is using heuristics to determine which group of agents to re-plan. MAPF-LNS selects the set of agents to re-plan using randomized procedures in all three of its heuristics. Additionally, it updates the initial plan whenever the re-planned paths are better than the starting paths, rather than generating multiple solutions from the same initial plan and selecting the best one. While this approach is efficient for instances with a high number of empty nodes, a more exhaustive approach may be necessary when the number of agents is close to the number of nodes, even in small graphs. In contrast, our algorithm does not rely

on heuristics or randomized procedures. Instead, it explores the entire neighborhood until it reaches a local minimum. This approach, even if slower for some instances, can be really helpful when the solution to be improved is a long and complicated one. As already explained, the MAPF-LNS algorithm does not assure to reach a local minimum, while reaching a solution that is the best in a defined neighborhood is the founding idea of our work. To compare the two algorithms, we tested them on three simple graphs with a high agent density. We first ran the MAPF-LNS algorithm using the implementation found in [22], available in the GitHub repository [38], with **Conflict Based Search** (CBS) [4] as the re-planning algorithms. We then used the initial solution generated by the **Parallel Push and Swap** algorithm [37] during the first part of the execution as the initial solution for our algorithm. The numbers of agents in our tests were chosen to highlight the differences between the two approaches when facing highly crowded scenarios. We used sum-of-cost as the objective, as it is the objective used in [22], and we used the Alternate Neighborhood Search algorithm.

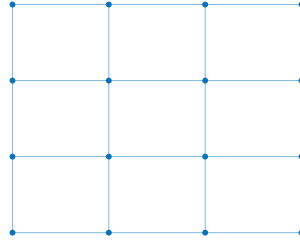


Figure 3.55: Empty 4x4 graph.

The first graph used is the empty 4x4 grid graph in Figure 3.55. We randomly generated 10 scenarios with 14 agents.

	MAPF-LNS		ANS	
original SoC	SoC	time(s)	SoC	time(s)
599	90	200	82	19.64
535	59	200	80	19.54
861	677	200	113	45.59
1685	1678	200	98	64.81
458	58	200	56	52.02
872	423	200	111	33.73
857	416	200	98	52.65
797	59	200	74	43.74
715	206	200	49	42.99
913	54	200	65	86.02

Table 3.5: Results of tests on graph 3.55 with 14 agents. The best result for each instance is highlighted in red

As we can see in Table 3.5, even with an executing time much lower than the cutoff time of 200, the solutions found are similar or much better.

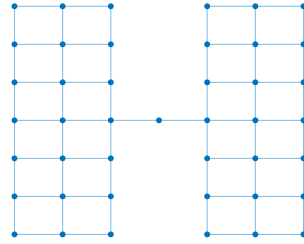


Figure 3.56: Bottleneck graph.

We then tested the two algorithms on the graph in Figure 3.56 made by two empty parts connected by a node. This was done to test the two algorithms in a bottleneck scenario. The graph has 43 nodes, and we randomly generated 10 scenarios with 40 agents. The radius is set to 1 to reflect the imposed cutoff time of 200 seconds for the MAPS-LNS algorithm.

original SoC	MAPF-LNS		ANS	
	SoC	time(s)	SoC	time(s)
8161	7926	200	3976	88.68
5312	4782	200	3333	101.54
3602	2293	200	1797	66.89
6745	6466	200	3112	160.78
5464	4849	200	3001	84.51
7781	7633	200	3514	154.33
3689	3114	200	1843	112.85
3900	3458	200	2560	47.35
6285	5869	200	3503	132.23
6059	5756	200	2997	165.97

Table 3.6: Results of tests on graph 3.55 with 14 agents. The best result for each instance is highlighted in red

In Table 3.6, it is shown how, for these crowded scenarios, the Alternate Neighborhood Search algorithm finds better solutions than the MAPF-LNS algorithm.

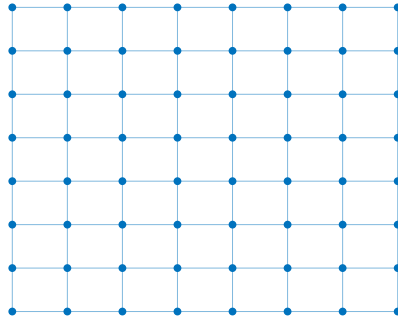


Figure 3.57: Empty-8-8 graph from MAPF benchmark suite.

Lastly, we compared the performances of the two algorithms on an empty 8x8 graph, also present in the MAPF benchmark suite from the Moving AI Lab [2]. We randomly generated 10 scenarios with 60 agents. The time cutoff is set to 400 seconds.

	MAPF-LNS		ANS	
original SoC	SoC	time(s)	SoC	time(s)
34252	34246	400	20833	327.84
18496	18340	400	13227	204.32
22619	22200	400	14670	117.43
9501	9307	400	7388	101.93
22352	21782	400	11632	209.80
7407	7302	400	6345	28.89
13002	12847	400	9028	139.78
19959	19107	400	12449	377.05
12825	12562	400	8063	82.73
7407	7301	400	6345	33.25

Table 3.7: Results of tests on graph 3.57 with 14 agents. The best result for each instance is highlighted in red

Even in this case, we can see in Table 3.7 that the ANS algorithm performs better for crowded scenarios than the MAPF-LNS algorithm.

Conclusion and Future Research

This thesis introduces new approaches to the Multi-Agent Path Finding (MAPF) problem, developing distinct methods for both undirected graphs and strongly connected directed graphs, along with a local search technique to optimize MAPF solutions. The work is organized into three chapters.

The first chapter addresses MAPF on undirected graphs, presenting a hierarchical approach that combines spectral clustering with two rule-based algorithms: BiBOX and Push-and-Swap. This method decomposes the graph into highly connected components, creating a new structure where each node represents a component of the original graph. The MAPF problem is then solved on this component graph, and within each component individually, using two new algorithms, ChiBOX and Chi-Push-and-Swap. By partitioning the graph through spectral clustering, the method enables parallel execution, significantly improving computational efficiency, particularly on densely populated graphs. The total time complexity of the algorithms is $O(n^3)$. Experimental results demonstrate that this approach leads to solutions with fewer moves, shorter paths, and reduced computational times compared to applying rule-based algorithms directly on unprocessed graphs.

The second chapter examines MAPF on strongly connected digraphs, focusing on feasibility determination and efficient solution construction. The thesis provides a necessary and sufficient condition for MAPF feasibility on these graphs and demonstrates that feasibility can be determined in linear time with respect to the number of nodes. We introduce the diSC (digraph Strongly Connected) algorithm, which reduces the MAPF problem on digraphs to a Pebble Motion on Trees (PMT) prob-

lem, making it simpler to analyze and solve. The diSC algorithm then remaps the PMT solution back to the original digraph. The algorithm has a length complexity of $O(|P|n^2c)$, where $|P|$ represents the number of agents, n is the node count, and c is the maximum corridor length. Testing of the diSC algorithm in both random and real-world scenarios, such as warehouse layouts, shows its effectiveness in producing feasible solutions; however, opportunities remain for optimizing solution length further.

The final chapter presents an iterative local search procedure designed to improve sub-optimal MAPF solutions through dynamic programming. Starting from a feasible solution, the algorithm defines a neighborhood using a distance function, then searches within this neighborhood for an improved solution. This process is repeated, redefining the neighborhood based on each new solution, until no further improvements are possible. Two types of distance metrics are introduced: path distances, which compute distances based on the path lengths between nodes, and agent distances, which consider the agents whose paths change between solutions. Another variation of the algorithm alternates between these distances to achieve shorter and more efficient solutions. The search within each neighborhood uses dynamic programming, and due to the locality constraint, the overall computational complexity remains polynomial. Experimental results confirm significant improvements in solution quality, with the algorithm alternating distances achieving the shortest plans compared to other algorithms.

Future Research

As the field of Multi-Agent Path Finding (MAPF) continues to evolve, future research could build on the findings of this thesis by improving the proposed algorithms and exploring their application to real-world challenges.

One potential direction is to improve the clustering techniques used in the algorithms of the first chapter. By testing alternative clustering methods and fine-tuning parameters, researchers could achieve more effective graph decomposition, which would help boost the performance of algorithms like ChiBOX and Chi-Push-and-Swap. Ap-

plying these algorithms to real-world environments, such as AGVs in warehouses, offers further research directions. To make this research applicable in real scenarios, future research could address additional challenges like physical size limits or the ability to adjust to dynamic environments. Another area to explore is parallelization within these algorithms, especially where computing resources are limited. Developing algorithms that make more efficient use of multi-core processors would allow for MAPF solutions that are not only computationally efficient but also scalable to much larger agent sets. Furthermore, the iterative local search algorithm can be expanded by exploring more methods for defining neighborhoods. Exploring alternative distances could lead to improved solutions. Additionally, further optimization of the solution could be achieved by using multi-objective optimization during the local search, or alternating objective functions to escape local minima.

In summary, MAPF research has numerous paths for further exploration. By refining existing methods, optimizing solutions, and addressing real-world constraints, researchers can help create new and more efficient ways to solve MAPF problems on graphs.

Bibliography

- [1] Athanasios Krontiris, Ryan Luna, and Kostas Bekris. From feasibility tests to path planners for multi-agent pathfinding. In *International Symposium on Combinatorial Search*, volume 4, 2013.
- [2] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, pages 151–158, 2019.
- [3] Jingjin Yu and Steven LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, pages 1443–1449, 2013.
- [4] 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.
- [5] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 961–962. IOS Press, 2014.

- [6] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. EECBS: A bounded-suboptimal search for multi-agent path finding. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*, pages 12353–12362. AAAI Press, 2021.
- [7] Jordan Tyler Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *IJCAI*, volume 2011, pages 674–679, 2011.
- [8] David Silver. Cooperative pathfinding. In *Proceedings of the Aaai Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 1, pages 117–122, 2005.
- [9] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *International Joint Conference on Artificial Intelligence*, 2011.
- [10] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *2009 IEEE International Conference on Robotics and Automation, ICRA 2009*, pages 3613–3619. IEEE, 2009.
- [11] Pavel Surynek. Solving abstract cooperative path-finding in densely populated environments. *Comput. Intell.*, 30(2):402–450, 2014.
- [12] Ebtehal Turki Saho Alotaibi and Hisham Al-Rawi. Push and spin: A complete multi-robot path planning algorithm. In *2016 14th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1–8. IEEE, 2016.
- [13] Boris De Wilde, Adriaan W Ter Mors, and Cees Witteveen. Push and rotate: a complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research*, 51:443–492, 2014.
- [14] Vincenzo Auletta, Angelo Monti, Mimmo Parente, and Pino Persiano. A linear-time algorithm for the feasibility of pebble motion on trees. *Algorithmica*, 23(3):223–245, 1999.

- [15] Mokhtar M Khorshid, Robert C Holte, and Nathan R Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Fourth Annual Symposium on Combinatorial Search*, 2011.
- [16] Daniel Martin Kornhauser, Gary Miller, and Paul Spirakis. *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications*. PhD thesis, M. I. T., Dept. of Electrical Engineering and Computer Science, 1984.
- [17] Gabriele Röger and Malte Helmert. Non-optimal multi-agent pathfinding is solved (since 1984). In *Proceedings of the International Symposium on Combinatorial Search*, volume 3, pages 173–174, 2012.
- [18] Pierre Le Bodic and Edward Lam. Optimal unlabeled pebble motion on trees. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, pages 218–222, 2024.
- [19] Bernhard Nebel. On the computational complexity of multi-agent pathfinding on directed graphs. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 212–216, 2020.
- [20] Adi Botea and Pavel Surynek. Multi-agent path finding on strongly biconnected digraphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.
- [21] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.
- [22] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. Any-time multi-agent path finding via large neighborhood search. In *International Joint Conference on Artificial Intelligence 2021*, pages 4127–4135. Association for the Advancement of Artificial Intelligence (AAAI), 2021.
- [23] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. Mapf-Ins2: Fast repairing for multi-agent path finding via large neighborhood

- search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10256–10265, 2022.
- [24] Bin Luo, Richard C. Wilson, and Edwin R. Hancock. Spectral clustering of graphs. In *Graph Based Representations in Pattern Recognition, 4th IAPR Workshop, GbRPR 2003, Proceedings*, volume 2726 of *LNCS*, pages 190–201. Springer, 2003.
- [25] Gilad Goraly and Refael Hassin. Multi-color pebble motion on graphs. *Algorithmica*, 58(3):610–636, 2010.
- [26] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14, 2001.
- [27] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17:395–416, 2007.
- [28] Vincenzo Auletta and Pino Persiano. Optimal pebble motion on a tree. *Information and Computation*, 165(1):42–68, 2001.
- [29] Christos H Papadimitriou, Prabhakar Raghavan, Madhu Sudan, and Hisao Tamaki. Motion planning on a graph. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 511–520. IEEE, 1994.
- [30] Zhilin Wu and Stéphane Grumbach. Feasibility of motion planning on acyclic and strongly connected directed graphs. *Discrete Applied Mathematics*, 158(9):1017–1028, 2010.
- [31] Zhilin Wu and Stéphane Grumbach. Feasibility of motion planning on directed graphs. In *Theory and Applications of Models of Computation: 6th Annual Conference, TAMC 2009, Changsha, China, May 18–22, 2009. Proceedings 6*, pages 430–439. Springer, 2009.
- [32] Mehmet Hakan Karaata. A stabilizing algorithm for finding biconnected components. *Journal of Parallel and Distributed Computing*, 62(5):982–999, 2002.

- [33] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.
- [34] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.
- [35] Bernhard Nebel. The small solution hypothesis for mapf on strongly connected directed graphs is true. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, pages 304–313, 2023.
- [36] David Pisinger and Stefan Ropke. Large neighborhood search. *Handbook of metaheuristics*, pages 99–127, 2019.
- [37] Qandeel Sajid, Ryan Luna, and Kostas Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *Proceedings of the International Symposium on Combinatorial Search*, volume 3, pages 88–96, 2012.
- [38] Jiaoyang-Li. MAPF-LNS, 2021. URL: <https://github.com/Jiaoyang-Li/MAPF-LNS>.

Publications

- [Pub1] Stefano Ardizzoni, Irene Saccani, Luca Consolini, Marco Locatelli, and Bernhard Nebel. An algorithm with improved complexity for pebble motion/multi-agent path finding on trees. *Journal of Artificial Intelligence Research*, 79:483–514, 2024.
- [Pub2] Irene Saccani, Kristýna Janovská, and Pavel Surynek. Spectral clustering in rule-based algorithms for multi-agent path finding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, pages 281–282, 2024.
- [Pub3] Stefano Ardizzoni, Irene Saccani, Luca Consolini, and Marco Locatelli. Multi-agent path finding on strongly connected digraphs. In *2022 IEEE 61st Conference on Decision and Control (CDC)*, pages 7194–7199. IEEE, 2022.
- [Pub4] Stefano Ardizzoni, Irene Saccani, Luca Consolini, and Marco Locatelli. Multi-agent path finding on strongly connected digraphs: feasibility and solution algorithms. accepted by Artificial Intelligence, 2024.
- [Pub5] Irene Saccani, Stefano Ardizzoni, Luca Consolini, and Marco Locatelli. Dynamic programming based local search approaches for multi-agent path finding problems on directed graphs. under review, 2024.
- [Pub6] Stefano Ardizzoni, Irene Saccani, Luca Consolini, and Marco Locatelli. Local optimization of mapf solutions on directed graphs. In *2023 62nd*

IEEE Conference on Decision and Control (CDC), pages 8081–8086. IEEE, 2023.

Ringraziamenti

Desidero esprimere la mia sincera gratitudine a tutte le persone che hanno reso possibile la realizzazione di questo progetto di dottorato.

Innanzitutto, un sentito grazie ai professori Marco Locatelli e Luca Consolini per la loro guida, pazienza e costante supporto, che hanno rappresentato un punto di riferimento lungo tutto il percorso.

I would also like to thank Professor Pavel Surynek, who supported me during my time abroad at CVUT in Prague, for welcoming me into a new environment and for the growth opportunities this experience provided. Thank you also to Kristyna for the support in the work we did together.

Un ringraziamento speciale va al gruppo di ricerca e ai miei colleghi dottorandi: in particolare Stefano e Mattia per il lavoro svolto insieme, e Marina, Andrea, Giammarco e Giulia per aver condiviso con me questi anni, rendendoli un percorso ricco di confronti e collaborazione. Grazie anche a Davide per essermi sempre accanto.

Non posso non menzionare la mia famiglia, il cui amore e supporto sono stati fondamentali in ogni momento di questa esperienza.

Grazie infine ai miei amici, in particolare Chiara, Valeria e Nicolò, per esserci sempre stati, nelle sfide e nelle soddisfazioni.

A tutti voi, grazie per aver fatto parte di questo percorso.