Cooperative Multi-Agent Path Planning

Boris de Wilde
Cooperative Multi-Agent Path Planning

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Boris de Wilde
born in Assen, the Netherlands

TU Delft
Algorithmics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
Cooperative Multi-Agent Path Planning

Author: Boris de Wilde
Student id: 1549189
Email: boreus@gmail.com

Abstract

Multi-Agent Path Planning is the problem of finding routes between pairs of start and destination vertices in a graph such that these routes are conflict-free in space and time. Application domains usually feature automated vehicles, for example in areas like warehouse management, aircraft taxiing and video games.

Finding an optimal set of routes is an NP-hard problem; sub-optimal approaches usually decouple the problem into the problems of finding the individual routes sequentially, where each consecutive problem is constrained by previously determined routes. One of these approaches is the Push and Swap algorithm. The Push and Swap algorithm has been presented as complete for the class of problems in which at least two vertices in the graph do not contain an agent. We demonstrate, however, that there exist instances of this type in which a solution exists, but the Push and Swap algorithm fails to find one.

By combining our analysis of the Push and Swap algorithm with results from the literature on the feasibility of finding solutions in the problem of moving ‘pebbles’ over graphs, we present the Push and Rotate algorithm, which is complete for the class of problems in which at least two vertices are unoccupied. The algorithm runs in polynomial time and is presented with a proof of completeness. Furthermore, we present a revised version of an important post-processing operation that makes the algorithm practically usable on large instances.

Thesis Committee:

Chair: Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft
Daily supervisor: Dr. Ir. A.W. ter Mors, Faculty EEMCS, TU Delft
Committee Member: Dr. T.B. Klos, Faculty EEMCS, TU Delft
Committee Member: Dr. Ir. F.A. Kuipers, Faculty EEMCS, TU Delft
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contents</strong></td>
<td>iii</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>v</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem statement</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Overview</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Context</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Problem statement</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Complexity</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Algorithms</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Conclusion</td>
<td>9</td>
</tr>
<tr>
<td><strong>3 Push and Swap</strong></td>
<td>11</td>
</tr>
<tr>
<td>3.1 Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Analysis</td>
<td>20</td>
</tr>
<tr>
<td><strong>4 Push and Rotate</strong></td>
<td>29</td>
</tr>
<tr>
<td>4.1 Problem decomposition</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Push and Rotate</td>
<td>35</td>
</tr>
<tr>
<td>4.3 Analysis</td>
<td>51</td>
</tr>
<tr>
<td>4.4 Heuristic improvements</td>
<td>58</td>
</tr>
<tr>
<td><strong>5 Experimental evaluation</strong></td>
<td>61</td>
</tr>
<tr>
<td>5.1 Instances</td>
<td>61</td>
</tr>
<tr>
<td>5.2 Solution quality</td>
<td>63</td>
</tr>
<tr>
<td>5.3 Runtime</td>
<td>64</td>
</tr>
<tr>
<td>5.4 Smooth</td>
<td>65</td>
</tr>
<tr>
<td>5.5 Heuristic shortest path</td>
<td>66</td>
</tr>
</tbody>
</table>
###Contents

<table>
<thead>
<tr>
<th>6 Conclusion</th>
<th>69</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Future work</td>
<td>69</td>
</tr>
</tbody>
</table>

**Bibliography**

| 71 |
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The famous 14-15 puzzle. The objective of the puzzle is to slide the blocks such that blocks 14 and 15 switch places, and the other blocks are on their original positions.</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Wilson’s $\theta_0$ graph.</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Illustration of isthmuses connecting non-separable components.</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Reduction from 3XC to multi-agent path planning.</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Illustration of (part of) the Slidable class of instances. Suppose the path of some agent to its destination is from left to right, each of the vertices in this path will have an “alternative path” that does not use this vertex, as shown for the vertex $v$ in the middle.</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of agent $r$ pushing agent $s$.</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of a swap between agents $r$ and $s$.</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Sequence of states of the exchange operation.</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Clearing of vertex $n$. The red vertices, $v$, $v'$ and $\epsilon$ are the “obstacles” in $\mathcal{U}$.</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Second stage of the clear operation.</td>
<td>15</td>
</tr>
<tr>
<td>3.6</td>
<td>Stage three of the clear operation.</td>
<td>16</td>
</tr>
<tr>
<td>3.7</td>
<td>Example of redundant moves.</td>
<td>18</td>
</tr>
<tr>
<td>3.8</td>
<td>Example in which no swap operation is possible.</td>
<td>20</td>
</tr>
<tr>
<td>3.9</td>
<td>State of the example after agent $r$ has been sent to its goal vertex.</td>
<td>21</td>
</tr>
<tr>
<td>3.10</td>
<td>Even though agent $a_1$ is unable to swap positions with the other agents, this instance is solvable.</td>
<td>22</td>
</tr>
<tr>
<td>3.11</td>
<td>Example instance in which resolve is recursively executed many times, leading to an invalid state. The pink vertex indicates the position of the agent ($a_8$) that is next to be sent to its destination.</td>
<td>23</td>
</tr>
</tbody>
</table>
3.12 (a.) The agent $a_8$ is the first to be sent to its destination vertex. In order to get there, it has just swapped with agent $a_6$. The resolve operation will proceed to bring $a_8$ to its goal, swapping it with agent $a_{11}$. (b.) When agent $a_8$ is at its goal position, agent $a_6$ is still unable to push back to its goal position, hence the resolve operation will send agent $a_{11}$ to its goal, first swapping it with agent $a_3$. (c.) Now agent $a_3$ will resolve. Yellow vertices indicate the positions of agents that are resolving.

3.13 (d.) Agent $a_{11}$ swaps with agent $a_9$ to reach its goal vertex. Since agent $a_3$ is unable to push back to its goal vertex, agent $a_9$ will now be sent to its goal vertex. In order to reach its goal vertex, agent $a_9$ has to swap agent $a_2$ away from its goal vertex. (e.) Now agent $a_2$ is also resolving. (f.) Once agent $a_9$ has reached its goal position, $a_{10}$ will be sent to its goal position. Agent $a_{10}$ needs to swap with the already resolving agent $a_6$ in order to reach its goal position.

3.14 (g.) Once agent $a_6$ has been swapped, it is more than one step away from its goal vertex. (h. and i.) Another problem is that the agent that is in the way of agent $a_6$ (which is agent $a_{12}$) is at its goal position, and will not move out of the way when it is sent to its goal. Finally there is no agent left that can be chosen to send to its destination.

3.15 The clear operation will not be able to properly clear two neighbours of the node of $v$.

3.16 Second case in which the clear operation does not consider all necessary options.

3.17 Second example sequence of moves containing redundancy.

4.1 Decomposition of a problem: nontrivial biconnected components.

4.2 Decomposition of a problem: vertices with degree $\geq 3$.

4.3 Decomposition of a problem: the components, after adding the vertices with degree $\geq 3$.

4.4 Decomposition of a problem: nodes assigned to the subproblems in case there are 3 unoccupied vertices.

4.5 Decomposition of a problem: the agents assigned to the yellow (leftmost) subproblem are $a_1$, $a_2$, $a_3$ and $a_4$.

4.6 Decomposition of a problem: the agents assigned to the red (middle) subproblem are $a_5$, $a_6$, $a_7$ and $a_8$.

4.7 Decomposition of a problem: the agents assigned to the red (middle) subproblem are $a_5$, $a_6$, $a_7$ and $a_8$.

4.8 Decomposition of a problem: the agents assigned to the blue (rightmost) subproblem are $a_9$ and $a_{10}$.

4.9 An instance in which propagation of the priority relation is required.

4.10 Illustration of the clear_vertex and push operations. Picture (a.) shows the path to an unoccupied vertex in $G$, the agents move along that path in order to clear the next vertex the path of agent $a_3$ to its goal.

4.11 Sequence of states of the exchange operation.

4.12 The stages of the clear operation.

4.13 The stages of the clear operation.
4.14 Illustration of rotate: (line 13) the clear vertex operation is executed on vertex $v'$. ................................................. 45
4.15 Illustration of rotate: (line 17) agent $r'$ moves to vertex $v'$. ................. 45
4.16 Illustration of rotate: (line 18) agents $r$ and $r'$ swap positions. ................. 45
4.17 Illustration of rotate: (line 19) all agents in the cycle move forwards, begin-

ning with agent $a_3$. ......................................................... 46
4.18 Illustration of rotate: (line 20) the reversal of the clear vertex moves is executed (replacing agent $r$ with agent $r'$). ......................... 46
4.19 Illustration of rotate: the operation is completed successfully. ......................... 46
4.20 Illustration of the paths $p$ (pink) and $q$ (yellow) in the operations plan and

resolve. In (a.) the plan operation will perform two swap operations in order to move agent $a_6$ to the right-hand side. Picture (b.) shows that agent $a_6$ has been swapped away from its goal position. In order for agent $a_6$ to return to its position, the agent $a_{11}$ that is currently occupying that position needs to be moved towards its goal position. ................................................. 48
4.21 Illustration of the paths $p$ (pink) and $q$ (yellow) in the operations plan and

resolve. $q$ now (picture (c.)) already contains two agents that want to return to their goal positions. Swapping agent $a_9$ with agent $a_2$ adds another. In picture (d.) the paths $p$ and $q$ intersect and it all the agents on the cycle that is formed want to move (or “rotate”) forward through this cycle. ......................... 49
4.22 Illustration of the paths $p$ (pink) and $q$ (yellow) in the operations plan and

resolve. After the rotate operation is completed, only one more swap operation is enough to solve the instance. ................................. 49
4.23 Agent $r$ and $s$ are unable to swap on the isthmus. ......................... 54
4.24 Illustration of reachable free vertices. ................................................. 56
4.25 Example illustrating the importance of agent ordering. A number of random

orderings is used for the Push and Rotate algorithm. Clearly some orderings yield a lot better results than others. ................................................. 59
4.26 Example illustrating the importance of agent ordering. ................................. 59
4.27 Example illustrating the different shortest paths. ................................. 60
4.28 An example in which taking the shortest path is clearly sub-optimal. ................................. 60

5.1 The roadmap of an instance from the Starcraft benchmark set: TheFrozenSea.

It contains 754304 vertices (the white pixels). The instance has 3700 agents. ................................. 63
5.2 Solution quality (number of moves, left-hand side), and the difference with the

lower bound ($\frac{\Pi_l - l_0}{l_0}$, right-hand side) for the benchmark sets. ................................. 64
5.3 Runtime on benchmark instances, plotted against number of agents (left) and

solution quality (right). ................................. 64
5.4 Runtime on different grid sizes, plotted against number of agents (left) and

solution quality (right). ................................. 65
5.5 Runtime (left) and effects (right, ($\frac{\text{smooth}(\Pi) - l_0}{\Pi - l_0}$) of smooth for the benchmark sets. ................................. 66
5.6 Runtime and effects of smooth on different grid sizes. ................................. 66
5.7 Runtime (left) and solution quality (right) with and without the shortest path heuristic on the grid instances. .................................................. 68
5.8 Runtime (left) and solution quality (right) with and without the shortest path heuristic for the benchmark instances. ............................... 68
Chapter 1

Introduction

From [12]:

The multi-robot path planning problem is a well-known and extensively studied problem. In these instances, a set of autonomous mobile robots operate in a common discrete graph (roadmap), with the objective of computing a set of collision free [sic] paths on the graph for each robot from its unique start to its unique goal. There are many interesting and exciting uses for an effective solution to this problem, including:

3. Intelligent transportation networks [1] where an external system is employed to effectively navigate vehicles.
4. Assembly and disassembly problems [5] where a sequence of actions is used to construct/deconstruct a product from its parts.
5. Autonomous robotic mining [19] where teams of heterogeneous robots operate in a common environment to achieve a common goal.
6. Computer games [17] where teams of autonomous agents need to traverse collision free paths from their start to their goal [reference to figure].

With such an extensive array of important application domains, it is no surprise that algorithms have been formulated. In his master’s thesis [12], Ryan Luna presented the Push and Swap algorithm. One by one, the algorithm moves agents to their respective destination locations along a shortest path, pushing still-to-be-planned-for agents out of the way, and swapping positions with those already at their destination. The Push and Swap algorithm is touted as complete (i.e., if a solution exists, then the algorithm will find one) for all instances in which at least two locations are unoccupied, and runs in polynomial time. The algorithm
1.1 Problem statement

The main problem is to develop an algorithm for the multi-robot path planning problem:

Design an algorithm that finds a sequence of moves to transform an initial configuration of agents on the roadmap to a goal configuration, that is complete for the class of problems with at least two unoccupied locations, and that runs in polynomial-time.

Since the problem of finding the shortest sequence of moves for any instance is NP-hard (cf. [18]), finding optimal solutions may require an exponential number of steps, assuming P \( \neq \) NP.

Since we will base our algorithm on Luna’s Push and Swap, an additional problem is to analyze for which classes of instances Push and Swap is not complete, and to identify the shortcomings in the specification of Push and Swap, in order to rectify these for our Push and Rotate algorithm.

1.2 Contributions

This thesis makes the following contributions to the field of multi-robot path planning:

- **Push and Rotate:** a complete, polynomial-time algorithm for the class of multi-robot path planning problems in which at least two locations are unoccupied.

- An analysis of the Push and Swap algorithm: we will demonstrate that Push and Swap is not complete for the following classes of problems:

  1. polygons (or cycles): a roadmap with only vertices of degree 2,

---

1 The International Conference on Intelligent Robots and Systems (IROS) and International Joint Conference on Artificial Intelligence (IJCAI) are both mainstream conferences in robotics and artificial intelligence, respectively.

2 Whether or not the Push and Swap algorithm finds a solution depends, for some types of instances, on the order in which the agents are planned for.
2. roadmaps which contain isthmuses (strings with length $\geq m - 1$ of vertices of degree 2, where $m$ is the number of unoccupied vertices in the instance) such that agents are not able to move from one part of the roadmap to another part,

3. instances which cause the \texttt{resolve} operation to be invoked twice on a single agent, and

4. certain agent configurations in which the \texttt{swap} operation will not succeed.

- The Push and Swap algorithm contains a procedure \texttt{smooth} that can remove any redundant moves (that may be introduced when one agent moves out of the way of another agent) from a solution. It runs in $O(n^3)$ time, where $n$ is the number of moves in the solution, which makes it a dominant factor in the overall running time of the algorithm. We present a quicker, but otherwise equivalent, $O(n)$-time \texttt{smooth} procedure.

- We present a heuristic way of computing a favourable shortest path along which to move an agent towards its destination vertex, and run it on a number of problem benchmarks from the games industry, comparing it to a standard shortest path. In those cases that there was a significant difference between our heuristic and the standard shortest path, our heuristic was significantly better.

### 1.3 Overview

This thesis will proceed as follows. In the next chapter, we will give a formal definition of the multi-robot path planning problem, we will discuss some related algorithms, and some algorithms for related problems. In chapter 3, we will first replicate (using the terminology of [12]) and explain the Push and Swap algorithm in its entirety, before presenting a richly-illustrated demonstration of the gaps in the completeness of the Push and Swap algorithm.

In chapter 4, we present our Push and Rotate algorithm, which derives its name from the \texttt{rotate} operation that deals with cyclic resolve issues that can trip up Push and Swap. The idea behind Push and Rotate is first to perform a reachability analysis, to partition agents into groups of agents that can exchange locations with each other, and finally to prioritize agents to ensure that agents from one group finish before agents in another group, in case this is necessary for a solution to be found.

In chapter 5, we show the performance of our algorithm on a number of gaming benchmarks, as well as grids, and we analyze the performance of the shortest path heuristic. In chapter 6, we present some ideas for future work.
Chapter 2

Context

2.1 Problem statement

Can a collection of agents in a roadmap move from one set of locations to another, without collision? We define the Multi-Agent Path Planning problem as follows:

Given a roadmap \( G = (V, E) \), which is a connected graph, a set of agents \( R \), an initial assignment of agents to vertices \( S : R \rightarrow V \), and a goal assignment of agents to vertices \( T : R \rightarrow V \). Let a move \( \pi : (R \rightarrow V) \rightarrow (R \rightarrow V) \) be the change in assignment of exactly one agent such that there is an edge in the graph between its current and its next vertex. Find a sequence of moves \( \Pi = [\pi_1, \pi_2, \ldots, \pi_l] \), such that applying the sequence of moves takes the assignment of agents from \( S \) to \( T \), i.e. \( (\pi_l \circ \pi_{l-1} \circ \cdots \circ \pi_2 \circ \pi_1)(S) = T \).

A move \( \pi \) can, for the sake of simplicity, also be described with just one agent \( r \in R \) and one vertex \( v \in V \), and is a valid move for the “current” assignment \( \mathcal{A} \) if there is no \( r' \) for which \( \mathcal{A}(r') = v \) and there exists an edge \( (\mathcal{A}(r), v) \in E \) in the roadmap. The “next” assignment \( \mathcal{A}' = \pi(A) \) can be described as follows:

\[
\mathcal{A}'(a) = \begin{cases} 
  v & \text{if } a = r \\
  \mathcal{A}(a) & \text{otherwise}
\end{cases}
\]

2.2 Complexity

A problem related to the multi-agent path planning problem is the Warehouseman’s problem (from [7]):

Given an initial and final configuration of objects in question, determine whether there exists a continuous coordinated motion of the objects between the initial and final configurations during which they do not penetrate either the walls of the enclosing box or each other.

Hopcroft et al. show that the coordinated motion planning problem for a collection of two-dimensional rectangular objects moving inside a two-dimensional rectangular box is PSPACE-hard [7]. Later, this result was strengthened by Hearn and Demaine using their non-deterministic constraint logic model of computation [6]. From Hopcroft et al.:
This result should be viewed as a guide to the difficulty of the general problem and should lead researchers to consider more tractable restricted classes of motion problems of practical interest.

One of the possible restrictions is to replace the geometry in the problems by adjacencies in a graph.

Figure 2.1: The famous 14-15 puzzle. The objective of the puzzle is to slide the blocks such that blocks 14 and 15 switch places, and the other blocks are on their original positions.

Inspired by Sam Loyd’s 15-puzzle (figure 2.1), a problem statement quite similar to the one in section 2.1 is defined by Wilson [27]. He considers the labelings on a graph $G$ that has one unoccupied vertex. Define the simple graph $\text{puz}(G)$ as a graph with vertex set $V(\text{puz}(G))$ consisting of all the labelings on $G$. Two labelings are joined by an edge in $\text{puz}(G)$ if and only if one labeling can be obtained from the other in one move. Wilson shows that, for simple non-separable graphs, $\text{puz}(G)$ is connected, unless $G$ is biconnected, in which case $\text{puz}(G)$ has two components. There are two exceptions in his result: the polygon (or cycle) graph, and remarkably a single specific graph $\theta_0$ (figure 2.2).

Figure 2.2: Wilson’s $\theta_0$ graph.

Kornhauser extends this result to general graphs with any number of unoccupied vertices and provides a P-time decision procedure with an $O(n^3)$ bound on the number of moves required [10]. The result by Kornhauser is based on the fact that a graph may be viewed as a tree of biconnected (non-separable) components that are linked by chains of vertices with degree 2 (called isthmuses). Note that if two of these components are linked by a chain that contains more vertices than the number of unoccupied vertices minus two, then it is impossible for an agent to cross this isthmus (see figure 2.3).
Note that agent $a_6$ can take one step off the right-hand side isthmus on both sides, but it is only able to further enter the component on the left-hand side. The fact that agents $a_1 \ldots a_6$ can move to the position that is occupied in the figure by agent $a_7$ is important and will return later in this work. The isthmuses that are impossible to cross induce a decomposition of the problem instance into smaller subproblems, which can be solved separately.

A “natural algorithmic question arises”: is it feasible to find the shortest solution in polynomial time? Multi-agent path planning can be considered a generalization of single-agent path planning, which is called graph motion planning with one robot by Papadimitriou et al. [18]. They show NP-completeness of the optimization variant of motion planning problem in which only one agent has a specific destination vertex, and the other agents are only considered (movable) obstacles. This result holds on planar graphs. Papadimitriou et al. also provide approximate algorithms.

By reducing 3-Exact-Cover to it, Goldreich shows that it is NP-hard to find the shortest solution to such a problem [3]. The basic idea behind the reduction is shown in figure 2.4.
Recall that the 3XC problem [2] can be stated as follows:

Given a set of elements \( U = \{ e_i \} \) and a set of subsets \( S = \{ s_j \} \), where for each subset \( s_j \subseteq U \) and \( |s_j| = 3 \). Is there a subset of subsets \( S' \subseteq S \) such that \( \bigcup_{x \in S'} x = U \) and all subsets in \( S' \) are pairwise disjoint (\( s_a \cap s_b = \emptyset \) for each \( s_a \in S', s_b \in S', \) and \( s_a \neq s_b \))? 

Each element in \( U \) is represented by two vertices that contain agents that want to swap positions. The subsets in \( S \) are represented by a single vertex containing an agent. This vertex is connected to both vertices of each element that is contained in the set. All set vertices are connected to an empty vertex \( v_0 \). In order to enable two agents within an element to swap positions, first one of the connected set vertices needs to be vacated by moving the agent in that vertex to \( v_0 \).

When a set vertex is vacated (i.e. agent \( a_i \) moves to \( v_0 \)), it is possible for the agents in all the three connected elements to swap positions. This means that at least \( \frac{1}{3} |U| \) set vertices need to be vacated in any solution. By setting the \( K \) parameter (the maximum allowed number of moves in the decision variant) to \( \frac{1}{3} |U| \), such that at most \( \frac{1}{3} |U| \) set vertices can be vacated, the solution to the SMS problem will match a solution to the 3XC problem: the set vertices (containing agents \( a_i \)) that are vacated in the SMS solution, are the subsets in a solution to the 3XC instance. Goldreich’s Shortest-Move-Sequence (SMS) problem differs from our problem statement only in formulation, hence the multi-agent path planning problem is NP-complete.

### 2.3 Algorithms

As seen in the previous section, many formulations of the problem have been proven impossible to efficiently solve optimally, unless \( P=NP \). There are a number of algorithms available to solve instances of the multi-agent path planning problem, we focus mainly on centralized algorithms. Some of these algorithms are complete for certain classes of the multi-agent path planning problem, while other algorithms provide no completeness guarantees.

A wide class of incomplete algorithms are based on a reservation system. Once an agent finds a path to its destination, it places a reservation on the required vertices at the required time slots. In context-aware routing [23], the agents only appear on the graph at a chosen “release time”. Since in our problem formulation there is no such release time (or, it is always “\( t = 0 \)”), this approach is not complete.

A reservation based system can also be used in a similar way to create a distributed cooperative routing algorithm [24]. Once an agent has planned a route to its destination, it broadcasts this path to all the other agents. When planning its path, an agent has to take into account the paths that higher priority agents have broadcast. If an agent receives a path from a higher priority agent, and this path conflicts with its own path, then the agent will have to change its plan. In order to be able to solve harder instances, a windowed approach is presented in [20]. In the WHCA* algorithm, the agents only make reservations for a restricted time horizon, which has three advantages. First, the agents can continue cooperating after they reach their destination vertices. The second advantage is that the
sensitivity to agent ordering (or prioritization) is reduced. Finally, there is no need to plan for long-term contingencies that may not occur.

\[ \text{Path} \rightarrow \text{Alt.path} \rightarrow \ldots \]

\[ \text{\ldots} \rightarrow v \rightarrow \text{\ldots} \]

Figure 2.5: Illustration of (part of) the Slidable class of instances. Suppose the path of some agent to its destination is from left to right, each of the vertices in this path will have an “alternative path” that does not use this vertex, as shown for the vertex $v$ in the middle.

The MAPP algorithm [26] works on a class of problem instances that is called “Slidable”. This is a set of conditions that include the existence of an alternate path for each of the steps in the path of an agent to its destination, and there is minimal interference between the initial and final position of the agents (see figure 2.5). Furthermore, the MAPP algorithm requires two unoccupied vertices.

A tree-based agent swapping strategy coupled with a tree decomposition approach is presented by Khorshid [9]. An important limiting factor of this algorithm is that not all the instances remain solvable after the tree decomposition. Specifically on instances that have a tree roadmap, or a roadmap that is like a tree, the algorithm performs very well. The authors show it performs better on these tree-like instances than the Push and Swap algorithm which is a main topic of this work.

The BIBOX algorithm [22] is complete for biconnected graphs with (at least) two unoccupied vertices. Note that a biconnected graph can be seen as a series of connected loops or cycles. The algorithm works by decomposing the roadmap into a sequence of loops and solving the outermost loop, thus creating a smaller problem, and repeating the process until only the “original” loop remains. The BIBOX algorithm solves instances in $O(|V|^3)$ steps.

### 2.4 Conclusion

It has been found that determining feasibility of Warehouseman’s Problem instances is PSPACE-hard. When the geometry issues are removed from the problem and replaced by a roadmap, then determining feasibility is in P, while finding the shortest solution is still NP-hard.

The Push and Swap algorithm seems favourable since it does not impose as many restrictions on the roadmap. However, as we show in the next chapter, the Push and Swap algorithm contains some issues that cause it to be incomplete. One of these issues arises when subproblems from Kornhauser’s decomposition interact. In chapter 4 we introduce the Push and Rotate algorithm, which performs the decomposition and also includes other improvements intended to solve the completeness issues of Push and Swap.
Chapter 3

Push and Swap

The Push and Swap algorithm is presented as an algorithm that is complete for a wide class of instances and runs in polynomial time. The class of instances for which it is claimed to be complete consists of all the instances in which at least two vertices in the roadmap are not occupied.

In the first section of this chapter we present the algorithm exactly as it was presented by Luna et al. in [15], while explaining the way the algorithm works. The second section contains examples of instances for which the Push and Swap algorithm is not complete.

As the name of the algorithm suggests, it is built from two operations: push and swap. The algorithm works as follows: on the way to their goal vertex, agents attempt to push blocking agents out of the way. If pushing the blocking agents fails, then the algorithm attempts to swap the agent with the blocking agent (figures 3.1 and 3.2). An important difference between the push operation and the swap operation is the fact that the swap operation is allowed to temporarily move agents away from their goal vertex.

![Figure 3.1: Example of agent $r$ pushing agent $s$.](image)

![Figure 3.2: Example of a swap between agents $r$ and $s$.](image)
3.1 Algorithm

The main algorithm is listed in algorithm 3.1.1. The current assignment of agents to vertices $\mathcal{A}$ is initially set to the start positions $\mathcal{S}$. Since the moves are represented by a list of assignments, the sequence of moves $\Pi^*$ is initialized containing only the initial assignment $\mathcal{S}$. As moves are executed in the algorithm, additional assignments will be appended to the list $\Pi^*$. The set $\mathcal{U}$ keeps track of (the positions of) those agents that are already at their goal position, indicating that these agents should not be moved. $\mathcal{U}$ is initialized to the empty set. Once agent $r$ has reached its goal position $T[r]$, this position is added to the set $\mathcal{U}$ indicating that agent $r$ should not be moved away from this position. In some undefined order the agents in $\mathcal{R}$ are selected to move to their goal vertex in $\mathcal{T}$. The agents will always move to their goal vertex in $\mathcal{T}$ along the shortest path $p$ from their current position in $\mathcal{A}$, using the push and the swap operation.

The push operation is only allowed to move the agents occupying vertices that are not in $\mathcal{U}$. Only when it is not possible to push a blocking agent out of the way, the swap operation is executed. The swap operation is allowed to temporarily move agents away from their goal positions, but it has mechanisms in place to make sure that all agents that are not involved in the swap will be returned to their goal positions at the end of the swap operation. An important operation that is used to return agents to their goal positions, after being swapped away from it, is the resolve operation.

Algorithm 3.1.1 push_and_swap($\mathcal{G}, \mathcal{R}, \mathcal{S}, \mathcal{T}$)

1: $\mathcal{A} \leftarrow \mathcal{S}, \Pi^* \leftarrow \{\mathcal{S}\}, \mathcal{U} \leftarrow \emptyset$
2: for all $r \in \mathcal{R}$ do
3:  while $\mathcal{A}[r] \neq \mathcal{T}[r]$ do
4:  $p = \text{shortest\_path}(\mathcal{G}, \mathcal{A}[r], \mathcal{T}[r])$
5:  if push($\Pi^*, \mathcal{G}, \mathcal{A}, r, p, \mathcal{U}$) = false then
6:    if swap($\Pi^*, \mathcal{G}, \mathcal{A}, T, r, \mathcal{U}$) = false then
7:      return $\emptyset$
8:  end if
9: end if
10: end while
11: $\mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{A}[r]$
12: end for
13: return $\Pi^*$

3.1.1 Push

The push operation (algorithm 3.1.2) tries to move agent $r$ along a specific path $p^*$ that is passed as a parameter. This path is usually the (shortest) path from $\mathcal{A}[r]$ to $\mathcal{T}[r]$, but not always, since the resolve operation also uses the push operation. As long as the vertices on path $p^*$ are unoccupied, the agent can simply move forward. When an agent occupies the next vertex on the path, an attempt is made to move this blocking agent away from this vertex. If the blocking agent is in the set $\mathcal{U}$, then the push operation will return false,
since it is not allowed to move this agent. Otherwise, the shortest path from vertex \( v \) to an empty vertex \( v_e \) is computed, considering the vertices in \( \mathcal{U} \) as obstacles, which means that effectively \( G \setminus \mathcal{U} \) will be searched for the path \( p \). If no such path can be found, the push operation will return false, since there is no way to clear vertex \( v \) without moving agents in \( \mathcal{U} \). All the agents on path \( p \) are moved forward along this path (toward the empty vertex), thus clearing the vertex \( v \). Agent \( r \) is now free to advance another step along the path \( p^* \) in the next iteration of the while loop. It seems that the assignment of \( v \) on line 2 is intended to be inside the while-loop.

**Algorithm 3.1.2** push\((\Pi^*, G, A, r, p^*, \mathcal{U})\)

1: \( t = \text{last vertex in } p^* \)
2: \( v \leftarrow \text{vertex in } p^* \text{ after } A[r] \)
3: \( \text{while } A[r] \neq t \) do
4: \( \text{Advance } r \text{ along } p^* \text{ until blocked, inserting intermediate actions into } \Pi^* \)
5: \( \text{if } A[r] \neq t \) then
6: \( \text{Mark } A[r] \text{ and } \mathcal{U} \text{ as blocked on } G \)
7: \( v_e \leftarrow \text{reachable empty vertex to } v \text{ on } G \)
8: \( p \leftarrow \text{shortest path}(G, v, v_e) \)
9: \( \text{if } p = \emptyset \text{ then return false } \text{end if} \)
10: \( \text{Mark } A[r] \text{ and } \mathcal{U} \text{ as free on } G \)
11: \( \text{Move robots on } p \text{ toward } v_e; \text{ insert actions into } \Pi^* \)
12: \( \text{end if} \)
13: \( \text{end while} \)
14: \( \text{return } \text{true} \)

### 3.1.2 Swap

The swap operation (algorithm 3.1.3) is executed when agent \( s \) is blocking path \( p^* \) of agent \( r \) towards its destination. The operation attempts to swap the positions of agents \( r \) and \( s \). The swap operation attempts to execute an exchange of positions as shown in figure 3.3. In order to perform this exchange, two requirements must be met: agent \( r \) must be on a vertex \( v \) with degree \( \geq 3 \), agent \( s \) must be on a vertex adjacent to \( v \), and (at least) two other vertices adjacent to \( v \) must be unoccupied. If these requirements are met, then the agents \( r \) and \( s \) can simply move into the unoccupied vertices in one order, and then move back out in the reverse order.

The two operations that are used to meet these requirements are push (algorithm 3.1.2) and clear (algorithm 3.1.4). Both these operations do not have to consider the set \( \mathcal{U} \), since all the moves that they execute, will be executed in reverse after the exchange operation is complete, putting all agents, except agents \( r \) and \( s \), back to their original positions. In order to find a suitable vertex \( v \) with degree 3 or more to perform the swap, simply all vertices that have degree 3 or more are tried.

The push operation pushes the “composite agent” made of agents \( r \) and \( s \) to vertex \( v \), but without considering the set \( \mathcal{U} \) as obstacles. The clear operation moves agents occupying the
Algorithm 3.1.3 \( \text{swap}(\Pi', G, A, T, r, \mathcal{U}) \)

1: \( p^* \leftarrow \text{shortest\_path}(G, A[r], T[r]) \)
2: \( s \leftarrow \text{robot on first vertex in } p^* \text{ after } A[r] \)
3: \( \text{success} = \text{false} \)
4: \( S \leftarrow \{ \text{Vertices of degree } \geq 3, \text{ sorted by dist. from } r \} \)
5: \( \text{while } S \neq \emptyset \text{ and } \text{success} = \text{false} \text{ do} \)
6: \( v \leftarrow S.\text{pop}(), \Pi \leftarrow \emptyset \)
7: \( p \leftarrow \text{shortest\_path}(G, A[r], v) \)
8: \( \text{if } \text{push}(\Pi, G, A, \{r, s\}, p, \emptyset) = \text{true} \text{ then} \)
9: \( \quad \text{if } \text{clear}(\Pi, G, A, A[r], A[s]) = \text{true} \text{ then} \)
10: \( \quad \quad \text{success} = \text{true} \)
11: \( \quad \text{end if} \)
12: \( \text{end if} \)
13: \( \text{end while} \)
14: \( \text{if } \text{success} = \text{false} \text{ then } \text{return} \text{ false } \text{ end if} \)
15: \( \Pi^* = \Pi + \Pi \)
16: \( \text{execute\_swap}(\Pi^*, G, A[r], A[s]) \)
17: \( \Pi = \Pi.\text{reverse}(), \text{exchanging paths for } r \text{ and } s \)
18: \( \Pi^* = \Pi^* + \Pi \)
19: \( \text{if } T[s] \in \mathcal{U} \text{ then} \)
20: \( \quad \text{return } \text{resolve}(\Pi^*, G, A, T, \mathcal{U}, p^*, r, s) \)
21: \( \text{end if} \)
22: \( \text{return} \text{ true} \)

Figure 3.3: Sequence of states of the exchange operation.
neighbour vertices of v away from v in order to create 2 unoccupied vertices adjacent to v.

Clear

The clear operation (algorithm 3.1.4) attempts to clear two neighbouring vertices of the vertex v in three stages. First, it checks for any already empty vertices in the neighbourhood of v. If there are two or more empty neighbours, the clear operation returns true. Otherwise (stage two, lines 5-17) it attempts to push agents in the neighbourhood of v away from v (figure 3.5), considering the vertices v, v' and ε as “obstacles” (i.e. adding them to the U parameter of clear_vertex, indicating that it is not allowed to move agents from or to these vertices, as shown in figure 3.4). If this yields two empty neighbours, then clear returns true.

![Figure 3.4: Clearing of vertex n. The red vertices, v, v' and ε are the “obstacles” in U.](image)

In stage three (figure 3.6), the clear operation requires one of the neighbours of v to be successfully cleared in order to make a final attempt at clearing another neighbour of v. By pushing the agents r and s away from v, the way is clear for an agent in a neighbouring vertex of v to move to the already cleared vertex ε. This agent is then pushed away from v. If this succeeds, then the originally empty vertex is cleared again, and the initial position of the agent is now also cleared, yielding two empty neighbour vertices. After moving agents r and s back towards v, the clear operation returns true in this case.

![Figure 3.5: Second stage of the clear operation.](image)
Algorithm 3.1.4 \texttt{clear}(\Pi^*, \mathcal{G}, \mathcal{A}, v, v')

\begin{algorithmic}[1]
\STATE $\epsilon \leftarrow \{\text{free neighbours of } v\}$, $\mathcal{U} \leftarrow \{v, v', \epsilon\}$
\IF{$|\epsilon| \geq 2$} \RETURN \texttt{true} \ENDIF
\FORALL{$n \in \text{neighbours}(v) \setminus \{\epsilon, v'\}$}
\FORALL{$n' \in \text{neighbours}(n) \setminus \{\epsilon, v\}$}
\STATE $p \leftarrow \text{shortest\_path}(\mathcal{G}, n, n')$
\IF{push($\Pi^*, \mathcal{G}, \text{robot}(\mathcal{G}, n), p, \mathcal{U}$)}
\STATE $\epsilon \leftarrow \epsilon \cup \{n\}$
\IF{$|\epsilon| = 2$} \RETURN \texttt{true} \ENDIF
\ELSEIF{$|\epsilon| = 1$} \BREAK \ENDIF
\ENDFOR
\ENDFOR
\STATE $p \leftarrow \text{shortest\_path}(\mathcal{G}, v, v')$
\IF{push($\Pi^*, \mathcal{G}, \text{robot}(\mathcal{G}, v), p, \epsilon$)}
\STATE $v'' \leftarrow \text{vertex held by robot that formerly held } v'$
\STATE $\mathcal{U} \leftarrow \{v, v', v'', \epsilon\}$
\FORALL{$n \in \text{neighbours}(v) \setminus \{\epsilon, v'\}$}
\FORALL{$n' \in \text{neighbours}(\epsilon) \setminus v$}
\STATE $p \leftarrow \text{shortest\_path}(\mathcal{G}, n, n')$
\IF{push($\Pi^*, \mathcal{G}, \text{robot}(\mathcal{G}, n), p, \mathcal{U} \cup \{n\}$)}
\STATE \text{move \text{robot}(\mathcal{G}, v') to } v, \text{robot}(\mathcal{G}, v'') \text{ to } v'$
\RETURN \texttt{true} \ENDIF
\ENDFOR
\ENDFOR
\ENDIF
\ELSEIF{$|\epsilon| = 1$} \BREAK \ENDIF
\RETURN \texttt{false}
\end{algorithmic}
Resolve

Whenever an agent in $U$ is swapped off its goal position, it needs to be returned to its goal position, since when returning from the swap operation, no agents in $U$ other than $r$ and $s$ should be moved. $r$ is the agent that is advancing towards its goal position along $p^*$. Agent $s$ is the “resolving” agent that wants to return to its goal position, and agent $r$ has just swapped with $s$ and is now occupying the goal position $T[s]$ of $s$.

Algorithm 3.1.5 resolve($\Pi^*, G, A, T, U, p^*, r, s$)

1: $t =$ vertex in $p^*$ after $A[r]$; $U' \leftarrow \{U \cup \{A[s]\}\}\{T[s]\}$
2: $p =$ shortest_path($G, A[r], t$)
3: if push($\Pi^*, G, A, r, p, U'$) then
4:     move $s$ from $A[s]$ to $T[s]$
5:     return true
6: else
7:     $r' = r, p' \leftarrow \{A[s], T[s]\}$
8:     while swap($\Pi^*, G, A, T, r', U'$) do
9:         if $A[s] = T[s]$ or push($\Pi^*, G, A, s, p', U'$) then
10:            return true
11:        else if $A[r'] = T[r']$ then
12:            $U' \leftarrow U' \cup \{T[r']\}$
13:            $r' =$ robot $r'$ just swapped with
14:            $p' =$ shortest_path($G, A[r'], T[r']$)
15:            return resolve($\Pi^*, G, A, T, U', p', r', s$)
16:     end if
17: end while
18: end if
19: return false

The resolve operation (algorithm 3.1.5) attempts to return agent $s$ to its goal position by moving agent $r$ towards its goal position $T[r]$ and attempting to push agent $s$ back to goal position $T[s]$ each time agent $r$ advances a step towards its goal. Initially, the resolve operation only tries to push agent $r$ further towards its goal in order to clear $T[s]$, by pushing $r$ along the shortest path $p$ from $A[r]$ to $t$. There seems to be no good reason that shortest_path is used, since $A[r]$ and $t$ should be adjacent to each other. It would seem that the way the two-vertex path is constructed on line 7 would also suffice on line 2.

If the push does not work, swap operations are used to advance agent $r$ towards its goal. After each successful swap, an attempt is made to return agent $s$ to its goal by pushing it along $p'$. If agent $r$ reaches its goal, but agent $s$ is still unable to move to its goal position, a new agent $r'$ is selected and the resolve process of agent $s$ continues with agent $r'$ instead of agent $r$ with the tail-call on line 15.

Note that the resolve operation invokes the swap operation, and that the swap operation in its turn invokes the resolve operation.
3.1.3 Smooth

The Push and Swap algorithm is capable of outputting redundant moves. This may happen in the following example shown in figure 3.7: agents $r$ and $s$ will be swapping positions, after which agent $t$ also moves into its goal vertex. In order for agents $r$ and $s$ to swap, two vertices adjacent to the only vertex with degree $\geq 3$ must be cleared. Agent $t$ is in one of these vertices and will be pushed away, into its goal position $T[t]$ by the clear operation. Since the moves that are generated by the clear operation will be executed in reverse after the exchange of position between agents $r$ and $s$ has been completed, agent $t$ will be moved back to its initial position. After the swap, agent $t$ will visit its goal vertex $T[t]$ for the second time. Clearly, the final two moves of the sequence shown in the figure are redundant.

The redundancy is detected by smooth (algorithm 3.1.6) as follows: if an agent visits a vertex twice, while no other agent visits this vertex in between, then all the moves of this agent from the first visit (exclusive) to the next visit (inclusive) are redundant and can be removed. Once some moves are removed, it is possible that additional moves can now also match this criteria.
Algorithm 3.1.6 smooth($\Pi$)

1: removed = true
2: while removed = true do
3:     removed = false
4:     for all $\pi \in \Pi$.reverse() do
5:         $r = \text{robot}(\pi), v =$ last vertex in $\pi$
6:         $\pi' \leftarrow$ next path in $\Pi$.reverse() containing $v$
7:         if $\pi' \neq \emptyset$ and $r = \text{robot}(\pi')$ then
8:             for all $\pi'' \in \Pi(\pi', \pi]$ do
9:                 if $\text{robot}(\pi'') = r$ then
10:                    remove ($\pi''$) from $\Pi$
11:             end if
12:         end for
13:         remove portion of $\pi'$ after $v$
14:     end if
15: end while
16: return $\Pi$
3.2 Analysis

There are several issues concerning the completeness of the Push and Swap algorithm. The completeness proof given in [15] is based on several lemmas that are incorrect.

- Lemma 4.3: `clear` considers all essential cases when evacuating two vertices in the neighbourhood of a vertex \( v \) for the purposes of swapping robots \( r \) and \( s \) at \( v \). If `clear` fails, then freeing \( v \)'s neighbourhood for `swap` is not possible.

- Lemma 4.4: A multi-robot path planning problem is solvable if and only if `swap` can bring robots \( r \) and \( s \) to a vertex \( v \) with degree \( \geq 3 \) along with two empty vertices.

Another issue concerns the recursive nature of the `resolve` operation. All these issues will be further explained in the following subsections.

3.2.1 Polygons

Consider the instance shown in figure 3.8. First note that this instance can be solved by moving both agents either clockwise or counter-clockwise, one agent for one step and the other agent \( n - 1 \) steps. This type of graph is called a polygon in the pebble motion literature. The graph does not contain any nodes of degree 3 or more, which means that a `swap` operation cannot possibly succeed. When solving this instance, Push and Swap will try to perform a `swap` operation, and hence will fail to solve the solvable instance with \( m \leq n - 2 \).

![Figure 3.8: Example in which no swap operation is possible.](image)

Push and Swap will attempt to solve this instance as follows. Since the example is symmetrical, any agent can be chosen first without loss of generality. Let agent \( r \) be the first agent that is sent to its destination. Agent \( r \) is sent along the shortest path between its current position and its goal position. First a `push` operation is attempted, which will push the other agent \( s \) out of the way. Agent \( r \) is now at its destination vertex, and is added to \( U \). The state of the algorithm is shown in figure 3.9. Next the other agent \( s \) is sent to its destination along the shortest path. This shortest path will go through the current position of \( r \), which is now in \( U \). This means that a `push` operation will not be able to move \( r \). Now the `swap` operation is executed, but this operation will also fail since there are no vertices with sufficiently high degree. Once this `swap` operation returns false, the algorithm terminates without having found a solution.

This example contradicts the part of the completeness proof of Push and Swap where it is claimed that a `swap` operation will always succeed if there is a solution. The reasoning behind this claim is that two agents need to switch relative positions on the path from the
position of an agent to its goal vertex in order to solve the problem. It is clear that the two agents can swap position, where the \texttt{swap} operation does not find a way to do it.

At first glance, there are two possible solutions to this problem. First, the algorithm could not only consider the shortest path to the destination vertex, but also different paths. The other option is that the \texttt{swap} operation should consider additional options for swapping the positions of two agents, specifically when there are no vertices with sufficiently high degree.

The author of Push and Swap suggests planning an additional “shortest” path [13]. When sending an agent to its destination, first an attempt is made to find a shortest path that considers all the agents in \( \mathcal{U} \) as obstacles. If this fails, the original version of the shortest path algorithm is executed. An additional advantage of this shortest path may be that it also requires fewer steps to move along a path without having to use the \texttt{swap} operation. With this extension, the Push and Swap algorithm will be able to correctly solve all polygon instances.

If this problem on polygons is just a symptom of a problem that may occur also in larger instances, it is not so easy to see that searching one additional shortest path will completely solve the incompleteness problem of Push and Swap. Since the polygon is a well-known example of a hard to solve class of instances for the pebble motion problem [10], in which the problem seems to exist only in polygons, it may be safe to assume that solving the problem for polygon instances will solve the overall problem.

### 3.2.2 Isthmuses

Figure 3.10 shows an instance that the Push and Swap algorithm may fail to solve. Whether or not Push and Swap solves the instance, depends on the order in which the agents are sent to their destinations. It is possible to solve the instance by moving \( a_1 \) away from the “junction” and then letting agents \( a_2 \) and \( a_3 \) swap on that vertex. Note that the instance meets the requirements for the completeness of Push and Swap, since there are two unoccupied vertices in the graph.

If the Push and Swap algorithm starts with agent \( a_1 \), it will simply be added to \( \mathcal{U} \), since it is already at its goal vertex. Now one of the other agents will be sent to its destination vertex. Because of the symmetry in the instance, we assume without loss of generality that agent \( a_2 \) is the next agent to be sent to its destination vertex. The agent \( a_2 \) needs to pass through the vertex of \( a_1 \) in order to reach its destination vertex. Since \( a_1 \) is in \( \mathcal{U} \), the push operation is not allowed to move it, and the \texttt{swap} operation will be executed. When looking carefully at the instance, it should become clear that it is entirely impossible for agent \( a_1 \) to
reach the vertices in which \(a_2\) and \(a_3\) start. The result of a successful swap between \(a_2\) and \(a_1\) will be that \(a_1\) occupies the initial position of \(a_2\). This result would contradict the fact that it is impossible for \(a_1\) to reach that vertex. Hence a successful swap is not possible and the Push and Swap algorithm will fail, contradicting the lemma 4.4.

In the pebble motion literature, the idea of an isthmus is defined [10]. This is a part of the graph that has such low degree that, in combination with the number of unoccupied vertices in the graph, makes it impossible for agents to cross from one side of this “bridge” to the other side. This means that agents on one side of an isthmus are constrained to stay on that side. Agents may also be trapped on the isthmus itself.

The instance shown in figure 3.10 does contain an isthmus between two parts of the graph. While agents are restricted by an isthmus to what parts of the graph they can reach, it is often possible for agents to reach the vertex on the edge of another part of the graph that connects it to the part of the graph to which the agent is constrained. Whenever an agent has reached its goal vertex on the “edge” of a subgraph into which it cannot move, it will be impossible to swap it with agents in this subgraph. If the shortest path of an agent in this subgraph happens to pass through this vertex, the Push and Swap algorithm will attempt to swap the two agents and it will fail.

A solution to the example instance is to delay the planning of agent \(a_1\) until all other agents are done. This will make sure that agent \(a_1\) can be pushed out of the way to enable agents \(a_2\) and \(a_3\) to swap. Generalizing this solution to all instances: all agents that have their goal vertex on the edge of a subgraph, should be planned at a late stage in the algorithm, allowing them to be pushed by other agents. This idea will be further explained in the next chapter.

![Figure 3.10: Even though agent \(a_1\) is unable to swap positions with the other agents, this instance is solvable.](image)

### 3.2.3 Recursive resolve

The `resolve` operation is executed when a `swap` operation removes an agent that is in \(U\) from its goal position. It is possible that the `resolve` operation needs to execute another `swap` operation which will lead to recursive behaviour of this operation. Figure 3.11 shows an instance in which this recursive behaviour occurs. This example requires agent \(a_8\) to be the first to be sent to its destination. In the execution of the Push and Swap algorithm, it will enter a state in which agent \(a_6\) is resolved for the second time, meaning that it can no longer move to its goal vertex in one step. This appears to be in invalid state for the Push
and Swap algorithm to be in, since the \texttt{resolve} pseudocode seems to assume that an agent can reach its goal vertex in one step, judging from the following lines of pseudocode:

- line 4: move $s$ from $A[s]$ to $T[s]$, and
- line 7: ..., $p^r \leftarrow \{A[s], T[s]\}$.

![Diagram](image)

Figure 3.11: Example instance in which \texttt{resolve} is recursively executed many times, leading to an invalid state. The pink vertex indicates the position of the agent ($a_8$) that is next to be sent to its destination.

The execution of the Push and Swap algorithm is shown in figures 3.12-3.14. First the agent $a_8$ moves to its destination, leaving agent $a_6$ resolving from the original position of $a_8$. Once $a_8$ is at its goal vertex, agent $a_6$ still cannot move to its goal position and agent $a_{11}$ will be sent to its goal position. Agent $a_{11}$ will swap with agent $a_3$, which will then be resolving. This process continues, until the point where agent $a_6$ is swapped for the second time. Now $a_6$ requires two steps to reach its goal position, while the algorithm description clearly assumes that its goal position should be reachable in one step.

The \texttt{resolve} operation works when the agent that is being resolved is the only agent that wants to go to its goal vertex. This means that any agent that is in the way of this agent can be sent to its goal, which will make it move away from the goal of the resolving agent. When an already resolving agent is swapped for the second time, it is unclear how the algorithm should proceed. At first it seems that the \texttt{resolve} operation should not be executed again, but in this case the \texttt{resolve} operation would move an agent in $U$, while not updating $U$. Either way, the algorithm does not appear to be designed to handle this situation.

A solution to this circular resolving situation becomes clear when looking at the state of the agents shown in figure 3.13f. All the agents want each others positions in the cycle. Performing a “rotate” operation would move all the resolving agents and the currently
3.2 Analysis

Push and Swap

Figure 3.12: (a.) The agent $a_8$ is the first to be sent to its destination vertex. In order to get there, it has just swapped with agent $a_6$. The resolve operation will proceed to bring $a_8$ to its goal, swapping it with agent $a_{11}$. (b.) When agent $a_8$ is at its goal position, agent $a_6$ is still unable to push back to its goal position, hence the resolve operation will send agent $a_{11}$ to its goal, first swapping it with agent $a_3$. (c.) Now agent $a_3$ will resolve. Yellow vertices indicate the positions of agents that are resolving.

Figure 3.13: (d.) Agent $a_{11}$ swaps with agent $a_9$ to reach its goal vertex. Since agent $a_3$ is unable to push back to its goal vertex, agent $a_9$ will now be sent to its goal vertex. In order to reach its goal vertex, agent $a_9$ has to swap agent $a_2$ away from its goal vertex. (e.) Now agent $a_2$ is also resolving. (f.) Once agent $a_9$ has reached its goal position, $a_{10}$ will be sent to its goal position. Agent $a_{10}$ needs to swap with the already resolving agent $a_6$ in order to reach its goal position.
Figure 3.14: (g.) Once agent $a_6$ has been swapped, it is more than one step away from its goal vertex. (h. and i.) Another problem is that the agent that is in the way of agent $a_6$ (which is agent $a_{12}$) is at its goal position, and will not move out of the way when it is sent to its goal. Finally there is no agent left that can be chosen to send to its destination.

planning agent to the required positions in one operation. It turns out that it is possible to perform such a rotate operation, as shown in our lemma 4.3 in the next chapter. To apply the rotate operation at the appropriate time, the algorithm needs to keep track of where the agents are that are currently resolving. This can be done by delaying the execution of the resolve operation. Details of this rotate operation and the way that the resolve operation should be delayed can be found in the next chapter.

3.2.4 Clear issue

The completeness proof of Push and Swap states that the clear operation considers all the required possibilities of clearing two neighbour vertices of a selected vertex. The instance in figure 3.15 shows a situation in which clear does not successfully clear two neighbour vertices of vertex $v$. It is possible to clear two neighbour vertices of $v$ by moving agent $a_2$ through vertex $e$ to vertex $e'$. In its original description, the clear operation does not consider this option.

When executing the clear operation, it first looks if there are any neighbour vertices already unoccupied. If there are two or more unoccupied neighbour vertices, the clear operation does not need to clear additional vertices and it will return true. In the case that there is one unoccupied vertex, this vertex is considered an obstacle throughout the execution of the rest of the operation. The clear operation does consider the option of moving an agent from one neighbouring vertex to the already cleared vertex and then trying to push it away in order to clear it. This would be the solution to the problem, but the clear operation only considers moving this agent through the vertex $v$, and it is not possible to clear $v$ by moving agents $a_1$ and $a_3$ backwards.

In some cases it is impossible to clear the $v$ vertex in order to move an agent through that vertex to the unoccupied vertex. The operation should consider the possibility of there being some link between two neighbour vertices of the $v$ vertex. Another case in which the
clear operation does not consider sufficient options for clearing the neighbourhood of \( v \) is shown in figure 3.16. Details of all the possibilities that the clear operation should consider are in the next chapter.

![Diagram of agents and vertices](image)

**Figure 3.15:** The clear operation will not be able to properly clear two neighbours of the node of \( v \).

![Diagram of agents and vertices](image)

**Figure 3.16:** Second case in which the clear operation does not consider all necessary options.

### 3.2.5 Smooth

The Push and Swap algorithm may sometimes output moves that are not required in the final solution. These occur for example in situations where agent \( t \) is moved towards its goal vertex while clearing neighbour vertices of some vertex \( v \), in order for two other agents \( r \) and \( s \) to swap. After agents \( r \) and \( s \) have successfully completed their swap at vertex \( v \), all the moves leading to the exchange (\texttt{execute_swap}) are executed in reverse (line 19 of algorithm 3.1.2) in order to restore the positions of all agents, including agent \( t \). In this case (figure 3.7 on page 18), agent \( t \) did not need to return to its initial vertex.

In order to remove these redundant moves, the smooth operation is designed to detect and remove them. The smooth operation looks at the situation where an agent returns to a vertex that it has visited before. If no other agents have visited this vertex in between, then the agent is free to remain at the vertex and all intermediate moves of this agent can be discarded. The simplest way to implement such an operation is to scan over the list of moves of each agent and check for each move if there is a corresponding move in that list that has the same destination vertex. If the moves in between these two moves all have different destination vertices, then all the intermediate moves (of this agent) can be removed. Once some moves have been removed, the algorithm has to start from scratch, since this may render additional moves redundant. The fictional example that is shown in figure 3.17, in which the agents \( a_1 \) and \( a_2 \) wander about without much of a goal other than to illustrate how smooth works, contains the following moves:

\[
(a_1 \rightarrow v_3), (a_1 \rightarrow v_1), (a_2 \rightarrow v_2), (a_2 \rightarrow v_3), (a_2 \rightarrow v_2), (a_1 \rightarrow v_3)
\]
First, the second and third moves of agent $a_2$ to $v_3$ and back to $v_2$ can be discarded, since no other agent visits $v_2$ in between it is free to remain at $v_2$. This leaves the following moves:

$\left( a_1 \rightarrow v_3 \right), \left( a_1 \rightarrow v_1 \right), \left( a_2 \rightarrow v_2 \right), \left( a_1 \rightarrow v_3 \right)$

Only now most of the moves of agent $v_1$ can be considered redundant, since agent $a_2$ now no longer visits vertex $v_3$. This results in the following final list of moves:

$\left( a_1 \rightarrow v_3 \right), \left( a_2 \rightarrow v_2 \right)$

The most straightforward way to implement the smooth operation, which is also the way it is presented along with the Push and Swap algorithm, is to perform a nested for-loop that looks in $O(n^2)$ time for a pair of moves $(\pi', \pi)$ for which the redundancy condition holds. Once a pair like this has been found, and removed, the entire loop must start over, since new redundancy may be discovered, as can be seen in the above example. In a naive implementation, this will lead to $O(n^3)$ time complexity. Since in this case $n$ is the output size, and from the analysis in section 4.3.2 (which also roughly applies to the Push and Swap algorithm) it can easily be seen that this $O(n^3)$ will be a dominant factor in the overall runtime of the algorithm.

The smooth operation can also be performed in $O(n)$ time. This can be done by using a doubly-linked data structure where each move is linked to the next move with the same destination vertex and the next move with the same agent. Details on this optimized smooth operation are in the next chapter.

### 3.2.6 Conclusion

The following issues were shown that contradict the completeness claim of Push and Swap:
• When an instance contains no vertices of degree $\geq 3$, the Push and Swap algorithm may attempt to (unsuccessfully) swap two agents while it is possible to find a solution in another way.

• A problem instance can be separated into several smaller subproblems where no agent can move from one subproblem to the other. These subproblems can interfere in such a way that the swap operation is executed on two agents that are unable to swap in an instance that is solvable.

• It is possible for the algorithm to reach a state in which it is unclear how to proceed. This can happen if the resolve operation is called twice on one agent in one recursion.

• Not all required cases are considered in the clear operation to clear the neighbours of a vertex $v$, contradicting Luna’s lemma 4.3.

• The runtime of the smooth operation has such bad performance, that it will be a dominant factor in the overall runtime of the Push and Swap algorithm.

Additionally, a naive implementation of the smooth operation would yield an unusable Push and Swap algorithm, because of its time complexity.

All of these issues will be solved in the next chapter, where we introduce the Push and Rotate algorithm that is still based on the principle of pushing and swapping agents. We will prove that this new algorithm is complete for the class of instances in which there are two unoccupied vertices.
Chapter 4

Push and Rotate

Push and Rotate is a revised version of the Push and Swap algorithm. Like the Push and Swap algorithm, it solves an instance using push and swap operations. Most of the operations have been altered in order to resolve the issues presented in section 3.2. A short overview of the changes to the various operations:

solve The push and swap operation has been split into the solve and plan operations.

plan The agent is now only moved ahead one step at a time, in order to check if a rotate is required. The resolve operation is executed at the end of the plan operation, instead of in the swap operation.

plan' This operation solves polygon instances.

push The agent is only pushed ahead by one step. Clearing the vertex, if required, is done by the clear vertex operation, instead of in the push function itself.

swap The resolve operation is no longer executed from the swap operation.

clear Additional possibilities are considered for clearing two neighbours of vertex \( v \).

resolve Instead of resolving only one agent, all agents on path \( q \) are resolved.

4.1 Problem decomposition

In order to detect and prevent certain states for which the swap operation is unable to succeed (as shown in section 3.2.2), some preprocessing can be done. An additional advantage of this preprocessing is that unsolvable instances can be detected before executing the main solving algorithm, saving some time. This preprocessing operation is based on work in pebble motion [10]. The result of the preprocessing will be an assignment of priorities to agents. When the agents are sent to their destinations in order of priority in the main solving algorithm, the algorithm is guaranteed to be complete, which is shown in section 4.3.1.

The preprocessing divides the problem into several subproblems \( G_i \). A subproblem is not the same as a subgraph. Agents and vertices from the problem instance can be assigned
to zero or one subproblems. Additionally, an agent initially occupying a vertex that is assigned to a subproblem may or may not be assigned to this subproblem (for example, in figure 4.9 agent $a_5$ is not assigned to subproblem $D$). A subproblem is a part of the multi-agent path planning problem, such that agents assigned to two different subproblems have minimal interaction with each other. Two agents that do belong to the same subproblem, will always be able to swap with each other, as shown in lemma 4.5. The subproblem that an agent is assigned to indicates which parts of the roadmap it is able to reach.

An agent that is assigned to subproblem $C_i$ will be unable to enter subproblem $C_j$ ($i \neq j$), with the exception of one single vertex in $C_j$ connecting the two subproblems. This vertex will be called on the “edge” of subproblem $C_j$. Since an agent assigned $C_i$ is unable to enter (most of) $C_j$, it is also incapable to swapping with agents assigned to $C_j$. If an agent in $C_i$ has its goal position on the edge vertex in $C_j$ that connects the components, then it should be planned after the agents in subproblem $C_j$, since otherwise the problem described in section 3.2.2 may occur, i.e. agents assigned to subproblem $C_i$ will attempt to swap with it.

Preprocessing is divided into three stages. The first stage (algorithm 4.1.1) assigns vertices to subproblems, the second stage (algorithm 4.1.2) assigns agents to subproblems, and finally these assignments are used to determine a priority relationship over the subproblems (algorithm 4.1.3). The order in which the agents need to be planned needs to satisfy the condition that agents assigned to $C_i$ need to precede agents assigned to $C_j$, if $C_i \prec C_j$. The agents that are not assigned to a subproblem need to be planned last, i.e. after all agents that are assigned to some subproblem.

**Algorithm 4.1.1 Division into subproblems**

1: $C \leftarrow$ all nontrivial biconnected components in $G$
2: $C \leftarrow C \cup$ vertices with degree at least 3
3: for all $C_i, C_j \in C$ do
4: \hspace{1em} if $(\min_{v \in C_i, u \in C_j} d(v, u)) \leq m - 2$ then
5: \hspace{2em} Replace $C_i$ and $C_j$ in $C$ with $C_k = C_i \cup C_j \cup$ shortest path$(u, v)$
6: \hspace{1em} end if
7: end for

The first stage starts by computing all nontrivial biconnected components (figure 4.1). Biconnected components are defined as components for which the remaining vertices remain connected when one of the vertices from the component is removed. Non-trivial biconnected components are those of size at least 3. These components can be obtained in linear time $O(|V| + |E|)$ [8]. All remaining vertices with degree at least 3 (figure 4.3) are added as components of size 1. At this point the algorithm considers 4 components (figure 4.2).

Let $m = |V| - |R|$ denote the number of unoccupied vertices. All pairs of components that have distance less than or equal to $m - 2$ will then be joined into one subproblem recursively, until all pairwise subproblems have a distance of at least $m - 1$ from each other. This distance is significant, since it is impossible for an agent to cross this distance and go from one subproblem to another. Considering the example,

- if $m = 2$, then the components shown in figure 4.2 will remain separate.
4.1 Problem decomposition

Figure 4.1: Decomposition of a problem: nontrivial biconnected components.

Figure 4.2: Decomposition of a problem: vertices with degree $\geq 3$.

Figure 4.3: Decomposition of a problem: the components, after adding the vertices with degree $\geq 3$.

Figure 4.4: Decomposition of a problem: nodes assigned to the subproblems in case there are 3 unoccupied vertices.
4.1 Problem decomposition

- If $m > 3$, then the components will all be joined into one single component.
- The assignment of nodes to subproblems for $m = 3$ is shown in figure 4.4.

Algorithm 4.1.2 Assigning agents to subproblems

```plaintext
1: for all $C_i \in C$ do
2:   for all $v \in C_i$ do
3:     if $\exists u \notin C_i$ for which $(u, v) \in G$ then
4:        $m' \leftarrow$ number of unoccupied vertices reachable from $v$ in $G \backslash \{u\}$
5:        $m'' \leftarrow$ number of unoccupied vertices reachable from $C_i$ in $G \backslash \{v\}$
6:     if $(m' \geq 1 \land m' < m) \lor m'' \geq 1$ then
7:        Assign agent on position $v$ to $C_i$ (if any)
8:     end if
9:     Follow path from $u$ away from $v$ and assign the first $m' - 1$ agents on this path
to $C_i$
10:    else
11:     Assign agent on position $v$ to $C_i$ (if any)
12:    end if
13: end for
14: end for
15: return Assignment of agents to $C$
```

Assignment of the agents to subproblems is based on the distance of the agent to the subproblem and the fact that all agents within the vertices of a subproblem can reach any configuration as long as one of the vertices in the subproblem is unoccupied [10]. The value of $m'$ indicates the number of additional unoccupied vertices that can be introduced into the subproblem, without moving the agents under consideration further away from the subproblem. Considering the vertices on the edge of the subproblem, the agents on these vertices belong to this subproblem only if it is possible to move an unoccupied vertex into the subproblem without moving the agent out of the subproblem, which is why $m''$ is also checked.

The assignment of agents to subproblems is illustrated in figures 4.5, 4.6, 4.7 and 4.8. The agents are moved as far as possible into the subproblem to indicate reachability: it is easy to verify that all agents assigned to the same subproblem are able to swap with each other, and agents assigned to different subproblems are not able to swap with each other.

For each of the vertices $v$ on the edge of each subproblem, the number of unoccupied vertices reachable from the subproblem is counted, not counting the vertices that are (only) reachable by exiting the subproblem from vertex $v$, and assigned to $m'$. If $m'$ is not zero and also not $m$, then the agent (if any) occupying $v$ is assigned to the subproblem. Next, the path away from the subproblem is followed, and the first $m' - 1$ agents encountered along this path are assigned to the subproblem, since these agents are able to enter the subproblem leaving an additional unoccupied vertex available for movement.

The purpose of the preprocessing is to detect and prevent situations where an agent has reached its goal position that is in a subproblem to which the agent is not assigned. These
Figure 4.5: Decomposition of a problem: the agents assigned to the yellow (leftmost) subproblem are $a_1, a_2, a_3$ and $a_4$.

Figure 4.6: Decomposition of a problem: the agents assigned to the red (middle) subproblem are $a_5, a_6, a_7$ and $a_8$.

Figure 4.7: Decomposition of a problem: the agents assigned to the red (middle) subproblem are $a_5, a_6, a_7$ and $a_8$.

Figure 4.8: Decomposition of a problem: the agents assigned to the blue (rightmost) subproblem are $a_9$ and $a_{10}$.
agents can only reach the vertices on the edge of a subproblem, hence only these vertices need to be checked. Additionally an agent may be pushed into an edge position and locked there by other agents that do not have their goal positions in the subproblem.

The agents that do not belong to any subproblem will, for the sake of simplicity, be planned last. If an agent from subproblem $C_j$ either

1. has its goal position on the edge of subproblem $C_i$, or

2. will push another agent to the edge of subproblem $C_i$ and lock it there,

then the priority relation $C_i \prec C_j$ is added, which means that agents from subproblem $C_i$ should be planned before the agents from subproblem $C_j$. If both relations $C_i \prec C_j$ and $C_j \prec C_i$ hold, then two agents from two different subproblems need to swap relative positions, which is impossible and the problem cannot have a solution.

### 4.1.1 Propagation of priorities

Consider the special case in figure 4.9. Note that the assignment of the agents to subproblems is as follows: $A \leftarrow [a_1, a_7]$, $B \leftarrow [a_6, a_{10}]$, $C \leftarrow [a_2, a_3, a_4, a_5]$ and $D \leftarrow [a_8, a_9]$. There exists a priority relation between $C$ and $D$: $C \prec D$. The priority relation exists because in order for agents $a_8$ and $a_9$ to switch positions, agent $a_5$ needs to be pushed away.

Only prioritizing the agents according to the priority relation will not be sufficient to solve the instance. If subproblems $A$ and $B$ get solved before $C$, then the agents in subproblems $A$ and $B$ will block all the free vertices from the rest of the graph. This can be solved by propagating the priority relation through the entire tree of subproblems.
4.2 Push and Rotate

The \texttt{solve} operation (algorithm 4.2.1) takes the roadmap $\mathcal{G}$, start positions $\mathcal{S}$ and goal positions $\mathcal{T}$ as input and returns a list of moves $\Pi$ that will take all agents from their start positions to their goal positions. If this is not possible, like all other operations that can fail, it returns false.

Each agent takes its turn, according to the priorities assigned from the problem decomposition, moving along the shortest path from their current position to their goal position. This is done by the \texttt{plan} operation (algorithm 4.2.2). Throughout all the operations, the position of the agents that are already on their goal position will be preserved or restored. The agents that have already been sent to their goal are added to the set $\mathcal{F}$.

If the roadmap $\mathcal{G}$ is a polygon, which is easily checked ($\forall v \in V : \deg(v) = 2$), then the \texttt{plan*} operation (algorithm 4.2.3) should be used instead of the \texttt{plan} operation.

Consider a heavily congested shortest path $p$ of agent $r$ to its goal position ($\mathcal{T}[r]$). In order to move along this path, a number of \texttt{swap} operations is required. This is especially the case where the path is congested by agents (in $\mathcal{F}$) that are already at their goal positions, since the other agents may be pushed out of the way. The result of a \texttt{swap} operation is that

![Diagram](image-url)
Algorithm 4.2.1 solve($G, S, T$)

1: \( \Pi \leftarrow [ ] \)
2: \( \mathcal{A} \leftarrow S \)
3: \( \mathcal{F} \leftarrow \emptyset \)
4: for all agents \( r \in \mathcal{R} \) do
5: \hspace{1em} if \( \forall v \in V : \text{degree}(v) = 2 \) then
6: \hspace{2em} if \( \text{plan}^*(\Pi, G, \mathcal{A}, T, r, \mathcal{F}) = \text{false} \) then
7: \hspace{3em} return \text{false}
8: \hspace{1em} end if
9: \hspace{1em} else
10: \hspace{2em} if \( \text{plan}(\Pi, G, \mathcal{A}, T, r, \mathcal{F}, [ ]) = \text{false} \) then
11: \hspace{3em} return \text{false}
12: \hspace{1em} end if
13: \hspace{1em} end if
14: end for
15: return \( \Pi \)

Algorithm 4.2.2 plan($\Pi, G, \mathcal{A}, T, r, \mathcal{F}, q$)

1: \( p \leftarrow \text{shortest path in } G \text{ from } \mathcal{A}[r] \text{ to } T[r] \)
2: \( \text{while } \mathcal{A}[r] \neq T[r] \) do
3: \hspace{1em} \( v \leftarrow \text{vertex after } \mathcal{A}[r] \text{ on } p \)
4: \hspace{1em} if \( v \in q \) then
5: \hspace{2em} \text{rotate}(\Pi, G, \mathcal{A}, q, v)
6: \hspace{1em} else
7: \hspace{2em} if \( \text{push}(\Pi, G, \mathcal{A}, r, v, \mathcal{A}[\mathcal{F}]) = \text{false} \) then
8: \hspace{3em} if \( \text{swap}(\Pi, G, \mathcal{A}, r, \mathcal{A}^{-1}[v]) = \text{false} \) then
9: \hspace{4em} return \text{false}
10: \hspace{2em} end if
11: \hspace{1em} end if
12: \hspace{1em} end if
13: \( q \leftarrow q + [v] \)
14: end while
15: \( \mathcal{F} \leftarrow \mathcal{F} \cup \{r\} \)
16: return \text{resolve}(\Pi, G, \mathcal{A}, T, q, \mathcal{F})
Algorithm 4.2.3 plan\((\Pi, G, \mathcal{A}, \mathcal{T}, r, \mathcal{F})\)

1: \( p \leftarrow \) shortest path in \( G \setminus \mathcal{A}[\mathcal{F}] \) from \( \mathcal{A}[r] \) to \( \mathcal{T}[r] \)
2: if \( p = \) false then return false end if
3: while \( \mathcal{A}[r] \neq \mathcal{T}[r] \) do
4: \( v \leftarrow \) next vertex on \( p \)
5: if push\((\Pi, G, \mathcal{A}, r, v, \mathcal{A}[\mathcal{F}])\) = false then
6: return false
7: end if
8: end while
9: \( \mathcal{F} \leftarrow \mathcal{F} \cup \{r\} \)
10: return true

the current agent \((r)\) moves forwards along \( p \), and that the swapped agent moves backwards along \( p \). Note that the swap operation does not execute the resolve operation anymore, but the entire path \( p \) is added to \( q \), and \( q \) is resolved at the end of the plan operation. Sending an agent along path \( p \) to its goal leaves a string of “resolving” agents on \( q \) that want to move back forwards along \( q \).

Now if the next vertex on \( p \) intersects \( q \), there is a cycle in \( q \). In this cycle each agent either wants to move forward, or is indifferent to moving away from its vertex (i.e. it is not in \( \mathcal{F} \)). This move is executed by the rotate operation.

Move operation

The specific move operation is dependent on implementation choices, hence it is not specified like the other operations. For the algorithm to work, the move operation, when moving agent \( r \) to vertex \( v \), must update at least the following:

- \( \mathcal{A} \) should reflect the new position of agent \( r \), and
- the move should be appended to the list of moves \( \Pi \).

In order for the reverse operation to work, the information that is recorded in \( \Pi \) should include the original vertex of agent \( r \).

Algorithm 4.2.4 push\((\Pi, G, \mathcal{A}, r, v, \mathcal{U})\)

1: if vertex \( v \) is occupied then
2: \( \mathcal{U}' \leftarrow \mathcal{U} \cup \{\mathcal{A}[r]\} \)
3: if clear_vertex\((\Pi, G, \mathcal{A}, v, \mathcal{U}')\) = false then
4: return false
5: end if
6: end if
7: move\((\Pi, \mathcal{A}, \) agent \( r \) to vertex \( v)\)
8: return true
4.2 Push and Rotate

4.2.1 Push operation

Note that in the Push and Swap algorithm, the push operation would attempt to push an agent all the way along \( p \), if possible. The new push operation (algorithm 4.2.4) moves agent \( r \) only one step forward along path \( p \), if possible. When necessary, it clears the next vertex on path \( p \) using the clear_vertex operation. When clear, agent \( r \) moves to this vertex. This operation makes sure that no agent in \( \mathcal{U} \) is moved. The push and clear_vertex operations are illustrated in figure 4.10.

**Algorithm 4.2.5 clear_vertex(\( \Pi, \mathcal{G}, \mathcal{A}, v, \mathcal{U} \))**

1: for all unoccupied vertex \( u \in \mathcal{G} \) do
2: \( p \leftarrow \) shortest path in \( \mathcal{G} \setminus \mathcal{U} \) from \( u \) to \( v \)
3: if \( p \neq \) false then
4: \( x' \leftarrow \) false
5: for all vertices \( x \) on path \( p \) (in order) do
6: if \( x' \neq \) false then
7: \( r \leftarrow \mathcal{A}^{-1}(x) \)
8: \( \text{move}(\Pi, \mathcal{A}, \text{agent} \ r \text{ to vertex} \ x') \)
9: end if
10: \( x' \leftarrow x \)
11: end for
12: return true
13: end if
14: end for
15: return false

Clear vertex operation

In order to clear vertex \( v \), the clear_vertex operation (algorithm 4.2.5) finds the shortest path from \( v \) to an unoccupied vertex \( u \). It moves all the agents on this path in the direction of \( u \), thus clearing \( v \). When looking for the shortest path, only the part of the roadmap is considered that does not contain vertices in \( \mathcal{U} \), i.e. the vertices in \( \mathcal{U} \) are considered obstacles. The clear_vertex operation is illustrated in figures 4.10a and 4.10b.
4.2 Push and Rotate

Algorithm 4.2.6 \( \text{swap}(\Pi, G, A, r, s) \)

1. \( S \leftarrow \{ \text{vertex } x \in G \mid \text{degree}(x) \geq 3 \} \)
2. for all vertex \( v \in S \) do
3. \( A' \leftarrow A \)
4. \( \Pi' \leftarrow [] \)
5. if multipush(\( \Pi', G, A', r, s, v \)) = true then
6. if clear(\( \Pi', G, A', r, s, v \)) = true then
7. \( \Pi \leftarrow \Pi + \Pi' \)
8. \( A \leftarrow A' \)
9. exchange(\( \Pi, G, A, r, s, v \))
10. reverse(\( \Pi, A, \Pi'_{r/s} \))
11. return true
12. end if
13. end if
14. end for
15. return false

4.2.2 Swap operation

The swap operation (algorithm 4.2.6) has not changed much from the version in the Push and Swap algorithm. The operation attempts to swap agent \( r \) with the agent \( s \) on the vertex that is next on path \( p \). In order to perform this swap, it attempts to create a state (see figure 4.11a) where agents \( r \) and \( s \) are on \( v \) and a neighbour of \( v \), and two neighbours of \( v \) are unoccupied. When this state has been reached, the exchange operation can swap the positions of agents \( r \) and \( s \) (as shown in figure 4.11).

Until the desired state has been reached, an attempt is made with each vertex of degree at least 3. First the agents \( r \) and \( s \) are moved to vertex \( v \), pushing all blocking agents out of the way. Since these moves will be reversed later, it is not required to take \( F \) into account. When agent \( r \) or \( s \) is on vertex \( v \), the clear operation is used to clear two neighbours of \( v \). When the clear operation succeeds, the required state for an exchange operation has
been reached and the swap will be executed. After the exchange, the moves that were used to reach the required state are executed in reverse. The result is that all agents are returned to their positions prior to the execution of swap, with the exception of agents $r$ and $s$, which have swapped position.

**Algorithm 4.2.7** multipush($\Pi, G, A, r', s', v$)

```plaintext
1: $r \leftarrow$ agent $r'$ or $s'$ closest to $v$
2: $s \leftarrow$ the other agent
3: $p \leftarrow$ shortest path in $G$ from $A[r]$ to $v$
4: if $p = \text{false}$ then
5: return false
6: end if
7: for all vertices $x$ on path $p$ do
8: $v_r \leftarrow A[r], v_s \leftarrow A[s]$
9: if vertex $x$ is occupied then
10: $U' \leftarrow \{v_r, v_s\}$
11: if clear_vertex($\Pi, G, A, x, U'$) = false then
12: return false
13: end if
14: end if
15: move($\Pi, A$, agent $r$ to vertex $x$)
16: move($\Pi, A$, agent $s$ to vertex $v_r$)
17: end for
18: return true
```

**Multipush operation**

The multipush operation (algorithm 4.2.7) works very similar to the push operation, with the exception that it does not take the agents that are already at their goal positions in $\mathcal{F}$ into account. This is not necessary, since the swap operation will execute the resulting moves in reverse after the swap has been completed. In order to push agents $r$ and $s$ to vertex $v$, the clear_vertex operation is used to clear each node on the path, if required. Compared to
the use of clear_vertex in the push operation, another set \( \mathcal{U}' = \{ A[r], A[s] \} \) is passed as a parameter. This results in the ability to move agents that are in \( \mathcal{F} \).

### Clear operation

The clear operation (algorithm 4.2.8) works in four stages. In the first stage (figure 4.12(a.)) it simply attempts to push agents, that are occupying neighbours of \( v \), away from \( v \). If there are two vertices cleared in this stage, the operation is done. Otherwise, the other three stages need one unoccupied vertex next to \( v \) to work. If there is no unoccupied vertex after stage one, the clear operation fails.

![Figure 4.12: The stages of the clear operation.](image)

In the second stage (figure 4.12(b.)) a neighbour \( n \) is required that has a path to the already empty neighbour \( \varepsilon \) that does not go through \( v \). If this vertex \( n \) is able to clear when the vertex \( \varepsilon \) is not blocked (i.e. not passed in the \( \mathcal{U} \) parameter to clear_vertex), then this vertex has the required path, and some agent will have been pushed into \( \varepsilon \) (note that vertex \( n \) was not able to clear in the first stage). Next, an attempt is made to clear the vertex \( \varepsilon \) again, keeping \( n \) blocked. If this succeeds, then two vertices \( n \) and \( \varepsilon \) have been cleared. After finding one \( n \) that has the required path (and failing to clear it), no other neighbours of \( v \) need to be checked in this stage, since other neighbours will yield the same result.

Agents occupying neighbours of \( v \) may be able to vacate these vertices through the vertex \( v' \) that agent \( s \) is occupying. The third stage (figure 4.13(c.)) checks if this is possible, by moving agent \( r \) into the empty vertex \( \varepsilon \), and agent \( s \) into \( v \). If any neighbour \( n \) of \( v \) has a path to \( v' \) that does not go through \( v \), then it will now be able to clear. Only one such neighbour needs to be checked, because any other neighbour with the required path will yield the same result. Next, the vertex \( v' \) should be occupied, since the neighbour \( n \) previously (stage one) was unable to clear. If the vertex \( v' \) can be cleared, while keeping vertex \( n \) unoccupied, then these two vertices are the required unoccupied neighbours of \( v \).

The idea behind the final stage (figure 4.13(d.)) is that it may be possible to create additional space behind the already empty vertex \( \varepsilon \). To use this space, an agent from an occupied neighbour \( n \) of \( v \) is moved to \( \varepsilon \) and then this vertex is cleared with the clear_vertex operation. In order for the agent at \( n \) to move to \( \varepsilon \), it needs to go through vertex \( v \), since no other link was found in stage 2 of the operation. To make way, the agents \( r \) and \( s \) first need to be pushed away from \( v \). When vertex \( \varepsilon \) is successfully cleared a second time, two vertices are empty, which are \( n \) and \( \varepsilon \).
Algorithm 4.2.8 clear($\Pi, G, A, r', s', v$)

1: $r \leftarrow$ agent $r'$ or $s'$ on $v$; $s \leftarrow$ agent $r'$ or $s'$ not on $v$; $v' \leftarrow A(s)$
2: $E \leftarrow \{\text{unoccupied } n \in \text{neighbours}(v)\}$
3: if $|E| \geq 2$ then return true end if
4: for all $n \in \text{neighbours}(v) \setminus (E \cup \{v'\})$ do
5: if clear_vertex($\Pi, G, A, n, E \cup \{v, v'\}$) = true then
6: if $|E| \geq 1$ then return true end if
7: $E \leftarrow E \cup \{n\}$
8: end if
9: end for
10: if $|E| = 0$ then return false end if
11: $\varepsilon \leftarrow$ the vertex in $E$
12: for all $n \in \text{neighbours}(v) \setminus \{v', \varepsilon\}$ do
13: $\Pi' \leftarrow [\cdot] ; A' \leftarrow A$
14: if clear_vertex($\Pi', G, A', n, \{v, v'\}$) = true then
15: if clear_vertex($\Pi', G, A', \varepsilon, \{v, v', n\}$) = true then
16: $A \leftarrow A' ; \Pi \leftarrow \Pi + \Pi'$
17: return true
18: end if
19: break
20: end if
21: end for
22: for all $n \in \text{neighbours}(v) \setminus \{v', \varepsilon\}$ do
23: $\Pi' \leftarrow [\cdot] ; A' \leftarrow A$
24: move($\Pi', A'$, agent $r$ to vertex $\varepsilon$) ; move($\Pi', A'$, agent $s$ to vertex $v$)
25: if clear_vertex($\Pi', G, A', n, \{v, \varepsilon\}$) = true then
26: if clear_vertex($\Pi', G, A', \varepsilon, \{v, v', \varepsilon\}$) = true then
27: $A \leftarrow A' ; \Pi \leftarrow \Pi + \Pi'$
28: return true
29: end if
30: break
31: end if
32: end for
33: if clear_vertex($\Pi, G, A, v', \{v\}$) = false then return false end if
34: move($\Pi, A$, agent $r$ to vertex $v'$)
35: if clear_vertex($\Pi, G, A, \varepsilon, \{v, v', A[s]\}$) = false then return false end if
36: $n \leftarrow$ any vertex from neighbours($v$) \setminus \{v', \varepsilon\} , $t \leftarrow A^{-1}(n)$
37: move($\Pi, A$, agent $t$ through vertex $v$ to vertex $\varepsilon$)
38: move($\Pi, A$, agent $r$ to vertex $v$) ; move($\Pi, A$, agent $s$ to vertex $v'$)
39: return clear_vertex($\Pi, G, A, \varepsilon, \{v, v', n\}$)
Figure 4.13: The stages of the clear operation.

Algorithm 4.2.9 \texttt{exchange}(\Pi, G, A, r', s', v)

\begin{algorithm}
\begin{enumerate}
\item $r \leftarrow \text{agent $r'$ or $s'$ on } v$
\item $s \leftarrow \text{agent $r'$ or $s'$ not on } v$
\item $(v_1, v_2) \leftarrow \text{two unoccupied neighbours of } v$
\item $v_s \leftarrow A[s]$
\item \texttt{move}(\Pi, A, \text{agent } r \text{ to vertex } v_1)$
\item \texttt{move}(\Pi, A, \text{agent } s \text{ through vertex } v \text{ to vertex } v_2)$
\item \texttt{move}(\Pi, A, \text{agent } r \text{ through vertex } v \text{ to vertex } v_s)$
\item \texttt{move}(\Pi, A, \text{agent } s \text{ to } v)$
\end{enumerate}
\end{algorithm}

Exchange operation

The \texttt{exchange} operation (algorithm 4.2.9) is fairly simple. Let the agent occupying $v$ be called $r$ and the other agent be called $s$. Agent $r$ (which is on $v$) is first moved to one empty neighbour of $v$. Then the other agent $s$ is moved into the other empty neighbour. Agent $r$ that started on $v$ is now free to move to the vertex ($v_s$) that agent $s$ initially occupied. After this, agent $s$ can move to $v$. This operation is illustrated in figure 4.11 (on page 40).

Reverse operation

The \texttt{reverse}($\Pi, A, \Pi'_{r/s}$) operation reverses all the moves in the sequence $\Pi'$. This means that the moves $\pi \in \Pi'$ are added to $\Pi$ in reverse order, with the following changes:

- the agents move to the original vertex of $\pi$, and
- moves of agents $r$ are now moves of agent $s$, and vice versa.
Algorithm 4.2.10 rotate($\Pi, G, A, q, v$)

1: $c$ ← the tail of $q$ starting from $v$ (inclusive)
2: $q$ ← the head of $q$ up to (not including) $v$
3: for all vertices $v' \in c$ do
4:     if $v'$ is unoccupied then
5:         Move all agents in $c$ forward, starting with the agent moving to $v'$
6:         return true
7:     end if
8: end for
9: for all vertices $v' \in c$ do
10:     $r \leftarrow A[v']$
11:     $\Pi' \leftarrow []$
12:     if clear_vertex($\Pi', G, A, v', c \setminus \{v'\}$) = true then
13:         $\Pi \leftarrow \Pi + \Pi'$
14:         $v'' \leftarrow$ vertex in $c$ before $v'$
15:         $r' \leftarrow A[v'']$
16:         move($\Pi, A$, agent $r'$ to vertex $v'$)
17:         swap($\Pi, G, A, r, v'$)
18:     Move all agents in $c$ forward, starting with the agent moving to $v''$
19:     reverse($\Pi, A, \Pi', r/r'$)
20:     return true
21: end if
22: end for
23: return false

4.2.3 Rotate operation

The rotate operation (algorithm 4.2.10) relies on the fact that there is a cycle $c$ present in $q$. Each of the agents within this cycle either does not care where it moves to, or it wants to move forward in this cycle. Whenever there is an unoccupied vertex $v'$ in $c$, it is easy to rotate the agents in $c$: each time move the agent forward that wants to move into the empty vertex $v'$.

Rotating a fully occupied cycle is a little bit harder, but it can be done. The procedure is shown in figures 4.14-4.19. In section 4.3.1 it is shown that:

- there is always at least one vertex for which clear_vertex will succeed, and
- the swap operation on line 18 can always be executed successfully.
4.2 Push and Rotate

Figure 4.14: Illustration of \texttt{rotate}: (line 13) the \texttt{clear\_vertex} operation is executed on vertex \( v' \).

Figure 4.15: Illustration of \texttt{rotate}: (line 17) agent \( r' \) moves to vertex \( v' \).

Figure 4.16: Illustration of \texttt{rotate}: (line 18) agents \( r \) and \( r' \) swap positions.
4.2 Push and Rotate

Figure 4.17: Illustration of rotate: (line 19) all agents in the cycle move forwards, beginning with agent $a_3$.

Figure 4.18: Illustration of rotate: (line 20) the reversal of the clear vertex moves is executed (replacing agent $r$ with agent $r'$).

Figure 4.19: Illustration of rotate: the operation is completed successfully.
**Algorithm 4.2.11 resolve(Π, G, A, T, q, F)**

1: while |q| > 0 do
2:    v ← the last vertex on q
3:    r ← A⁻¹(v)
4:    if r ∈ F ∧ A[r] ≠ T[r] then
5:        if push(Π, G, A, r, T[r], A[F]) = false then
6:            s ← A⁻¹(T[r])
7:            return plan(Π, G, A, T, s, F, q)
8:    end if
9: end if
10: remove v from q
11: end while
12: return true

### 4.2.4 Resolve operation

After an agent has made its way to its goal vertex, it may leave a trail (q) of agents that have been moved from their positions by the swap operation. When these agents were originally on their goal positions, they need to be moved back. These agents that want to return to their goal positions, will be called “resolving” agents.

The resolve operation (algorithm 4.2.11) iterates backwards through q. Whenever an agent on vertex v is found in F, it is checked if this is an agent that is resolving, or an agent that is on its goal position. Note that recursive calls to plan and resolve may also resolve agents in q.

Whenever an agents r needs resolving, it is one move away from its goal position. First, a push operation is attempted to restore the position of r. If this does not work, the goal position of r must be occupied by another agent s. Since this position is already the goal position of r, the agent s has a different goal position. By executing the plan operation, agent s is moved towards its goal position.

Once agent s is at its goal position, there are two possibilities:

1. Agent r can now be moved (pushed) to its goal position.
2. There is another agent occupying the goal position of agent r.

In the latter case, the procedure is repeated, which means that the plan operation is executed bringing the agent now occupying the goal position of agent r to its goal.

In each iteration of this loop, the plan operation is executed. This operation brings one agent to its goal position, without removing agents in F from their goal. The loop will end when the push operation succeeds, which will happen after at most k iterations, since at that point all (other) agents must be at their goal position, hence no other agent can be occupying the goal position of agent r anymore.

The interaction between the plan and resolve operations is illustrated in figures 4.20, 4.21 and 4.22. Note that this instance is one that the original Push and Swap algorithm is not capable of solving.
Figure 4.20: Illustration of the paths $p$ (pink) and $q$ (yellow) in the operations plan and resolve. In (a.) the plan operation will perform two swap operations in order to move agent $a_8$ to the right-hand side. Picture (b.) shows that agent $a_6$ has been swapped away from its goal position. In order for agent $a_6$ to return to its position, the agent $a_{11}$ that is currently occupying that position needs to be moved towards its goal position.
Figure 4.21: Illustration of the paths \( p \) (pink) and \( q \) (yellow) in the operations \texttt{plan and resolve}. \( q \) now (picture (c.)) already contains two agents that want to return to their goal positions. Swapping agent \( a_9 \) with agent \( a_2 \) adds another. In picture (d.) the paths \( p \) and \( q \) intersect and it all the agents on the cycle that is formed want to move (or “rotate”) forward through this cycle.

Figure 4.22: Illustration of the paths \( p \) (pink) and \( q \) (yellow) in the operations \texttt{plan and resolve}. After the \texttt{rotate} operation is completed, only one more \texttt{swap} operation is enough to solve the instance.
4.2.5 Post-processing

To eliminate redundant moves from the solution, some post-processing can be applied. A sequence of moves that move an agent to a vertex that it has visited before is redundant, if no other agent has visited that vertex in between. These moves should be removed from the solution. When moves are removed from the solution, other moves may become redundant too. This means that a naive approach may require the algorithm to restart each time after removing redundancy. In order to do this more efficiently, a linked-list-like structure can be used. This is done in algorithm 4.2.12.

Algorithm 4.2.12 smooth(Π)

1: \( Q \leftarrow \) empty queue
2: \( \text{for all } (a, v) = \pi \in \Pi \text{ do} \)
3: \( P_A(\pi) \leftarrow L_A(a), N_A(L_A(a)) \leftarrow \pi \)
4: \( P_V(\pi) \leftarrow L_V(v), N_V(L_V(v)) \leftarrow \pi \)
5: \( \text{if } A^{-1}(L_V(v)) = a \text{ then} \)
6: \( Q.\text{add}(L_V(v)) \)
7: \( \text{end if} \)
8: \( L_A(a) \leftarrow \pi, L_V(v) \leftarrow \pi \)
9: \( \text{end for} \)
10: \( \text{while } |Q| > 0 \text{ do} \)
11: \( \pi \leftarrow \text{retrieve and remove next element from } Q \)
12: \( \pi' \leftarrow N_A(\pi) \)
13: \( \text{while } \pi' \neq N_V(\pi) \text{ do} \)
14: \( \Pi \leftarrow \Pi \setminus \pi' \)
15: \( P_A(N_A(\pi')) \leftarrow P_A(\pi'), N_A(P_A(\pi')) \leftarrow N_A(\pi') \)
16: \( P_V(N_V(\pi')) \leftarrow P_V(\pi'), N_V(P_V(\pi')) \leftarrow N_V(\pi') \)
17: \( \text{if } A^{-1}(P_V(\pi')) = A^{-1}(N_V(\pi')) \text{ then} \)
18: \( Q.\text{add}(P_V(\pi')) \)
19: \( \text{end if} \)
20: \( \pi' \leftarrow N_A(\pi') \)
21: \( \text{end while} \)
22: \( \text{end while} \)
23: \( \text{return } \Pi \)

The \( P_A \) (previous-agent) and \( N_A \) (next-agent) functions are back and forward pointers respectively that form a doubly-linked chain of all the moves of a specific agent. The \( P_V \) (previous-vertex) and \( N_V \) (next-vertex) functions are similar pointers that form a chain of all the moves to a specific vertex. To construct this linked-list-like structure, the \( L_A \) and \( L_V \) functions are used to store the last move from a specific agent or to a specific vertex respectively. Once two adjacent nodes in the chain induced by \( P_V \) and \( N_V \) are of the same agent, there are redundant moves. The first move of the redundant sequence is added to the queue \( Q \).

In order to remove the moves starting at move \( \pi \), observe that all but the first move from
the sequence \([\pi, N_A(\pi), N_A(N_A(\pi)), \cdots, N_V(\pi)]\) between \(\pi\) and \(N_V(\pi)\) can be removed. To remove a move from the structure, the pointers need to be updated. When the pointers have been updated, it is possible to check again for a redundant sequence of moves between each \(P_V(\pi')\) and \(N_V(\pi')\).

### 4.3 Analysis

#### 4.3.1 Proof of completeness

In order to prove the completeness of the Push and Rotate algorithm, the following predicates will be used.

- \(c(r)\) the component to which agent \(r\) is assigned. \(c(r) = c(s)\) iff agents \(r\) and \(s\) are both assigned to some component \(C_{k}\). Note that if agent \(r\) is not assigned to any component, then there is no agent \(s\) for which \(c(r) = c(s)\).

- \(\text{next}(r) = v\) vertex \(v\) is next on a shortest path from \(A[r]\) to \(T[r]\).

- \(\text{blocked}(r, s)\) agent \(s\) is “in the way” of agent \(r\)’s path to its goal vertex, i.e.

\[
A^{-1}(\text{next}(r)) = s
\]

- \(\text{push}(r, s)\) if \(\text{blocked}(r, s)\), then the push operation can push agent \(s\) away from the vertex \(\text{next}(r)\).

- \(\text{swap}(r, s)\) if \(r\) and \(s\) are adjacent (i.e. \((A[r], A[s]) \in G)\), then there is a sequence of moves for which agents \(r\) and \(s\) swap positions.

- \(\text{multipush}(r, s, v)\) the multipush operation can push two adjacent agents \(r\) and \(s\) towards vertex \(v\), such that one agent ends up on \(v\) and the other agent ends up on a vertex adjacent to \(v\).

- \(\text{clear}(r, s, v)\) it is possible to clear two neighbour vertices of \(v\), while keeping agents \(r\) and \(s\) on \(v\) and a vertex adjacent to \(v\).

**Theorem 4.1.** The Push and Rotate algorithm is complete for the class of multi-agent path planning problems in which there are two or more unoccupied vertices in each connected component.

**Proof.** We will show that, if an instance in the two-unoccupied-vertices class of the multi-agent path planning problem has at least one solution, then the Push and Rotate algorithm will find one.

If the roadmap \(G\) is not connected, the specified algorithm should simply be executed separately on each component. Hence, from now on, we will assume that \(G\) is connected.

First, order the agents according to the priorities based on the decomposition into sub-problems.
As stated in lemma 4.1, the plan operation is able to send each agent to its goal position, while keeping agents in \( F \) on their positions. The solve operation will call plan for each agent \( r \), in order of priority, after which agent \( r \) is added to \( F \), which means that it will still be on its goal position after future calls to plan.

At the end of the solve operation, the assignment of agents to vertices \( A \) will match the goal assignment \( T \), i.e. each agent is on its goal position. The sequence of moves \( \Pi \) will be a solution to the problem instance.

Lemma 4.1. After execution of the plan operation, agent \( r \) will have been added to \( F \), and all agents in \( F \) will be at their goal positions.

Proof. The plan operation will send agent \( r \) to its goal position, while leaving agents in \( F \setminus A^{-1}(q) \) on their goal positions, and moving agents in \( F \cap A^{-1}(q) \) back to their goal positions.

In each iteration, agent \( r \) moves to the vertex \( \text{next}(r) \), by using either rotate, push, or swap. Since the distance of agent \( r \) to its goal position decreases in each iteration, the agent \( r \) will reach its goal position in at most \( n \) steps (note \( |p| \leq n \)).

- If \( \text{next}(r) \in q \), then the rotate operation is used. Since there is an edge between the last vertex in \( q \) and \( \text{next}(r) \), this means that there is a cycle \( c \) in \( q \). The rotate operation can (lemma 4.3) and will move all agents, including \( r \), forward along the cycle. The result of the rotate is that all agents in \( F \cap A^{-1}(c) \) are returned to their goal positions (lemma 4.2) and agent \( r \) moves to \( \text{next}(r) \). Note that \( q \) will now be updated such that the cycle is removed.

- If \( \neg \exists s : \text{blocked}(r,s) \), agent \( r \) will simply move to \( \text{next}(r) \).

- If \( \text{push}(r,s) \), agent \( s \) will be pushed out of the way, and agent \( r \) will move to \( \text{next}(r) \).

- Otherwise the swap operation will be executed. \( \neg \text{push}(r,s) \rightarrow c(r) = c(s) \) (lemma 4.4), and \( c(r) = c(s) \rightarrow \text{swap}(r,s) \) (lemma 4.5), hence agents \( r \) and \( s \) will be swapped successfully.

After moving agent \( r \) such that \( A[r] = T[r] \), the resolve operation will be executed, which will return all agents in \( A^{-1}(q) \cap F \) back to their goal positions (lemma 4.6).

Lemma 4.2. All agents in \( F \cap A^{-1}(q) \) in the plan operation have been swapped backwards along the path \( q \).

Proof. In the plan operation, if \( s = A^{-1}(\text{next}(r)) \in F \), then the push operation will not be able to move agent \( s \) away from its position. Hence the swap operation will be used, the result of which is that \( r \) moves forward along the path \( p \), and \( s \) moves backward along this path.

Lemma 4.3. The rotate operation can successfully rotate all agents that occupy a vertex in a cycle.
Proof. Each agent in \( F \) that is occupying a vertex in the cycle \( c \), has been swapped backwards in \( c \) away from its goal vertex (lemma 4.2), and hence will be returned to this vertex when all agents in \( c \) are moved forward. All vertices in \( c \) are assigned to the same component, since they are clearly (part of) a biconnected component. Using these properties, we can show that the rotate operation as described in algorithm 4.2.10, will always succeed.

The most trivial case is when there is a free vertex in the cycle \( c \). In this case the rotate can be executed in a very straightforward manner, starting with the agent that needs to move to this empty vertex.

In case the cycle does not contain an empty vertex, the clear\_vertex and swap operations are used.

- There is always at least one vertex \( v \in c \) for which the clear\_vertex operation succeeds. This is because there are at least two empty vertices, and \( G \) is connected. Hence there must be a path from some vertex \( v \in c \) to an empty vertex, along which agents can be pushed in order to clear vertex \( v \).

- Since all the agents in \( c \) are assigned to the same component, the robot that initially occupied vertex \( v \), and the robot that wants to move to \( v \), are able to swap (lemma 4.5).

\[ \neg \text{push}(r,s) \rightarrow c(r) = c(s) \]

Proof. We will prove the equivalent statement:

\[ c(r) \neq c(s) \rightarrow \text{push}(r,s) \]

If \( r \) is not assigned to any subproblem, it means that \( r \) is restricted to an isthmus, then \( c(r) \neq c(s) \) is always true by definition. Agent \( r \), like any agent that is “trapped” on an isthmus, is incapable of swapping positions with any other agent. This means that it can only reach its goal position by pushing other agents away (figure 4.23). The prioritization of agents from the decomposition makes sure that all agents that are assigned to a subproblem are in \( F \), since agents that are not assigned to any subproblem are planned last. If agent \( s \in F \) is blocking the way of agent \( r \) to its goal, then agents \( r \) and \( s \) need to swap relative positions, and this is impossible. If the instance has a solution, then \( \text{push}(r,s) \) must be true in this case.

Let us now consider the case where \( r \) is assigned to some component \( (c(r)) \), and the blocking agent \( s \) (note that if \( s \) is not blocking \( r \), then \( \text{push}(r,s) \) is trivially true) has to move out of the way. Since agent \( s \) is not assigned to \( c(r) \), it can only reach the vertex on the edge of \( c(r) \). First, we consider the case where agent \( s \) is not occupying a vertex (on the edge) of \( c(r) \). This means that agent \( r \) has its goal position on an isthmus connected to \( c(r) \). Since an isthmus is a string of vertices with degree 2, there is no possibility of two agents swapping their relative positions on it (figure 4.23). This means that in the case that agent \( s \in F \), the instance has no solution, since \( T[r] \) must be “behind” \( T[s] \). The criteria for assigning agents to subproblems guarantees that there are sufficient free vertices reachable.
4.3 Analysis

Push and Rotate

Figure 4.23: Agent $r$ and $s$ are unable to swap on the isthmus.

from $c(s)$ (algorithm 4.1.2, lines 4-9). Since $s \notin \mathcal{F}$, none of the agents assigned to $c(s)$ is in $\mathcal{F}$, and there will be a path from $A[s]$ to a free vertex, which is sufficient to push agent $s$.

This leaves the case of agent $r$ that is assigned to $c(r)$ and agent $s$ that is occupying a vertex on the edge of $c(r)$. This is exactly the criterium on which priority relations between subproblems are determined in algorithm 4.1.3. Since the priorities are propagated, all subproblems that are reachable through the adjacent subproblem will also have lower priority than $c(r)$, hence the agents in these subproblems will not be in $\mathcal{F}$, and will not block a path to an empty vertex, which is required to execute the push operation.

Lemma 4.5. $c(r) = c(s) \rightarrow \text{swap}(r,s)$

Proof. Now we show $c(r) = c(s) \rightarrow \text{swap}(r,s)$. When one of the agents $r$ and $s$ occupies a vertex $v$ with $\text{degree}(v) \geq 3$ and two empty neighbour vertices, and the other agent occupies a neighbour vertex of $v$, then the agents $r$ and $s$ can exchange positions by moving into the empty vertices in one order, and exiting them in the other order. Hence the following statement holds:

$$\text{swap}(r,s) \iff \exists v : (\text{multipush}(r,s,v) \land \text{clear}(r,s,v))$$

This leaves the following statement to prove:

$$c(r) = c(s) \rightarrow \exists v : (\text{multipush}(r,s,v) \land \text{clear}(r,s,v))$$

The assignment criteria for agents to subproblems guarantees that an agent must be either
1. inside a biconnected component, or

2. at least one of the agents is less than $m$ steps away from a vertex within $c(r)$, with degree $\geq 3$.

In the first case, an unoccupied vertex is needed inside the component in order to freely move the agents inside the component. Either an unoccupied vertex is already inside the same component, or this unoccupied vertex can be moved to the biconnected component by pushing a agent out of the component. If this movement is blocked by the agents, then one or both agents are already on a vertex with degree $\geq 3$.

In the second case the agents are either between two vertices of degree $\geq 3$, or they are not occupying vertices that are assigned to the subproblem. Consider the two agents between two vertices with degree $\geq 3$. There are not more than $m - 2$ vertices between these two vertices, two of which are occupied by agents $r$ and $s$. Suppose $l_1$ and $l_2$ steps are required to reach the vertices, and there are $m_1$ and $m_2$ free vertices on the respective sides of the agents. This leads to:

$$l_1 + l_2 \leq m - 4 \quad (4.1)$$
$$m_1 + m_2 = m \quad (4.2)$$

Assume one of the vertices is unreachable:

$$l_1 > m_1 = m - m_2 \quad (4.3)$$

Then the other vertex is reachable:

$$l_2 \leq m - 4 - l_1 \quad (4.4)$$
$$l_2 < m - 4 - (m - m_2) \quad (4.5)$$
$$l_2 < m_2 - 4 \quad (4.6)$$

If the agents are outside the vertices assigned to the subproblem, then the assignment criteria of agents to subproblems states that it must be possible for any agent assigned to a subproblem to enter the subproblem while one additional empty vertex remains in the subproblem.

Consider the amount of free vertices on both sides of the two agents (see figure 4.24).

1. Having $\geq 1$ free vertices on one side of the two agents and $> 1$ on the other side is sufficient to make clearing two vertices easy.

2. In the case that all free vertices are on one side of the two agents, either the agents are not in the same subproblem, or another vertex $v''$ with degree $\geq 3$ is reachable in $m - 2$ steps. Moving the agents towards $v''$ will leave $\geq 2$ free vertices “behind” $v''$ and $\geq 1$ free vertex “behind” the trailing agent. This means that the first case applies to vertex $v''$. 

55
3. Having exactly one free vertex on both sides of the agents implies that $m = 2$. This means that each subproblem is either a single vertex, or a biconnected component of the graph, since no two components are joined together in algorithm 4.1.1. In the single vertex case, it is impossible for both agents to belong to the same subproblem, as moving agent $s$ into this single-vertex subproblem will not leave sufficient free vertices reachable as is required in algorithm 4.1.2.

When the agents belong to the same biconnected component, stage three of the clear operation applies: it is possible to push the agents towards one of the free vertices, such that the trailing agent ends up on vertex $v$. This clears the path for an additional vertex next to $v$ to push towards the other free vertex through the original position of the trailing agent. This clears vertices next to $v$.

Lemma 4.6. The resolve operation will return all agents in $F$ to their goal positions.

Proof. While $|q| > 0$ the resolve operation considers the agent at the last vertex of $q$. For each agent $r$ that is on last$(q)$, either:

1. agent $r$ is not in $F$ and can be ignored for now, or

2. agent $r$ can move or push to $T[r]$, or

3. an agent $s$ is in the way, and $\neg$push$(r,s)$.

In case 2, agent $r$ can be resolved by executing the push operation. Once $r$ has been resolved, or $r \notin F$ (case 1), the last vertex is removed from $q$.

In case 3, $\neg$push$(r,s)$ means that $c(r) = c(s)$ (lemma 4.4), and thus executing the plan operation on agent $s$ will not violate the prioritization of the agents. The remaining $q$ is passed to plan. At the end of operation plan, the resolve operation is executed, with the remaining $q$ extended by the path of agent $s$ to its goal position.

Note that the plan and resolve operations are recursively invoking each other. After each time the plan operation is executed, one additional agent will added to $F$. After at
most \( k - 1 \) recursive invocations, all agents in the subproblem have been sent to their goal vertices, and are in \( F \), which makes it impossible for an agent to be blocking \( T[r] \), since \( \neg \text{push}(r,s) \to c(r) = c(s) \) contradicts with the facts that all agents in the subproblem are in \( F \), but agent \( s \) (which is in the same subproblem as \( r \)) is still on vertex \( T[r] \) (which is not \( T[s] \)).

### 4.3.2 Runtime analysis

Let \( k \) denote the number of agents in the instance and \( n \) the number of vertices in the roadmap. In order to solve an instance, each agent needs to be sent to its goal position by the plan operation, which may be invoked by the solve operation, or recursively through the resolve operation. Note that even though the plan can be invoked from two separate operations, the total times that it will be invoked is at most \( k \). The plan computes the shortest path for the agent to reach its goal position. The length of this path (or any simple path in the graph) is upper bounded by \( n \). For each step along this shortest path, the plan operation will do a rotate, push or swap operation. Out of these operations, the runtime of swap operation is dominant, which simplifies the following equation:

\[
t_{\text{solve}} = O(k \cdot n \cdot t_{\text{swap}})
\]

The swap simply tries multipush and clear on all \( v \) with sufficient degree. While further analysis may show that only a very limited \((O(1))\) number of vertices need to be checked, for now we will have the following:

\[
t_{\text{swap}} = O(n \cdot (t_{\text{multipush}} + t_{\text{clear}}))
\]

Both these operations take \( O(n \cdot t_{\text{clear vertex}}) \). The clear_vertex operation can find a free vertex with a simple breadth-first search, resulting in \( O(|V| + |E|) = O(n^2) \). This leads to the following runtime complexity for the solve operation:

\[
t_{\text{solve}} = O(n^5 \cdot k)
\]

### 4.3.3 Solution quality

For the same reasons as above, the moves generated by the resolve operation are omitted in this analysis. Each of the \( k \) agents has to move along its (shortest) path (of length \( \leq n \)) towards its goal position. Just like in the runtime analysis, the swap operation is a dominant factor in the output of the solve operation. This yields the following expression for the number of moves that the solve operation outputs:

\[
l_{\text{solve}} = O(k \cdot n \cdot l_{\text{swap}})
\]

The swap operation moves two agents to a vertex \( v \) by using multipush, and then clears two neighbours of \( v \). While many different attempts are made in the clear operation to clear the two neighbours of \( v \), the total output of moves is not more than constant times the number of moves generated by clear_vertex plus some constant number of moves:
4.4 Heuristic improvements

To further optimize the quality of the solutions that Push and Rotate produces, some heuristics can be considered. Two aspects that are open to heuristic improvement are: the order in which agents are planned within one subproblem and the shortest path algorithm that is used in various operations.

4.4.1 Agent order

While the order of the agents is essential to the solvability of the instance by the Push and Rotate algorithm, the order of the agents within a subproblem is not defined by the decomposition.

The example in figure 4.26 shows how much difference the ordering of agents can make. The agent $a_1$ needs to pass the goal vertex of agents $a_2...a_6$ to reach its own goal vertex. If the route of any agent other than $a_1$ is planned before $a_1$, then this agent will have to again move from its goal position in order to let agent $a_1$ pass. Obviously, any agent ordering that has agent $a_1$ plan its route first is preferred in this extreme example. When considering the order in which agents are planned, it is interesting to find out how much this order can influence the solution quality. For a very small number of agents, it may be possible to try all possible agent orders. When the number of agents grows, this is no longer possible and two options remain:

- a (fixed-size) sample of random orderings can be used, or
- some heuristic ordering can be used.

In figure 4.25 it is shown how much effect the ordering of agents can have on solution quality. While it is possible to order all the agents before any planning is done using a heuristic, the other (third) option is that the solve operation heuristically picks the next agent.
4.4.2 Shortest paths

With respect to the shortest path algorithm, consider the example in figure 4.27. The agent has three possible shortest paths to its goal vertex. One of these paths is completely free of obstacles, and the other two paths contain one or more agents that are at their goal positions. Clearly in any case, any shortest path that is completely free of obstacles should be preferred. When having to choose between multiple paths with obstacles, these obstacles may be divided into two categories: agents on their goal positions and agents not on their goal positions. As the push operation is almost always less expensive than the swap operation, it may be less expensive to have a path contain agents that are not at their goal instead of agents that are at their goal. When taking the costs for push and swap into consideration, it is possible to assign costs for each obstacle in a shortest path and then choosing the path with the lowest sum of costs.
4.4 Heuristic improvements

Push and Rotate

Considering the case in Figure 4.28, it can be seen that taking a shortest path can be sub-optimal. Instead of taking the shortest (minimal hops) path to its goal, the agent should just avoid the agent that is already at its goal and plan a route through the other vertex in order to solve the instance.

In order to see if a “weighted” shortest path implementation would have an advantageous effect on plan quality and runtime, the following weights have been implemented and tested (the results of which can be found in the next chapter):

- empty vertices have weight 0,
- vertices containing an agent not in \( F \) have weight 1, and
- vertices containing an agent in \( F \) have weight 10.

Only shortest minimum-cost paths are used. Which means that the paths always have the minimum number of hops, i.e. routing around \( a_1 \) in Figure 4.28 will not be done, even though that path may have lower weight.
Chapter 5

Experimental evaluation

The goals of this experimental evaluation are:

- Annotate the theoretical worst-case bounds with the performance of the algorithm on different instance classes. Often algorithms perform significantly better than their worst-case time complexity would indicate.

- Provide insight in the performance of the algorithm for the purpose of allowing it to be compared with other approaches. For this purpose instances from a public benchmark set [21] have been included, since these have been used to test other algorithms [26].

- Test the shortest path heuristic and confirm statistically that it on average improves the solution quality. Because using the heuristic shortest path yields shorter solutions, the runtime is likely also shorter when using the heuristic.

- Show the improved runtime of smooth and how much improvement in solution quality the operation yields.

The Push and Rotate algorithm has been implemented in Java. The experiments were run on a system with an i7-870 CPU running at 2.93GHz and 8GB of RAM.

5.1 Instances

In order to evaluate the algorithm, random instances as well as large public benchmarks were used. It is possible to construct the random instances in such a manner that they are guaranteed to have a solution. Three types of instances were used in the evaluation of the Push and Rotate algorithm:

- random solvable instances,
- grid instances, and
- large instances from a public benchmark set [21].
Each of these instances is used for a specific purpose like finding counter examples and measuring performance. While the benchmark instances are large in terms of the size of the roadmap, the agent-to-vertex ratio is very low, i.e. there are vast amounts of unoccupied vertices. The other instances were mostly generated with only 2 unoccupied vertices. In order to enable reproduction of the results, these are the procedures for generating random instances:

1. Parameters: \( n \): the number of vertices, \( a \): the number of agents, \( \varepsilon \): the edge probability, \( t \): the number of steps. These parameters can also be chosen randomly, but of course \( a \leq n - 2 \).

2. Create \( n \) vertices \( V = \{ v_1, v_2, \ldots, v_n \} \).

3. For each \( i < j \), add edge \((v_i, v_j)\) to the graph with probability \( \varepsilon \).

4. If \( v_j \) is not connected to any other vertex, pick a \( j \) uniformly random from \([1, 2, \ldots, j-1]\) and add edge \((v_i, v_j)\) to the graph.

5. The start positions \( S \) is a random subset of \( V \).

6. Perform \( t \) random moves to obtain the goal positions \( T \).

The grid instances are generated as follows:

1. Parameters: \( n, m \): the dimensions of the grid, \( a \): the number of agents.

2. Create \( n \cdot m \) vertices \( V = \{ v_{11}, v_{12}, \ldots, v_{1m}, v_{21}, v_{22}, \ldots, v_{2m}, \ldots, v_{nm} \} \).

3. Add edges to the graph such that \( v_{ij} \) can reach vertices \( v_{i-1j} \) (if \( i > 1 \)), \( v_{i+1j} \) (if \( i < n \)), \( v_{ij-1} \) (if \( j > 1 \)), \( v_{ij+1} \) (if \( j < m \)). Note that the grid does not “wrap around” at the side of the grid, i.e. between \( v_{11} \) and \( v_{n1} \) there is no edge (unless of course \( n = 2 \)).

4. Both the start positions \( S \) and the goal positions \( T \) are random subsets of \( V \).

The large instances from the benchmark set required a little modification in order to be applicable to the chosen problem domain, since in some scenarios there were multiple agents on the same start and/or goal position. This was solved by performing a simple breadth-first search from the specified start or goal position to find the closest free position.

Solving the random instances has been a great tool for finding bugs in the algorithm and/or the implementation. On the final implementation of the final algorithm, millions of random instances were generated and all solved without problem. Since the runtime and/or solution quality while running the algorithm on a (completely) random instance would result in too much variance to analyze, only the grid instances and large instances are used in the further evaluation of the algorithm.
5.2 Solution quality

While the optimal solution for any test instance with more than a handful of agents is too costly to compute, the following lower bound may give a useful insight into the quality of the solutions of Push and Rotate on the large instances. Define $l_0$ as follows:

$$l_0 = \sum_{r \in A} |\text{shortest\_path}(S[r], T[r])|$$

The generated grid instances turned out to far too congested for this lower bound to be of interest. The values of $|\Pi|$ were somewhere between 6.18 and 18.5 times the value of $l_0$. Since these instances contain exactly the minimum of two unoccupied vertices, almost every agent will need to push and swap multiple times in order to get to its goal position.

Since the large instances of the benchmark set are still very different from each other, the solution quality is presented in the following way (right-hand side of figure 5.2):

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

which is the difference between the solution quality and the lower bound as a fraction of the lower bound. Note that on average the number of moves in the solution was only 101.3% of
5.3 Runtime

The runtime complexity of the three most time consuming parts of the algorithm have been analyzed in the previous chapter. While this indication of worst-case performance is important, it does not always give insight into what the performance on real instances is like.

Figure 5.3 shows the runtime for the large instances on double-logarithmic scale. In the last section, it is shown that on these benchmark instances, the makespan of the solutions is very close to the lower bound value. This could explain why the runtime of the instances scales so nicely in the number of agents. Another factor that is very significant in the
performance of the algorithm is the number of vertices in the roadmap. As instances in the benchmark set have more agents, they also tend to have more vertices.

Figure 5.4: Runtime on different grid sizes, plotted against number of agents (left) and solution quality (right).

Figure 5.4 shows the runtime for the grid instances, where the number of agents (left-hand side) and the makespan (right-hand side) is used to distinguish between the instances. There is a large random factor in the grid instances, since the start and goal positions are completely random, hence one instance may have all pairs of start and goal positions very close together, while another instance may have more distance between these positions (low $l_0$ versus high $l_0$ respectively).

When comparing the results for the large benchmark instances to other algorithms [25], the Push and Rotate algorithm seems to be able to solve instances about as fast as the MAPP algorithm. The MAPP algorithm seems to solve instances with 2000 agents in a little over 100 seconds. Note that we use the benchmark instances in a slightly different way than Wang et al. with respect to the placement of agents in the instances.

Calculating the lower bound $l_0$ for the benchmark set, requires on average 18% of the time required to solve the instances, while the lower bound was calculated using a highly optimized heuristic pathfinding routine as opposed to the BFS used by the weighted shortest path heuristic. This is an indication that for these instances, most of the time may be spent by the algorithm finding the paths between $A[r]$ and $T[r]$ for each agent.

5.4 Smooth

The smooth operation was changed so that it takes time linear ($O(n)$) in the number of moves, instead of cubic ($O(n^3)$) time. The runtime behaviour of the smooth algorithm on the grid and benchmark instances can be seen in the left-hand side graphs of figures 5.5 and 5.6.

It is also interesting to find out how useful the smooth operation really is. Redundant moves are mostly added to a solution when a lot of swap operations have to be performed. In some cases, the push operation also generates redundant moves when there are blocking agents. The grid instances are very congested, which leads to a lot of swap and push operations.
5.5 Heuristic shortest path

The shortest path heuristic, as introduced in 4.4.2, attempts to find the best possible route through which an agent can reach its goal. The weighted shortest path function tries to avoid
paths that go through vertices in $U$, because that would lead to a costly swap operation. Note that the heuristic may still cause the resulting solution to be longer. It is conceivable that avoiding swap operations early in the execution of the algorithm may lead to extra required swap operations in later stages.

Figures 5.7 and 5.8 show in the graphs on the left-hand side the difference in runtime when either running with or without the weighted path heuristic. On the right-hand side the difference in solution quality is shown. The difference in plan quality for the grid instances, while not very large, is shown to be significant in table 5.1.

<table>
<thead>
<tr>
<th>Grid size (number of vertices)</th>
<th>Solution quality with heuristic</th>
<th>Solution quality without heuristic</th>
<th>99 percent confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>218.7584</td>
<td>231.5002</td>
<td>-16 .. -10</td>
</tr>
<tr>
<td>25</td>
<td>619.6392</td>
<td>642.4134</td>
<td>-30 .. -17</td>
</tr>
<tr>
<td>36</td>
<td>1326.218</td>
<td>1389.309</td>
<td>-71 .. -52</td>
</tr>
<tr>
<td>49</td>
<td>2350.656</td>
<td>2441.159</td>
<td>-103 .. -75</td>
</tr>
<tr>
<td>64</td>
<td>3882.4</td>
<td>4042.265</td>
<td>-183 .. -143</td>
</tr>
<tr>
<td>81</td>
<td>5879.628</td>
<td>6123.011</td>
<td>-270 .. -217</td>
</tr>
<tr>
<td>100</td>
<td>8213.011</td>
<td>8621.493</td>
<td>-444 .. -374</td>
</tr>
<tr>
<td>121</td>
<td>12003.75</td>
<td>12453.85</td>
<td>-474 .. -381</td>
</tr>
<tr>
<td>144</td>
<td>16018.3</td>
<td>16648.82</td>
<td>-677 .. -555</td>
</tr>
<tr>
<td>169</td>
<td>21483.34</td>
<td>22313.58</td>
<td>-885 .. -739</td>
</tr>
<tr>
<td>196</td>
<td>27703.17</td>
<td>28598.58</td>
<td>-933 .. -746</td>
</tr>
<tr>
<td>225</td>
<td>35896.65</td>
<td>36993.91</td>
<td>-1167 .. -908</td>
</tr>
<tr>
<td>256</td>
<td>45037.54</td>
<td>45947.9</td>
<td>-1091 .. -763</td>
</tr>
<tr>
<td>289</td>
<td>56272.52</td>
<td>57597.51</td>
<td>-1487 .. -1121</td>
</tr>
<tr>
<td>324</td>
<td>68655.72</td>
<td>69715.26</td>
<td>-1275 .. -843</td>
</tr>
<tr>
<td>361</td>
<td>83012.98</td>
<td>84254.52</td>
<td>-1382 .. -846</td>
</tr>
<tr>
<td>400</td>
<td>97684.7</td>
<td>99501.36</td>
<td>-2072 .. -1480</td>
</tr>
</tbody>
</table>

Table 5.1: Results of the paired Wilcoxon statistic test.

Whenever a (heuristic) improvement to the algorithm improves the quality of the found solution, it is also very likely that this affects the runtime of the main Push and Rotate operation, since the algorithm does not contain a lot of backtracking. For this reason both the difference in solution quality and the runtime difference is measured.
Figure 5.7: Runtime (left) and solution quality (right) with and without the shortest path heuristic on the grid instances.

Figure 5.8: Runtime (left) and solution quality (right) with and without the shortest path heuristic for the benchmark instances.

### 5.5.1 Conclusion

The following interesting observations can be made from these experiments:

- While the algorithm is complete for a wider class of problem instances, the Push and Rotate algorithm seems to perform comparably to the MAPP algorithm on the benchmark instances.

- The algorithm turns out to perform statistically better with the heuristic shortest path function, but the difference is not very large.

- It turns out that the smooth operation can often remove more than 50% of the moves in the solution. This means that the algorithm outputs a lot of redundant moves.
Chapter 6

Conclusion

We have presented the Push and Rotate algorithm for the Multi-Agent Path Planning problem, and provided a proof of its completeness for the class of instances in which there are two unoccupied vertices in the roadmap. Furthermore, an important post-processing operation was improved such that it is usable for large instances.

6.1 Future work

Push and Rotate algorithm can be improved in several areas. First, there can be a lot more interaction between the decomposition into subproblems and the rest of the Push and Rotate algorithm. Ways to improve the performance of the algorithm through interaction, include the following:

- The assignment of agents to subproblems can be used to fully determine if an instance has a solution or not, since agents are limited to the subproblem they are assigned to, and vertices that are close to this subproblem.

- In order to perform swap operations, the algorithm requires empty vertices. Using the decomposition, it may be possible to manage the availability of empty vertices in a smart way. For example, agents can be grouped by subproblem (which is already done), then two empty vertices can be transferred to the current subproblem, where they will be used by all agents in that subproblem.

- Agents can only swap at vertices with degree $\geq 3$ that are also in their subproblem. We conjecture that one of the two closest vertices with degree $\geq 3$ will always be usable for the swap operation.

- There is quite possibly a much simpler way to prevent the unwanted interaction between subproblems that caused the Push and Swap algorithm to be incomplete. One approach that could be attempted is to always allow agents from one component to push away agents from other components, regardless of whether these agents are at their goal positions already.
On a related note, it may be possible to remove the propagation of priorities when the priorities are actually implemented in such a way that in case $A \prec B$, the agents in $B$ get planned first, instead of planning the agents in $A$ last (which is how the current implementation works, since propagation combined with “planning last” results in the “planning first” approach).

- If it is possible to show that there is always a vertex inside the subproblem to which the \texttt{multipush} operation can transfer two swapping robots in $O(n)$ moves, then the worst-case bound can be tightened to $O(n^3)$ moves.

The work on pebble motion [10] on which the decomposition into subproblems is based already outlines an algorithm that is complete for all Multi-Agent Path Planning problems. This approach could be used to solve instances that have only one unoccupied vertex, by implementing a different version of the swap operation based on it.

It may be interesting to study the performance of the Push and Rotate algorithm and other algorithms on different classes of instances, specifically instances which have practical relevance. This would allow for a better comparison between algorithms. It may be possible to create a multi-agent path planning portfolio from several of the algorithms. The problem formulation that we use in this work is restricted to only one agent per move. It is possible to formulate the problem in such a way that agents can move in parallel. This could potentially make the algorithm far more useful for practical purposes. This may be possible by integrating the Push and Rotate algorithm with other existing algorithms, where it solves the hard (dense) parts of instances that other algorithms may not be capable of solving.

The performance of the algorithm can be improved by using heuristics, as shown in the experimental results with the shortest path heuristic. It may be possible to improve this heuristic in order to further improve the overall performance of the algorithm. Options to improve the heuristic include different weights to different kinds of obstacles on the shortest paths. These weights can be measured during runtime: it is easy to compute the average number of steps required to execute for example a swap operation. Another heuristic improvement can be made in the prioritization of the agents within a subproblem.

In the experimental evaluation, it can be seen that the smooth operation yields significant performance improvements. Since the smooth operation only removes redundant moves, it is clear that the other operations in the algorithm output quite a large number of redundant moves. If these redundant moves originate from a specific operation, then it may be possible to reduce the number of redundant moves that is being output directly instead of removing them afterwards. While the resulting solution would not be different, this kind of insight into the algorithm may enable for further optimizations or may reveal a type of redundancy in the solutions that cannot currently be detected by the smooth operation.
Bibliography


