Master Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Cooperative path planning for a team of mobile robots

Tomáš Novák

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Systems and Control
January 2018

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Novák Tomáš**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Diploma Thesis: **Cooperative path planning for a team of mobile robots**

Guidelines:

1. Get acquainted with current approaches to collision-free path planning for a team of cooperating agents/robots, especially the Push and Rotate method [1,2].
2. Propose an extension of the method, which will allow concurrent motion of several robots.
3. Implement the proposed extension, verify it experimentally and compare its behaviour with a selected state-of-the-art method.
4. Describe and discuss obtained results.

Bibliography/Sources:

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

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

Valid until the summer semester 2017/2018

L.S.

prof. Ing. Michael Šebek, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 21, 2017

Acknowledgements

I wish to thank Miroslav Kulich for his patience and support. I am very grateful for the frequent helpful consultations and his constant will to direct me during the process of writing this thesis.

Declaration

I hereby declare that this thesis is my own work and that I stated all the resources used in accordance with “Metodický pokyn o dodržování etických principů při přípravě vysokoškolských závěrečných prací”.

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

In Prague, January 9th 2018
V Praze, 9. ledna 2018

....
signature

Abstract

The purpose of this thesis is to design an algorithm that is able to calculate collision-free trajectories for robots moving in a warehouse using a map of said warehouse. For this, the challenges of the problem are discussed and several state-of-the-art methods are explored with emphasis on the Push and Rotate method.

A new algorithm inspired by approach used in the Push and Rotate algorithm is proposed. This algorithm aims to overcome challenges that arise from expected use in a real warehouse environment.

The algorithm was implemented in C++ programming language. Its properties are shown through experiments on a map of a real warehouse.

Keywords: robot, planning, warehouse

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Jugoslávských partyzánů 1580/3,
160 00 Praha 6

Abstrakt

Tato práce se zabývá návrhem algorithmu, který umožnuje vypočítat bezkolizní trajektorie robotů pohybujících se ve skladu za použití grafu. Nejprve jsou prezentovány výzvy, které tento problém přináší, a několik známých metod je diskutováno s důrazem na metodu Push and Rotate.

V práci je představen nový algoritmus inspirovaný algoritmem Push and Rotate. Tento algoritmus se zaměřuje na překonání nedostatků diskutovaných metod pro využití v prostředí reálného skladu.

Algoritmus byl implementován v programováním jazyce C++. Jeho vlastnosti jsou prezentovány několika experimenty provedenými na mapě reálného skladu.

Klíčová slova: robot, plánování, sklad

Překlad názvu: Kooperativní plánování pro tým mobilních robotů

Contents

Figures

Tables

Chapter 1

Introduction

Industry 4.0 (Figure 1.1) is a concept of modernization and automation of factories and production in general. One of its goals is to eliminate or limit the human presence in the process. Highly automated production lines can be already seen in automotive, food, or electronics industries, but in most of them the human presence remains. As Prof. Ing. Vladimír Mařík DrSc. says, the revolution is using methods of cybernetics, artificial intelligence, and the Internet of things. [ČR]

Warehouses are used by industries to store assembly parts or goods to be sold. These warehouses often already have a computer system that tracks the position of the product in the racks but the goods are usually moved to and from the racks by human beings. They navigate trough the space between the racks searching for the correct rack and then they search for the item. Their movement requires a lot of space between the racks. This space could have been used more efficiently. When assembling an order composed from several different items, they usually spend a lot of the time walking around the warehouse. Some companies, for example Amazon, have implemented a system for automated warehouses where robots bring whole racks to pick-stations. Here the human workers pick up the desired items and put them into a boxes according to orders. Such a system requires the robots to be able to navigate trough the warehouse and avoid obstacles which could be static – such as walls and racks – or dynamic, mainly other robots and

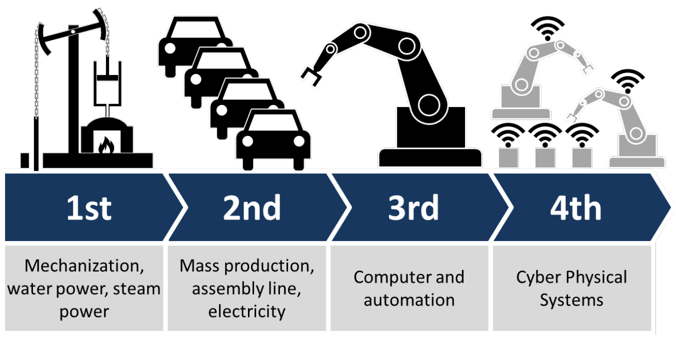

1st
2nd
3rd
4th
Mechanization, water power, steam power
Mass production, assembly line, electricity
Computer and automation
Cyber Physical Systems

Figure 1.1: The four "industrial revolutions". Author: Christoph Roser at [http://www.allaboutlean.com] AllAboutLean.com

occasionally human beings.

This type of problem that needs to be solved is called Multi-agent Pathfinding. When searching for an optimal solution, this problem is NP-complete for a discrete graph and PSPACE-complete for real environment [HSS84]. The Pebble-motion problem is a subcategory of multi-agent pathfinding problems and consists of moving multiple pebble-like objects from a node to a node in a graph, while only one pebble can occupy one node at a given time. The most famous application of this problem is the 15-puzzle where the goal is to rearrange fifteen squares on a 4×4 grid leaving only one free node. The recently published algorithm called Push and Rotate [dWtMW14] is complete for instances with at least two empty nodes and consist of three operations – *push*, *swap*, and *rotate*. The main shortcoming of the pebble-motion solving algorithms is that individual agents cannot move at the same time. Therefore, the real usage of such a solution in a warehouse would be time-wasting and ineffective. All of the above-mentioned algorithms also only assume one type of agent while in the automated warehouse the agents might be separated to those with and those without racks and assume constant node-to-node movement time which is not possible in real application.

In this thesis, I am proposing a modification of the Push and Rotate algorithm. I aim for a real warehouse application that allows parallel movement of two types of agents and accounts for non-constant node-to-node movement time. It is applicable for only a subset of graph types with a limited number of agents but it is designed to be suitable for typical automated warehouses. The proposed algorithm moves agents on the shortest possible trajectories to their destinations and in case of conflict it uses a modified *push* operation or one of the newly proposed operations: *stop* and *replan*.

In Chapter 2 we define the problem and investigate the different types of the state-of-the-art algorithms and their usability in a real warehouse environment. The proposed algorithm is described in detail in Chapter 3 with all its components and implementations. The performed experiments can Chapter 4. Finally, the results of this thesis are summarized in Chapter 5.

Chapter 2

Problem background

2.1 Problem description

The multi-agent path-planning problem in general is described in this section. The existing approaches to the solution with more detailed description of the Push and Rotate algorithm are examined. The challenges resulting in real warehouse environment path-planning are also explored.

2.1.1 Multi-agent path-planning problem

Suppose a simple connected graph $G = (N, E)$, where N is a set of nodes and E is a set of directed or undirected connections between the nodes which are called edges. Also, suppose a set of agents $A = \{(n_s^i, n_g^i)\}_{i=1}^n$, where n_s^i are the starting nodes and n_g^i are the goal nodes to which the given agent is supposed to get. The algorithm solving the multi-agent path-planning problem aims to find a set of trajectories from n_s^i to n_g^i for each agent, while no agents occupy the same node at any given time. The graph can have any possible shape, but there are conditions that must be met for an instance to have a solution. For example an instance in Figure 2.1a) has no solution because it is not possible to exchange agent a_1 with agents a_2 or a_3, while the other agents starting and goal nodes are identical. The instance on the same graph in Figure 2.1b) can be solved easily.

As mentioned above, the graph must be connected. If it is not the case, it can be separated into several simple connected subgraphs that are solved separately assuming no agent has starting and goal nodes in different subgraphs.

2.1.2 Addressed problem

The problem of path-planning in a real warehouse environment has many challenges, however, is also introduces simplifications. Consider two types of robots: one type moving with the rack and the other type moving without the rack. They are both moving on a known graph with directed edges and several types of nodes. Figure 6.1 is a graph of real warehouse that was used for the algorithm design and testing. For better visualization a more detailed cutout can be seen in Figure 6.2. All types of nodes that are considered and

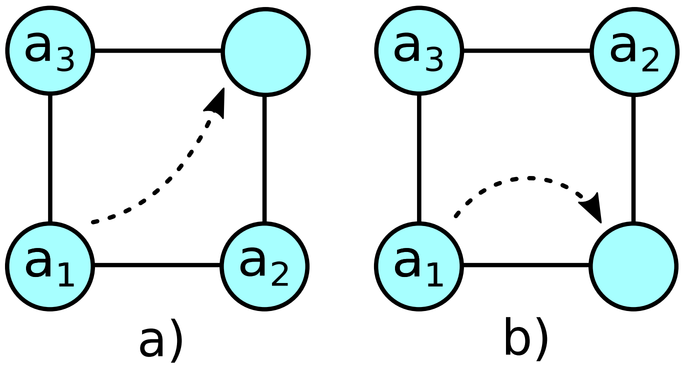


Figure 2.1: The polygon graph with unsolvable (a) and solvable (b) tasks.

described below are visible so that it is easy to make an idea about the shape of the typical warehouse.

Road nodes are used by all robots for movement but never as start or goal nodes (red nodes in Figure 6.1).

Storage location nodes are used for rack storage . All robots can use them as start and goal nodes, but only robots without racks can move over two or more nodes consecutively (grey nodes with rectangles around in Figure 6.1).

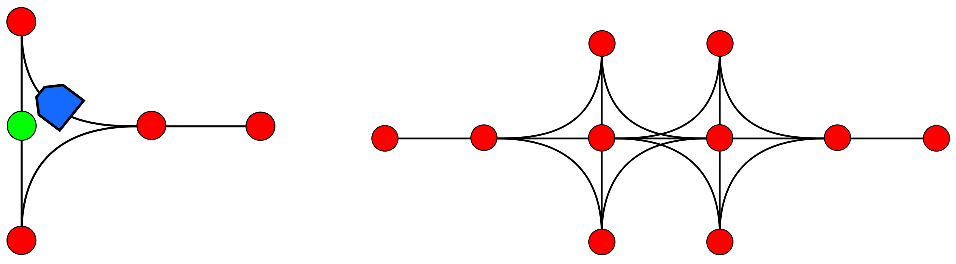
Pick-station nodes serve as goal nodes for robots with racks. The robots stop for an indeterminate time at the station while a human worker picks the goods from the rack. There can be multiple pick stations at different places in one warehouse (dark blue nodes in Figure 6.1).

Queue nodes form parallel lines just before the pick-station nodes to make a queue of robots that were requested at the pick-station (light blue nodes in Figure 6.1).

Maintenance nodes are used by robots get charge or to be repaired by human workers. They are the only nodes in the graph neighboring with only one node (orange nodes in Figure 6.1).

The shape of the warehouse graph is biconnected – a connected graph which remains connected even if any node is removed. The graph in Figure 6.1 has exactly this property with the exception of the nodes neighboring maintenance nodes. Thus by restriction of their usage as only start or goal nodes, we can suppose the graph is biconnected. This is going to simplify the required algorithm. Another assumption decreasing the requirements is that the number of robots in the warehouse is much smaller than the number of nodes. This assumption is justifiable by solving an optimization problem of makespan minimalization with the number of robots or by simple reasoning. The number of trips to bring n racks to the pick station that robots have to perform is asymptotically approaching zero with increasing number of robots while the complexity of the maneuvers to avoid crashes is increasing polynomially, making the trips longer and the solution more resource demanding.

The challenges originating from real usage are mainly based on physical properties of agents that have a physical shape and non-discrete dynamical movement. The non-constant movement time is caused by different distances between nodes, different velocities at which robots travel and their acceleration and deceleration. This allows for the robots to be anywhere between two nodes at any time and while one robot can move trough three nodes in 9 seconds, others might be able to travel only trough two. This forces us to use a much finer time step and define conflicts between nodes and edges. The common shape of the warehouse creates a graph with straight corridors and sharp corners. To avoid frequent stops and on place rotations in one of the most critical sections where agents meet, spline edges as in Figure 2.2a and 2.2b are added into the graph. The extra edges cause a spatial problem where the blue agent crossing the spline edge would collide with agent on the green node (Figure 2.2a). The definition of the conflict must be adjusted accordingly.



(a) : Spline shortcuts

(b) : Complex spline junctions

Figure 2.2: Complex spline junctions at the corners of aisles in the warehouse.

The warehouse map (Figure 6.1) with all the nodes, edges, and conflicts defined is given by the SafeLog project [saf]. The project aims for a human-robot collaboration in a flexible warehouse. Part of the project is the problem solved in this theses, the trajectory planning for a group of robots in an real warehouse. The ability for a flexible planning and prediction of the future state of the robots will help the project to implement their goals in safe and efficient collaboration of robots and humacs. The Figure 2.3 shows the safety concept of robots and humans navigating trough the common environment.

2.2 Existing approaches

The multi-agent path-planning problem is well researched, but in comparison with the single robot path-planning problem it is not solved as comprehensively and efficiently. Most of the approaches can be divided into two categories: coupled and decoupled planning.

In coupled planning, the joint configuration space of all possible states (meaning the positions of agents) in the graph is searched by standard search

Figure 2.3: The safety concept of a human-robot collaboration in a warehouse. Author: SafeLog at [http://safelog-project.eu/index.php/safety-concept/] safelog-project.eu.

algorithms, such as A*. Every state is a set of agent positions on a graph and every expansion of the state adds all the possible combinations of robot movement from the current state into the open set. Considering the most common search algorithms, time complexity usually grows exponentially with the number of agents. The number of possible expansions also increases exponentially with the number of agents. Therefore the algorithm takes an enormous amount of processing time and memory to find the solution when used for problems with more than a few agents. To challenge this problem, different approaches to state representation or expansion may be considered. Many approaches often relax the optimality of the solution but extremely decrease the complexity and allow scalability to large problems. For example by using the modification of the RRT algorithm [SSH13].

The decoupled planning methods plan for each agent separately while using different ways od manipulation to prevent conflicts. Mors et al. [tM11] present a decoupled planning algorithm that searches a minimum-time path for a single agent while avoiding the already planned paths of others. The algorithm can find optimal conflict-free routes in low-polynomial time, but it is not complete. These types of algorithms are usually non-optimal and incomplete, mostly because of their high dependency on the sequence in which the agents are driven along the path to their destinations (Figure 2.4). One of the algorithms proven to be complete is Push and Rotate [dWtMW14] which is further discussed in detail in Subsection 2.2.1.

If we focus on the addressed problem, solutions have already been developed by private companies. These companies sell their products as complete solutions with hardware and software together and do not publish their approaches, thus there is no publicly available algorithm solving this issue.

2.2.1 Push and Rotate

The Push and Swap algorithm [LB11] by Luna and Bekris was published in 2011 as a complete solution for any connected graph with two or more unoccupied nodes. The completeness was disprooved and an improved version was proposed by De Wilde at al. in 2014 as the Push and Rotate algorithm [dWtMW14]. This algorithm is proved to be complete for setups with two or

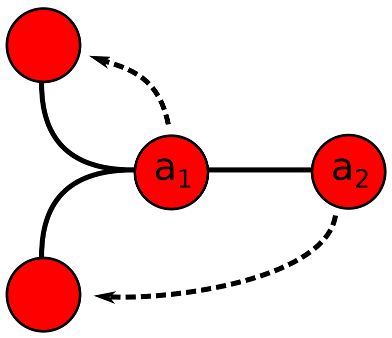


Figure 2.4: If agent a_2 has a higher priority and was planned first, solution does not exist. If agent a_1 was planned first, the solution is found.

more unoccupied nodes.

The main idea of this algorithm is to divide the problem into subproblems and then drive the agents one by one to their goal positions along the shortest path, performing one of the operations – *push*, *swap*, or *rotate* – on every step in the path.

One of the flaws the original Push and Swap algorithm had was that it did not take into account Kornhauser's [KMS84] result that agents cannot swap if there is an isthmus (an edge whose deletion would separate connected graph into 2 mutually disconnected graphs also called bridge) between them longer than the number of empty nodes minus two. This was solved by the decomposition of the problem into biconnected components and an introduction of the *rotate* operation.

When the graph is decomposed into biconnected components, the agents are assigned to the subproblems according to their initial position and number of empty nodes in that subproblem. Next, the priority between subproblems is evaluated. This depends on the final position of the robots. The detailed description of the decomposition and priority evaluation can be found in [dWtMW14]. This is out of the scope of this thesis. When the priority calculation is finished, the robots are moved one by one to their final destination along the shortest path. When moved from one node to another, one of the operations described below is used. When the solution is found, redundant steps generated during the evaluation must be removed. If any agent returns back to a node that it has already visited and no other agent visited the same node in the meantime, the redundant moves are excluded.

Push

The *push* operation attempts to move the agent a_1 from current node v to the adjacent node u. When the u node is not empty, the *clear* operation is evoked to empty it.

The *clear* operation finds the shortest path from the node u to the closest unoccupied node n. The operation must not use a set of blocked nodes that mainly consist of the node v to avoid moving the agent backward. Then all the agents alongside the path are moved in the direction of n, clearing the node u, and agent a_1 can move to said node. Only one agent moves toward its goal node in this algorithm; thus the agents moved aside does not have to be brought back to their original position after the *push* operation. If no path is found, then the *swap* operation is triggered.

In Figure 2.5, the agent a_1 performs three *push* operations. In the first two operations, the *clear* operation is evoked to clear agent a_2 from the adjacent node.

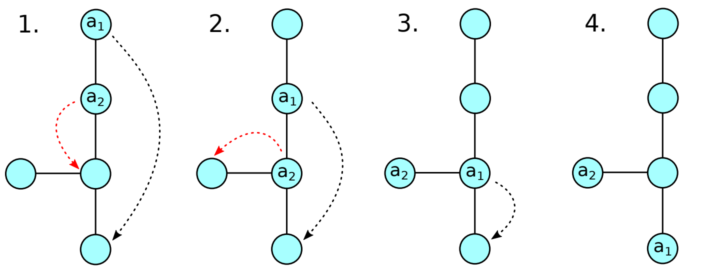


Figure 2.5: Three *push* operations performed by agent a_1.

Swap

The *swap* operation attempts to exchange the position of two agents a_1 and a_2. This can only be done at a node with a degree (number of edges coincident with the node) three or higher (green node in Figure 2.6). The agents are moved to the closest node n with this property, their positions are swapped, and then they are moved back to their original position. When moved to the node n, some other agents may be moved out of the way. The movements done are recorded and reversed after the *swap* is finished. All the agents eventually go back to their original position; therefore the operation can even use the blocked nodes. The operation is proved to find a solution for swapping of any two agents if and only if they belong to the same subproblem.

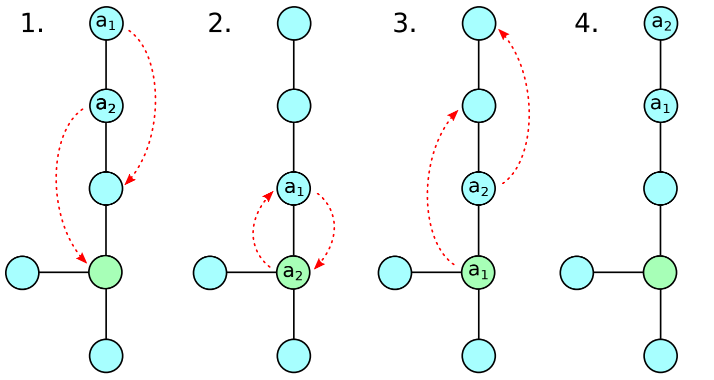


Figure 2.6: Example of *swap* operation.

Rotate

The *rotate* operation is used when the agent visits the same node it has already visited before. This circle is then removed from the agents path and the robots occupying the nodes in the circle are moved one step forward. If at least one node in the circle is empty, the rotation is trivial. Otherwise, the algorithm searches for a node v (green node in Figure 2.7) in the circle that can be cleared and the agent at the v swaps with the previous agent in the circle. Similarly, as in the *swap* operation, all the changes in other agents positions are reversed.

In Figure 2.7 the agent a_1 is first moved from the circle, then it is swapped with a_2 using *swap* operation (somewhere outside the displayed figure) and then all the agents are rotated one step forward.

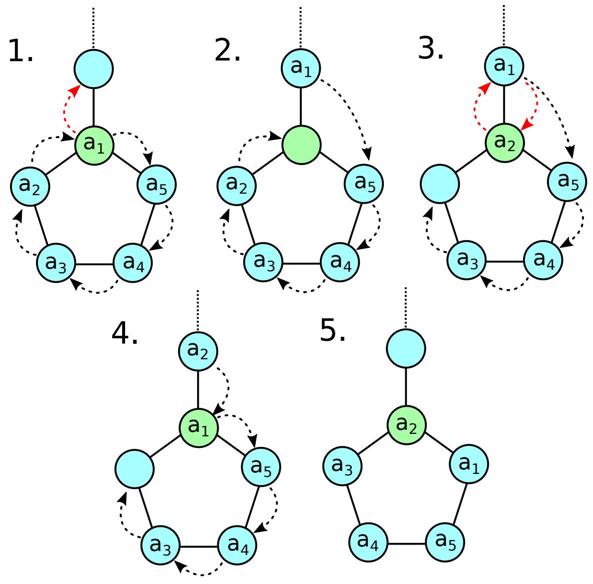


Figure 2.7: Example of *rotate* operation.

Real environment usage

The algorithm as originally presented does not account for any of the challenges mentioned in Subsection 2.1.2. The modification for non-constant node-to-node movement would be simple and only required post-processing procedure because only one agent is moving towards the goal at a time.

Although it is an advantage for continuous-time usage, the restriction of one at the time agent movement is degrading the performance and limits the real use scenarios. There would be no reason to have more than 2 agents in the whole warehouse. One agent would be always moving around the warehouse and the other would wait at the pick-station after the goods were collected from it. The modification of the algorithm for parallel movement of agents will require fundamental changes in the overall design of the algorithm.

Chapter 3
Proposed algorithm

3.1 Basic definitions

The problem discussed in this chapter is the problem of planning paths for real robots in the real warehouse environment considering two types of robots: with and without the rack. The result is a set of trajectories that robots can simultaneously follow without any collision. The trajectories consist of movement from a node to a node across an edge, rotation in place and waiting on a node.

Consider a graph $G = (N, E)$, where N is a set of nodes connected by directed edges E. The task is defined by a set of robots R. Each node has attributes described in Table 3.1. The node type attribute can be set to one of the types described in Subsection 2.1.2. Oriented node (attributes described in Table 3.2) is defined by its orientation (0, 90, 180 or 270 deg) and defines a set of other edges and oriented nodes that are in conflict (no robot should occupy them while there is a robot occupying this oriented node). Each node has a set of oriented nodes. The edge attributes (Table 3.3) consist of basic parameters and same sets of conflicts as in Oriented node. The start and end angle is included to distinguish between the spline and direct edges. The cost attribute is used for A* planning algorithm and specifies the desirability of edge usage. Attributes of a robot (Table 3.4) define the start and goal state, if the robot carries the rack, if the robot is in its final position, its priority, temporary priority, its current velocity and heading angle.

ID	identification number
Type	node type
Position	point in 2D Euclidean space
Entering edges	set of edges ending in node
Exiting edges	set of edges starting in the node
Oriented nodes	set of oriented nodes

Table 3.1: Node attributes

ID	identification number
Orientation	(0, 90, 180 or 270) deg
Parent node	reference to its parent node
Edges conflicts	set of edges in conflict
Oriented nodes conflicts	set of oriented nodes in conflict

Table 3.2: Oriented node attributes

ID	identification number
Start angle	$\langle 0, 360)$ deg
End angle	$\langle 0, 360)$ deg
Cost	$(0, \inf\rangle$
Start node	reference to a start node
End node	reference to a end node
Edges conflicts	set of edges in conflict
Oriented nodes conflicts	set of oriented nodes in conflict

Table 3.3: Edge attributes

3.2 Warehouse problem requirements

As mentioned in Subsection 2.1.2, there are several challenges when designing an algorithm for real continuous environment usage instead for a discrete world of graphs. On the other with assumption of several properties of the warehouse graph, one can employ several simplifications in comparison with complete graph algorithms. The main challenges that had to be resolved during the algorithm design are discussed in this section.

One could use a robot mathematical model to adjust any discrete algorithm for the continuous time by making sure that all the robots always cross a node together by adjusting their velocity. This algorithm would generate solutions with high makespan as many robots would be significantly slowed down by robots who have to rotate on nodes to change the direction of movement.

3.2.1 Non-constant time of movement between nodes

The non-constant time of movement from a node to another node is one of the main difference from standard graph algorithm. The robots are not tied to a single node, but rather can occupy space somewhere between the nodes. To allow the algorithm to work with robots between nodes, the movement from node to node is represented by a sequence of time steps containing necessary information about the robots: time, position, rotation, velocity and a set of occupied nodes. To generate these sequences a model of robot's movement is necessary. The precision of this model defines the usability on real scenario, however in the proposed algorithm we only use very simple model and propose modification to deal with its inaccuracy in the real world scenario.

ID	identification number
Start node	reference to an initial node
Goal node	reference to a target node
Start orientation	$\langle 0, 360)$ deg
End orientation	$\langle 0, 360)$ deg
Rack	true/false
Finished	true/false
Priority	priority of the robot
Temporary priority	accumulated priority of robot
Velocity	current velocity
Heading angle	current robot orientation $\langle 0, 360)$ deg

Table 3.4: Robot attributes

3.2.2 Node/edge conflicts, mainly at spline edges and complicated junctions

Unlike standard graph algorithms where each agent occupies only the node that it is located at, in the warehouse graph considered the robots can be located either at an oriented node or edge, each having defined a set of other oriented nodes and edges that no robot can be present at the same time. This can mean that no robot is allowed to be present at neighboring node, which is mainly case of spline edges and complicated junctions as in Figure 2.2. The set of conflicts for an oriented node typically consists of all oriented nodes of the occupied node and oriented nodes of neighboring nodes oriented in a direction of an edge entering the parent node of the oriented node and said edges. The set of conflicts for an edge is typically the occupied edge, all oriented nodes of the start and end node of the edge and edges entering the said nodes.

3.2.3 Parallel movement of robots

Parallel movement of robots is a crucial goal of this algorithm to reduce the makespan of the solution and make it viable for usage in a real warehouse. To overcome this challenge, all robots are moved one time step together. When one robot happens to conflict with another robot, one of the operations described in Section 3.3 is invoked to resolve the conflict. All of the operations involve only the robots in conflict and position of the finished robots. The consideration of parallel movement of robots causes an increase in complexity of the algorithm, but also add more flexibility which results in new *stop* operation that aims to reduce the frequency of more complicated *push* usages.

3.2.4 Simplifications

The real warehouse environment increases the complexity of the algorithm significantly, however the graph of the warehouse has certain propeties that we can use to simplify the original algorithm. In a real warehouse environment we assume that the graph is always biconnected; thus the decomposition used in Push and Rotate algorithm will always end up with one component of the whole graph. Therefore the decomposition can be omitted from the algorithm. When the whole graph is biconnected, the *swap* operation will always succeed as proved in [dWtMW14] and the *rotate* operation can be omitted.

The remaining operations are now *push* and *swap*. To include the *swap* operation, the implementation might become unbearable, while it would be used in practice only when the number of robots would get close to the number of nodes; thus I have omitted it and introduced new operation *replan* that is supposed to help to solve situations that the *push* operation is not able to solve and even directly in the *push* operation to find new path for robots that had to be diverted from their original path. This simplification brings limitation on number of robots explained in Section 3.4.1.

3.2.5 Runtime requirements

Consider usage in a real warehouse. The robots are not given the tasks at once, but rather the goals are assigned one by one from warehouse management software. When a goal is assigned to the robot, the time to find the solution is supposed to be as fast as possible, which is a challenging requirement to meet. Thus the algorithm should be able to reuse the already evaluated solution and just add new robot's movement as fast as possible. The proposed algorithm produces sequences of time steps for each robot that are collision-free. If we add a goal for a new robot, we can run the shortest path planning phase only for the new robot and reuse the already evaluated paths for the rest of the robots. When running the algorithm again, only new conflicts caused by the impact of the added robot are resolved again. To further reduce the impact of the added robot, the shortest path might avoid nodes at which most of the operations occur if possible.

Also there is no need to wait for the algorithm to finish. When solving the problem, time is moved forward and all calculated paths until current time-step are collision-free. When there is a conflict, the time is usually moved backwards. In extreme case where one operation immediately causes new conflicts, the time can be moved backwards significantly, however this can be statistically evaluated to calculate how long the solution buffer must be to start the movement of the robots. In extreme case the buffer would get too small during the movement of the robots, the movement would have to be paused.

3.3 Algorithm description

The full algorithm that is supposed to navigate the robots in the warehouse is composed of 3 layers. The highest layer is the warehouse manager which creates a queue of tasks that are supposed to be accomplished from an order that is actually processed. The order may consist of several items that are stored at different locations, thus their position is determined and the manager adds the information which racks has to be brought to which pick station. This layer is also supposed to handle queues in front of the pick station, thus the other layers can move the robots only to the isthmus leading to the queue nodes. The robots can be sorted in the queue part of warehouse using complete algorithm such as Push and Rotate with only one robot moving at the time.

The middle layer is a part of the planner, that processes the queue from the manager and selects the robot for each task. The planner selects a robot that is idle or the first one to finish its current task. If there are multiple robots available, then the one with shortest path to the goal is selected. This layer also performs the initial state estimation and reuse of the already planned paths, if they are available. When all the information is collected, the planner runs the path-planning algorithm that generates collision-free trajectories that solve the given task.

The lowest layer, the path-planning algorithm, takes the set of robots R, the graph G describing the warehouse and reduced graph $G_r \subseteq G$ from which all storage location nodes and all edges connected to them are removed as an input. The algorithm is divided into three parts: the single robot path planning phase, the initial trajectory generation and the robot maneuvering phase (Algorithm 1). In the single robot path planning phase, shortest path using A* algorithm is calculated for each robot from its start node to its goal node. In the second phase, the trajectories following the calculated paths are generated using the robot model. The robot maneuvering phase simulates movement of the robots following calculated trajectories and uses operations *stop*, *push* and *replan* to modify the trajectories in case of conflict. Each robot has its planned trajectory J_{r_x} and pointer to the current state. When the time is shifted forward or backwards, the pointer is incremented or decremented for all robots. This way the state of the whole warehouse is moved.

At the beginning, the robots have assigned main, unchangeable priorities according to the length of the shortest path to their destination. The temporary priority can be incremented by the *push* operation and reset to the value of main priority. The *stop* operation is trying to resolve the conflict by stopping one of the robots, the *push* operation pushes robot with lower priority.

3.3.1 Single robot path planning phase

The lines 1 to 8 of the Algorithm 1 describe the single robot path planning phase. First the shortest paths for all robots are generated on the graph G or the reduced graph G_r depending on the attribute Rack of the actual robot. The shortest paths from a start node n_s to an goal node n_e are found by widely used path-finding algorithm A* [Bee]. This algorithm is complete and optimal for consistent heuristic. The heuristic function $H(n_x)$ is the Euclidean distance to n_e while $G(n_s, n_x)$ is the known cost from n_s to n_x. The heuristic function $H(n_x)$ is used to sort nodes in the open list (lost of not yet explored nodes). In the proposed algorithm, three costs for long spline

Algorithm 1: Robot path-planning algorithm

```
Data: Set of robots $R$ with tasks, graph $G$, reduced graph $G_r$, robot
      model $L$
Result: Collision free trajectories for robots
1  $trajectories \leftarrow$ empty vector
2  $paths \leftarrow$ empty vector
3  forall $r \in R$ do
4      if $Robot\ r\ has\ rack$ then
5          $P \leftarrow shortest\_path(r, G_r)$
6      else
7          $P \leftarrow shortest\_path(r, G)$
8      end
9      $J_r \leftarrow generate\_trajectories(P, G, L)$
10     $trajectories \leftarrow trajectories + J_r$
11     $paths \leftarrow paths + P$
12 end
13 $J_{sol} \leftarrow solver(trajectories, paths, G, R)$
```

edges C_l, edges to storage location nodes C_s and default edges C_d are used. The default cost C_d is chosen to be 1. The cost C_l is based on the length of the spline edges. Traveling trough spline edge is shorter than travel trough two default edges, but longer than traveling trough one; thus $C_d < C_l < 2C_d$ and is set to 1.5. The edges that end in the storage location nodes are in most cases traveled by the robots without racks. To enforce their preference to travel under the racks and leave more space on the road nodes for robots with racks, the cost C_s must be $0 < C_s < C_d$ and $0 < C_s < \frac{C_I}{2}$, and is set to 0.1. The robots that carry a rack cannot use edges starting or ending at the storage location nodes with exception of initial and goal state of the robots.

3.3.2 Initial trajectory generation phase

The generated paths are processed by the robot model (line 13). The output for each robot r_x is a list of time steps $J_r = \{j_1, j_2, \ldots, j_n\}$, where n is the number of time steps in the path. The output data depends on the model parameters. Because we need to discretize continuous movement, the time steps are actually samples of real trajectory movement; thus sample frequency must be defined reasonably. The algorithm directly processes the time samples, therefore the computational difficulty grows with the sampling frequency. Choosing too small number of samples could lead to failure in case that there are not at least 2 samples between two nodes – each where robot occupies one of the two nodes on the edge. The robots could into conflict in moments that were not captured by the samples; thus the algorithm would not detect it. It is reasonable to have at least 10 samples for each edge. To make sure, the sampling frequency F_s is sufficient, it should meet the condition $F_s > \frac{l_m}{v_m} \times 10$, where l_m is the minimum length of an edge and v_m

is the maximal robot velocity. The sample time $T_s = \frac{1}{F_s}$ is usually used in the algorithm.

Algorithm 2: Solver algorithm

```
   Data: generated trajectories $J$
   Result: Trajectories $J$, modified to be collision-free.
1  $solved \leftarrow false$
2  while $!solved$ do
3      if $conflict\_detect()$ then
4          $resolve\_crash(crash)$
5      if $check\_solved()$ then
6          $solved \leftarrow true$
7      else
8          $resolve\_priority\_reset()$
9          $resolve\_replan()$
10         $state\_shift(1)$
11     end
12 end
```

3.3.3 Robot maneuvering phase

This phase is described by Algorithm 2. The loop (line 2) is repeated until the problem is solved. First it is checked if there is conflict in the current state of robots (line 3). If there is one, it is immediately resolved using Algorithm 4. Then it is checked if the problem is not yet solved by checking if all robots reached their destination (line 5). When the problem is not solved yet the state is shifted forward by one step (line 10). The two resolve methods preceding the state shift are used to envoke certain operations, their purpose is described further in the operations description.

The *Conflict detect* (Algorithm 3) searches trough all the blocked oriented nodes and edges and checks whether no robot occupy any of them. It only skips the blocked nodes and edges that originated from currently tested robot to avoid robot conflicting with itself.

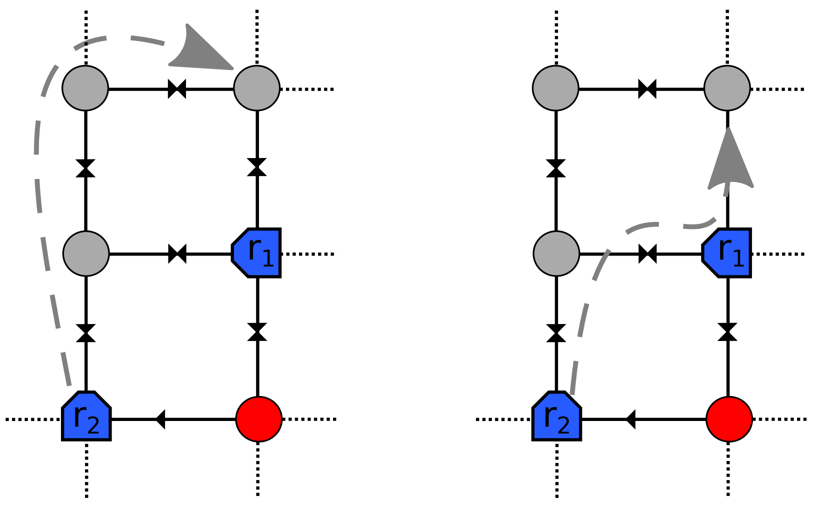
The *Resolve crash* Algorithm 4 solves only one crash at the time. If more than one crash occurs at the same time only the first found is resolved, however all the operations will cause the time t (and state of all robots) to decrement for at least T_s; thus the other crashes will be also resolved. In the algorithm, it is first decided which operation should be used (line 1) and then it is executed. The executed algorithm now depends on operation that was decided to be used and are described further in separate sub-chapters.

The *Decide operation* (Algorithm 5) first decides which robot has higher priority and which one has lower priority. If any of the robots is finished, the *replan* operation is used. This is because after robot reaches its final destination we do not want to move it. It is possible, because no robot can finish on Road node, thus there is always another path to the destination of the robot. When no robot is finished, the *stop* is tested if it can be used. This is done by testing two conditions for both robots.

The first one is that the other robot's path does not cross the node that the tested robot would be stopped at. In Figure 3.1b, the robot r_2 has a path planned in a way, that stopping the robot r_1 would not help to resolve the conflict. If the path was planned differently (Figure 3.1a), the *stop* operation helps resolve the conflict completely.

The second condition is implemented to prevent a dead-lock situations that might rise from *stop*. When the robot is stopped, it is put into idle state and waits until the first edge that it needs to travel trough is empty. This might lead into a dead-lock situation. In Figure 3.2b the deadlock would occur when the robot r_1 would get stopped because of the robot r_2, the robot r_2 would get stopped because of the robot r_3 and the robot r_3 would get stopped because of the robot r_1. To avoid this situation, the robot that is supposed to be let go cannot have any idle robots on its way. This condition will assure there will be no deadlock situation, but also can cause that *stop* operation is not used in situations it would help.

When no other operation is selected, the *push* operation is chosen.



(a) : *stop* operation is possible. Stopping robot r_1 allows robot r_2 to

(b) : If r_1 is stopped, the robot r_2 would still crash into it.

Figure 3.1: Examples showing when the *stop* operation is possible and when it is not.

3.3.4 Stop

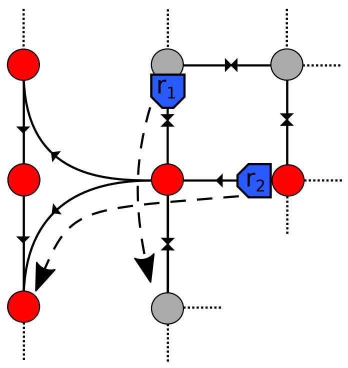
The idea of the *stop* operation (Algorithm 6) is very simple – to stop one robot that the other one can continue without disruption. Most of the edges between road nodes are one way; thus most of the time if one robot stops for a short time, the other one can easily pass and the conflict is resolved (Figure 3.2a).

Algorithm 3: Conflict detect algorithm

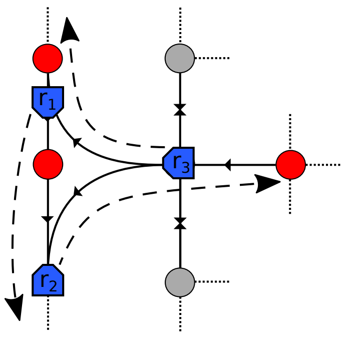
```
   Result: Indicator whether there is conflict in current robot state is
           returned. If there is conflict, it is reported which robots
           conflicted.
 1 forall blocked oriented nodes do
 2     forall robots do
 3         if robot occupy current oriented node and node was not blocked
            by current robot then
 4             Report conflict.
 5             return true
 6     end
 7 end
 8 forall blocked edged do
 9     forall robots do
10         if robot occupy current edge and edge was not blocked by current
            robot then
11             Report conflict.
12             return true
13     end
14 end
15 return false
```

Algorithm 4: Resolve crash algorithm

```
   Data: Robots that crashed $r_1$ and $r_2$
   Result: Resolves the actual crash between two robots.
 1 operation ← decide_operation()
 2 Execute operation.
```



(a) : The *stop* operation will be always successful.



(b) : Deadlock example

Figure 3.2: Examples of *stop* operation situations.

Algorithm 5: Decide operation algorithm

```
Data: Robots that crashed $r_1$ and $r_2$
Result: Decides operation that should be executed to resolve the
        conflict.
1  if temporary priority of $r_1$ > temporary priority of $r_2$ then
2      $r_{high\_priority} \leftarrow r_1$
3      $r_{low\_priority} \leftarrow r_2$
4  else
5      $r_{high\_priority} \leftarrow r_2$
6      $r_{low\_priority} \leftarrow r_1$
7  end
8  if $r_1$ is finished or $r_2$ is finished then
9      return $Replan\_operation(r_{high\_priority}, r_{low\_priority})$
10 if $can\_be\_stopped$ then
11     return $Stop\_operation(r_{high\_priority}, r_{low\_priority})$
12 return $Push\_operation(r_{high\_priority}, r_{low\_priority})$
```

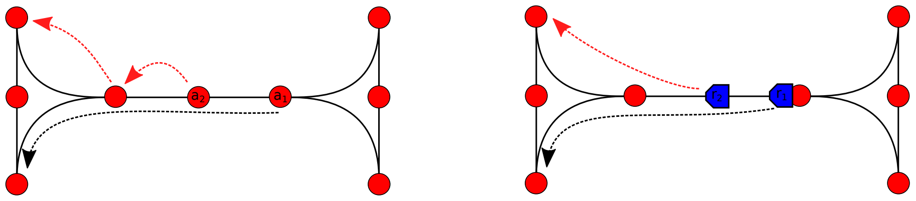
At first it is decided which robot should be stopped (line 1). The algorithm already has a set of robots (of size 1 or 2) which can be stopped and selects the one with lower temporary priority. Then the time is shifted back until the r_{stop} is occupying a node (line 2). At this node n_s the robot r_{stop} will be stopped. The wait sequence is generated (line 3). It is a sequence of $steps\_shift$ time steps where the robot r_{stop} stands still on the node n_s ending with time step with special *idle* flag. When this special time step is the next step, it is checked during the state shift (Algorithm 2, line 10) if the edge that the robot r_{stop} is going to move at is not in conflict with any other robot (if by traveling the edge, the robot does not get into conflict with other robots). Until there is a conflict, the wait sequence is prolonged. The generated wait sequence is then added into the path after the current state prolonging the planned trajectory $J_{r_{stop}}$ (line 4).

Algorithm 6: *stop* operation

```
Data: Robots that crashed $r_1$ and $r_2$, robot model $L$
Result: Stops one of the robots and resolves conflict.
1  $[r_{stop}, r_{go}] \leftarrow$ Decide which robot to stop
2  $steps\_shift \leftarrow$ Shift time back until $r_{stop}$ is occupying a node
3  $J_{wait} \leftarrow$ Generate wait sequence using $L$.
4  Update $J_{r_{stop}}$ with $J_{wait}$
```

3.3.5 Push

The *push* operation is based on the operation from the Push and Rotate algorithm with the same name. The original *push* operation is used for the movement of the agents even when there is no conflict (Subsection 2.2.1). In this algorithm, only the part of the operation that is invoked when the next node is occupied by an agent is considered, because it is used for resolving conflicts and not moving the robots itself. In the original algorithm, when the other agents are pushed away, they are always moved only by one node; thus the operation can be executed several times for a single robot if the robots have a conflict on a long isthmus. The red arrows in Figure 3.3a show the operation had to be carried consecutively twice in order to let the agent a_1 trough. This version of the operation moves the agents arbitrarily far, when moving on an isthmus or when the closest nodes cannot be used for example if they are occupied by finished robots (Figure 3.4).



(a) : Original *push* operation. **(b) :** New *push* operation.

Figure 3.3: Comparison of the original and new *push* operation on simple case.

The new *push* (Algorithm 7) has to decide first which robot r_{push} will be pushed away and which robot r_{go} will continue on its path (line 1). Due to the properties of warehouse graph, two (non-finished) robots can always perform this operation, because at least one robot can always be pushed. The robot's temporary priority is the main factor in the decision of the robot roles. The preferred robot to be pushed is the robot with lower temporary priority, because it is less likely that the robot was pushed recently and the robot with higher priority has more likely longer trajectory to travel trough. In some situations, one of the robots cannot be pushed, because the warehouse graph can have directed edges. There are nodes with only one exiting edge from the node and the other robot might occupy the end node of this edge (see Figure 3.5. The operation selects the robot that can be pushed with respect to the directed edges.

The operation needs to find the closest node that is not in the path of the robot r_{go}. This search needs a lists of nodes and edges that are in a path of the robot r_{go} to know which nodes cannot be selected (lines 3 and 4). It also needs a list of nodes that the algorithm is forbidden to expand during the search. First, the node where the robot r_{go} is going to wait is added to the nodes list to prohibit robot r_{push} being pushed trough this node (line 5) and all nodes with finished robots are added since it is not possible to move them (line 6). Then the path is found (line 8) using Dijkstra's algorithm [Sie] modified to respect the blocked edges and nodes with forbidden expansion. The state of the algorithm is shifted back until the robot r_{push} occupies a node (line 9) and all its future time steps are removed from it's planned trajectory to be later replaced with the push trajectory (line 10).

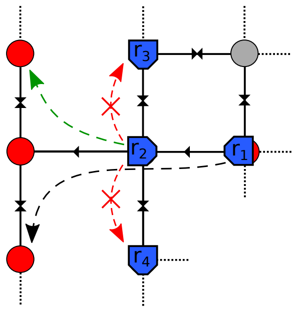


Figure 3.4: The robot r_2 cannot be pushed to the closest nodes, because those are occupied by finished robots r_3 and r_4 and must be pushed

The trajectory is generated using the model of the robot and special *reset* flag is added to the last generated step (line 11). The temporary priority of a robot r_{push} is always increased by r_{go} (line 14) to push other robots that might get into conflict with the robot r_{push} while being pushed. This way only a robot with really high priority would be able to push this robot back. The priority system ensures the non-finished robot with highest priority always moves towards its destination. When the push is finished, the robot r_{push} resets its temporary priority when the time step with *reset* flag is encountered in the Algorithm 2 (line 8).

As the push of the robot r_{push} is executed on directed edges, moving back to the original position using the same path might be impossible. Instead, the algorithm generates trajectory from the last node of *push_path* to the goal node of r_{push} (line 12) using *replan* (see Subsection 3.3.6). The trajectories are added together to form a new trajectory for the robot r_{push}.

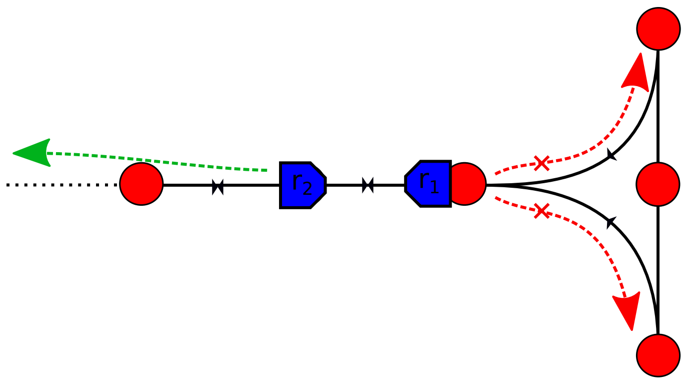


Figure 3.5: It is impossible to push the robot r_1, because the only exiting edge ends on a node occupied by the robot r_2, which is trying to push it. The robot r_2 must be pushed.

The algorithm needs to stop the robot r_{go} on the last visited node before the conflict. First we need to shift the time to a state where the robot r_{go} is at the node. However, the state was shifted back before thus the state must be shifted by $(steps\_shift - t_{back\_to\_node})$, where $t_{back\_to\_node}$ is the number of steps from robot r_{go} leaving the previous node before the state shift back (line 15). This shift can be either forward or backwards. Similarly as in the *stop* operation, the operation generates the wait sequence J_{wait} for the robot r_{go} with minimal wait of $(steps\_shiftt_{back\_to\_node})$ steps to ensure that the robot r_{push} will get to the state of conflict. The special *idle* flag is added to the last step of the wait (line 16). The operation adds J_{wait} sequence into the $J_{r_{go}}$ trajectory right after the current step (line 17) and the operation is complete.

Algorithm 7: The *push* operation.

```
    Data: Robots that crashed $r_1$ and $r_2$, robot model $L$, graph $G$
    Result: Resolves conflict with push operation.
1   $[r_{push}, r_{go}] \leftarrow$ Decide which robot to let go and which robot to push.
2   $t_{back\_to\_node} \leftarrow$ Number of steps from when $r_{go}$ left last node.
3   $blocked\_nodes \leftarrow$ Nodes in path of $r_{go}$.
4   $blocked\_edges \leftarrow$ Edges in path of $r_{go}$.
5   $no\_expansion\_nodes \leftarrow$ Node where $r_{go}$ waits.
6   $no\_expansion\_nodes \leftarrow$ Nodes with finished robots.
7   $n_p \leftarrow$ Last node that $r_{push}$ occupied.
8   $push\_path \leftarrow$ Find path to closest node to $n_p$ with respect of
      $blocked\_nodes$, $blocked\_edges$ and $no\_expansion\_nodes$ on graph
      $G$.
9   $[steps\_shift, t_{now}] \leftarrow$ Shift state back until $r_{push}$ is occupying a node.
10  $J_{r_{push}} \leftarrow J_{r_{push}}(j_0, \ldots, j_{(t_{now})})$.
11  $J_{push} \leftarrow$ Generate push trajectory using $push\_path$ and $L$.
12  $J_{replanned} \leftarrow$ Generate trajectory from last node of $push\_path$ to goal
      node of $r_{push}$ using replan.
13  $J_{r_{push}} \leftarrow J_{r_{push}} \cup J_{push} \cup J_{replanned}$.
14  temporary priority of $r_{push} \leftarrow$ temporary priority of $r_{push}+$ temporary
      priority of $r_{go}$.
15  Shift state by $(steps\_shift - t_{back\_to\_node})$.
16  $J_{wait} \leftarrow$ Generate wait sequence with at least
      $(t_{back\_to\_node} - steps\_shift)$.
17  Update $J_{r_{go}}$ with $J_{wait}$.
```

3.3.6 Replan

The *replan* operation (Algorithm 8) is used when one of the robots is finished (Figure 3.8). The finished robots cannot be moved, thus *push* and *stop* would not help in this case. The operation plans a new trajectory from current node n_c that the robot r_x occupies to its goal node n_g, while avoiding all nodes that the finished robots occupy. The property of graph that by removing any storage location nodes, the graph will not become disconnected, is considered. This property assures that there is always a path to the goal destination if any storage location node is removed, assuming the removed node is not the node the robot is occupying or its goal node. The graph could become disconnected by removing road nodes, but no robot can finish on a road node.

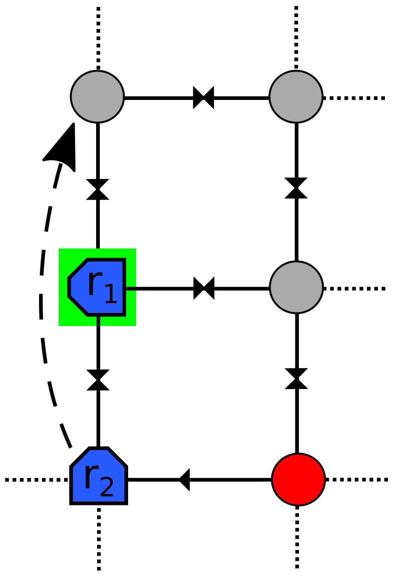


Figure 3.6: The both operations *stop* and *push* would fail, because the robot r_1 is finished and cannot be moved. This situation require the *replan* operation.

At first, the algorithm identifies which of the robots is not finished (line 1) to select the one that needs to be replanned. This robot needs to avoid all robots that are already finished, therefore all nodes occupied by finished robots are added to the list *nodes_to_avoid* (line 2). The solution state is shifted back until the robot r_x is occupying a node (line 3). The operation removes the planned trajectory of the robot r_x from the current state to replace it further with the new replanned one (line 6). Similarly as in Subsection 3.3.1, the algorithm calculates the shortest path from the currently occupied node n_c to the goal node n_g using A* algorithm (line 7). However all the nodes in the *nodes_to_avoid* list are removed from the graph. From the calculated path, the operation generates trajectory for the robot r_x describing its movement from node n_c to its goal node n_g (line 8). This trajectory is then added to the planned trajectory J_{r_x} (line 9).

The *replan* operation removed part of the planned trajectory and replaced it with a new one. If the robot r_x was preforming the *push* operation, the special *reset* flag for resetting the priority would be deleted. To avoid this loss, the algorithm resets the priority (line 10), because the change of the trajectory causes the robot is no longer performing the *push* operation.

The *push* operation uses the *replan* operation as described in Subsection 3.3.5 to *replan* a path of a robot after being pushed. The main difference is that the operation is not supposed to solve conflict, but only generate a new trajectory. To fit the description of Algorithm 8, one can simply assume that $r_{push} = r_1 = r_2$. The algorithm still benefits from the avoidance of the finished robots saving future *replan* operation calls.

Algorithm 8: *replan* operation

```
    Data: Robots that crashed $r_1$ and $r_2$, robot model $L$, graph $G$
    Result: Replan the trajectory $J_{r_x}$ of the non-finished robot $r_x$.
 1  $r_x \leftarrow$ The robot ($r_1$ or $r_2$) that is not finished.
 2  $nodes\_to\_avoid \leftarrow$ Nodes occupied by all finished robots.
 3  $t_{now} \leftarrow$ Shift state back until $r_x$ is occupying a node.
 4  $n_c \leftarrow$ Node occupied by $r_x$.
 5  $n_g \leftarrow$ Goal node of $r_x$.
 6  $J_{r_x} \leftarrow J_{r_x}(j_0, \ldots, j_{(t_{now})})$.
 7  $P \leftarrow$ Calculate the shortest path from $n_c$ to $n_g$ avoiding nodes in
      $nodes\_to\_avoid$.
 8  $J_{replan} \leftarrow$ Generate trajectory using path $P$ and the robot model $L$.
 9  $J_{r_x} \leftarrow J_{r_x} \cup J_{replan}$
10  temporary priority of $r_x \leftarrow$ main priority of $r_x$.
```

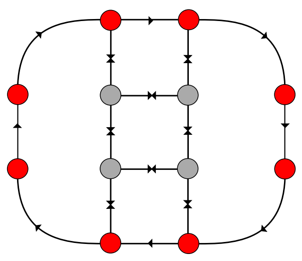
3.4 Algorithm properties

The algorithm has several properties that are discussed in this section. First there are some limitations given the assumed graph structure and goal nodes. Also there are several advantages pointed out in comparison with standard graph algorithms.

3.4.1 Algorithm limitations

Limitations of the proposed algorithm are caused by simplifications of the Push and Rotate algorithm and specialization on the real environment. We have the assumptions to the graph properties and possible goal nodes for the robots. The directed graph must be connected and biconnected at nodes which robots are allowed to have their goal nodes with exception of maintenance nodes. The other nodes do not need to be biconnected.

Another limitation is the maximal number of robots. In theory, the number of robots that should be able to navigate is same as in the Push and Rotate algorithm, therefore $n-1$ where n is the number of robots. However in the proposed algorithm the finished robots cannot be moved which requires the warehouse-like graph structure with road nodes surrounding storage location nodes. Also, no robots can finish on the road nodes. The maximum number of robots on the graph is $n - n_r$, where n_r is the number of road nodes. In Figure 3.7a, there is an example of small warehouse graph with 4 storage location nodes in the center surrounded by 8 road nodes, thus only 4 robots can navigate this 12-node graph. This can be extended by maintenance nodes (or they can also be storage location nodes) connected to each road node (Figure 3.7). This way the robots to nodes rate is much better, 12 robots can navigate on 20 node graph. The width of the storage location nodes block must be maximally 2, but the length can be arbitrary. In an ideal case, if there would be only 1 block with a length close to infinity, the rate of robots to nodes would be $\frac{2}{3}$. In the warehouse used during development and testing of the algorithm, the rate (ignoring pick-station, isthmuses leading to and from pick-stations and queue parts of graph) is approximately 0.52.


(a) : Small graph with 4 storage location nodes.

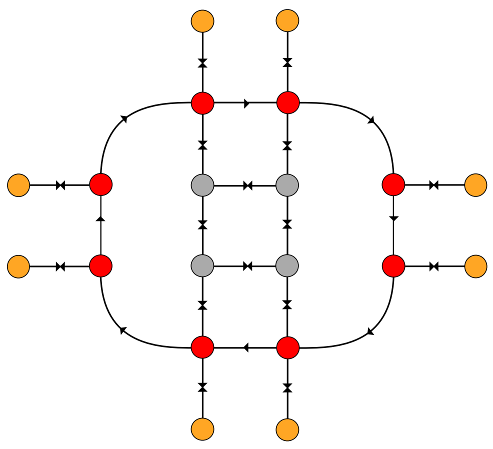

(b) : Small graph with extended capacity using 4 maintenance nodes.

Figure 3.7: Small warehouse graphs.

3.4.2 Algorithm advantages

The calculation of trajectories for a high number of robots in a big warehouse is computationally demanding. Also the goals for robots do not have to be known at the same time and some might be added during the execution of the tasks by robots. Existing approaches usually require to finish the calculation to obtain paths. The proposed algorithm solves both of these issues.

The algorithm moves the state forward in time and only when any conflict occurs it moves the state back in time. For one operation, the time the state is moved back is the maximal time of the conflicted robots moving from last node on the edge. However if the operation causes another conflict before the operation is finished (for example stopping one robot causes a new conflict with another robot), the state could be moved back again. The shift back is not limited and it could be shifted arbitrarily far, but moving back significantly is highly improbable. The algorithm can start running, buffering the solution for some time and then the robots can start moving in real time with low risk of the solution state moving behind the state of the warehouse. Of course an implementation of safety halt of the system when the state of the robots get close to the state of the solution should be implemented. The buffering time must be decided by numerous simulations on a planned warehouse graph with given number of robots. The computational power that is available must also be considered. Results showing how long this buffering time must be in the case of the warehouse graph used in this thesis are shown in Section 4.

The algorithm allows for tasks being added during the calculation. The state of the solution must be moved back to the time when the new robot is supposed to start moving and the robot is simply added with its shortest path to its destination. It might cause new conflicts in previously calculated trajectories, but for conflicts that it does not affect, there is no need for recalculation, while the trajectories of these robots are already collision-free. One issue that might occur is that due to the impact of the newly added robot, some robots will perform operations that are no longer needed. For example the newly added robot r_1 affects another robot r_2 that in previous calculation would get into conflict with the robot r_3. In previous iteration, the robot r_2 pushed the robot r_3 and this trajectory was added to its trajectory. In the next iteration the robot r_1 stops the robot r_2 and it will not get into conflict with the r_3, but while the robot r_3 has the trajectory of the operation already calculated it will still perform it. This might lead to a unnecessary movements and delays.

3.5 Algorithm implementation

The implementation aims to prove the usability of this proposed algorithm. It allows two types of robots (with and without the rack) to move from arbitrary position to a storage location or a maintenance node.

3.5.1 Support applications

The visualization of the solution is important to understand what is actually happening and how the robots are moving. To accomplish this a GUI application had to be developed. I have started working on the basic GUI that is able to load the warehouse maps from xml files. This project was taken over and has grown into three separate applications: FleetManager, CarryFleetSimulator and FleetManagerTerminal. FleetManager simulates the warehouse manager software. The application allows to load the map and the solution from algorithm generated as a list of operations for a list of robots. These messages are sent to the CarryFleetSimulator that simulates the movement of the robots. The state of the warehouse is displayed in FleetManagerTerminal, which is basically the GUI I have developed.

Currently this simulation software is bypassed to only display the calculated trajectories, because of the current incompatibility of its simulator with the proposed trajectory planner. In future this planner is supposed to be running in parallel with these applications to calculate the trajectories and recalculate them when new tasks for robots are given by the FleetManager.

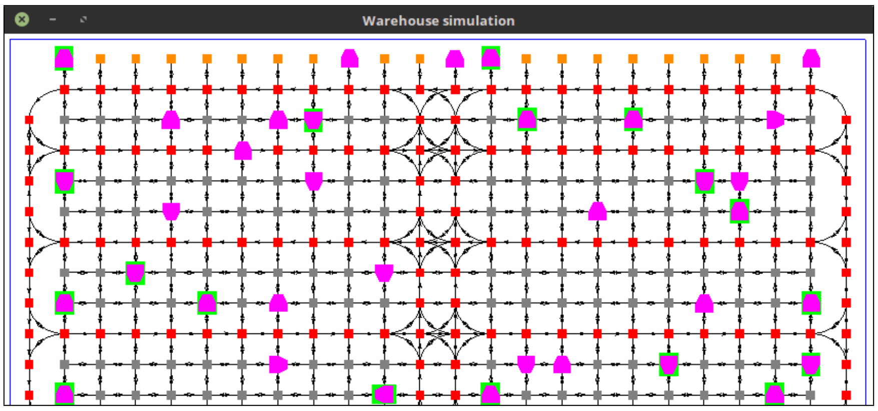


Figure 3.8: The FleetManagerTerminal window.

3.5.2 Programming language and tools

The algorithm and the support applications are written in C++. The algorithm was developed in Linux Mint operating system using CodeLite application to write the source code and CMake to grenerate Makefiles. Compatibility with OSX was added later and the development was done in Xcode to benefit from its debugging capabilities.

For the GUI in FleetManagerTerminal, the SFML [Gom] and SFGUI [ea] libraries were used. To load the map in the FleetManager and in the implemented algorithm the libxml2 library [Vei] is used to process the map files which were created in xml file format. For A* algorithm the boost [Bee] library is used as very fast and reliable implementation.

3.5.3 Arena representation

The warehouse graph is represented as a pair of two vectors. One vector contains all nodes with all needed information (Table 3.1). The second vector contains edges of the graphs (Table 3.3). The Arena contains the warehouse graph, description of tasks and references to the algorithms used during the solution (A* and dijkstra algorithms).

3.5.4 Generated trajectories representation

Each robot has its trajectory which is represented as a list of nodes (in the sense of a list node, not a graph node). These nodes contain full state of the robot, reference to the previous node and next node, but also the first and last node of a segment. The segment is a part which starts with the first list node when the robot occupies a given graph node, contains the rotation at the graph node or time the robot is idle at the graph node and the movement on the next edge. The last list node of a segment is always the last list node before the robot occupies the next graph node.

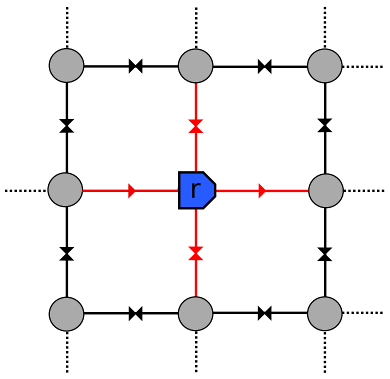
The list object has implemented methods for removing any node and adding nodes or full parts of the trajectory between arbitrary nodes. It also has a function used in *push* and *replan* operations to remove all nodes after certain node.

3.5.5 Robot model

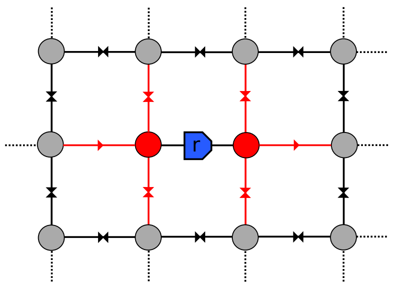
The robot model that is implemented is very simple. One of the main simplifications is that it allows instantaneous change of velocity, allowing the robots to accelerate to maximal velocity and stop immediately. The model allows to generate trajectory from a given path, rotating robots on the spot, moving on straight and elliptic edges. The simplicity of the model is important for the trajectory modification to be implemented. For example to stop a robot, the trajectory to the node at which it will be stopped would have to be modified. The instantaneous change of velocity allows to avoid this complexity and simplify the implementation of the prototype algorithm.

3.5.6 Conflicts

The conflicts definition is one of the most essential parts that needs to be defined correctly for the algorithm to work well. As mentioned in Section 3.1, all edges and oriented nodes have a list of their conflicts with other edges and oriented nodes. The definitions for simple grid-like parts of the graph are simple.



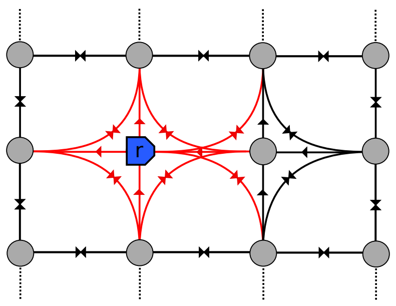
(a) : Conflicts definition for the oriented node the robot r occupies.



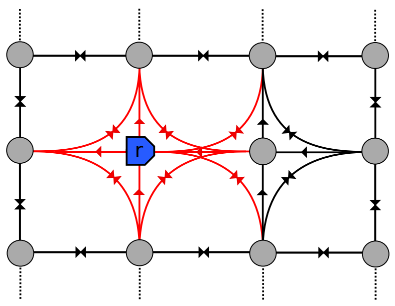
(b) : Conflicts definition for the edge the robot r occupies.

Figure 3.9: Simple conflict definitions.

In Figure 3.9a the robot r occupies the 0-degree oriented node. The conflicts defined for it are other oriented nodes for 90, 180 and 270 degrees of the same node at which it stands and all the exiting and entering edges of the node. It is important that the other nodes around are not considered to be in conflict for other robots to freely move, mainly then solving an conflict. All of the operations move robots back to nodes to ensure they are not in conflict after the operation is competed. Figure 3.9b shows the conflict definition for robots traveling on edges. There are considerably more conflicts than when a robot is occupying a node. The start and end nodes of the edge and all their entering and exiting edges are in conflict while the robot could physically collide with any robots traveling on such edges or standing at such nodes.



(a) : Conflicts definition for the node the robot r occupies.



(b) : Conflicts definition for the edge the robot r occupies.

Figure 3.10: Complex conflict definitions.

The conflict definition gets more complicated at more complex parts of the graph. In the center of the graph used during the development of this algorithm there are junctions with many elliptical edges that cross each other. The robot r in Figure 3.10a occupies the node in the center. In this case the situation is similar as in Figure 3.9a, having all the edges entering and exiting the node in the conflict. But when the robot enters an edge in this part, especially one of the elliptical edges, the definition of conflicts gets very complicated. The situation can be seen in Figure 3.10b. Not only edges entering and exiting the start and end node, but also edges crossing these edges and nodes that are close to the curved edge must be considered to ensure space for the robot to move without collision, in particular for the robots carrying the racks. These conflicts are very complicated and there is no simple way to automate their definition, thus must be chosen by hand.

The used warehouse graph has these definition already set up and they are loaded during the loading of the map from the provided xml file.

3.5.7 Implementation difficulties

The implementation brought several challenges that made it very difficult. The algorithm is very sensitive for any mistakes in the code that leads to crash or a unsolvable situations that cycles the solution. Most of the bugs were fixed but one major issue remains unsolved. The map that was provided has its conflicts defined in its xml file, but some of the conflicts are too strict and do not allow for the algorithm to function. When robots are in conflict at least one of them is moved back to a node, but sometimes the node has defined conflicts in a way that it is impossible for the other robot to move without causing a new conflict. Most of these problematic conflicts do not make sense, while the other robot would not physically hinder it. There are some conflicts that are helping to overcome crash when the robots are rotated is a specific direction while carrying a rack, but the algorithm is not designed for such case. The solution for this issue is to put more space between the nodes to make those conflicts obsolete. These issues were found later in development and to allow the algorithm to work, some conflicts are ignored. It may still happen that there conflicts that cannot be solved are created. To improve the functionality, new conflict generator must be developed, or the conflicts must be cautiously defined by hand.

Chapter 4

Experiments

The goal of the experiments is to assess the usability of the proposed algorithm in practice. The experiments were performed on Linux Mint on a computer with the Intel i7-4771 processor and 8GB RAM. The computer is an older one, thus it is supposed that one with more modern CPU would perform significantly better. While most of the late development was done on macOS, a lot of testing was also done on MacBook Air (early 2014) which performed equally and sometimes even faster than the PC setup.

During the experiments, the warehouse map displayed in Figure 6.1 was used. I have created a task for 50 robots with various distances from the start node to the goal node. There are 22 robots that carry racks and 28 robots without the rack. Only maintenance nodes and storage location nodes were used as start and goal positions for the robots. During the experiments, the algorithm was executed with 2 to 50 robots to study the influence of the growing number of robots to the performance.

4.1 Execution delay

One of the main advantages of the algorithm is discussed in Subsection 3.4.2. The algorithm moves forward in time and the calculated trajectories can be performed before the algorithm finishes. The algorithm can move back in time, thus the risk of the execution being prior to the calculation must be addressed.

The algorithm was executed with 50 robots and the time of the solution state was compared to real-time. In Figure 4.1 one can see that the difference between state time and real-time grows (logarithmic scale was used for an easier visual comparison), thus it is highly unlikely that the lines ever cross. In this case, the buffering time can be very small, thus the robots can start moving towards their goals instantly.

The more difficult the problem is (bigger warehouse, more robots), the higher is the risk of the execution catching up to the algorithm. This can be overcome with more computational resources and reasonable buffering time. One can also assume that not all robots will move at the same time. Some robots might be charging at the maintenance stations while others might be waiting in queue for the pick station.

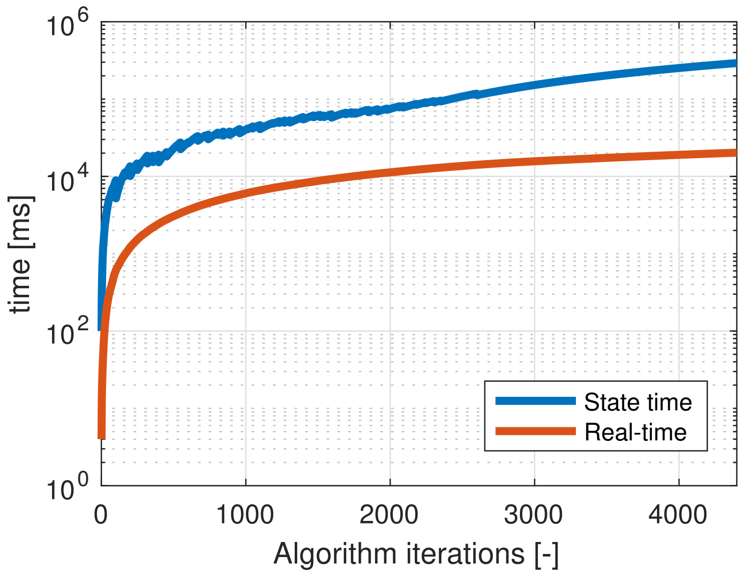


Figure 4.1: Comparison of real-time during the state time of the algorithm during the calculation.

4.2 Two approaches comparison

The ability of the algorithm to add robots to the plan during the calculation was discussed in Subsection 3.4.2. The extreme case of adding robots one by one to the beginning of the solution was tested to assess the impact on the results. The algorithm always calculates the complete solution. Then another robot is added, while the other robots keep their original trajectories. This approach is named *sequential*, while the approach of calculating all n robots at once is named *standard*. Several indicators, for example the run-time and solution time impact, are compared to the standard approach which calculates all robots at the same time.

4.2.1 Number of conflicts comparison

The number of conflicts for each amount of robots from 2 to 50 was recorded using a *standard* approach and then the *sequential* approach was tested. The conflicts from each run accumulated to be comparable. Figure 4.2, one can see that the cumulative sum of conflicts for the *sequential* approach is comparable with the *standard* approach for the amount of robots ranging from 2 to 39. This shows that the newly added robots in this task only cause few new conflicts with comparison with the full calculation.

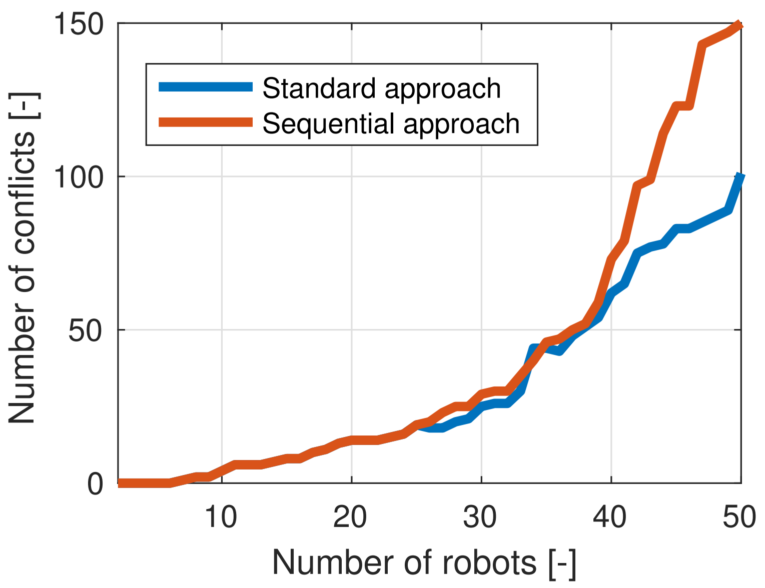


Figure 4.2: Comparison of the number of conflicts between the *sequential* approach and the *standard* approach.

With the growing number of robots, the probability of long parts of the trajectories of robots being replanned and changed completely due to the addition of new robots is growing, thus the conflicts that have been solved previously might be thrown away with the trajectory and thus new conflicts must be calculated. The shape of the curves can vary highly according to the current tasks and situations. The order of the robots in which they are being added also plays a significant role. In this case the robots were added for *standard* approach ranging from 2 to 50 robots in the same order as they were added during the *sequential* approach. This result also confirms the expected property that the number of conflicts grows exponentially with the number of robots.

4.2.2 Calculation time comparison

Perhaps the most significant impact of the *standard* approach is on the solution time. Running the algorithm multiple times trough the whole plan demands significantly more computational resources. In each run less resources are needed due to the fact that most conflicts were already solved. However the cumulative value of calculation time for the *sequential* approach is always significantly higher as seen in Figure 4.3 (a logarithmic scale is used for better visualization). In less extreme cases, adding a task during the calculation will still cause a delay, however, this is not as significant than the recalculation of the whole solution.

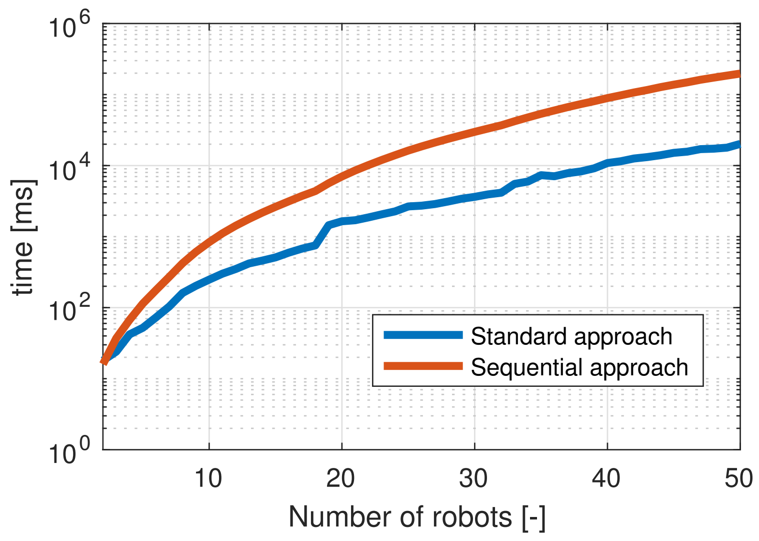


Figure 4.3: Comparison of the calculation time between the *sequential* approach and the *standard* approach.

For example, 40 robots start moving at the same time and the algorithm gets far in front of the real execution. After a few seconds, when the algorithm is almost finished, a task for a robot is added 2 seconds in advance to the real execution. The algorithm will go back and use the already calculated trajectories, thus only newly caused conflicts need to be calculated again. Once more the algorithm gets again far in front of the execution and it continues without any issue. Some extreme cases might occur, thus it is very important to cautiously choose the optimal time reserve when adding a task for a robot.

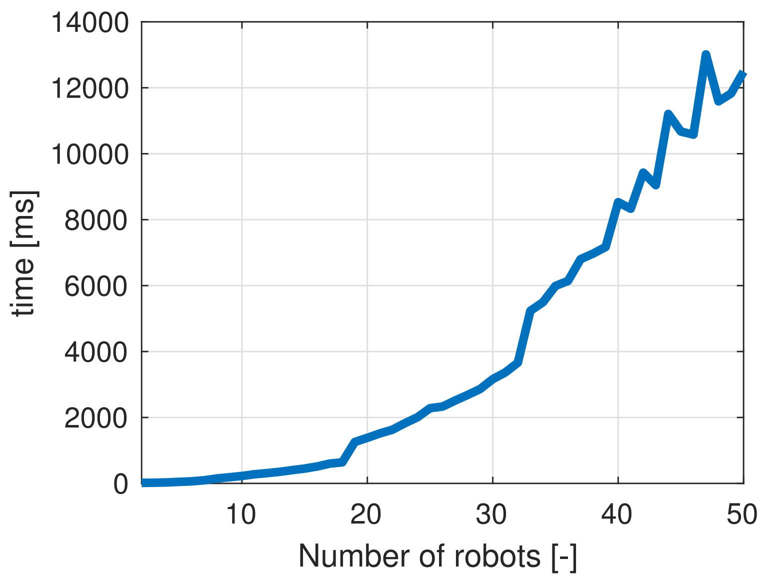


Figure 4.4: Calculation time of adding one robot to the solution.

Figure 4.4 shows the calculation time of adding one robot for a range of 2 to 50 robots. The calculation time grows significantly with the number of robots added, however the main influence is the length of the total solution time, given by the robot with longest trajectories. This can be seen when comparing the calculation time on Figure 4.4 with the solution time on Figure 4.5. The sudden growth of the solution time cause steeper growth of the calculation time. In the real world usage, the robots do not have to wait for the calculation to finish to start execution, not even to add a new task for a robot. The state of the robots can be immediately shifted back to a desired time of start of the new robot and the calculation continues from this point. In case of adding robots in a very short periods (for example 1 robot each 100 ms), the calculation time would be closer to the calculation time of the *standard* approach.

4.2.3 Solution time comparison

In this part I compare the solution time for 2 to 50 robots for both approaches. The solution time is the time it takes for all robots to reach their goal nodes from the beginning of the execution to the end of the execution. The curves in Figure 4.5 show that the solution time is almost identical for both approaches. This is due to the fact that the solution time is highly influenced by the robot with the longest path. This will most probably be a robot with a rack, because these robots cannot move trough the storage-location nodes. They can only move on the road nodes that are most often connected with directed edges. This may mean that the robot will have to travel on a long trajectory. This issue could be shorten simply by adding undirected edges into the warehouse, however that would lead into more conflicts in parts of the warehouse where avoidance of the robots with rack would be difficult.

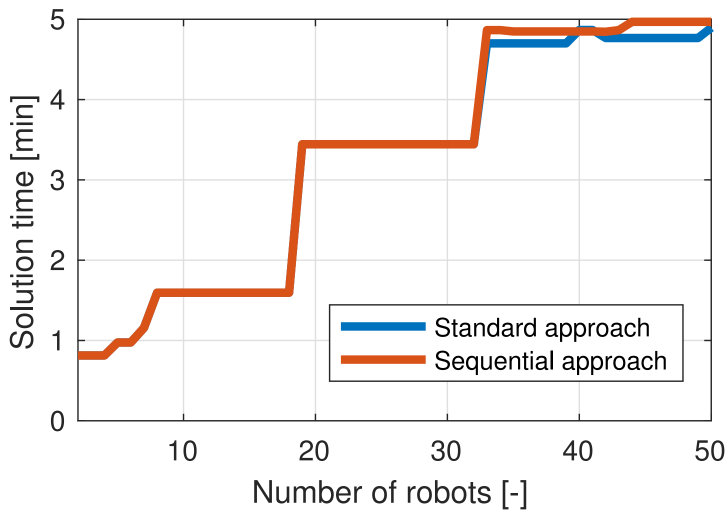


Figure 4.5: Comparison of the solution time between the *sequential* approach and the *standard* approach.

When the number of robots exceeds 31, one can see that the robot with the longest trajectory is being affected differently by the two approaches. In conclusion, the effect of the *sequential* approach on the solution time is not significant.

4.3 Calculated trajectory length

One of the concerns of the multi-agent path-finding task is the optimality of the solution. To obtain an optimal solution for example when using the A* algorithm, would take an unfeasible amount of time or computational resources. Instead, the optimal trajectories for respective robots were calculated using the A* algorithm. Each trajectory is optimal to the respective robot, however the trajectories are not mutually collision-free. This approach is used as a the lowest threshold for the length of the trajectories. The sum of their steps in their trajectories is used. In Figure 4.6, the sum of all robot trajectory steps for an lowest threshold and conflict-free trajectory is shown. Similarly to the previous experiments, the difference starts to be significant in the 30 and more robots range and it diverges fast.

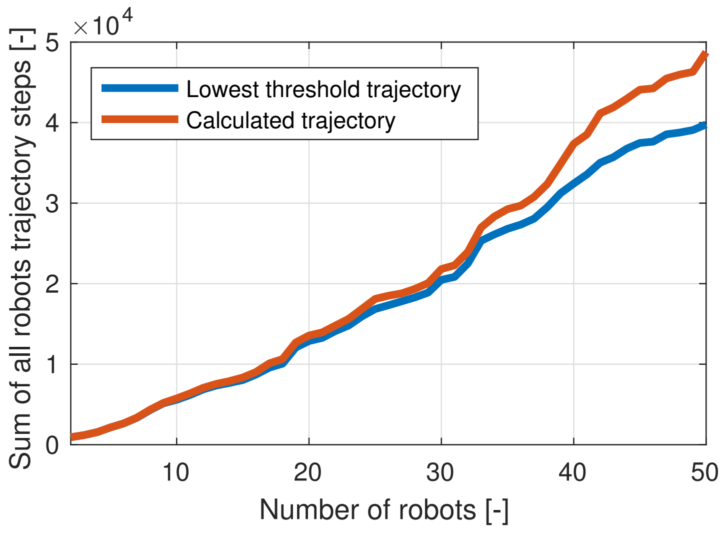


Figure 4.6: Comparison of the sum of the lengths of the lowest threshold trajectories and the conflict-free trajectories.

In Figure 4.7, the difference of trajectory lengths is shown as a growth in percentage. This represents the effect of the operations on the trajectories with an increasing number of robots. There are no conflicts between the first 6 robots, thus their cumulative trajectory length is the same as for the lowest threshold trajectories. The cumulative trajectory length grows with the increasing number of conflicts. The figure is highly correlated with Figure 4.2 which represents the number of the conflicts with increasing number of the robots. The results from this experiment and the others suggest that the optimal number of robots for the warehouse of this size is between 30 to 40.

We suppose that 4 robots will always be charging at maintenance nodes and that the average time spent by picking things by the human worker from the rack is 15 seconds. This means that we are left with 36 robots, thus we have 18 robots for each pick-station. That allows each robot to have 270 seconds or 4.5 minutes to pick up a rack and bring it back to the pick-station. The solution times shown in Figure 4.5 suggest that the longest trajectory for 40 robots is about 5 minutes. This represents the worst case scenario.

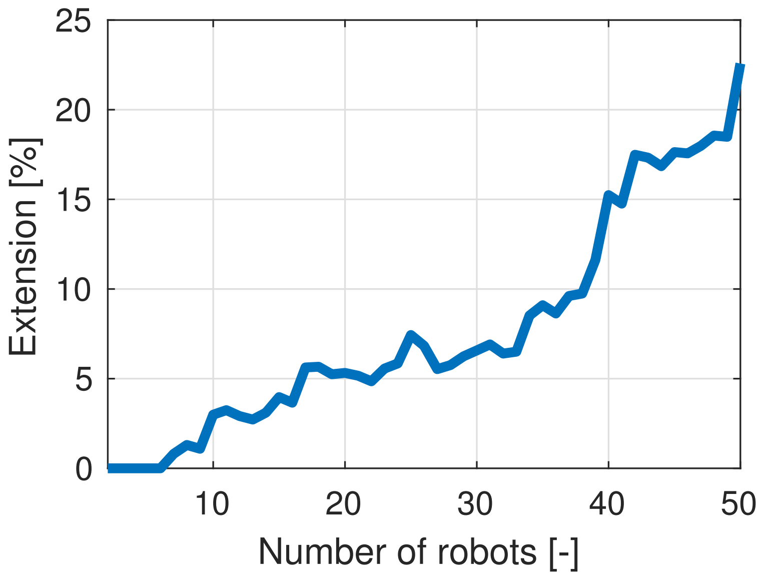


Figure 4.7: Extension of the sum of the lengths of the trajectories compared to the sum of the lengths of the lowest threshold trajectories.

Chapter 5
Conclusion

The goal of this thesis was to design a trajectory planning algorithm for a large number of robots navigating trough a warehouse. The use of such algorithm has a high potential as we are now at the beginning of the 4th industrial revolution and the adoption of automated warehouses is growing. Such solutions are being developed and used by major companies such as Amazon, but the spread of such technology in smaller companies did not yet reach high levels. The goal of this algorithm is to overcome challenges that come with real environment usage which differs highly from common multi-agent path-finding algorithms. These do not consider different velocities of robots, various distances of the nodes in the warehouse graph, and the delay of the robots which is caused by their on place rotation.

Several multi-agent path-finding algorithms had to be studied to gain the necessary knowledge. I have studied the Push and Rotate algorithm in detail. This algorithm was implemented in my colleague's thesis [Luk16].

The designed algorithm is overcoming all of the above-mentioned challenges. It allows a continuous movement of the robots on their trajectories instead of a discrete movement between nodes. It also takes into consideration the rotation of the robots and their different velocities. One of the main achievements is that all the robots can move together which was not possible with the Push and Rotate algorithm. This is achieved by an implemented system of priorities that assures that the robot with the longest trajectory always moves towards its final destination. Whenever the robots are involved in a *push* operation, the accumulation of priorities overcomes problems with deadlocks. The computational time for the solution that has to be calculated before the result can be executed was another challenge. The algorithm solves this issue inherently by taking the shortest trajectories to destinations. The trajectories are only modified during the calculation. Thus if the algorithm is faster than real-time, the solution can be executed before the calculation is finished. This also allows to add robots during the calculation.

Several experiments to show the properties of the algorithm were executed. These experiments present the usability of the algorithm in a real warehouse. Thanks to the solution being generated before the calculation is finished, robots can execute movement without waiting or with a buffer defined delay. Then the influence of adding robots one by one is studied to gain insight of the performance in the case of tasks being added during the calculation. The influence on the final computational resources in extreme cases is significant. It also shows that adding one new robot does not necessarily mean that many new conflicts will be generated and that it is much faster than it would have been if all the trajectories had to be calculated from the beginning. Finally, the calculated trajectories were compared to the lowest threshold trajectories showing that for up to 50 robots in a smaller warehouse, the trajectories will prolong less than 25%. However, this result is highly dependent on the number of conflicts generated by the given task.

The implementation at its current state aims to prove the functionality. Nevertheless there is still room for improvements and some steps must be done for a full implementation in the a automated warehouse. From the functional point of view, the conflict definition must be redone to fit the specifications of the space needed between the robots. An automated algorithm to generate this might be proposed, but for a real-world usage in a warehouse with complicated edges I would rather suggest a manual identification by an expert who understands the space and algorithm limitations related to the conflicts.

Another thing that might be improved is the complexity of the robot model. An extension to better represent the dynamics of the robots could be developed. This improvement would require a reworking of the operations that modify the trajectories. For example if the robot needs to be stopped on a node, the preceding trajectory must be modified to stop it on its spot.

Continuous usage and queue parts of the graphs must be implemented for usage in a real warehouse. The continuous usage should allow the warehouse manager and assign tasks to the robots. The algorithm can simply calculate or wait for new tasks. The parts of trajectories that were already executed by the robots might be deleted to reduce memory usage. The sections of the graph around pick-stations should be managed separately to sort incoming robots into a correct order for a pick-station.

Chapter 6
Appendix

6.1 CD content

File/Folder	Content
TomasNovakThesis.pdf	electronic version of this thesis
parallelpush	the C++ implementation of the proposed algorithm
readme.md	instructions for calculation and replay of tasks
maps	contains the warehouse map
tasks	contains several task files
experiments	video and data recorded during experiments
safelog_fleet_management_system	GUI for trajectories replay

6.2 Figures

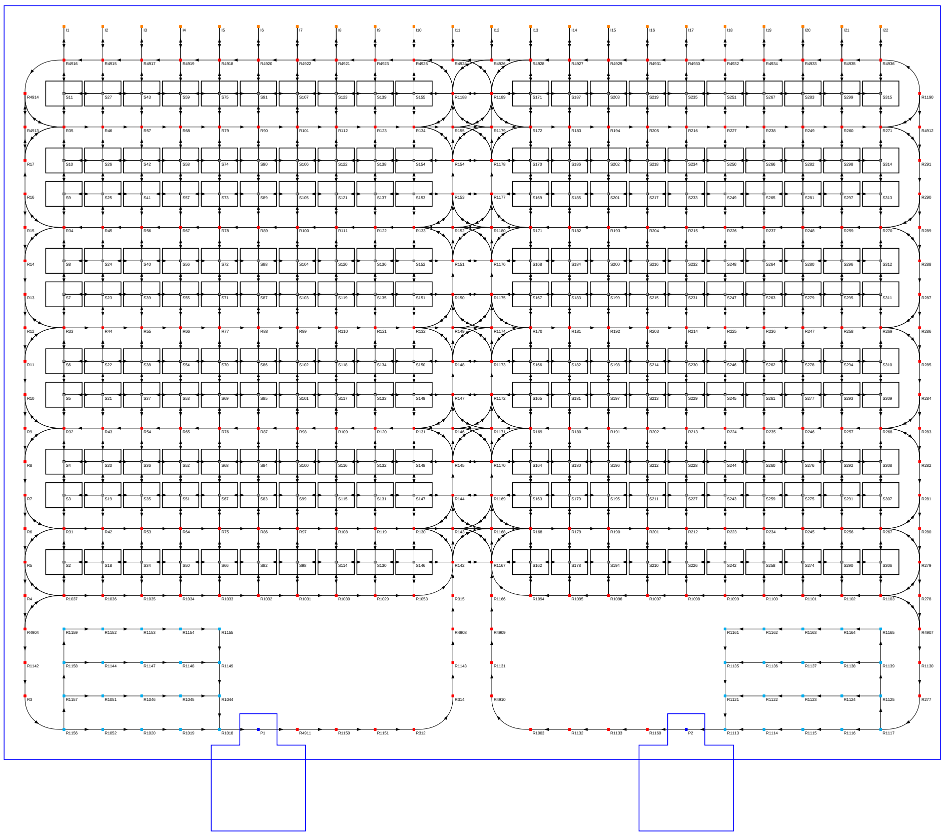


Figure 6.1: Map of a real warehouse.

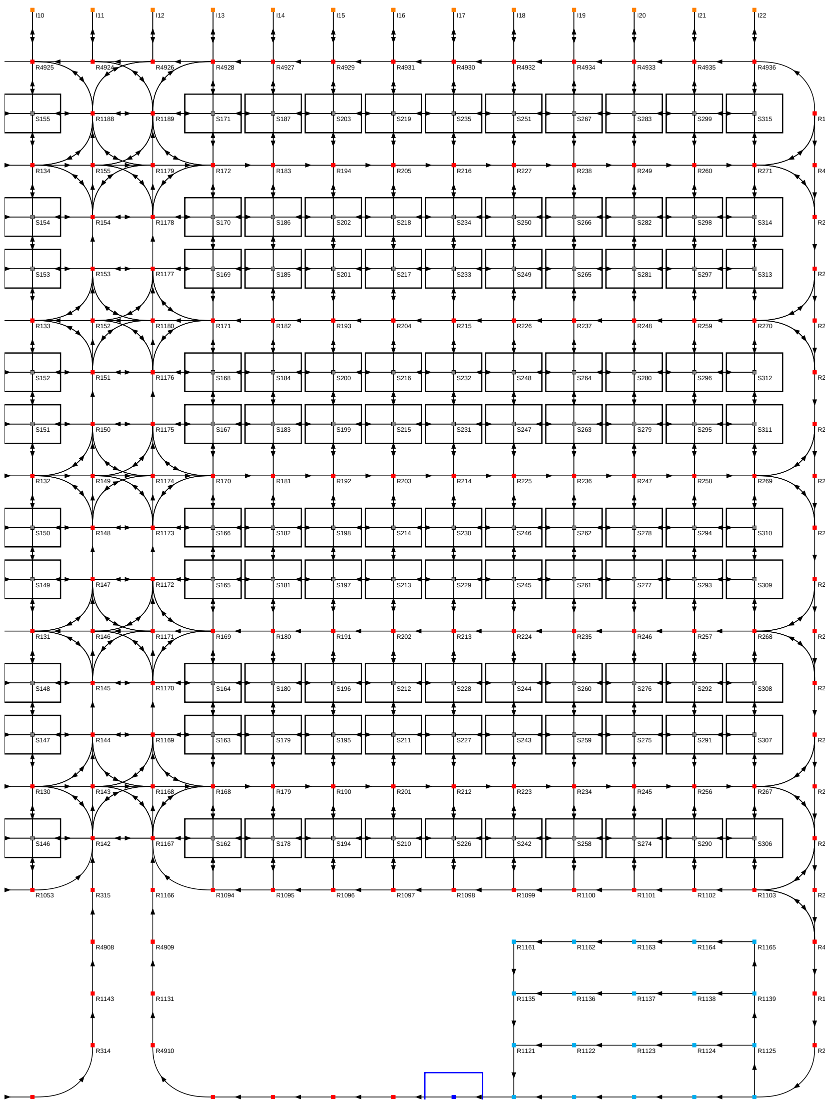


Figure 6.2: Cutout from the map of a real warehouse.

Bibliography

[Bee] Kristopher Beevers, *Boost graph library: A* heuristic search - 1.64.0*, `http://www.boost.org/doc/libs/1_64_0/libs/graph/doc/astar_search.html`, Accessed: May 9, 2017.

[ČR] TA ČR, *TAČR - Vladimír Mařík: Česká republika může na Industry 4.0 hodně vydělat*, `https://goo.gl/85ySQY`, Accessed: May 3, 2017.

[dWtMW14] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen, *Push and Rotate: a complete multi-agent pathfinding algorithm.*, J. Artif. Intell. Res. (JAIR) **51** (2014), 443–492.

[ea] Stefan Schindler et al., *Github - TankOs/SFGUI: Simple and fast graphical user interface*, `https://github.com/TankOs/SFGUI`, Accessed: May 9, 2017.

[Gom] Laurent Gomila, *SFML*, `https://www.sfml-dev.org/index.php`, Accessed: May 19, 2017.

[HSS84] J.E. Hopcroft, J.T. Schwartz, and M. Sharir, *On the complexity of motion planning for multiple independent objects; PSPACE-hardness of the "warehouseman's problem"*, The International Journal of Robotics Research **3** (1984), no. 4, 76–88.

[KMS84] D. Kornhauser, G. Miller, and P. Spirakis, *Coordinating pebble motion on graphs, the diameter of permutation groups, and applications*, 25th Annual Symposium onFoundations of Computer Science, 1984., Oct 1984, pp. 241–250.

[LB11] Ryan Luna and Kostas E. Bekris, *Push and Swap: Fast cooperative path-finding with completeness guarantees*, Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One, IJCAI'11, AAAI Press, 2011, pp. 294–300.

[Luk16] Ing. Jakub Lukeš, *Cooperative path planning for big teams of robots*, Ph.D. thesis, Czech Technical University in Prague, 2016.

[saf] *SafeLog – safe human-robot interaction in logistic applications for highly flexible warehouses*, `http://safelog-project.eu/`, Accessed: January 8, 2018.

[Sie] Jeremy Siek, *Dijkstra's shortest paths*, `http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/dijkstra_shortest_paths.html`, Accessed: December 26, 2017.

[SSH13] Kiril Solovey, Oren Salzman, and Dan Halperin, *Finding a needle in an exponential haystack: Discrete RRT for exploration of implicit roadmaps in multi-robot motion planning*, CoRR **abs/1305.2889** (2013).

[tM11] A. W. ter Mors, *Conflict-free route planning in dynamic environments*, 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sept 2011, pp. 2166–2171.

[Vei] Daniel Veillard, *The XML C parser and toolkit of Gnome*, `http://xmlsoft.org/`, Accessed: January 2, 2018.