Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Cooperative Path Planning for Big Teams of Robots

Bc. Jakub Lukeš

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Robotics
March 2016

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student:	Bc. Jakub L u k e š
Study programme:	Cybernetics and Robotics
Specialisation:	Robotics
Title of Diploma Thesis:	Cooperative Path Planning for Big Teams of Robots

Guidelines:

1. Get acquainted with current approaches to collision-free path planning for a teams of cooperating agents/robots.
2. Choose a most promising method and implement it. The selection should be done with respect to ability of the method to be extended by adding constrains to robots and trajectories (heterogeneous team of robots, different maps for different types of robots, etc.) and its computational complexity.
3. Propose extensions of the method, which will consider the previously mentioned constrains.
4. Verify experimentally the proposed solution and describe and discuss obtained results.

Bibliography/Sources:

[1] B. de Wilde, A. W. ter Mors and C. Witteveen. Push and Rotate: a Complete Multi-agent Pathfinding Algorithm, Volume 51, pages 443-492, 2014
[2] B. de Wilde. Cooperative Multi-Agent Path Planning, Ph.D. thesis, Delft, the Netherlands, 2012
[3] W. Wang and W. B. Goh. A stochastic algorithm for makespan minimized multi-agent path planning in discrete space. Appl. Soft Comput. 30, C, May 2015, 287-304
[4] Peasgood, M.; Clark, C.M.; McPhee, J. A Complete and Scalable Strategy for Coordinating Multiple Robots Within Roadmaps, in Robotics, IEEE Transactions on , vol.24, no.2, pp.283-292, April 2008
[5] Cap, M.; Novak, P.; Kleiner, A.; Selecky, M., Prioritized Planning Algorithms for Trajectory Coordination of Multiple Mobile Robots, in Automation Science and Engineering, IEEE Transactions on , vol.12, no.3, pp.835-849, July 2015

Diploma Thesis Supervisor: RNDr. Miroslav Kulich, Ph.D.

Valid until: the end of the summer semester of academic year 2016/2017

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, December 22, 2015

Acknowledgements

I thank here to my supervisor RNDr. Miroslav Kulich, Ph.D for helping me with this topic and sharing great deal of information and experiences from his own previous projects and for always being capable of helping me. His different point of view speeded up the solution of many problems which occurred.

Declaration

Abstract

The goal of this thesis is to choose a most promising method for solving the movement of goods from one place to another. This task is called the Warehouse problem and it is defined a task where agents (usually robots) on a graph must reach their goal positions while avoiding obstacles and other agents. The graph also contains racks which have an ability that an agent not carrying a rack can go under them but agent with a rack cannot. Few algorithms are shortly described based on their computational complexity and completeness and the *Push and rotate* algorithm is selected as the most fastest and also suitable the extension. *Push and rotate* is implemented using C++ programming language, the implementation is described in detail and experimental results are discussed. The most important result is acquiring a solution to a task with 58 581 nodes and 982 agents in 42 seconds from which 10 seconds can be removed when a replanning on the same map is invoked.

Keywords: path planning with two types obstacles, teams of heterogeneous agents, complete algorithm, *Push and rotate*

Supervisor:
RNDr. Miroslav Kulich, Ph.D.

Abstrakt

Cílem této diplomové práce je vybrat nejvíce slibnou metodu pro řešení přemístění zboží z jednoho místa na druhé. Tato úloha je nazvána problém skladu a je definována jako úloha, kde agenti (obvykle roboty) musí na grafu dosáhnout cílové pozice a zároveň se vyhnout překážkám a ostatním agentům. Graf také obsahuje regály, které mají tu vlastnost, že agent, který nějaký nepřeváží, se může pod ním pohybovat, ale agent s regálem ne. Několik algoritmů je krátce popsáno v závislosti na jejich výpočetních rychlostech a úplnosti a algoritmus *Zatlač a zatoč* je vybrán jako nejrychlejší a zároveň vhodný pro uvedené rozšíření. *Zatlač a zatoč* je implementován v jazyce C++, implementace je do detailu popsána a experimentální výsledky jsou popsány. Nejvýznamějším výsledkem je získání řešení pro úlohu s 58 581 uzly a 982 agenty za 42s, kde 10s může být odstraněno při přeplánování na té samé mapě.

Klíčová slova: plánování cesty s dvěma typy překážek, týmy heterogenních robotů, úplný algorithm, *Zatlač a zatoč*

Contents

1 Introduction **1**

2 Problem description **3**

2.1 Common approaches 4

3 Push and rotate **7**

3.1 Division into subproblems 8

3.1.1 Algorithm description 8

3.2 Assigning agents to subproblems . 9

3.2.1 Algorithm description 10

3.3 Priority relation between subproblems 11

3.3.1 Algorithm description 12

3.4 Solve 13

3.4.1 Plan 15

3.4.2 Push 16

3.4.3 Move 16

3.4.4 Clear_vertex 16

3.4.5 Swap 17

3.4.6 Multipush 18

3.4.7 Clear 19

3.4.8 Exchange 20

3.4.9 Reverse 21

3.4.10 Rotate 22

3.4.11 Resolve 22

3.5 Smooth 23

3.5.1 Smooth algorithm description 24

3.5.2 Detect redundancies 25

3.5.3 Remove redundancies 26

3.6 Example solution 27

4 Push and rotate implementation **31**

4.1 Agents 31

4.2 Nodes 32

4.3 Arena 32

4.4 Division into subproblems 32

4.5 Assigning agents to subproblems 33

4.6 Priority relation between subproblems 33

4.7 Solve 34

4.7.1 Plan 34

4.7.2 Push 34

4.7.3 Clear_vertex 34

4.7.4 Swap 34

4.7.5 Multipush 34

4.7.6 Clear 35

4.7.7 Exchange 35

4.7.8 Reverse 35

4.7.9 Rotate 35

4.7.10 Resolve 35

4.8 Smooth 35

4.9 Move 36

4.10 Extension for two types of agents 37

5 Experiments **39**

5.1 maze27-11 40

5.2 maze512-1-0 41

5.3 AR0411SR 41

5.4 random512-40-0 42

5.5 warehouse 42

5.6 Time complexity 51

5.7 Available maps types 51

5.7.1 Low node count map type . . . 51

5.7.2 Benchmark map type 52

6 Conclusion **53**

7 CD content **55**

Bibliography **57**

Figures

3.1 Agent $R0$ has a goal position at node pointed by arrow. If the agent $R1$ has a higher priority than $R0$ then the task has no solution. 8
3.2 Demonstration how the methods in *Solve* are call each other. 14
3.3 The visualization of a move with its four pointers N_A (next agent), N_V (next vertex), P_A (previous agent) and P_V (previous vertex). Each pointer points to another move. . . . 24
3.4 The solution for a map called vertex produced by the *Smooth* algorithm. It can be seen that the moves 3.4a to 3.4d are redundant because 3.4a is same as 3.4e. 28
3.5 The smoothed solution for a map called vertex produced by the *Smooth* algorithm on the solution from the *Solve* algorithm. Two redundant sequences have been removed. The original solution can be found in figure 3.4. 29

5.1 `Maze27-11` overview. The map is taken from [1]. 40
5.2 The result acquired on the map `maze27-11` using 10 agents. 41
5.3 `Maze512-1-0` overview. The map is taken from [1] and the picture as well. 43
5.4 The result acquired on the map `maze512-1-0` using 101 agents. . . . 44
5.5 The result acquired on the map `maze512-1-0` using 499 agents. . . . 44
5.6 The result acquired on the map `maze512-1-0` using 993 agents. . . . 44
5.7 `AR0411SR` overview. The map is taken from [1] and the picture as well. 45
5.8 The result acquired on the map `AR0411SR` using 101 agents. 46
5.9 The result acquired on the map `AR0411SR` using 497 agents. 46
5.10 The result acquired on the map `AR0411SR` using 982 agents. 46
5.11 `Random512-40-0` overview. The map is taken from [1] and the picture as well. 47
5.12 The result acquired on the map `random512-40-0` using 100 agents. 48
5.13 The result acquired on the map `random512-40-0` using 499 agents. 48
5.14 The result acquired on the map `random512-40-0` using 991 agents. 48
5.15 The map `warehouse` created for this thesis. Each agent has its own colour. The coloured squares are the agents under the rack. The goal destination of an agent is marked by dotted arrow. 49
5.16 The result acquired on the map `warehouse` using 47 agents. The map can be seen in Fig. 5.15. It has 218 nodes and single subproblem of size 118. 50
5.17 Map result using 575 agents. . . 50
5.18 The result acquired on the map `warehouse` using 575 agents. The map can be seen in Fig. 5.15. This map was created copying the `warehouse` map next to it until a width 112 and height 113 nodes was reached. It has 11 472 nodes and single subproblem of size 5 552. . . . 50
5.19 The calls made during the computation of map AR0411SR(Fig. 5.7 done by QTProfiler [2]). 52
5.20 The calls made during the computation of map maze (Fig. 5.3 done by QTProfiler [2]). 52

Tables

Chapter 1
Introduction

Assume a movement of goods from one place to another is needed. This is a common task in warehouses when a customer buys something and the movement is done by humans. They go to one place then they pick goods and then move to another and avoid obstacles (racks, walls) and they try not to hit other humans as well. The transport done by humans is rather inefficient because they are expensive and they cannot move a large quantities quickly moreover they can cause traffic jams as well.

A wide range of algorithms were created to solve the problem with traffic jams and they try to optimize paths of the movements of the goods. The algorithms consider agents (usually robots in reality) for the movement of the goods but they consider only one type of agent. The topic of this thesis is to choose a most promising method for solving this task, implement it and mainly extend it to allow two types of agents.

This task we refer as the Warehouse problem. The goods are stored in racks which either stay still or are moved by some robot. The racks have an ability that a robot which is not carrying a rack can go through them (in reality underneath them) and off course a robot which is carrying a rack cannot move under them. These additional specifications are considered as an extension to the Warehouse problem.

The Warehouse problem, the definition of the problem and the task, and common approaches are described at first. A description of few algorithms is stated based on their computational complexity and completeness and the *Push and rotate* algorithm is selected as the fastest and also suitable for the extension. Methods using multi agent systems are not considered(Chapter 2).

Push and rotate is theoretically explained at first and all its parts are desribed in detail then. Each of the original algorithms is described and recounted how it works and what are its main advantages. These pseudo codes are shown and explained what they compute line by line. An example using all algorithm parts is shown at the end (Chapter 3).

The implementation and the extension for more types of obstacles is reported and short information about the programming language C++ and its standard template library is added. The Boost library is also shortly told out and the used algorithms from it are described. The extension for the two types of agents is also introduced (Chapter 4).

The experiments are created using the maps available from the Pathfinding benchmark page [1] which contains problems with around 80 thousand nodes and these maps come with a problem set containing description for over one thousands agents. These maps are utilized in experiments and the results for varying number of agents are shortly discussed. A note about available map types is depicted. The Algorithm is capable of processing two types of maps and their syntax is shown (Chapter 5).

Chapter 2

Problem description

The definition of the warehouse problem as well as the common approaches can be found in this chapter.

Imagine a big warehouse with goods. In such warehouse a common operations are to store goods, to retrieve them and sometimes even to reorganize goods locations. All these actions can be simplified to a movement of goods from one place to another one. Let's call a task the movement of a good from one place to another one. Solving one task is easy but what about more than one to be done in parallel, for example 5? The tasks surely influence each other because the goods cannot pass through one another. Let's name those many tasks as a problem. The problem must be solved in real time, i.e. usually in tens of seconds. Let's say that the number of tasks increases to 50. The question is can it be planned still in tens of seconds even for this big problem of tasks?

Let's formulate the Warehouse problem at first. Given graph $\mathcal{G} = (\mathcal{U}, \mathcal{E})$, where $\mathcal{U}$ is a set of nodes and $\mathcal{E}$ is a set of connection between the nodes from $\mathcal{U}$, a set of agents $\mathcal{A}$, an initial assignment of agents to nodes $\mathcal{I} : \mathcal{A} \to \mathcal{U}$, a goal assignment of agents to nodes $\mathcal{T} : \mathcal{A} \to \mathcal{U}$. Let a move $\pi : (\mathcal{A} \to \mathcal{U}) \to (\mathcal{A} \to \mathcal{U})$ be a change of an assignment of agents in exactly one agent (A). A can only change its position from a node u to another node v if and only if $(u, v) \in \mathcal{E}$ and v does not contain another agent before moving with A. It is expected that the number of free space in the graph is much higher than the number of agents [3].

The solution for the problem is a finite sequence of moves $\Pi = [\pi_1, \pi_2, \cdots, \pi_k]$ which satisfies that it moves the agents to their destinations, i.e. $\mathcal{T} = \mathcal{I} \circ \Pi$. In this thesis such sequence is sometimes called list of done moves.

The agent represents the entity which carries out the task. In the reality it can be for example a robot. The nodes represent the important places in the warehouse, for example picking stations or storage.

2.1 Common approaches

Coupled or decoupled methods are usually reasoned for solving the Warehouse problem. Centralized (coupled) approaches plan all agents simultaneously. These methods provide an optimal solution and they are complete, i.e. they always find a solution if one exists or report that no solution exists otherwise. Downside is that the solution usually takes long to be found even for a few agents.

Decoupled methods decouple the problem into subproblems and then solve each subproblem separately. Each agent plans its path separately and solve the task independently. If agent encounters another agent then it tries to found a path along it. This can cause deadlocks. These methods are usually a magnitude faster but they usually are not complete. The solution found by these methods is very often deviated from the optimal solution.

In *Windowed Hierarchical Cooperative A* algorithm* (WHCA*) [4], each agent searches its own path in a three dimensional space-time and uses a reservation table thus taking into account the planned routes of other agents. It also uses precomputed optimal paths to the destination from a few nodes in its windows size. An agent gains better ability to avoid other agents with increasing size of the window but the computation demands increases. The windows size is usually denoted as a number in brackets, i.e. WHCA*(16). If the problem is too big and the window is chosen small then some agents are not capable of reaching their destination.

A scalable multi-agent path planning algorithm with tractability and completeness guarantees (MAPP) [5] is complete for a class of problems called *Slidable* which basically means that an alternate path around the node can be found. With a problem containing m nodes and n mobile units (agents) the worst cast performance for the running time is stated as $O(m^2n^2)$.

A stochastic algorithm for makespan minimized multi-agent path planning in discrete space (PIMM) [6] defines the Warehouse problem as an optimization problem. An objective function is defined using maximum entropy then it is later minimized by a probabilistic iterative algorithm. The algorithm has at least two parameters which must be tuned to receive some useful results. The authors compare PIMM, WHCA*(16) and WHCA(128) on a problem with 80 tasks which they are able to solve in 10 003 ms, 1 650ms and 33 260ms.

A scheduling and routing of autonomous moving objects on a mesh topology [7] is an algorithm for moving $4n^2$ autonomous moving objects (AMOs) on a $n \times n$ mesh topology. An assumption is made on a graph $\mathcal{G}$ that $\mathcal{G}$ allows simultaneous movement of agents in opposite direction on a single vertex. This is too strong constrain for our application. On the other hand the computing requirements are $O(n^2)$.

An efficient and complete centralized multi-robot path planning was proposed by Luna and Bekris [8]. An algorithm *Push and swap* is presented as a complete heuristic which solves general problems with at most $n-2$ agents on a graph with n nodes. The algorithm uses two methods (*push*, *swap*) for manipulating the agents in the graph. *Push* pushes the agent along its path and *swap* swaps an agent with another adjacent to it along the path.

An extension of the algorithm *Push and swap* algorithm was written by de Wilde [9] which divides a graph into subgraphs and solves them independently. It introduces additional methods such as rotate which deals in agents in a cycle.

In this thesis, an algorithm adaptation of *Push and rotate*[3] is presented. The inspiration is taken not from [9] but rather from [3] because it the other source it goes more into detail. This algorithm belongs to the decoupled methods but the author extended it in such way that it is expected to be complete. The main task here is to confirm that this algorithm is as fast as written in [3] because the claim is that it is capable of planning team of 100 agents under 1s.

Many decentralized methods were published last years, but these are not considered in this thesis.

Chapter 3
Push and rotate

The *Push and rotate* algorithm was chosen based on the state of the art as the most fastest and the most suitable for the extension with two types of agents. This algorithm was presented in a PhD thesis [3] where it started as Push and Swap and was extended to solve some other specific problematic tasks, e.g. maps where all nodes have a degree at most two. This entire chapter is based on this thesis and all pseudo code is taken from [3]. Some parts are reformulated and explained more in detail when the algorithm was not clear. The extension for two types of robots is done in next chapter as well as the implementation details.

The main idea of this algorithm is to find subproblems which have an ability to circle agents, i.e. each agent can reach any desired position in the subproblem. Some agents are assigned to this subproblem if they are close enough to it. The next step decides priority of each subproblem based on agents belonging to other subproblems and their goal position. All agents are planned in order which reflects priority of the subproblem they belong to and are moved around the graph until the desired assignment is reached. At the end the algorithm removes all redundant moves from the solution.

The algorithm can be split into three parts. The first one decides the agent planning order (steps: find suproblems, assign agents to them, order subproblems). The second one solves the task itself (step: solve). The third one tries to shorten the solution (step: smooth). Each step is explained in more detail in next subsections. The two types of agents (one carries a rack and the other does not) is mainly used in the second part thus all neighbours of a specific node are considered the ones without a rack if not state otherwise.

When *Push and rotate* computes a path for an agent then other agents are not considered as an obstacle. Either the part using this path does not move with an agent at all (the first part) or it is capable of pushing the agents on path away (the second part).

Why is the planning order of agents is important? It is because in some cases the priority of one agent over another can cause the task to have no solution even if one exists because the agent with higher priority cannot be moved (pushed away) from its position by a lower priority agent.

An example of this situation can be seen in Fig. 3.1. The task has only two agents $R0$ and $R1$ and they are located at their starting locations. The agent

$R1$ is already at its goal location. The doted line marks the goal position of the agent which means that $R0$ needs to move $R1$ from its position. If $R1$ is planned first then $R0$ cannot reach its goal position because it cannot push $R1$ from its position thus the algorithm does not find a solution.

It should be noted that if the priority is assigned fittingly for this task then it only solves this specific case. It can happen that the roles of the agents are reversed and the task would have no solution again. The fixed assignment of priorities cannot be used.

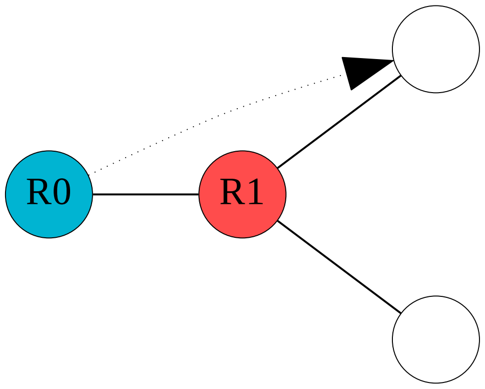


Figure 3.1: Agent $R0$ has a goal position at node pointed by arrow. If the agent $R1$ has a higher priority than $R0$ then the task has no solution.

3.1 Division into subproblems

The planning order of agents influences both the solution quality and its existence. The first three steps of the algorithm *Push and rotate* ensure that this algorithm is complete. The first step is to find biconnected parts of the graph, i.e. subproblems.

All nodes which have degree three or higher are checked if they belong to some subproblem. The check is done by searching all subproblems. If not then they are marked as a single one. Then it is tested if it is possible to merge some subproblems along with their path between them. Note that a path between two subproblems is unique because if another path exists then the subproblems are biconnected.

Let's take a look at this step more in detail. The *Division into subproblems* algorithm can be found in the Algorithm 1.

3.1.1 Algorithm description

The algorithm starts with finding the biconnected components in the graph (line 1). Let a component be a non-empty subgraph. Let a trivial biconnected component is a component of size lesser than 3. Components of size 2 are biconnected but for this application they are not treated as such and they are removed from this set (line 2).

Nodes which have degree more than three can remain assigned to no subproblem. These nodes are important for finding a solution because two agents can swap position using them. These nodes are marked as a subproblem of size one and added to the set of all subproblems (line 3).

Subproblems can be merged together if they are close enough to each other. Let $d(u, v)$ denote the length of the path between nodes u and v, i.e. the number of nodes between them including the start and end nodes. Let $d(\mathcal{C}_1, \mathcal{C}_2) = \min_{u\in\mathcal{C}_i, v\in\mathcal{C}_j} d(u, v)$ denote the length of the shortest path from subproblem $\mathcal{C}_1$ to $\mathcal{C}_2$. Let $m = |\mathcal{U}| - |\mathcal{A}|$ denote the number of unoccupied vertices. If some subproblems, let's say $\mathcal{C}_1$ and $\mathcal{C}_2$, are close enough to each other, i.e. $d(\mathcal{C}_1, \mathcal{C}_2) \leq m - 2$, then they are merged to a single subproblem including the path between them.

On the line 4 starts a while loop which iterates until there exist two subproblems which can be merged together. When two subproblems are merged to a single one then the shortest path between them is added as well. This can cause other subproblems to be mergeable with some other subproblems which was previously not close enough.

Algorithm 1 Division_into_subproblems($\mathcal{G}$)

```
1: $\mathcal{C} \leftarrow$ all biconnected components from $\mathcal{G}$
2: $\mathcal{C} \leftarrow \mathcal{C} \setminus \{$trivial biconnected components$\}$
3: $\mathcal{C} \leftarrow \mathcal{C} \cup \{v \in \mathcal{U} \mid \text{degree}(v) \geq 3 \wedge v \notin C\}$
4: while $\exists \mathcal{C}_i, \mathcal{C}_j \in \mathcal{C} \mid \left(\min_{v\in\mathcal{C}_i, u\in\mathcal{C}_j} d(v, u)\right) \leq m - 2$ do
5:     $\mathcal{C}_k \leftarrow \mathcal{C}_i \cup \mathcal{C}_j \cup \{v' \in \text{shortest\_path}(u, v)\}$
6:     $\mathcal{C} \leftarrow (C \setminus \{\mathcal{C}_i, \mathcal{C}_j\}) \cup \{\mathcal{C}_k\}$
7: end while
```

3.2 Assigning agents to subproblems

From the last section 3.1 the subproblems are known. Now agents are expected to be part of some subproblems, if any. This will later decide their planning order. Agents are assigned to a subproblem if they are capable of entering it, i.e. there is at least one free node in the subproblem [3]. This means that agents in the subproblem are capable of creating such free node for another agent, i.e. agents in the subproblem can leave it if needed.

Let v be a vertex in a subproblem $\mathcal{C}_i$. Let u be a vertex not in $\mathcal{C}_i$. Let m' be the number of unoccupied vertices reachable (a path exists) from v in $\mathcal{G} \setminus \{u\}$. Let m'' be the number of unoccupied vertices reachable from $\mathcal{C}_i$ in $\mathcal{G} \setminus \{v\}$. Let's take an agent a located at $v \in \mathcal{C}_i$. Agent a is assigned to subproblem $\mathcal{C}_i$ if v is located inside $\mathcal{C}_i$, i.e. $\neg\exists w \notin \mathcal{C}_i \mid (w, v) \in \mathcal{E}$.

If the agent a is not located inside the subproblem the nodes are called the edge of the subproblem, i.e. $\exists w \notin \mathcal{C}_i \mid (w, v) \in \mathcal{E}$. If this happens then it is assigned to $\mathcal{C}_i$ in two following cases. The first one is when a can reach at least one free space but not all when avoiding the node v, i.e. $m' \geq 1 \wedge m' < m$. The second one is when the node v does not block access to all free nodes available to $\mathcal{C}_1$, i.e. $m'' \geq 1$. It is possible that the agent located at the node at the edge of the subproblem is not assigned to that subproblem.

When all agents located at nodes belonging to subproblems then the remaining agents are solved. When a node u at an edge of the subproblem $\mathcal{C}_i$ is tested then algorithm adds each agent located at a node v which does not belong to subproblem and it is close enough, i.e. $v \notin \mathcal{C}_i \mid d(u, v) < m' - 1$.

If the agent a is not located at some u belonging to some $\mathcal{C}_i$ then it is assigned to closest subproblem $\mathcal{C}_j$ if it can enter it. Note that an agent can't belong to two subproblems simultaneously and an agent does not need to be assigned to some subproblem. Unassigned agents are always planned last.

3.2.1 Algorithm description

The task here is to found how many free nodes are reachable from a specific node u when avoiding a set of nodes. The set is usually a single node or a subproblem without a single node in it.

The algorithm is described in the Algorithm 2. For each component its all nodes are tested (line 3). If the node is inside a subproblem we assign an agent located at this node to this subproblem, if any. If the node has a connection to outside of subproblem then the values m' and m'' are computed (lines 4 and 5).

The line 9 means that the nearest nodes are followed away from the removed node v starting at the node u and ignoring the connection to v. If the node is empty then no agent is assigned but the value m' is lowered. This can cause the number of assigned agents to be lower than the value m'.

Algorithm 2 Assigning_agents_to_subproblems($\mathcal{G}, \mathcal{C}$)

```
1:  for all $\mathcal{C}_i \in \mathcal{C}$ do
2:      for all $v \in \mathcal{C}_i$ do
3:          if $\exists u \notin \mathcal{C}_i$ for which $(u, v) \in \mathcal{G}$ then
4:              $m' \leftarrow$ num. of unoccupied vertices reachable from $v$ in $\mathcal{G} \setminus \{u\}$
5:              $m'' \leftarrow$ num. of unoccupied vertices reachable from $\mathcal{C}_i$ in $\mathcal{G} \setminus \{v\}$
6:              if $(m' \geq 1 \land m' < m) \lor m'' \geq 1$ then
7:                  Assign agent on position $v$ to $\mathcal{C}_i$ (if any)
8:              end if
9:              Follow path from $u$ away from $v$ (include nodes with racks)
10:             and assign all agents at the first $m' - 1$ nodes
11:             on this path to $\mathcal{C}_i$
12:         else
13:             Assign agent on position $v$ to $\mathcal{C}_i$ (if any)
14:         end if
15:     end for
16: end for
```

Algorithm 3 Detect_priorities($\mathcal{C}$)

```
1:  $\mathcal{R} \leftarrow$ empty list
2:  for all $\mathcal{C}_i \in \mathcal{C}$ do
3:      $\mathcal{N}(\mathcal{C}_i) \leftarrow$ empty set
4:  end for
5:  for all $\mathcal{C}_i \in \mathcal{C}$ do
6:      for all $\mathcal{C}_j \in \mathcal{C}$ do
7:          $\mathcal{P} \leftarrow$ path from $\mathcal{C}_i$ to $\mathcal{C}_j$
8:          for all $v' \in \mathcal{P}$ do
9:              if $v' \in \mathcal{C}_i$ then
10:                 $\mathcal{P} \leftarrow \mathcal{P} \setminus [v']$
11:             else
12:                 break
13:             end if
14:         end for
15:         for all $v' \in \mathcal{P}$ do
16:             $S \leftarrow t(v')$
17:             if $S \neq \emptyset \wedge S \notin \mathcal{C}_i$ then
18:                 if $S \in \mathcal{C}_j$ then
19:                     $\mathcal{R} \leftarrow \mathcal{R} \cup \{(\mathcal{C}_i \prec \mathcal{C}_j\})$
20:                     break
21:                 end if
22:             else
23:                 break
24:             end if
25:         end for
26:         for all $v' \in \mathcal{P}$ do
27:             if $v' \in \mathcal{C}_j$ then
28:                 $\mathcal{N}(\mathcal{C}_i) \leftarrow \mathcal{N}(\mathcal{C}_i) + \{\mathcal{C}_j\}$
29:                 break
30:             end if
31:             if $\exists \mathcal{C}_k \in \mathcal{C} \mid v' \in \mathcal{C}_k$ then
32:                 break
33:             end if
34:         end for
35:     end for
36: end for
37: return $\mathcal{N}, \mathcal{R}$
```

3.3 Priority relation between subproblems

So far subproblems have been found and agents assigned to them. Each subproblem represent a pack of agents which have same planning priority because they can circle in the subproblem. Now the subproblems shall be ordered.

An agent was assigned to a subproblem if it was close enough. Its assignment depends on the starting position of the agent. On the other hand the priority of the subproblem depends on goal position of agents from other subproblems.

Let $\mathcal{C}_i$ and $\mathcal{C}_j$ be two subproblems. The priority relation $\mathcal{C}_i \prec \mathcal{C}_j$ (agents from subproblem $\mathcal{C}_i$ should be planned before agents from subproblem $\mathcal{C}_j$) is added if an agent $A \in \mathcal{C}_j$ pushes another agent $B \in \mathcal{C}_i$ to an edge of the subproblem $\mathcal{C}_i$ and lock it there or A has a goal position on the edge of the subproblem $\mathcal{C}_i$.

If an agent from subproblem $\mathcal{C}_j$ either has its goal position on the edge of subproblem $\mathcal{C}_i$, or will push another agent to the edge of subproblem $\mathcal{C}_i$ and lock it there, then , which means that .

If both relations exists then they are removed because they make a cycle. This is difference from the source [3] where it was used to detect if the task has solution.

3.3.1 Algorithm description

The algorithm has two parts. The relations between subproblems are detected at first (algorithm 3) Then the found relations are treated as a partially ordered set and the planning order is extracted (algorithm 4).

The first Algorithm starts with two possible same subproblems $\mathcal{C}_i$ and $\mathcal{C}_j$. Two subproblems $\mathcal{C}_i$ and $\mathcal{C}_j$ are chosen (lines 5 and 6). The path between those subproblems is found (line 7). If the path is empty then nothing happens and the order does not matter because, either those two subproblems are not connected and they cannot influence each other, or it is the same subproblem.

The nodes on the path are checked one after another. It is needed to get out of the subproblem $\mathcal{C}_i$ at first. This is done because the algorithm is interested only in testing nodes outside of the subproblem. The path, in general, can start at a node inside of $\mathcal{C}_i$. The next node is checked until the node on the path belongs to $\mathcal{C}_i$(line 9).

At this point the algorithm test a node outside of a subproblem (line 15). All nodes in this for cycle are outside of $\mathcal{C}_i$ because those which belonged to $\mathcal{C}_i$ were removed on line 10. If no agent has a goal position at this node then the search is done and the rest nodes are not checked.

If some agent has a goal position at this node then it is checked if it belongs to $\mathcal{C}_j$ (line 18). If this is true then an information that $\mathcal{C}_i$ must be planned before $\mathcal{C}_j$ (noted as $\mathcal{C}_i \prec \mathcal{C}_j$) is appended to a list of all priority relations $\mathcal{R}$ (line 19). $\mathcal{R}$ is initialized as empty on line 1. The algorithm stops in this case even though the other nodes on the path are not checked (line 20).

The important priorities have been found but for the next part the neighbourhood subproblems are needed as well. On line 26 a for cycle starts. In it the path is followed until a node belonging to $\mathcal{C}_j$ is found (line 27). Or until the path crosses another subproblem $\mathcal{C}_k$ (line 31). In the first situation $\mathcal{C}_j$ is added as a neighbour. In the second situation the search ends.

Neighbours are utilized in algorithm propagate which propagates priority. For all neighbourhood subproblems of $\mathcal{C}_j$ it is tested if the relation $\mathcal{C}_k \prec \mathcal{C}_j$ holds (line 2). If it does then nothing happens. If it does not then it can be said that $\mathcal{C}_j$ has greater priority than $\mathcal{C}_k$ a thus a relation $\mathcal{C}_j \prec \mathcal{C}_k$ is added [3] and the algorithm now propagates to $\mathcal{C}_k$ to its neighbours.

The previous algorithm 3 found all priorities. Now it is needed to extract an order of subproblems which is respecting them. This is done in algorithm 4.

It can happen that $\mathcal{R}$ contains same priorities more than once. If this happens the extra duplicates are removed (line 1). Then it is checked if $\mathcal{R}$ does not contain inverse priorities simultaneously, i.e. $\mathcal{C}_i \prec \mathcal{C}_j$ and $\mathcal{C}_j \prec \mathcal{C}_i$ (lines 2 to 6). Both priorities are removed in this case.

Then all priorities are propagated on line 7. All priorities are represented as an oriented graph and topological sorted at the end.[10]. Finally a priority of agents is assigned according to the subproblem they belong to. This creates the planning order of agents.

Algorithm 4 Fill_all_priorities($\mathcal{R}$,$\mathcal{N}$)

```
1:  $\mathcal{R} \leftarrow$ remove duplicates from $\mathcal{R}$
2:  for all $(\mathcal{C}_i \prec \mathcal{C}_j) \in \mathcal{R}$ do
3:      if $(\mathcal{C}_j \prec \mathcal{C}_i) \in \mathcal{R}$ then
4:          $\mathcal{R} \leftarrow \mathcal{R} \setminus \{(\mathcal{C}_i \prec \mathcal{C}_j), (\mathcal{C}_j \prec \mathcal{C}_i)\}$
5:      end if
6:  end for
7:  for all $(\mathcal{C}_i \prec \mathcal{C}_j) \in \mathcal{R}$ do
8:      Propagate($\mathcal{R}$,$\mathcal{N}$,$\mathcal{C}_j$)
9:  end for
10: return Extract priority order from $\mathcal{R}$
```

Algorithm 5 Propagate($\mathcal{R}$,$\mathcal{N}$,$\mathcal{C}_j$)

```
1:  for all $\mathcal{C}_k \in \mathcal{N}(\mathcal{C}_j)$ do
2:      if $(\mathcal{C}_k \prec \mathcal{C}_j) \notin \mathcal{R}$ then
3:          $\mathcal{R} \leftarrow \mathcal{R} \cup \{(\mathcal{C}_j \prec \mathcal{C}_k)\}$
4:          Propagate($\mathcal{R}$,$\mathcal{N}$,$\mathcal{C}_k$)
5:      end if
6:  end for
```

3.4 Solve

A planning order of agents is known at this point. It is needed now to find a sequence Π which moves the agents $\mathcal{A}$ from its starting position to its goal position.

The *Solve* algorithm plans agents sequentially. It is done by calling the method *Plan*. Note that all algorithms, from later on, return true if they succeeded or false if they did not. *Solve* (Algorithm 6) starts with initialization of an empty list of done moves Π (line 1) and with creation of a set of already planned agents F (line 2). Then each agent is planned sequentially until the a list of done moves is returned (lines 3 to 7). This list is either a solution (line 8) or an empty list (line 5) if no solution has been found. The overview what each method in *Solve* calls can be seen in Fig. 3.2.

In the original algorithm there was also the *Plan** algorithm. The difference is that *Plan** avoids other agents when finding a path for an agent. This issue can only happen on a map where all nodes have degree equal to 2. Those maps are not considered for our application.

Algorithm 6 Solve($\mathcal{G}, \mathcal{A}$)

```
1: $\Pi \leftarrow$ empty list [ ]
2: $\mathcal{F} \leftarrow$ empty set $\emptyset$
3: for all agents $A \in \mathcal{A}$ do
4:     if Plan($\Pi, \mathcal{G}, A, \mathcal{F}$, [ ]) = false then
5:         return empty list [ ]
6:     end if
7: end for
8: return $\Pi$
```

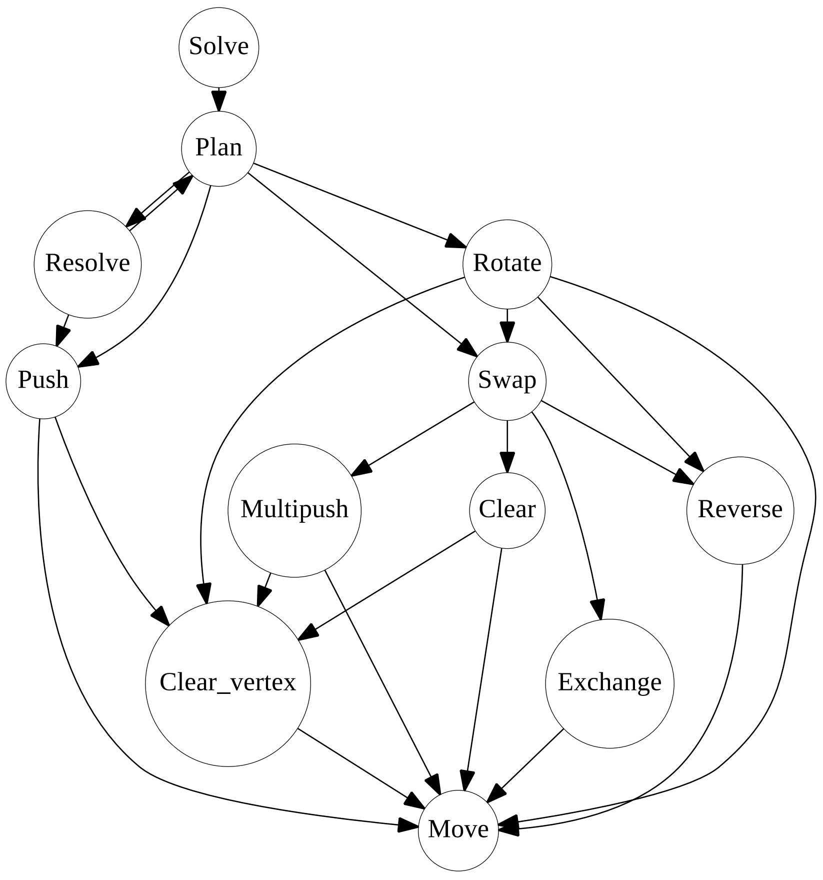


Figure 3.2: Demonstration how the methods in *Solve* are call each other.

3.4.1 Plan

The *Plan* algorithm (Algorithm 7) finds for the provided agent A the shortest path from the agent's current location $c(A)$ to its goal position $t(A)$ while avoiding other agents if that is possible (line 1) and pushes the agent along it.

In the while cycle (line 2) it is expected that the agent A ends at the goal position. This is done by three algorithms (*Push*, *Rotate*, *Swap*) which deal with specific situations which can occur during a planning. The next node on the path $\mathcal{P}$ is extracted (line 3) at first and removed from the path (line 4). If the v is already contained in a list of already visited nodes $\mathcal{Q}$ then the agents are forming a circle and thus the *Rotate* algorithm is called (line 5). $\mathcal{Q}$ is passed as an argument and starts as an empty list if passed from Solve.

If v is not contained in $\mathcal{Q}$ then the agent is pushed to the node. This is done by calling *Push* algorithm (line 8). *Push* does not succeed because another agent (the blocking agent) is located at v. The current agent and the blocking one must swap positions and thus *Swap* algorithm is called (line 9). Note that $a(v)$ returns the agent currently located at v. If *Swap* does not succeeded then it is not possible for the agent A reach $t(A)$ and task has no solution and *Plan* returns false in this case.

After A reached v the v is added to $\mathcal{Q}$ (line 14). When A is located at $t(A)$ then A is added to the set of already planned agents $\mathcal{F}$ (line 16) and *Resolve* is called (line 17). This returns already planned agents back to their positions.

Algorithm 7 Plan($\Pi, \mathcal{G}, \mathcal{A}, A, \mathcal{F}, \mathcal{Q}$)

```
1:  $\mathcal{P} \leftarrow$ shortest path in $\mathcal{G}$ from $c(A)$ to $t(A)$
2:  while $c(A) \neq t(A)$ do
3:      $v \leftarrow$ first node in $\mathcal{P}$
4:      $\mathcal{P} \leftarrow \mathcal{P} \setminus [v]$
5:      if $v \in q$ then
6:          Rotate($\Pi, \mathcal{G}, q, v$)
7:      else
8:          if Push($\Pi, \mathcal{G}, A, v, c[\mathcal{F}]$) = false then
9:              if Swap($\Pi, \mathcal{G}, A, a[v]$) = false then
10:                 return false
11:             end if
12:         end if
13:     end if
14:     $\mathcal{Q} \leftarrow \mathcal{Q} + [v]$
15: end while
16: $\mathcal{F} \leftarrow \mathcal{F} \cup \{a\}$
17: return Resolve($\Pi, \mathcal{G}, \mathcal{A}, g, \mathcal{F}$)
```

3.4.2 Push

The *Push* algorithm (Algorithm 8) moves the agent A to node v if the node is empty (line 7). But before it it is checked if this is possible (line 1). If not, then it means some other agent is occupying v. The algorithm *Clear_vertex* tries to empty v, i.e. to move the agent from it (line 3).

The argument $\mathcal{V}$ is filled from *Plan* as $c[\mathcal{F}]$ which means a list of nodes where already planned agents currently stand. The current node from agent A (marked as $c(A)$) is added to $\mathcal{V}$ and marked as $\mathcal{V}'$ (line 2). This is utilized in *Clear_vertex* algorithm which works with a set of forbidden nodes, i.e. nodes which cannot be used for moving an agent from vertex v. This prevents the creation of cycles. The algorithm *Move* is called (line 7) and A was successfully moved so the algorithm returns true (line 8).

Algorithm 8 Push($\Pi, \mathcal{G}, A, v, \mathcal{V}$)

```
1: if $a(v) \neq \emptyset$ then
2:     $\mathcal{V}' \leftarrow \mathcal{V} \cup \{c(A)\}$
3:     if Clear_vertex($\Pi, \mathcal{G}, v, \mathcal{V}'$) = false then
4:         return false
5:     end if
6: end if
7: Move($\Pi, A, v$)
8: return true
```

3.4.3 Move

The *Move* algorithm (Algorithm 9) just moves an agent A from its current node $c(A)$ to the node v. When this is done A treated as located at node v from now.

A move is created at first (line 1) and then it is appended as the last move in the list of done moves Π (line 2). Note that this action cannot fail because node v is empty and a just moves to the node. That is the reason why *Move* does not return true or false.

Algorithm 9 Move(Π, A, v)

```
1: $\pi \leftarrow (A, c(A), v)$
2: $\Pi \leftarrow \Pi + [\pi]$
```

3.4.4 Clear_vertex

The *Clear_vertex* algorithm (Algorithm 10) is called when the node v contains some agent and this agent is asked to leave the node. The main idea is that a free node somewhere in a graph $\mathcal{G} \setminus \mathcal{V}$ is found at first. $\mathcal{V}$ is a list of blocked nodes which cannot be used for moving agent from node v. Then all agents on a path from v to u are moved only one node ahead. This causes v to be free.

A distance array from node v to all other nodes is found at first (line 1). This array is sorted by value (line 2) and the nodes are tested one by one if it is possible to move agents there (lines 3). If the node u is not empty then next one is tested (line 4 to 20).

If the node u is empty then a path is extracted from u to v (line 7). If the path is not empty then the first element is extracted (line 9) and it is removed from $\mathcal{P}$ (line 10). Now the elements in $\mathcal{P}$ are processed sequentially starting with element after u until v (line 11). If the node x contains an agent then this agent is moved to x_{prev} (line 14) and the next node is checked. At the end *Clear_vertex* returns true because it has succeeded in clearing the node v.

When all nodes are checked and either no path to them is found, or they are not empty, then algorithm returns false (line 21). Note that it is possible that the path is empty since the search is done in $\mathcal{G} \setminus \mathcal{V}$.

Algorithm 10 Clear_vertex($\Pi, \mathcal{G}, v, \mathcal{V}$)

```
1:  $\mathcal{D} \leftarrow$ array of distances to all nodes in $\mathcal{G} \setminus \mathcal{V}$
2:  $\mathcal{D} \leftarrow$ sorted $\mathcal{D}$ according to the distance to each node
3:  for all $u \in \mathcal{D}$ do
4:      if $a(u) \neq \emptyset$ then
5:          continue
6:      end if
7:      $\mathcal{P} \leftarrow$ path from $u$ to $v$
8:      if $\exists x : x \in \mathcal{P}$ then
9:          $x_{prev} \leftarrow u$
10:         $\mathcal{P} \leftarrow \mathcal{P} \setminus [u]$
11:         for all $x \in \mathcal{P}$ do
12:             $B \leftarrow a(x)$
13:             if $B \neq \emptyset$ then
14:                 Move($\Pi, B, x_{prev}$)
15:             end if
16:             $x_{prev} \leftarrow x$
17:         end for
18:         return true
19:     end if
20: end for
21: return false
```

3.4.5 Swap

The algorithm *Swap* (Algorithm 11) swaps positions of two agents R and S. This is only possible on a node with a degree three or higher. The algorithm selects a node with such property (line 2) and tries to swap agents using it. A local list of done moves in this current attempt Π' is created because not all attempts are successful (line 3).

The *Multipush* algorithm is called at first(line 4). If *Multipush* succeeds then the algorithm *Clear* is called (line 5). If *Clear* managed to clear v then local moves are added to the list of done moves (line 6) and both agents are ready to be swapped. This is done in method exchange (line 7). After exchange it is needed to move all agents back where they were before the swap which is done in reverse (line 8). Swap succeeded and true is returned (line 9).

If any operation fails then the state of the graph $\mathcal{G}$ must be restored. Π' carries information which agents moved where. The algorithm *Restore* works same in the same way as reverse but without swapping roles of R and S (line 12).

It can happen that no such vertex exists or no vertex for is suitable for this type of operation. The algorithm returns false (line 14) in these cases.

Algorithm 11 Swap($\Pi, \mathcal{G}, R, S$)

```
1:  $\mathcal{D} \leftarrow \{$vertex $x \in \mathcal{G} \mid$ degree$(x) \geq 3\}$
2:  **for all** vertex $v \in \mathcal{D}$ **do**
3:      $\Pi' \leftarrow$ empty list [ ]
4:      **if** Multipush($\Pi', \mathcal{G}, R, S, v$) = true **then**
5:          **if** Clear($\Pi', \mathcal{G}, R, S, v$) = true **then**
6:              $\Pi \leftarrow \Pi \cup \Pi'$
7:              Exchange($\Pi, \mathcal{G}, R, S, v$)
8:              Reverse($\Pi, \Pi'_{r/s}, R, S$)
9:              **return** true
10:         **end if**
11:     **end if**
12:     Restore($\Pi'$)
13: **end for**
14: **return** false
```

3.4.6 Multipush

The algorithm *Multipush* (Algorithm 12) basically drags two agents A and B to a node v. This node has a degree more than 3 and it is passed as a parameter. Note that A and B are located at nodes which share an edge in $\mathcal{G}$ because B blocks A from moving to a desired goal node. The difference between *Multipush* and *Push* is that *Push* moves only one agent but *Multipush* simultaneously moves two agents.

It needs to be decided which agent is closer to v at first (lines 1 to 11). Path from agent A current node to v is found (A_{path}). This is done for B as well. If A_{path} is shorter then B_{path} then A is marked as R, B as S and A_{path} is saved to $\mathcal{P}$. The reason behind this is that the agent R is expected to be located at node v when moved to it. Note that R is not always blocking agent B because v can be on the A's side.

Path $\mathcal{P}$ can be empty. The algorithm returns false in that case (line 13). If the path is not empty then the first element is removed because R is standing at it (line 15).

In for cycle starting on line 16 both agents are moved along $\mathcal{P}$ until R reaches v. If some other agent is encountered during the process then it is asked through *Clear_vertex* to move away. In this case the list of blocked nodes consists only from a current nodes where R and S are, i.e. already planned agents can be moved because when swap ends then all moves are reversed and already planned agents are returned back.

At the end the algorithm returns true because R is located at v and S next to it (line 28).

Algorithm 12 Multipush($\Pi, \mathcal{G}, A, B, v$)

```
1:  $A_{path} \leftarrow$ path from $c(A)$ to $v$
2:  $B_{path} \leftarrow$ path from $c(B)$ to $v$
3:  if $A_{path} < B_{path}$ then
4:      $R \leftarrow A$
5:      $\mathcal{P} \leftarrow A_{path}$
6:      $S \leftarrow B$
7:  else
8:      $R \leftarrow B$
9:      $\mathcal{P} \leftarrow B_{path}$
10:     $S \leftarrow A$
11: end if
12: if $\neg\exists x : x \in \mathcal{P}$ then
13:     return false
14: end if
15: $\mathcal{P} \leftarrow \mathcal{P} \setminus [c(R)]$
16: for all $x \in \mathcal{P}$ do
17:     $r_c \leftarrow c(R)$
18:     $s_c \leftarrow c(S)$
19:     if $a(x) \neq \emptyset$ then
20:         $U' \leftarrow \{r_c, s_c\}$
21:         if Clear_vertex($\Pi, \mathcal{G}, x, \mathcal{U}'$) $=$ false then
22:             return false
23:         end if
24:     end if
25:     Move($\Pi, R, x$)
26:     Move($\Pi, S, r_c$)
27: end for
28: return true
```

3.4.7 Clear

The *Clear* algorithm (alg. 13) is called by *Swap* when at least two free nodes around a node v are required. *Clear* is prevented from moving agents R, located at node v, and S which is in some neighbourhood node of v.

Clear gets two agents R' and S' as a parameter and it is decided which one from them is located at node v at first(lines 1 to 9) and this agent is marked as R. The other one is marked as S and its node as v'. It is checked how many free nodes are in the neighbourhood of node v (line 10) and all those free nodes are added to the set $\mathcal{N}$. If $\mathcal{N}$ has at least two elements then it is sufficient and the algorithm returns true (line 12).

If $|\mathcal{N}|$ is lower or equal to 1 then Clear calls clear_vertex on a neighbourhood node avoiding the node where S is located (line 14). Each cleared node is added to $\mathcal{N}$ (line 19). If it happens that the size of $|\mathcal{N}| \geq 1$ then two nodes are found and true is returned because the other node was just cleared.

If $|\mathcal{N}| \leq 1$ after all attempts then algorithm returns false (line 22). The original algorithm tried to call *Push* and *Clear* if $|E| = 1$. This situation rarely happens and this part was not implemented.

Algorithm 13 Clear($\Pi, \mathcal{G}, R', S', v$)

```
 1: if $r' = a(v)$ then
 2:     $R \leftarrow R'$
 3:     $S \leftarrow S'$
 4:     $v' \leftarrow c(S')$
 5: else
 6:     $R \leftarrow S'$
 7:     $S \leftarrow R'$
 8:     $v' \leftarrow c(R')$
 9: end if
10: $\mathcal{N} \leftarrow$ free $neighbours(v)$
11: if $|\mathcal{N}| \geq 2$ then
12:     return true
13: end if
14: for all $n \in neighbours(v) \setminus (\mathcal{N} \cup \{v'\})$ do
15:     if Clear_vertex($\Pi, \mathcal{G}, n, \mathcal{N} \cup \{v, v'\}$) = true then
16:         if $|\mathcal{N}| \geq 1$ then
17:             return true
18:         end if
19:         $\mathcal{N} \leftarrow \mathcal{N} \cup \{n\}$
20:     end if
21: end for
22: return false
```

3.4.8 Exchange

The *Exchange* algoritm (Algorithm 14) exchanges position of two agents. One agent R is located at node v which has degree at least three. The other agent S is located at node s_c which has an connection with v. At least two unoccupied nodes are needed to guarantee that exchange is possible. R moves away from v to free neighbour, S moves through v to other free neighbour, R moves through v to s_c and S moves to v.

Exchange detects which agent from R' and S' (passed as an argument) is located where at start (lines 1 to 7) and finds two unoccupied nodes. Note that those nodes exist because they were cleared earlier by *Clear* algorithm. Exchange of positions of agents commences (line 10). For example line 10 means that algorithm moves R to the first free node v_1. This algorithm always succeeds.

Algorithm 14 Exchange($\Pi, \mathcal{G}, R', S', v$)

```
1: if $R' = a(v)$ then
2:     $R \leftarrow R'$
3:     $S \leftarrow S'$
4: else
5:     $R \leftarrow S'$
6:     $S \leftarrow R'$
7: end if
8: $(v_1, v_2) \leftarrow$ two unoccupied neghbours of $v$
9: $s_c \leftarrow c(S)$
10: Move($\Pi, R, v_1$)
11: Move($\Pi, S, v$)
12: Move($\Pi, S, v_2$)
13: Move($\Pi, R, v$)
14: Move($\Pi, R, s_c$)
15: Move($\Pi, S, v$)
```

3.4.9 Reverse

The algorithm *Reverse* (Algorithm 15) appends moves from $\Pi'_{r/s}$ to Π in reverse order. Each move π from the $\Pi'_{r/s}$ is reversed, i.e. the start node is now end node and vice versa. Then the order is reversed, i.e. the last done move is now the first in the list. Moreover moves done by an agent R are saved as done by agent S and vice versa (line 3 to 6). Other moves remains same but are only inverted (line 8). These all newly created moves π are added to the list of already done moves Π (line 11).

Algorithm 15 Reverse($\Pi, \Pi'_{r/s}, R, S$)

```
1: $\Pi_{inv} \leftarrow$ empty list [ ]
2: for all $(A, v_{start}, v_{end}) \in \Pi'_{r/s}$ do
3:     if $A = R$ then
4:         $\Pi_{inv} \leftarrow [(R, v_{end}, v_{start})] + \Pi_{inv}$
5:     else if $A = S$ then
6:         $\Pi_{inv} \leftarrow [(S, v_{end}, v_{start})] + \Pi_{inv}$
7:     else
8:         $\Pi_{inv} \leftarrow [(A, v_{end}, v_{start})] + \Pi_{inv}$
9:     end if
10: end for
11: $\Pi \leftarrow \Pi + \Pi_{inv}$
```

3.4.10 Rotate

The algorithm *Rotate* (Algorithm 16) is called when *Plan* found out that node on the agent's path is already visited by that agent. This means that nodes in the list of visited nodes $\mathcal{Q}$ form a cycle starting with node v. The cycle is extracted from $\mathcal{Q}$ and agents are moved forward.

The original algorithm considered also the situation when a circle had no free node. This is not important for this task and this is avoided. In such situation the algorithm returns false.

$\mathcal{Q}$ is divided into two parts at start. The first part is $\mathcal{D}$ which is from tail of $\mathcal{Q}$ until v but not including it (line 1). These are the nodes which form a circle. The second part is $\mathcal{Q}$ which is what is left after removing $\mathcal{D}$ from $\mathcal{Q}$ (line 2). Now the algorithm goes through all nodes in $\mathcal{D}$ (lines starting at 4). Each node is checked if it contains an agent (line 5). If the node contains an agent then the node is pushed to list of tested nodes $\mathcal{H}$ (line 13), which is initialized as a empty list (line 3).

If the node is free then all agents located at nodes in $\mathcal{H}$ are pushed one step ahead in $\mathcal{D}$ (lines 7 to 10). On line 6 the current element is saved. This is a destination node where the agent from $\mathcal{H}$ is moved (line 8). This means rotate succeeds and true is returned (line 11).

If the circle contains only nodes with agents then false is returned (line 15).

Algorithm 16 Rotate($\Pi, \mathcal{G}, \mathcal{Q}, v$)

```
1:  $\mathcal{D} \leftarrow$ the tail of $\mathcal{Q}$ starting from (including) $v$
2:  $\mathcal{Q} \leftarrow$ the head of $\mathcal{Q}$ up to (not including) $v$
3:  $\mathcal{H} \leftarrow$ empty list [ ]
4:  for all vertices $v' \in \mathcal{D}$ do
5:      if $a(v') = \emptyset$ then
6:          $w \leftarrow v'$
7:          for all $h \in \mathcal{H}$ do
8:              Move($\Pi, h, w$)
9:              $w \leftarrow h$
10:         end for
11:         return true
12:     end if
13:     $\mathcal{H} \leftarrow [v'] + \mathcal{H}$
14: end for
15: return false
```

3.4.11 Resolve

It is not normally possible to move already planned agents $\mathcal{F}$ but some algorithms (for example *Clear_vertex*) can push them out of the way. When an agent reach its goal position then all these agents must be returned back. That is when *Resolve* algorithm (Algorithm 17) do its job. *Resolve* utilize a list of visited nodes $\mathcal{Q}$. Each node is tested whether it is a goal position an agent A from $\mathcal{F}$. If it is that the case then A is returned back.

Resolve operates until $\mathcal{Q}$ contains a node (line 1). The last node in $\mathcal{Q}$ is extracted (line 2). An agent R which has a goal position at v is extracted (line 3). If no such agent exists then v is removed from $\mathcal{Q}$ (line 10).

If R exists then it is tested whether it is located at its goal position (line 4). If not then *Push* tries to move him back to its position. The agent at this point is at most one node away from it. *Push* can only fail if another agent S is at R's goal position. S is extracted (line 6) and planned using *Plan* (line 7). Note that S cannot be at its goal position because that position already belongs to R.

Resolve can call *Plan* at this situation (line 7). And this is in fact a cyclic calling because *Plan* calls always *Resolve*. The cycle is actually guaranteed to end because at most k agents are out of their positions and all were planned before thus the solution exists.

Algorithm 17 Resolve($\Pi, \mathcal{G}, \mathcal{A}, \mathcal{Q}, \mathcal{F}$)

```
1:  while $|\mathcal{Q}| > 0$ do
2:      $v \leftarrow$ the last vertex on $\mathcal{Q}$
3:      $R \leftarrow t(v)$
4:      if $R \neq \emptyset \wedge R \in \mathcal{F} \wedge c(R) \neq t(R)$ then
5:          if Push($\Pi, \mathcal{G}, R, t(R), c[\mathcal{F}]$) = false then
6:              $S \leftarrow a(t(R))$
7:              return Plan($\Pi, \mathcal{G}, \mathcal{A}, S, \mathcal{F}, \mathcal{Q}$)
8:          end if
9:      end if
10:     $\mathcal{Q} \leftarrow \mathcal{Q} \setminus [v]$
11: end while
12: return true
```

3.5 Smooth

All agents are planned at this point. A problem is that the algorithm *Solve* (section 3.4) produces redundant moves. These moves can be removed using the *Smooth* algorithm.

Smooth operates with pointers to other moves π in the list of done moves Π. A redundant sequence is a sequence of moves that moves an agent to a vertex that it has visited before and no other agent has visited that vertex in between [3].

An example of a redundant sequence consisting of two moves Move2 and Move3:

- Move1: Agent 1 moves from a node 1 to a node 2.
- Move2: Agent 1 moves from the node 2 to a node 3.
- Move3: Agent 1 moves from the node 3 to the node 2.

- Move4: Agent 1 moves from the node 2 to the node 3.

The moves Move2 and Move3 are redundant because no other agent visited the node 2 in between.

To solve this issue a double linked list of nodes is created [3]. Each move π consists of an agent A and a node u. It is needed to known which moves, if any, A did before π and which move follows after π. The same applies for u.

Let's define pointers to other moves. Let $N_A(v)$ (next agent) return a next move of the agent A. Let $N_V(v)$ (next vertex) return a next move where the vertex v is noted as the end vertex. A redundant sequence can be written as $\pi, N_A(\pi), N_V(\pi)$ [3].

When this redundant sequence is deleted it is possible that another redundant sequence appears. To update the double linked list additional two pointers are introduced. Let $P_A(v)$ (previous agent) return a previous move of the A and let $P_V(v)$ (previous vertex) return a previous move where the vertex v is noted as the end vertex. Figure 3.3 shows pointers of a single move [3].

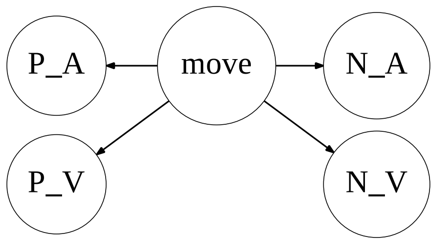


Figure 3.3: The visualization of a move with its four pointers N_A (next agent), N_V (next vertex), P_A (previous agent) and P_V(previous vertex). Each pointer points to another move.

It is now possible to update the list of done moves efficiently but this assumes that pointers are already constructed. Two extra types of pointers are introduced. Let $L_A(a)$ (last agent) return the last move ($\pi = (A, u)$) done by an agent A. u is some destination vertex and let $L_V(v)$ (last vertex) return the last move ($\pi = (B, v)$) where the vertex v is noted as the end vertex, i.e. some agent B moved to the vertex v [3].

3.5.1 Smooth algorithm description

The *Smooth* algorithm (Algorithm 18) has two parts. It detects redundancies in the list of done moves Π at first while simultaneously constructing double linked chains of moves (line 1). Then it removes redundancies and possibly detects some additional redundancies in the process (line 2). Finally Π is returned at (line 3). This is the same Π as passed as argument but without redundancies which have been removed on line 2. Both algorithms are now described.

Algorithm 18 Smooth($\mathcal{G}, \Pi$)

```
1: $\mathcal{D} \leftarrow$ Detect_redundancies($\mathcal{G}, \Pi$)
2: Remove_redundancies($\mathcal{D}$)
3: return $\Pi$
```

Algorithm 19 Detect_redundancies($\mathcal{G}, \Pi$)

```
1: $\mathcal{D} \leftarrow$ empty unique list [ ]
2: $L_A \leftarrow$ empty array
3: $L_V \leftarrow$ empty array
4: for all $A \in \mathcal{A}$ do
5:     $L_A(A) \leftarrow$ impossible move
6: end for
7: for all $u \in \mathcal{U}$ do
8:     $L_V(u) \leftarrow$ impossible move
9: end for
10: for all $(A, u) = \pi \in \Pi$ do
11:     $P_A(\pi) \leftarrow L_A(A)$, $N_A(L_A(A)) \leftarrow \pi$
12:     $P_V(\pi) \leftarrow L_V(A)$, $N_V(L_V(A)) \leftarrow \pi$
13:     if $a(L_V(u)) = A$ then
14:         $\mathcal{D} \leftarrow \mathcal{D} + [L_V(u)]$
15:     end if
16:     $L_A(A) \leftarrow \pi, L_V(u) \leftarrow \pi$
17: end for
18: return $\mathcal{D}$
```

3.5.2 Detect redundancies

The algorithm *Detect_redundancies* (Algorithm. 19) detects redundant sequences in Π and initializes double linked chain.

Denote an impossible move be a move of an non-existing agent from a non-existing node to a non-existing node -1. This is going to be used as a stop barrier for pointers. The arrays L_A and L_A are initialized with an impossible move (lines 4 to 9). Then a double linked chain is created on lines 11 to 12.

Let's explain the line 11 little in more details. The first part $P_A(\pi) \leftarrow L_A(A)$ means that the last move of agent A, $L_A(a)$ is now saved as a previous move of agent A of current move π, $P_A(\pi)$, i.e. the current move π has now filled a pointer to the previous move P_A.

The second part $N_A(L_A(A)) \leftarrow \pi$ saves the current move π as a next move of agent A, N_A, to the last move of agent A, L_A, i.e. the next move N_A of the move last move of agent A, L_A, is the move π.

It is checked if π is the last move of a redundant sequence when the double linked list is constructed (line 13). If it is then the first move of the redundant sequence is added to unique list $\mathcal{D}$, which is initialized as empty unique list (line 1). Note that $a(\pi)$ returns the agent A assuming that $\pi = (A, v)$.

Note about the uniqueness of the moves. Let's look back at the example in section 3.5. The moves Move2 and Move4 are unique even though they represent the same action. They are unique because the Move2 is done as second and Move4 is done as fourth. But for example when comparing Move2 and Move2 then they are not unique when considered by the $\mathcal{D}$.

Algorithm 20 Remove_redundancies($\mathcal{D}$)

```
1:  while $|\mathcal{D}| > 0$ do
2:      $\pi \leftarrow$ retrieve and remove first element from $\mathcal{D}$
3:      $\mathcal{D} \leftarrow \mathcal{D} \setminus [\pi]$
4:      $\pi' \leftarrow \pi$
5:      $\pi_L \leftarrow N_A(\pi)$
6:      while $\pi_L \neq \pi'$ do
7:          $\pi' \leftarrow N_A(\pi')$
8:          $\pi'.mark()$
9:          $P_A(N_A(\pi')) \leftarrow P_A(\pi'), N_A(P_A(\pi')) \leftarrow N_A(\pi')$
10:         $P_V(N_V(\pi')) \leftarrow P_V(\pi'), N_V(P_V(\pi')) \leftarrow N_V(\pi')$
11:         if $a(P_V(\pi') = a(N_V(\pi'))$ then
12:             $\mathcal{D} \leftarrow \mathcal{D} + [P_V(\pi')]$
13:         end if
14:     end while
15: end while
16: for all $\pi \in \Pi$ do
17:     if $\pi.isMarked()$ then
18:         $\Pi \leftarrow \Pi \setminus [\pi]$
19:     end if
20: end for
```

3.5.3 Remove redundancies

The algorithm *Remove_redundancies* (Algorithm 20) receives a list of detected redundancies $\mathcal{D}$ as a paramater. This list is iterated over again until the list is empty. An element is chosen from $\mathcal{D}$ and then the sequence is deleted. It is checked whether some new redundant sequence appeared. If yes then it is added to $\mathcal{D}$ and process is repeated. This procedure is finite because Π has a finite number of steps which can contain a finite number of redundancies. It is possible that the number of redundancies can be drastically temporary increased.

Remove_redundancies removes all redundancies in while cycle (lines 1 to 15). The first element is retrieved from $\mathcal{D}$ and removed from it (lines 2 to 3). A redundant sequence starts with π and ends $N_A(\pi)$. $N_A(\pi)$ is removed as well.

The redundant sequence is removed (lines 6 to 14). When a move is removed then it is marked as removed (line 8) and all four pointed moves, P_A, P_V, N_A, N_V, have some of their pointers updated (lines 9 to 10). After that it is checked whethera new redundancy appeared (line 11). If it is the case then it is added to the end of $\mathcal{D}$ (line 12). Note that at this moment π' is pointing to some other moves but nothing points back at it. Finally the entire Π is checked and redundant moves are removed (lines 16 to 20).

3.6 Example solution

Let's take a task which was already mentioned in Fig. 3.1. Assume that the task has only two agents $R0$ and $R1$ at their starting locations. $R1$ is already at its goal location. The doted line marks the goal position of $R0$. $R0$ needs to move $R1$ from its position.

Solution of this situation is displayed in Fig. 3.4. No biconnected components are detected, only a node with degree three or higher. The problem thus contains a single subproblem $\mathcal{C}_1$ which is the node where $R1$ is located. No subproblems can be merged.

Assignment of agents assigns both agents to $\mathcal{C}_1$. $R0$ is assigned because when the edge between them is tested then at most 2 agents outside of the subproblem can be assigned to it. $R1$ is assigned to $\mathcal{C}_1$ as well. When another edge is tested which is different from the edge with $R0$ then the condition $m' \geq 1 \wedge m' < m$ is satisfied because $m' = 1 \wedge m = 2$.

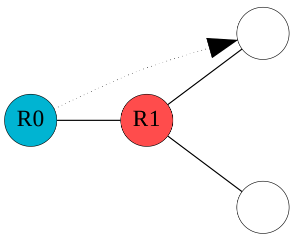
There is no priority between subproblems to be decided. Agents can be planned in any order.

Solve produces the moves. In this case $R0$ is planned first (Fig. 3.4a) and $R0$ pushes $R1$ away (Fig. 3.4b). Unfortunately, $R1$ chooses $R0$'s destination. $R0$ is pushed along the path to its goal and *Swap* is called because *Push* does not succeed (Fig. 3.4c). Both agents performs *Multipush*. $R0$ is chosen as an agent at node with degree three. *Clear* successfully cleared two neighbourhood nodes (they were already free).

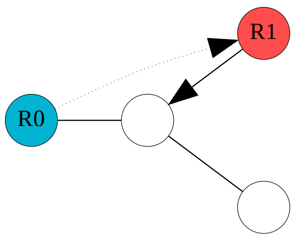
Exchange exchanges both agents positions (Figs. from 3.4d to 3.4i). It is verified that both agents are at their goal positions (3.4i).

It can be seen that moves from 3.4a to 3.4e are part of redundant sequence because 3.4a is the same as 3.4d. *Smooth* detects that $R0$ returns back to its starting position using moves 3.4a (initial move), 3.4c and 3.4d at first. Move 3.4b does not concern $R0$ because it is done by $R1$. Move 3.4a is added to $\mathcal{D}$.

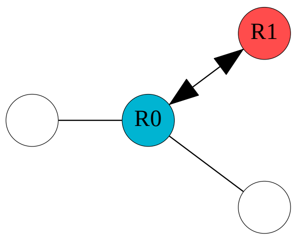
When the move 3.4c is removed and pointers are redirected it is detected that another redundant sequence appears because the condition $a(P_V(\pi') = a(N_V(\pi'))$ is satisfied. These are 3.4a (initial move), 3.4b and 3.4e. So the move 3.4a is added to $\mathcal{D}$. Then move 3.4d is removed because *Smooth* is still removing the first redundant sequence. Now Smooth starts removing the second redundant sequence (3.4a, 3.4b, 3.4e) by removing 3.4b and 3.4e. No new redundant sequences are found and algorithm returns the solution containing move 3.4a and moves from 3.4f to 3.4i. The complete solution can be seen in 3.5.



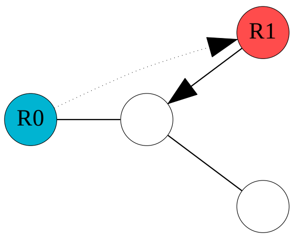
(a) : *R*0's destination is marked by the arrow.



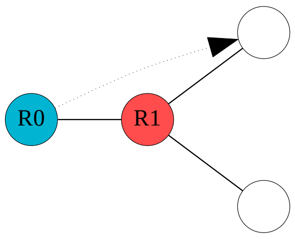
(b) : *R*1 is pushed away.



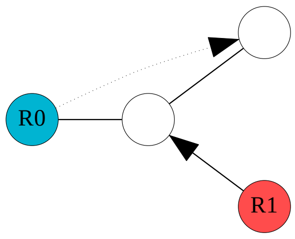
(c) : *R*0 advances.



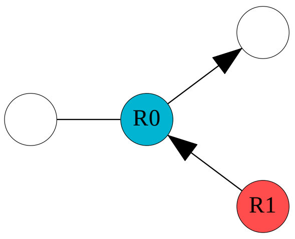
(d) : The *Swap* is called. *R*0 empties the node for exchange.



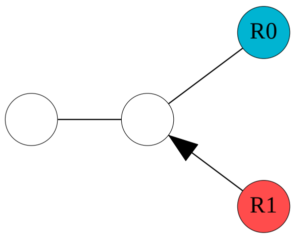
(e) : *R*1 returns to its starting position.



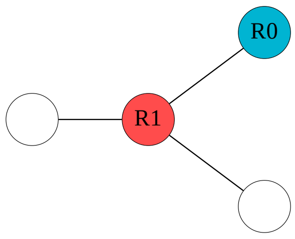
(f) : *R*1 moves to another node.



(g) : *R*0 advances.

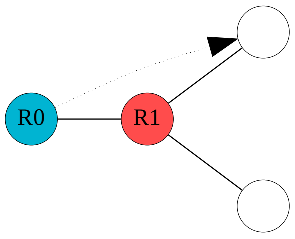


(h) : *R*0 finishes at goal position.

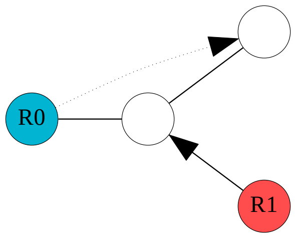


(i) : *R*1 returns to starting position.

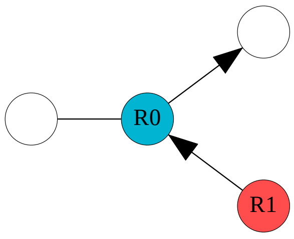
Figure 3.4: The solution for a map called vertex produced by the *Smooth* algorithm. It can be seen that the moves 3.4a to 3.4d are redundant because 3.4a is same as 3.4e.



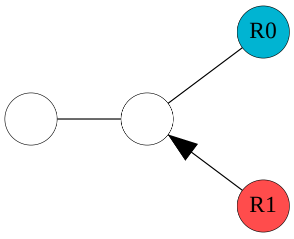
(a) : $R0$'s destination is marked by the arrow.



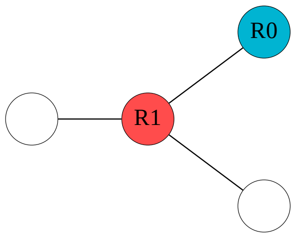
(b) : $R1$ moves to another node.



(c) : $R0$ advances.



(d) : $R0$ finishes at goal position.



(e) : $R1$ returns to starting position.

Figure 3.5: The smoothed solution for a map called vertex produced by the *Smooth* algorithm on the solution from the *Solve* algorithm. Two redundant sequences have been removed. The original solution can be found in figure 3.4.

Chapter 4

Push and rotate implementation

The theoretical description of *Push and rotate* is done in the chapter 3. It is appropriate to go more into detail now and to explain the implementation and how it was realized. The extension for two types of agents is located at the end of this chapter.

The algorithm was implemented in a programming language C++ [11]. This language allows using pointers and managing them. A pointer is variable storing an address of another variable. This allows many parts of programs to work with the structure which is pointed at without the need to copy it.

Some of the problems from the theoretical chapter can be reformulated or already are common solved problems. Usual approaches are taken in that case the or even, if it is possible, algorithms from public libraries are considered. A great example of this is the Boost library. Few of its graph algorithms are used in this thesis. The algorithm is noted and shortly explained in that case.

The standard template library (STL) is used when possible [12]. When a list (for example of nodes) is mentioned then it is a structure from STL.

A vector is used storing nodes $\mathcal{U}$ and agents $\mathcal{A}$. A list is used for storing the list of done moves. A set is utilized in this thesis on few places, for example when all planned agents are returned back to their goal position. A map is used when two subproblems are merged together. A pair is used for storing priority relation between two subproblems.

4.1 Agents

Each agent is represented as an object which knows the agent's id, the initial node id, the goal node id and if it carries a rack. These are expected parameters for creating an agent.

An agent has the private id which allows that the provided id (by a user for example) can have spaces between them, i.e. it is not expected that 10 agents in a task have ids from 0 to 9. This is useful for creating and removing tasks. Another agents' variable is *planning_priority_* which starts as -1 and it is assigned during the assigned agents to subproblems 3.2.

4.2 Nodes

A node is represented as an object which knows the node's id, the position (both x and y), a pointer to an agent currently located at the node, the subproblem id, if it contains a rack and pointers to all its neighbourhood nodes. Each node has the private id for same reasons like the agents.

Also a node has a pointer to an agent which has the node as a goal position. It is used for example in algorithm *Detect priorities* 3 on line 16.

4.3 Arena

The arena is an object which is created at the start and it us utilized by all algorithms. It knows all agents and all nodes. All agents are stored as a vector and nodes as well. This allows quick reference using their private id.

It contains a special variables, for example *show_info_about_running_*. If it is set to true then algorithm returns info about the running and about time elapsed. It contains a pointer to initialized *AStar* algorithm because it is called many times in *Solve*.

The arena is added as a parameter to all algorithms and it is noted as $\mathcal{G}$ because it represents mostly the graph.

4.4 Division into subproblems

The implementation details of the *Push and rotate* algorithm are explained one by one now. The advantage of this step is that it is only needed to perform once for each map unless some nodes or edges are removed or added. All other steps depend on all agents' starting and goal positions.

The biconnected components are found at first. The finding is done using the algorithm *Biconnected_components_and_articulation_points* from the Boost library [13]. This algorithm returns also subproblems of size two. These are not considered biconnected and are removed from the set. The articulation points are not used at all because *Push and rotate* has no use for nodes in a map which belongs simultaneously to two biconnected components.

Each subproblem is represented as a list of nodes which belong to it and it is stored as an element in the list of subproblems $\mathcal{C}$. Nodes of degree three or higher are added to main list as well. Then all subproblems receive a private id and the main list is replaced by a map where the id of subproblem is used as reference.

Subproblems are merged now. The algorithm chooses a first node in the first subproblem and starts *Dijkstra*'s algorithm [14]. *Dijkstra* searches shortest paths from a single node to all other nodes in the graph using the value of the edge between the nodes. For the search the edge between the nodes in the same subproblem has a value 0 and the remaining edges have value 1. This solves the issue with finding the closest node in each subproblem.

All distances provided by *Dijkstra*'s algorithm are sorted then and the testing for merging starts from the closest node. This prevents adding more paths more than once. If the node belongs to another subproblem and the condition for merging is satisfied then those subproblems are merged. When two subproblems $\mathcal{C}_1, \mathcal{C}_2$ are merged then all nodes of $\mathcal{C}_2$ are appended to the end of the list of $\mathcal{C}_1$. $\mathcal{C}_2$ is removed from the map and no longer exists. A path (a list of nodes) between $\mathcal{C}_1$ and $\mathcal{C}_2$ is added to the end of the list of $\mathcal{C}_1$, after nodes from $\mathcal{C}_2$. The next subproblem is processed when all nodes have been tested for merging. The algorithm restarts after all subproblems are checked, i.e. it starts from the first subproblem if any two subproblems were merged in the previous iteration.

4.5 Assigning agents to subproblems

The next task is to found how much free space is reachable from a specific node u when avoiding a set of nodes. The set is usually a single node or a subproblem without a node in it and the actual distance is not important even if the edges have different weights. The algorithm *breath first search* (*BFS* [15]) is used for computation of m' and m''.

BFS has two lists, an open list and a closed list. It starts the search from a single node it adds it to closed list and then it adds all nodes in node's neighbourhood into the open list. Then it chooses another node from the open list. Reachable nodes are stored in the open list and then processed if they are empty.

At some point in the algorithm (2 at line 9) it is asked to assign the first $m' - 1$ agents away from node v. It was not clear from the original paper what exactly this meant but *BFS* was used and the node edge between u and v is ignored and at most $m' - 1$ agents are assigned. Note that it is not possible to enter the subproblem containing v via another edge because the one between u and v is the only way.

4.6 Priority relation between subproblems

A relation $\mathcal{C}_i \prec \mathcal{C}_j$ is described as a pair (STL) where the first element is the id of $\mathcal{C}_i$ and the second element is the id of $\mathcal{C}_j$. That means the list of relations is represented as a list of pairs.

For each pair of subproblems $\mathcal{C}_i$ and $\mathcal{C}_j$ a path is found using the *AStar* algorithm [16]. *AStar* is initialized using the nodes and edges from map. A start node is the first node $\mathcal{C}_i$ and an end node is also the first node in $\mathcal{C}_j$. *AStar* uses a heuristics for finding the closest path to the destination node which depends on node's position. That's why all nodes have coordinates.

When the path from $\mathcal{C}_i$ to $\mathcal{C}_j$ is followed then it is actually tested for both another agent's goal position and $\mathcal{C}_i$'s neighbour simultaneously. The path is not restarted. For the sake of simplicity it is written in the pseudo code (alg. 3 lines 15 to 34).

The ordering is extracted using the *Topological sort* algorithm [10]. A directed graph is constructed from the found priorities and then passed as a parameter. *Topological sort* returns an array of ids of each subproblem where the first element has the highest priority, the second one slightly lower and so on with the last element having the lowest priority.

4.7 Solve

The list of done moves Π is actually a list. The list of already visited nodes $\mathcal{Q}$ is represented as a list of pointers to the specific nodes. The set of already planed agents $\mathcal{F}$ is represented as a map using the agent's private id as a key.

4.7.1 Plan

AStar is used in the same way like in the section 4.6 but in this case the graph $\mathcal{G}$ is created once and utilized every time a path is needed. The creation of $\mathcal{G}$ is done when the map is loaded. The weight of an edge has a default value of 1. If the node contains an agent then the weight of all edges containing this node has weight 10. This force *AStar* to avoid pushing other agents if it is not necessary.

4.7.2 Push

The set of blocked nodes of already planned agents $\mathcal{V}$ is represented as a set for fast searching. $\mathcal{V}$ contains only a nodes' ids.

4.7.3 Clear_vertex

Clear_vertex actually calls *BFS* ([15]) for finding the unoccupied nodes. The search is stop when a free node is found. The parent array is extracted from *BFS* as well. If the unoccupied node does not work then next closest unoccupied node is returned and so on.

4.7.4 Swap

The list of nodes with degree at least three $\mathcal{D}$ is saved during the map initialization and these nodes are tested one by one.

All done moves are saved in Π' (11 on line 3). The graph $\mathcal{G}$ is not copied in any iteration. This is possible because Π' stores where which agents moved and these moves are processed backwards and the agents are restored back to their initial position before *Multipush* is called (line 4). Note that *Restore* is called only when *Swap* did not succeeded.

4.7.5 Multipush

It is decided which agent, A or B, is assigned to variable R at the start of the algorithm 12. Actually the R starts as B and if the path is shorter for A

then it is reversed. This is done because B is expected to be a blocking agent and usually this is true and it is expected to have initialized variables when they are created. The $\mathcal{V}'$ is represented as a set containing ids of blocked nodes (line 20).

4.7.6 Clear

The set of all unoccupied nodes $\mathcal{N}$ of a node v is created just by processing the node's neighbours and adding only if there is no agent at the tested node. $\mathcal{N}$ contains only ids of free nodes. Another set is created and it contains all ids from $\mathcal{N}$ (Algorithm 13 line 15). These two sets exist and are maintained together for the rest of the for cycle.

4.7.7 Exchange

The comparison which agent stands at v is done only by using the agents id (Algorithm 14 line 1). Nothing else is checked. This is sufficient since the agent id is unique. The unoccupied neighbours are constructed once again thus it is possible that they are different from the ones found in *Clear* if the node has more than two free neighbours.

4.7.8 Reverse

The local list of done moves Π' is created using the STL. Π' allows the iteration from the end to the start and each move is replaced using the *Reverse* algorithm (Algorithm 15). Agents are distinguished based on their id only.

4.7.9 Rotate

One of the argument of the Algorithm 16 is a node v which marks the start of the cycle. The implementation actually does not need this at all. If the element exists the Algorithm already saves a special type of a pointer, an iterator, to it. This iterator is passed as an argument to *Rotate* instead of v. This allows the splitting of $\mathcal{Q}$ in c and $\mathcal{Q} \setminus c$.

4.7.10 Resolve

The set of already planned robots $\mathcal{F}$ contains a method *find* which returns an iterator to the element. The iterator points either to the desired element or to an inaccessible element. This is utilized when returning already planned agents back to their start position.

4.8 Smooth

One thing which is not clear is that move π can be marked (line 17). This is thanks to the structure π which has a local variable *removed_from_list_*.

If this *removed_from_list_* is marked (i.e. the variable is set to true) then *Smooth* removes the move at the end during the clean up.

4.9 Move

The impossible move is represented as a regular move with its values set to special values (-1). When a move is created, for example in *Push*, then all its pointers (P_A, P_V, N_A and N_V) point to the impossible move.

The unique move is a move on a specific position in list of done moves Π. Two moves are the same if they both move the same agent from some node u to some node v and are at the same position in the solution. This is solved by using an id which is assigned to all moves when they enter *Smooth*.

Algorithm 21 Plan_with_racks($\Pi, \mathcal{G}, \mathcal{A}, A, \mathcal{F}, \mathcal{Q}$)

```
1: if $A$ has a rack $\wedge$ $c(A)$ has a rack then
2:     $n \leftarrow$ free neighbour of $v$
3:     if Push($\Pi, \mathcal{G}, A, n, c[\mathcal{F}]$) = false then
4:         return false
5:     end if
6: end if
7: $\mathcal{P} \leftarrow$ shortest path in $\mathcal{G}$ from $c(A)$ to $t(A)$
8: while $c(A) \neq t(A)$ do
9:     $v \leftarrow$ first node in $\mathcal{P}$
10:    $\mathcal{P} \leftarrow \mathcal{P} \setminus [v]$
11:    if $v \in q$ then
12:        Rotate($\Pi, \mathcal{G}, q, v$)
13:    else
14:        if Push($\Pi, \mathcal{G}, A, v, c[\mathcal{F}]$) = false then
15:            if Swap($\Pi, \mathcal{G}, A, a[v]$) = false then
16:                if Push_with_racks($\Pi, \mathcal{G}, A, v, c[\mathcal{F}]$) = false then
17:                    if Swap_with_racks($\Pi, \mathcal{G}, A, a[v]$) = false then
18:                        return false
19:                    end if
20:                end if
21:            end if
22:        end if
23:    end if
24:    $\mathcal{Q} \leftarrow \mathcal{Q} + [v]$
25: end while
26: $\mathcal{F} \leftarrow \mathcal{F} \cup \{a\}$
27: return Resolve($\Pi, \mathcal{G}, \mathcal{A}, g, \mathcal{F}$)
```

4.10 Extension for two types of agents

The additional requirements (two types of robots and obstacles) cause the addition of few algorithms mainly into *Solve* because the idea is that an agent not carrying a rack tries to avoid all agents carrying it. If the agents with racks block a path then the agent without a rack is allowed to push them only after exhausting all other options. The algorithms *Division into subproblems*, *Assigning agents to subproblems*, *Priority relation* and *Smooth* remained same. The agents with racks are considered as a part of a subproblem but racks are not considered as a free space during the *Assigning agents to subproblems.*

Plan_with_racks (Algorith 21) is mainly same as *Plan.* If the agent has a rack and starts at a position with a rack then it carries it and it is moved away to some free node (line 1). Another difference is that if *Swap* fails the algorithm calls *Push_with_racks* and *Swap_with_racks* (lines 16 to 17).

If A does not carry a rack then *Push* treats other agents with racks as moving obstacles and automatically fails and false is returned. The *Push_with_racks* algorithm (Algorithm 23) works in a similar way as *Push* but agents without racks are allowed to push agents with racks away. *Push* was changed to take this into an account.

Algorithm 22 Push($\Pi, \mathcal{G}, A, v, \mathcal{V}$)

```
1: if $a(v) \neq \emptyset$ then
2:     if $a(v)$ has a rack $\wedge$ $A$ has no rack then
3:         return false
4:     end if
5:     $\mathcal{V}' \leftarrow \mathcal{V} \cup \{c(A)\}$
6:     if Clear_vertex($\Pi, \mathcal{G}, v, \mathcal{V}'$) = false then
7:         return false
8:     end if
9: end if
10: Move($\Pi, A, v$)
11: return true
```

Algorithm 23 Push_with_racks($\Pi, \mathcal{G}, A, v, \mathcal{V}$)

```
1: if $a(v) \neq \emptyset$ then
2:     $\mathcal{V}' \leftarrow \mathcal{V} \cup \{c(A)\}$
3:     if Clear_vertex_with_racks($\Pi, \mathcal{G}, v, \mathcal{V}'$) = false then
4:         return false
5:     end if
6: end if
7: Move($\Pi, A, v$)
8: return true
```

Clear_vertex_with_racks works in a similar way as *Clear_vertex* but since it has the ability to move agents with racks then the search for nearest free space is done in a graph where the agents with racks do not block the path. The rest remains same.

Swap_with_racks works in the same way as *Swap* but it is allowed to move agents carrying racks, e.g. *Multipush*, *Clear* and *Exchange* can move those agents. The pseudo code is not provided because it is almost the same.

Chapter 5
Experiments

An experimental evaluation can be founds in this chapter. The maps and tasks were taken from the Pathfinding bechmark site [1]. Each task is carried by exactly one agent. The tasks are loaded sequentially. If a task shares a start position with a processed task then this task is not added. The same goes for a goal position. It is possible that a task has a goal position equal to a start position of another task. If an agent carries a rack then it is removed if its goal position already contains a rack.

Three tables for each map are provided. The first one shows general info, such like a number of agents or subproblem sizes, the second one depicts the time taken for each part to complete. Note that the total time is measured over the entire algorithm run, i.e. it includes another calls besides the shown (for example notifying the user about the state of the algorithm). This means that the total time is slightly higher than the sum of all shown measurements. The problem time is the total time minus the time taken for both map loading and division into subproblems. This is done because when the tasks restarts on the same map then these two parts are not needed to be computed again in practise although the restarting is not used in this thesis.

The third one displays solution statistics. The shortest possible solution l_0 is a sum of all agents' shortest paths and each agent does not to consider other agents when reaching its destination, i.e.

$$l_0 = \sum_{A \in \mathcal{A}} \mathit{shortest\_path}\left(S[A], T[A]\right).$$

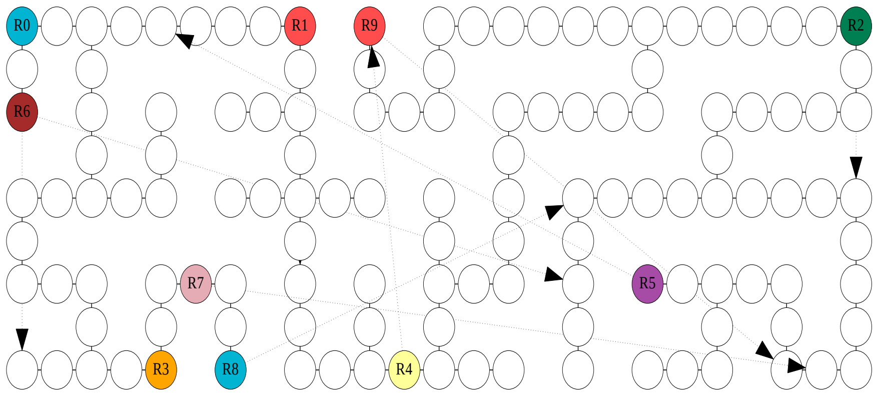
The value l_0 can usually never be reached. The last column shows how many extra moves are needed for solving the task:

$$x_1 = \frac{|\Pi| - l_0}{l_0},$$

where Π is the size of the list of done moves, i.e. the number of steps which solves the problem. The definition of l_0 and x_1 is taken from [3]. The experiments were done on Lenovo IdeaPad Y50-70 Black with Intel Core i7 4720HQ Haswell, RAM 8GB, NVIDIA GeForce GTX 960M 4GB, SSD 256GB and Windows 10 64-bit and the code optimization $-O2$ (Release) was used.

5.1 maze27-11

This map was extracted by cutting out only 27x11 elements from the original map 'maze512-1-101-101.map'. Description of the original map contains also specification of tasks, but in this case the set is not used because many agents are located outside of this small part of map. A special assignment was therefore created for this map. As can be seen in Fig. 5.1. An interesting fact about this map is that there are no biconnected components. This problem is relatively easy with small number of nodes and agents in comparison to the problems presented later in this chapter. On the other hand, this is the only problem the whole map with all agents' positions and their goals can be visualized.



(a) : The map. The goal position of each agent is marked with a dotted line and arrow.

MAP INFO	COUNT
Nodes	129
Subpr. before merging	11
Subpr. after merging	1
Size of subpr. 1	73

(b) : Map properties.

Figure 5.1: `Maze27-11` overview. The map is taken from [1].

The results are presented in Tab.5.2. The shortest possible route is about three times shorter than the solution found. This is because the agents have to call *Swap* many times in order to swap their positions.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	355	0.000
Push and rotate	933	1.628
Smoothed	883	1.487

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	0
Division into subpr.	0
Assign. $\mathcal{A}$ to subpr.	0
Priority relation	0
Solve	0
Smooth	0
Total time	2
Problem time	2

(b) : The duration of each part.

Figure 5.2: The result acquired on the map `maze27-11` using 10 agents.

5.2 maze512-1-0

The `maze512-1-0` can be seen in Fig. 5.3 and this is the map maze27-11 is cut from. The map is hard for *Push and rotate* because there are no biconnected components. All 11 506 are nodes with degree three or higher which are merged one by one together. Another issue is that corridors are exactly one node wide which causes problems with swapping the agents.

Another big issue is that the *Assigning agents to subproblems* algorithm has problems with this type of map. The subproblem has many nodes on the edge of the subproblem and for each of these nodes the number of reachable vertices (*Assigning agents to suproblems*, Algorithm 2 line 3) must be computed and agents around it evaluated even though they end up in same component thanks to the great number of free space.

If the *Assigning agents to subproblems* is solved or turned off (which in this case can be) then computation for 101 agents takes only 70 milliseconds, 499 takes only about 2 seconds and 993 agents takes about 29 seconds. The rest of the results are stored presented in Tab. 5.4 – 5.6

5.3 AR0411SR

Another experiment was done in map from the PC game Baldurs Gate II (published September, 2000) which is scaled to 512x512 [1]. The map can be seen in Fig. 5.7 and it actually has two parts. The first one is basically the entire map and the second one is hidden on the right side where a circle around a circle is located. That's why it has two subproblems even after merging.

The map has sufficient a number of nodes 58 581 and 101, 497 and 982 agents were planned on this map. For 101 agents the duration is only 610 milliseconds if the *map loading* and *division into subproblems* is not taken into an account. If the number of agents increases by 4.92 then the time required to solve the problem increases by 11.62 seconds which is nice given that the agents must plan around each other. Planning 982 agents takes about 33 seconds which is a result given that the path is only longer by 50.21%. On the other hand this map contains plenty of free space and agents can go around each other. All results are stored in Tab. 5.8 – 5.10

5.4 random512-40-0

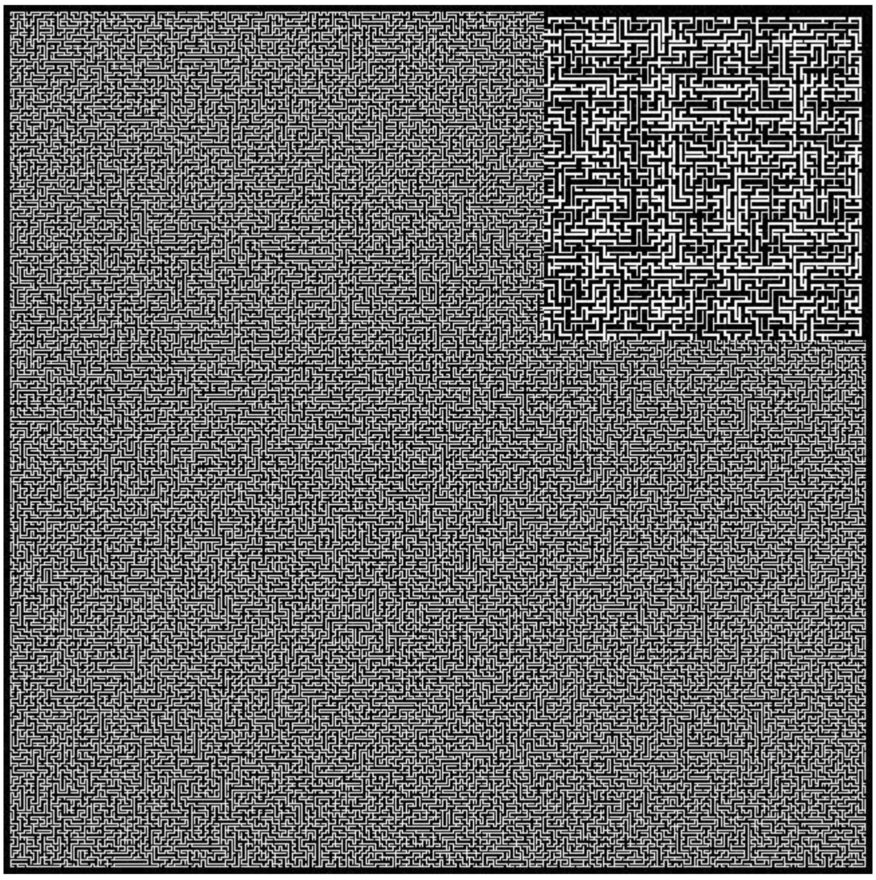
This section presents an experiment on a map named `random512-40-0` which contains two types of obstacles, normal ones (white) and trees (green) and both types must be avoided, see Fig. 5.11. The map has same issues as the map `maze512-1-0`, e.g. it has long corridors which end with a node od degree three or higher and *Assigning agents to subproblems* is too slow. These types of maps are not exactly the target of *Push and rotate* but they are part of the Benchmarking maps and thus they are considered and tested.

On the other hand the computation time of *Solve* is only 14 seconds for 991 agents. The rest of the results are depicted in Tab. 5.12 – 5.14.

5.5 warehouse

This experiment demonstrates feasibility of the extended *Push and rotate* algorithm to deal with two types of robots. The map can be seen in Fig. 5.15. The nodes of the map are marked as circles and the racks as squares. The map has 218 nodes. The problem with 47 agents takes only 17 milliseconds. The results are stored in Tab. 5.16.

Another map was created by copying the `warehouse` map next to and under it until a width reached 112 and height 113. A random generator was used for creating tasks for this map. Many tasks were not used because they either targeted an obstacle or they shared a start or goal position with others. The map has 11 472 nodes and 575 agents were assigned and the task was computed in 36 seconds with 25 seconds wasting on *Assigning agents to subproblems*. The results are stored in Tab. 5.18.



(a) : The map. Obstacles are black and free nodes are white.

MAP INFO	COUNT
Nodes	131 071
Subpr. before merging	11506
Subpr. after merging	1
Size of subpr. 1	89785

(b) : Map properties.

Figure 5.3: `Maze512-1-0` overview. The map is taken from [1] and the picture as well.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	2387	0.000
Push and Rotate	2419	0.013
Smoothed	2413	0.011

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	189
Division into subpr.	341
Assign. $\mathcal{A}$ to subpr.	819 800
Priority relation	0
Solve	62
Smooth	8
Total time	820 403
Problem time	819 873

(b) : The duration of each part.

Figure 5.4: The result acquired on the map `maze512-1-0` using 101 agents.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	51626	0.000
Push and Rotate	104555	1.025
Smoothed	104212	1.019

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	349
Division into subpr.	328
Assign. $\mathcal{A}$ to subpr.	819 079
Priority relation	0
Solve	1 953
Smooth	81
Total time	821 792
Problem time	821 115

(b) : The duration of each part.

Figure 5.5: The result acquired on the map `maze512-1-0` using 499 agents.

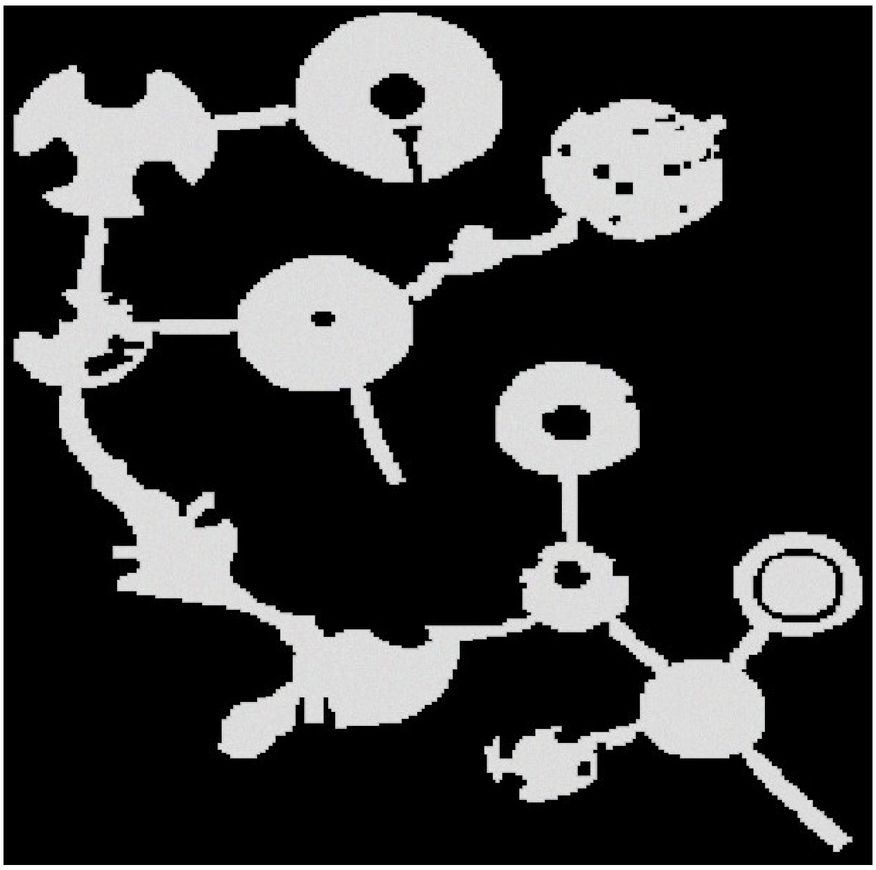
SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	201355	0.000
Push and Rotate	2991836	13.859
Smoothed	2767780	12.746

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	689
Division into subpr.	339
Assign. $\mathcal{A}$ to subpr.	840 430
Priority relation	0
Solve	27 385
Smooth	2 216
Total time	871 061
Problem time	870 033

(b) : The duration of each part.

Figure 5.6: The result acquired on the map `maze512-1-0` using 993 agents.



(a) : The map AR0411SR from a pc game Baldurs Gate II (published September, 2000) [1]. Obstacles are black and free nodes are white. The circle inside of a circle on bottom right side is the second subproblem.

MAP INFO	COUNT
Nodes	58 581
Subpr. before merging	2
Subpr. after merging	2
Size of subpr. 1	57 359
Size of subpr. 2	1 222

(b) : Map properties.

Figure 5.7: `AR0411SR` overview. The map is taken from [1] and the picture as well.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	35947	0.000
Push and rotate	36457	0.014
Smoothed	36266	0.009

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	453
Division into subpr.	7 180
Assign. $\mathcal{A}$ to subpr.	10
Priority relation	104
Solve	471
Smooth	22
Total time	8 243
Problem time	610

(b) : The duration of each part.

Figure 5.8: The result acquired on the map `AR0411SR` using 101 agents.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	164344	0.000
Push and rotate	178353	0.085
Smoothed	173116	0.053

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	1 544
Division into subpr.	5 717
Assign. $\mathcal{A}$ to subpr.	9
Priority relation	94
Solve	6 859
Smooth	128
Total time	14 352
Problem time	7 091

(b) : The duration of each part.

Figure 5.9: The result acquired on the map `AR0411SR` using 497 agents.

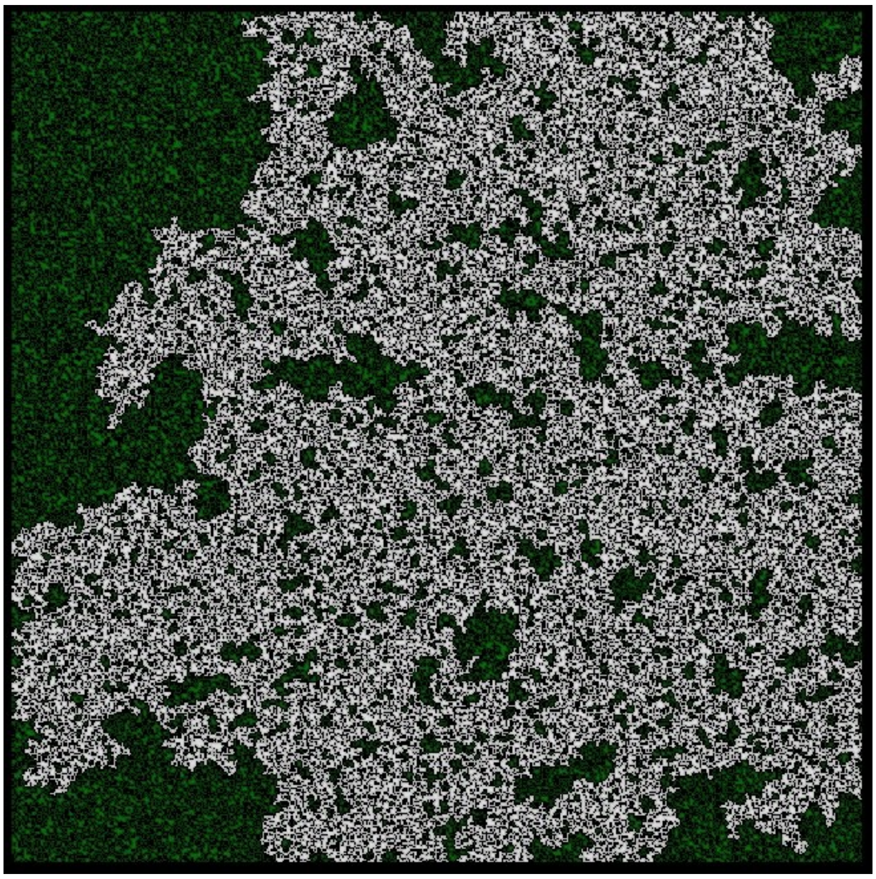
SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	337811	0.000
Push and rotate	554219	0.641
Smoothed	507415	0.502

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	3 032
Division into subpr.	6 182
Assign. $\mathcal{A}$ to subpr.	8
Priority relation	91
Solve	31 944
Smooth	442
Total time	41 703
Problem time	32 489

(b) : The duration of each part.

Figure 5.10: The result acquired on the map `AR0411SR` using 982 agents.



(a) : The map. Obstacles are black nodes, free nodes are white and trees are green. In this case the trees are considered as obstacles as well.

MAP INFO	COUNT
Nodes	104 950
Subpr. before merging	5482
Subpr. after merging	1
Size of subpr. 1	84 520

(b) : Map properties.

Figure 5.11: `Random512-40-0` overview. The map is taken from [1] and the picture as well.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	2658	0.000
Push and rotate	2662	0.001
Smoothed	2662	0.002

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	113
Division into subpr.	52 480
Assign. $\mathcal{A}$ to subpr.	857 561
Priority relation	0
Solve	41
Smooth	8
Total time	910 206
Problem time	857 613

(b) : The duration of each part.

Figure 5.12: The result acquired on the map `random512-40-0` using 100 agents.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	60380	0.000
Push and rotate	61822	0.024
Smoothed	61595	0.020

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	424
Division into subpr.	52 449
Assign. $\mathcal{A}$ to subpr.	848 536
Priority relation	0
Solve	1 454
Smooth	54
Total time	902 920
Problem time	850 047

(b) : The duration of each part.

Figure 5.13: The result acquired on the map `random512-40-0` using 499 agents.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	242384	0.000
Push and rotate	292718	0.208
Smoothed	286502	0.182

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	1 913
Division into subpr.	49 082
Assign. $\mathcal{A}$ to subpr.	848 026
Priority relation	0
Solve	13 897
Smooth	241
Total time	913 163
Problem time	862 168

(b) : The duration of each part.

Figure 5.14: The result acquired on the map `random512-40-0` using 991 agents.

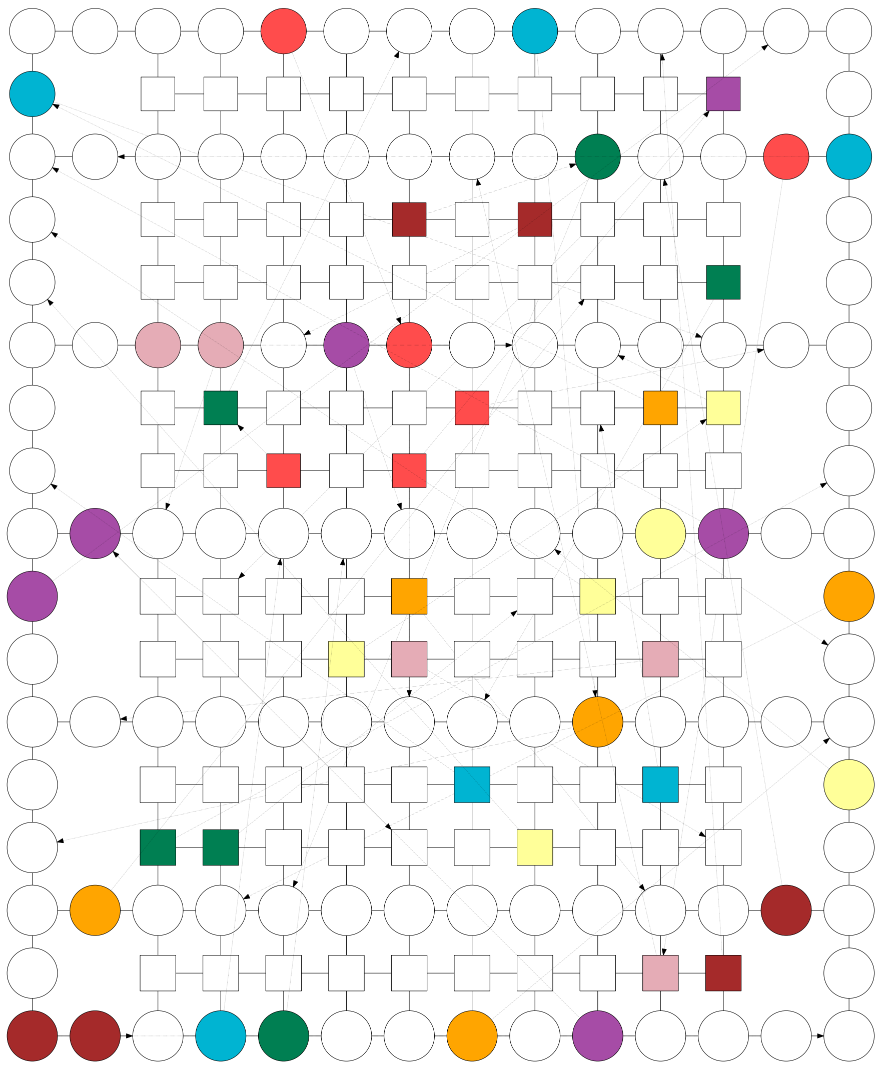


Figure 5.15: The map `warehouse` created for this thesis. Each agent has its own colour. The coloured squares are the agents under the rack. The goal destination of an agent is marked by dotted arrow.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	534	0.000
Push and Rotate	2803	4.249
Smoothed	2180	3.082

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	2
Division into subpr.	0
Assign. $\mathcal{A}$ to subpr.	4
Priority relation	0
Solve	10
Smooth	1
Total time	19
Problem time	17

(b) : The duration of each part.

Figure 5.16: The result acquired on the map `warehouse` using 47 agents. The map can be seen in Fig. 5.15. It has 218 nodes and single subproblem of size 118.

SOLUTION	MOVES	RATIO x_1
Shortest possible l_0	37476	0.000
Push and Rotate	49652	0.325
Smoothed	49498	0.321

(a) : Solution sizes.

ALG. PART	TIME [ms]
Map loading	267
Division into subpr.	80
Assign. $\mathcal{A}$ to subpr.	25 087
Priority relation	0
Solve	10 800
Smooth	44
Total time	36 281
Problem time	35 934

(b) : The duration of each part.

Figure 5.18: The result acquired on the map `warehouse` using 575 agents. The map can be seen in Fig. 5.15. This map was created copying the `warehouse` map next to it until a width 112 and height 113 nodes was reached. It has 11 472 nodes and single subproblem of size 5 552.

5.6 Time complexity

The algorithm *Push and rotate* was profiled by QTProfiler [2] which returned the Fig. 5.19. The profiling was done on a map `AR0411SR`. The problems with this map are that *Solve* takes too long. There is no clear winner here but apparently it would be possible to improve the call of *AStar* where there is too much time wasted on such easy algorithm.

Another profiling was done on a map `maze-512-1-0` by QTProfiler as well and it returned the Fig. 5.20. The issue and a clear winner here is the method *GetReachableFreeSpaceCount* which really takes too much time. This method shall be improved possibly by some better representation of queue.

5.7 Available maps types

The algorithm supports two map types. The first type is used for a maps with a low count of nodes, e.g. maps with node count lower than about 20. The maximum number of nodes is not defined because it depends on the user. The second type is taken from the benchmark page using the map description from the same source [1]. This is used for maps with thousands of nodes.

5.7.1 Low node count map type

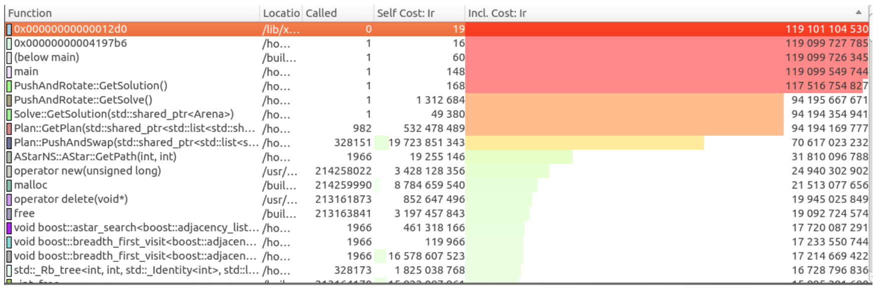
The map starts with the description of nodes and their neighbours. The node id is ended by ':' character. The delimiter between the nodes is ','. Each new node is on separate line. Then follows the list of the robots. Robots are marked with R before id. When at least one robot is loaded and then there is a free line beneath it then the loading is finished. Syntax is:

```
<NodeId>[<row>,<column>]:<NeighbourId>,<NeighbourId>,...
<no line>
R<AgentId>:<initial position>,<goal position>
<no line2>
```

It is possible to store some info about the map after <no line2> because the algorithm ignores everything after this free line. For example a name or what to expect from the task. A map `vertex` (Fig. 3.4) is described for example as:

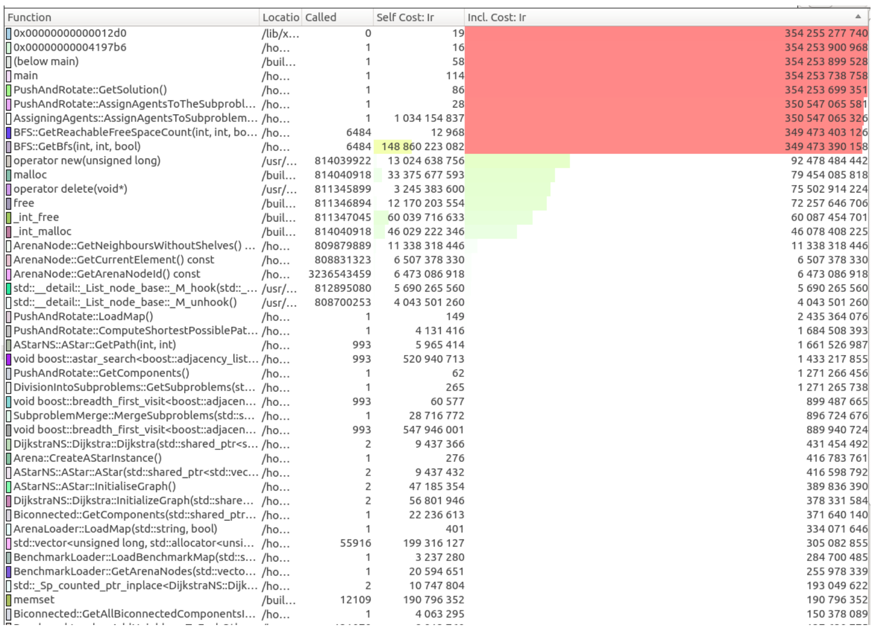
```
1[10,0]:2
2[10,5]:1,3,4
3[0,10]:2
4[20,10]:2

R1:1,3
R2:2,2
```



Function	Locatio	Called	Self Cost: Ir	Incl. Cost: Ir
0x00000000000012d0	/lib/x...	0	19	119 101 104 530
0x00000000004197b6	/ho...	1	16	119 099 727 785
(below main)	/buil...	1	60	119 099 726 345
main	/ho...	1	148	119 099 549 744
PushAndRotate::GetSolution()	/ho...	1	168	117 516 754 827
PushAndRotate::GetSolve()	/ho...	1	1 312 684	94 195 667 671
Solve::GetSolution(std::shared_ptr<Arena>)	/ho...	1	49 380	94 194 354 941
Plan::GetPlan(std::shared_ptr<std::list<std::sh...	/ho...	982	532 478 489	94 194 169 777
Plan::PushAndSwap(std::shared_ptr<std::list<s...	/ho...	328151	19 723 851 343	70 617 023 232
AStarNS::AStar::GetPath(int, int)	/ho...	1966	19 255 146	31 810 096 788
operator new(unsigned long)	/usr/...	214258022	3 428 128 356	24 940 302 902
malloc	/buil...	214259990	8 784 659 540	21 513 077 656
operator delete(void*)	/usr/...	213161873	852 647 496	19 945 025 849
free	/buil...	213163841	3 197 457 843	19 092 724 574
void boost::astar_search<boost::adjacency_list...	/ho...	1966	461 318 166	17 720 087 291
void boost::breadth_first_visit<boost::adjacen...	/ho...	1966	119 966	17 233 550 744
void boost::breadth_first_visit<boost::adjacen...	/ho...	1966	16 578 607 523	17 214 669 422
std::_Rb_tree<int, int, std::_Identity<int>, std::l...	/ho...	328173	1 825 038 768	16 728 796 836

Figure 5.19: The calls made during the computation of map AR0411SR(Fig. 5.7 done by QTProfiler [2]).



Function	Locatio	Called	Self Cost: Ir	Incl. Cost: Ir
0x00000000000012d0	/lib/x...	0	19	354 255 277 740
0x00000000004197b6	/ho...	1	16	354 253 900 968
(below main)	/buil...	1	58	354 253 899 528
main	/ho...	1	114	354 253 738 758
PushAndRotate::GetSolution()	/ho...	1	86	354 253 699 351
PushAndRotate::AssignAgentsToTheSubprobl...	/ho...	1	28	350 547 065 581
AssigningAgents::AssignAgentsToSubproblem...	/ho...	1	1 034 154 837	350 547 065 326
BFS::GetReachableFreeSpaceCount(int, int, bo...	/ho...	6484	12 968	349 473 403 126
BFS::GetBfs(int, int, bool)	/ho...	6484	148 860 223 082	349 473 390 158
operator new(unsigned long)	/usr/...	814039922	13 024 638 756	92 478 484 442
malloc	/buil...	814040918	33 375 677 593	79 454 085 818
operator delete(void*)	/usr/...	811345899	3 245 383 600	75 502 914 224
free	/buil...	811346894	12 170 203 554	72 257 646 706
_int_free	/buil...	811347045	60 039 716 633	60 087 454 701
_int_malloc	/buil...	814040918	46 029 222 346	46 078 408 225
ArenaNode::GetNeighboursWithoutShelves() ...	/ho...	809879889	11 338 318 446	11 338 318 446
ArenaNode::GetCurrentElement() const	/ho...	808831323	6 507 378 330	6 507 378 330
ArenaNode::GetArenaNodeId() const	/ho...	3236543459	6 473 086 918	6 473 086 918
std::__detail::_List_node_base::_M_hook(std::_...	/usr/...	812895080	5 690 265 560	5 690 265 560
std::__detail::_List_node_base::_M_unhook()	/usr/...	808700253	4 043 501 260	4 043 501 260
PushAndRotate::LoadMap()	/ho...	1	149	2 435 364 076
PushAndRotate::ComputeShortestPossiblePat...	/ho...	1	4 131 416	1 684 508 393
AStarNS::AStar::GetPath(int, int)	/ho...	993	5 965 414	1 661 526 987
void boost::astar_search<boost::adjacency_list...	/ho...	993	520 940 713	1 433 217 855
PushAndRotate::GetComponents()	/ho...	1	62	1 271 266 456
DivisionIntoSubproblems::GetSubproblems(st...	/ho...	1	265	1 271 265 738
void boost::breadth_first_visit<boost::adjacen...	/ho...	993	60 577	899 487 665
SubproblemMerge::MergeSubproblems(std::s...	/ho...	1	28 716 772	896 724 676
void boost::breadth_first_visit<boost::adjacen...	/ho...	993	547 946 001	889 940 724
DijkstraNS::Dijkstra::Dijkstra(std::shared_ptr<s...	/ho...	2	9 437 366	431 454 492
Arena::CreateAStarInstance()	/ho...	1	276	416 783 761
AStarNS::AStar::AStar(std::shared_ptr<std::vec...	/ho...	2	9 437 432	416 598 792
AStarNS::AStar::InitialiseGraph()	/ho...	2	47 185 354	389 836 390
DijkstraNS::Dijkstra::InitializeGraph(std::share...	/ho...	2	56 801 946	378 331 584
Biconnected::GetComponents(std::shared_ptr...	/ho...	1	22 236 613	371 640 140
ArenaLoader::LoadMap(std::string, bool)	/ho...	1	401	334 071 646
std::vector<unsigned long, std::allocator<unsi...	/ho...	55916	199 316 127	305 082 855
BenchmarkLoader::LoadBenchmarkMap(std::s...	/ho...	1	3 237 280	284 700 485
BenchmarkLoader::GetArenaNodes(std::vecto...	/ho...	1	20 594 651	255 978 339
std::_Sp_counted_ptr_inplace<DijkstraNS::Dijk...	/ho...	2	10 747 804	193 049 622
memset	/buil...	12109	190 796 352	190 796 352
Biconnected::GetAllBiconnectedComponentsI...	/ho...	1	4 063 295	150 378 089

Figure 5.20: The calls made during the computation of map maze (Fig. 5.3 done by QTProfiler [2]).

5.7.2 Benchmark map type

The benchmark maps are maps taken from Pathfinding benchmark site [1]. The algorithm supports their tasks are detected based on their file name extension, for example 'AR0411SR.map.scen'. A benchmark map consists of two files. The first one describes each task (start and goal position) and the second one depicts the map itself. The Information about the file format can be found [17].

A special char 'R' for describing a node with a rack was introduced because these maps do not support the idea of racks.

Chapter 6
Conclusion

The *Push and rotate* algorithm is described in this thesis. *Push and rotate* is moreover extended for use in maps with two types of agents (carrying a shelf and not) and with two types of obstacles (a wall, a rack) besides solving the Warehouse problem as well. The implementation of *Push and rotate* is explained and the implemented algorithm is used to provide results.

The Warehouse problem is defined at first. The common approaches (which are coupled and decoupled methods) are noted with the exception of multi agent planning systems. *Push and rotate* was chosen because its computation complexity promised that it is capable of planning of hundreds of agents in mere seconds which this thesis showed that it is possible on a certain type of maps. It was also suitable for the extension with shelves.

The algorithm is theoretically and thoroughly explained and the pseudo codes are upgraded and corrected when needed. The implementation notes are provided also with reasoning behind them for future work either for a revision or for an extension with real life applications.

Several experiments were performed based on problems from the Pathfinding benchmarks site. Tasks stored there are useful and they are guaranteed to be meaningful at least when an agent is planned alone (without other agents). The most important result is that *Push and rotate* is capable of planning 982 agents on a map `AR0411SR` with computing time 42 seconds about from which 10 seconds are used on parsing the map which can be done once and then used for another task on the same map unless some nodes or edges are removed or added. The extension with racks was tested on a map with 11 472 nodes and the task was to plan 575 agents. This was done in about 37 seconds from which about 25 seconds the assignment of agents was computed. This can be removed if the map remains the same.

The future work can extend this algorithm to work with a more types of agents. For example an agent which moves half the speed of other agents or double speed. A situation is only considered in this thesis when an agent is capable of moving to a next node in an instant. Another future task is that a human moves through the Warehouse and no agent can come to its proximity for safety reasons. This will cause some replanning among the agents and probably removing some graph nodes and edges. The last thing is that agents are expected to move one after another, i.e. they are taking turns.

The application can be extended in future in the way that the agents are moved concurrently. The solution provided can be used for the concurrently moving agents but sometimes an agent must wait for another one.

Chapter 7

CD content

Folder	Content
`/push_and_rotate`	The extended *Push and rotate* algorithm.
`/thesis`	This thesis in PDF format.
`/thesissource`	The source of this thesis.

Bibliography

[1] "Pathfinding benchmarks." http://www.movingai.com/benchmarks/. [Online; accessed 20-May-2016].

[2] "Profiling and memory checking tools." https://wiki.qt.io/Profiling_and_Memory_Checking_Tools. [Online; accessed 27-May-2016].

[3] B. De Wilde, *Cooperative multi-agent path planning.* PhD thesis, TU Delft, Delft University of Technology, 2012.

[4] D. Silver, "Cooperative pathfinding.," in *AIIDE*, pp. 117–122, 2005.

[5] K.-H. C. Wang and A. Botea, "Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees," *Journal of Artificial Intelligence Research*, pp. 55–90, 2011.

[6] W. Wang and W. Goh, "A stochastic algorithm for makespan minimized multi-agent path planning in discrete space," *APPLIED SOFT COMPUTING*, vol. 30, pp. 287–304, 2015.

[7] K. Chiew, "Scheduling and routing of autonomous moving objects on a mesh topology," *Operational Research*, vol. 12, no. 3, pp. 385–397, 2012.

[8] R. Luna and K. E. Bekris, "Efficient and complete centralized multi-robot path planning," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3268–3275, Sept 2011.

[9] B. de Wilde, A. W. ter Mors, and C. Witteveen, "Push and rotate: cooperative multi-agent path planning," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pp. 87–94, International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[10] "topological_sort." http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/topological_sort.html. [Online; accessed 02-May-2016].

[11] "C++." https://en.wikipedia.org/wiki/C%2B%2B. [Online; accessed 17-May-2016].

[12] "Standard template library." https://en.wikipedia.org/wiki/Standard_Template_Library. [Online; accessed 17-May-2016].

[13] "Biconnected_components and articulation_points." http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/biconnected_components.html. [Online; accessed 02-May-2016].

[14] "Dijkstra_shortest_paths." http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/dijkstra_shortest_paths.html. [Online; accessed 02-May-2016].

[15] "Breadth-first search." https://en.wikipedia.org/wiki/Breadth-first_search. [Online; accessed 17-May-2016].

[16] "Astar_search." http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/astar_search.html. [Online; accessed 02-May-2016].

[17] "Pathfinding benchmarks." http://www.movingai.com/benchmarks/formats.html. [Online; accessed 20-May-2016].