Title: Multi-objective multi-constrained large-scale planning

Deliverable: D3.3

Prepared by:

<table>
<thead>
<tr>
<th>Name</th>
<th>Miroslav Kulich</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organisation</td>
<td>CVUT</td>
</tr>
<tr>
<td>Date</td>
<td>July 11, 2018</td>
</tr>
<tr>
<td>Contributor(s)</td>
<td>Jakub Hvězda; Miroslav Kulich;</td>
</tr>
</tbody>
</table>

Approved by:

<table>
<thead>
<tr>
<th>First Reviewer</th>
<th>David Puljiz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second Reviewer</td>
<td>Jonas Stenzel</td>
</tr>
</tbody>
</table>
The SafeLog project is funded by the European Commission within Horizon2020 under GA-Nr. 688117.
Contents

1 Summary 4
2 Consortium 5
3 Glossary 7

Bibliography 9

4 Attachments (Referenced Objects) 9
  4.1 Attachments .................................................. 9
  4.1.1 d3.3-technical_report.pdf ............................. 9
# 1 Summary

The document overviews the main activities done within WPT-3.2 *Multi-objective multi-constrained large-scale planning*. More specifically, three different approaches to multi-robot trajectory planning, which were developed are described as well as a thorough experimental results demonstrating their key properties.

The structure of the attached document ([d3.3-technical_report.pdf](#)) is as follows. State of the art in multi-robot path planning and motion coordination is discussed in Chapter 1, while Chapter 2 is dedicated to the definition of fundamental terms used in the text, especially the solved planning problem. The first proposed approach - Sequential Context-Aware Route Planning (SCARP), which is based on the Context-Aware Route Planning (CARP) algorithm introduced by ter Morse, is introduced in Chapter 3. A probabilistic approach inspired by the Rapidly Exploring Random Tree algorithm is introduced in Chapter 4. We call this approach MRdRRT. De Wilde et al. presented their complete Push and Rotate path algorithm which allows to move a single agent at time. In Chapter 5 we present the extension (called Parallel Push and Rotate) of this algorithm which considers parallel movements of agents and deals with other challenges arising from a deployment in real warehouse.

The experimental results show that all presented methods have interesting properties. MRdRRT significantly improves particular steps of the original algorithm which allows it to solve problems assuming tens of robots in few seconds. This is in contrast to the original MRdRRT which can solve problems up to ten robots in tens of seconds. The experimental comparison moreover shows that the proposed approach can solve problems unsolvable for CARP. Finally, the approach is comparable to CARP in computational time and quality of the generated solutions for problems with up to 100 robots which are solvable by CARP. The main drawback is that the computational complexity with respect to the number of robots is still exponential which disqualifies its usage for large-scale problems.

Parallel Push and Rotate 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. On the other hand, computational complexity is too high to meet SafeLog requirements. This can be particularly solved by taking the shortest trajectories to destinations as initial solution and modifying them during the calculation. Thus if the algorithm is faster than real-time, the solution can be executed before the calculation is finished. Moreover, 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% in comparison to the low-bond trajectories (i.e., shortest trajectories that do not consider conflicts between robots). However, this result is highly dependent on the number of conflicts generated by the given task. Finally, the algorithm is very complex making its understanding and maintenance difficult. This reason together with its worse scalability makes the algorithm inappropriate for SafeLog.

The SCARP approach, on the other hand, performs best from the mentioned algorithms and is thus most promising for further use. It finds better solutions than the original CARP algorithm after several random shuffles of the robots’ priorities while requiring significantly less computational time for adding individual robots into the system. Moreover, it is much faster than running CARP 100 times with various orders of agents which produces similar results. The experimental results show that the algorithm is well scalable and useful to solve problems for hundreds of robots in the required time less than 2 seconds.
2 Consortium

Role of KIT

KIT has two roles:

1. KIT will be coordinator of SafeLog. Coordinating person will be Björn Hein. The department FOSRience of KIT will handle all management issues (s. previous paragraph, section Management structure and procedures and WP8 Project Management in document Part 1).

2. Regarding research and innovation KIT will mainly focus on the human-system interaction and assistive technologies in the envisioned flexible and collaborative warehouse WP4 Assisting technologies for a collaborative and flexible warehouse system with the corresponding relations to the other work packages.

Role of SLA

Swisslog provides expertise in automation and logistics ranging from industrial robot applications, electrical overhead monorails, transport AGVs and goods-to-man systems. Swisslog will handle the demonstrator based on a fleet of mobile goods-to-man robots. For this system prior work exists comprised of fleet-manager, standard safety infrastructure and also a 2D emulation environment. Swisslog will take the lead of WP1 and WP6.

Role of CVUT

CVUT will lead WP3 Planning and scheduling for a heterogeneous fleet manager. The target of the workpackage is to realize a planning module that will provide coordinated plans for robots and humans in the warehouse CVUT will also significantly contribute localization activities in WP2 Integrated safety concept for detecting and localizing of humans as well as specification and requirement analysis WP1 Requirements and Specifications and integration WP6 Integration and Demonstration.

Role of UNIZG-FER

UNIZG-FER will lead WP2 Integrated safety concept for detecting and localizing of humans. The target of the workpackage is development of a holistic safety concept that will allow safe collaboration of humans and robots in the warehouse. UNIZG-FER will also contribute in human aware planning in WP3 Planning and scheduling for a heterogeneous fleet manager, localization and human intention recognition in WP4 Assisting technologies for a collaborative and flexible warehouse system, specification and requirement analysis in WP1 Requirements and Specifications and integration in WP6 Integration and Demonstration.

Role of IML

IML has a comprehensive knowledge about a multitude of interlogistic applications as well as a deep knowledge about development of embedded eletronic components and robotic solution.

In this position IML will contribute to the overall integration of the different concepts by leading the WP6 Integration and Demonstration. Furthermore IML will bring in the expert knowledge in embedded systems and communication technologies to contribute majorly to the safety concept and hardware development of the vest as part of WP4 Assisting technologies for a collaborative and flexible warehouse system.
Role of KEEI

KEEI will lead WP5 Development of a Safety Vest. The goal of this work package is to develop a Safety Vest which enables humans to safely enter and work in a flexible warehouse system with AGVs. Special attention shall be given to safety certification of the safety west and the Safety Concept developed in WP2 Integrated safety concept for detecting and localizing of humans. KEEI will contribute to the Project with its experience in embedded systems design and in development and certification of safety critical control systems for railway applications.
3 Glossary

**AGV**
Automated Guided Vehicle: An Automated Guided Vehicle is a mobile robot that follows markers or wires in the floor, or uses vision, magnets, or lasers for navigation. They are most often used in industrial applications to move materials around a manufacturing facility or warehouse. Application of the automatic guided vehicle has broadened during the late 20th century. 5, 6

**CARP**
Context-Aware Route Planning: In context-aware route planning, there is a set of transportation agents each with a start and destination location on a shared infrastructure. Each agent wants to find a shortest-time route plan without colliding with any of the other agents, or ending up in a deadlock situation. This graph-based planning algorithm was proposed by Ter Mors et al. 4
4 Attachments (Referenced Objects)

4.1 Attachments

4.1.1 d3.3-technical_report.pdf

<table>
<thead>
<tr>
<th>Name</th>
<th>d3.3-technical_report.pdf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creator</td>
<td>kulich</td>
</tr>
<tr>
<td>Created</td>
<td>2018-07-03</td>
</tr>
<tr>
<td>Modified</td>
<td>2018-07-11</td>
</tr>
</tbody>
</table>

Description

Technical description of planning algorithms developed within *WPT-3.2 Multi-objective multi-constrained large-scale planning*. 
Chapter 1

State of the art

Multi-robot path planning and motion coordination has been studied since 1980s and many techniques have been developed during this period, see [1] for a nice overview. This problem (formulated as the warehouseman’s problem) was proved to be PSPACE-complete [2]. For the case where robots move on a predefined graph (like in SafeLog) complexity of the problem can be reduced, nevertheless it is still NP-hard [3], which means that optimal solutions cannot be generally found in a reasonable time for non-trivial instances (e.g., for a number of robots in order of tens).

Solutions to the problem consider either coupled or decoupled approaches. Centralized (coupled) approaches consider the multi-robot team as a multi-body robot for which classical single-robot path planning can be applied in a composite configuration space. Traditional centralized methods are based on complete (i.e., the algorithm finds a solution if it exists or reports that no solution exists otherwise) and optimal classical algorithms and provide optimal solutions [4], [5], [6]. However, these approaches require computational time exponential in the dimension of the composite configuration space so they are appropriate for small-sized problems only. This drawback lead to development of methods that prune the search space. For instance, Berg et al. [7] decompose any instance of the problem into a sequence of sub-problems where each subproblem can be solved independently from the others. The Biased Cost Pathfinding [8] employs generalised central decision maker that resolves collision points on paths that were pre-computed independently per unit, by replanning colliding units around a highest priority unit. Another approach is to design an algorithm based on a specific topology describing the environment. Peasgood [9] presents a multi-phase approach with linear time complexity based on searching a minimum spanning tree of the graph, while an approach for grid-like environments is introduced in [10]. A flow-annotated search graph inspired by two-way roads is built to avoid head-to-head collisions and to reduce the branching factor in search. Nevertheless, the computational complexity is still high (e.g., [7] solves a problem
with 40 robots in 12 minutes, \[10\] needs approx. 30 seconds for 400 robots).

On the contrary, decoupled methods present a coordination phase separated from the path planning phase. These approaches provide solutions typically in orders of magnitude faster times than coupled planners, but these solutions are sub-optimal. Moreover, the decoupled methods are often not complete as they may suffer from deadlocks. These approaches are divided into two categories: path coordination techniques and prioritized planning. Path coordination considers tuning the velocities of robots along the precomputed trajectories to avoid collisions [11], [12], while prioritized planning computes trajectories sequentially for the particular robots based on the robots' priorities. Robots with already determined trajectories are considered as moving obstacles to be avoided by robots with lower priorities [13], [14], [15].

Several computationally efficient heuristics have been introduced recently enabling to solve problems for tens of robots in seconds. Windowed Hierarchical Cooperative A* algorithm (WHCA*) employs heuristic search in a space-time domain based on hierarchical A* limited to a fixed depth [16]. Chiew [17] proposes an algorithm for \(n^2\) vehicles on a \(n \times n\) mesh topology of path network allowing simultaneous movement of vehicles in a corridor in opposite directions with computational complexity \(O(n^2)\). Luna and Bekros [18] present and complete heuristics for general problems with at most \(n - 2\) robots in a graph with \(n\) vertices based on combination on two primitives - "push" forces robots towards a specific path, while "swap" switches positions of two robots if they are to be colliding. An extension which divides the graph into subgraphs within which it is possible for agents to reach any position of the subgraph, and then uses "push", "swap", and "rotate" operations is presented in [19]. Wang and Botea [20] identify classes of multi-robot path planning problems that can be solved in polynomial time (Slidable) and introduces an algorithm with low polynomial upper bounds for time, space and solution length. Finally, Wang and Wooni [21] formulate multi-robot path planning as an optimization problem and approximate the objective function by adopting a maximum entropy function, which is minimized by a probabilistic iterative algorithm.

Although many of the approaches mentioned above have nice theoretical properties, the most practically usable algorithm is probably Context-Aware Route Planning (CARP) presented in [22] as it is fast, produces solutions of high quality, and although it is not complete, it finds a solution for a large number of practical setups.

The algorithm codes information about trajectories of robots with higher priorities into the planning graph rather than into the planning algorithm itself. It does so by constructing a resource graph, where each resource can be for example a node of the original graph or intersection graph edges. Every such resource holds information about time intervals in which it is not occupied by already planned robots. An adaptation of the A* algorithm is
used on this graph to find the shortest path through these intervals (called
free time windows) to obtain a path that avoids all already planned robots.

[23] deals with a real-life problem of routing vehicles in Container Ter-
minal Altenwerder in Hamburg harbor. They used a similar approach to
keeping a set of free time windows for path arcs in the graph. Their algo-
rithm contains a preprocessing of the graph for the use of specific vehicles
followed by computation of paths for individual vehicles on this preprocessed
graph.

Another similar approach is presented in [24], where each robot looks for
a viable path in a 2D spatial grid map and checks for collisions with moving
obstacles using a temporal occupancy table. [25] adopted a similar approach
to [22] with enhanced taxiway modeling approach to improve performance
on airport graph structures. [26] compares Context-Aware Route Plan-
ing (CARP) [22] with a fixed-path scheduling algorithm using $k$ shortest
paths [27] and a fixed-path scheduling algorithm using $k$ disjoint paths [28].
The experiments show that the CARP algorithm is superior in all measured
qualities.

[29] compares several heuristic approaches of assigning priority to robots
and concludes that the heuristics which plans longest paths first perform
best when a makespan is to be minimized. A greedy best-first heuristics
provides best results regarding joint plan cost. However, its downside is that
it calls the planning algorithm for all yet unplanned robots in every round
and it is thus very time-consuming.

Finally, Solovey et al. [30] presented the Multi-Robot Discrete Rapidly-
Exploring Random Tree (MRdRRT) algorithm, which is a probabilistic ap-
proach for path planning on predefined structures for relatively small number
of robots inspired by RRT algorithm [5]. Dobson et al. [31] improve upon
MRdRRT by presenting its optimal variant.
Chapter 2

Problem definition and used terms

2.1 Problem

Multi-agent pathfinding/coordination is a problem that is concerned about finding paths for multiple agents from their given start locations to their target locations without colliding with each other or obstacles in the environment while also optimizing a global cost function.

To specify the problem more precisely, assume:

- A set of \( k \) homogenous agents each labeled \( a_1, a_2, ..., a_k \).

- A graph \( G(V, E) \) where \( |V| = N \). The vertices \( V \) of the graph are all possible agent’s locations, while \( E \) represents a set of all possible agent’s transitions between the locations.

- A start location \( s_i \in V \) and a target location \( t_i \in V \) of each agent.

The aim is to find a set of collision-free trajectories on \( G(V, E) \) each of them specifying locations of an individual agent at all time moments so that agents are at their start locations initially and at their goal locations finally. Note that the time is discretized into time moments to simplify the problem.

We can also view nodes and edges as resources, where each resource has two main attributes associated with it. These are capacity \( c(r) \) which corresponds to the maximum number of agents that can occupy a resource at the same time and duration \( d(r) > 0 \) that represents the minimal time it takes the agent to traverse a given resource. Plans for every agent then contain not only sequence of resources on its path but also time intervals during which the agent visits them.

For general purpose planning the algorithm assumes that the resource graph is constructed such that resources are of two types: intersection resources (nodes) with capacity 1 and lane resources (edges) with capacity 1.
or greater. Another assumption is also that if multiple agents are present on the same resource then they are all traveling in the same direction and their order does not change, meaning they cannot overtake each other. The idea is that the lanes are not wide enough for two agents to drive in parallel but long enough so that agents can drive behind each other. The capacity is a simplifying assumption introduced because it eliminates the need to calculate collisions during the planning phase and only capacity of resource and free time windows overlap are thus taken into consideration.

The following paragraphs explain key used terms and specify additional constraints to the generated trajectories.

2.1.1 Actions

Every agent can perform two types of action at each time point: It can either move into one of the neighboring nodes, or it can wait at its current location. Every algorithm can make different assumptions regarding the cost of these actions, but we assume that staying idle has zero cost of distance traveled, but costs time. Furthermore, once an agent reaches its target location, it waits at this location for other agents to finish.

2.1.2 Constraints

The main constraints on agent movement assumed in this document are:

- No two agents $a_1$ and $a_2$ can occupy the same vertex $v \in V$ at the same time.

- Assume two agents $a_1$, $a_2$ located in two neighboring nodes $v_1, v_2 \in V$ respectively, they can not travel along the same edge $(v_1, v_2)$ at the same time in opposite directions. In other words, two neighboring agents cannot swap positions. However, it is possible for agents to follow one another assuming that they do not share the same vertex or edge concurrently. For example, if the agent $a_1$ moves from $v_2 \in V$ to $v_3 \in V$ then the agent $a_2$ can move from $v_1 \in V$ to $v_2 \in V$ at the same time.

2.1.3 Route plan

Given a start resource $r$ and a goal resource $r'$ the route plan is a sequence $\pi = (\langle r_1, T_1 \rangle, ..., \langle r_n, T_n \rangle)$ of $n$ plan steps where $T_i = [t_i, t_i']$, such that $r_1 = r$, $r_n = r'$, $t_1 \geq t$ and $\forall j \in \{1, ..., n\}$:

1. interval $T_j$ meets interval $T_{j+1}$ ($j < n$). This constraint means that exit time from $j^{th}$ resource must be equal to the entry time to the $j + 1^{st}$ resource.
2. \(| r_j | \geq d(r_j)\) meaning that the agents occupation time of a resource is at least sufficient to travel across the resource in the minimum travel time.

3. \((r_j, r_{j+1}) \in E_R\) means that if two resources follow each other in the plan, then there must be an edge between them in the resource graph.

2.1.4 Resource load

Given a set of agents plans \(\mathcal{P}\) and a set of all time points \(T\), the resource load is a function \(\lambda : R \times T \to N\) which returns a number of agents that are located at resource \(r\) in a given time \(t \in T\):

\[
\lambda(r, t) = |\{(r, T) \in \pi | \pi \in \mathcal{P} \land t \in T\}|
\]

What this means is that an agent can use a resource only in such time intervals where the resource is occupied by less agents than its capacity. By entering the resource only in these time windows it is ensured that no conflicts with other agents occur in agents plan.

2.1.5 Free time windows

Given a resource load function \(\lambda\), a free time window on resource \(r\) is a maximum interval \(w = [t_1, t_2)\) such that:

1. \(\forall t \in w : \lambda(r, t) < c(r)\)
2. \((t_2 - t_1) \geq d(r)\)

What these conditions mean is that for an interval to be a free time window the capacity in each time point of the interval must be always sufficient and also long enough so that the agent can travel across the resource. Because every agent that wants to traverse a free time window on a resource must enter the window, travel across and then leave it, it cannot enter at the end of a free time window or leave at the start of one because of non zero traversal time. For this reason every free time window \(w\) has entry window \(w_{\text{entry}}\) and exit window \(w_{\text{exit}}\) associated with it. These are limited by a time window \(w\) by the minimum traversal time through the resource:

- \(w_{\text{entry}} = [t_1, t_2 - d(r)]\)
- \(w_{\text{exit}} = [t_1 + d(r), t_2]\)

If agent desires to travel from one resource \(r_1\) to the neighbouring resource \(r_2\) it needs to find free time windows \(w, w'\) on both of these resources. Due to constraints on route plan discussed in Section 2.1.3 for \(w'\) to be reachable from \(w\) the entry window \(w'_{\text{entry}}\) must overlap with exit window \(w_{\text{exit}}\).
2.1.6 Free time window graph

Free time window graph $G_W = (W, E_W)$ is a directed graph where vertices $w \in W$ are a set of free time windows and edges $E_W$ specify reachability between free time windows of $W$. This means that given two free time windows $w, w'$ on resources $r, r'$ respectively it holds that $(w, w') \in E_W$ only if:

- $(r, r') \in E_R$
- $w_{exit} \cap w'_{entry} \neq \emptyset$

Each agent uses its own free time window graph for planning his route as every free time window graph contains only information about $n - 1$ previous agents. It does not contain information about movements of such agents. For this reason some assumptions need to be made about the graph of start and end resources for each agent because otherwise it would be possible that some agent $i$ could make it impossible for agent $i + 1$ to find his plan. These assumptions are for example that no two agents can have the same destination, the destination resources have sufficient capacity to hold all agents that have them as their goal or that once each agent reaches his destination he vanishes from the infrastructure.

2.1.7 Composite configuration space

The composite configuration space $G = (V, E)$ is a graph that is defined as follows. The vertices $V$ are all combinations of collision-free placements of $m$ agents on the original graph $G$. These vertices can also be viewed as $m$ agent configurations $C = (v_1, v_2, ..., v_m)$, where an agent $a_i$ is located at a vertex $v_i \in G$ and the agents do not collide with each other. The edges of $G$ can be created using either Cartesian product or Tensor product. For the purposes of this document the Tensor product is used because it allows simultaneous movement movement of multiple agents and thus for two $m$ agent configurations $C = (v_1, v_2, ..., v_m)$, $C' = (v'_1, v'_2, ..., v'_m)$ the edge $(C, C')$ exists if $(v_i, v'_i) \in E_i$ for every $i$ and no two agents collide with each other during the traversal of their respective edges.

The distance between two neighboring nodes $C_1 = (v_{11}, v_{12}, ..., v_{1n})$ and $C_2 = (v_{21}, v_{22}, ..., v_{2n})$ in a composite roadmap is calculated as the sum of Euclidean distances $d$ between the corresponding nodes:

$$\delta (C_1, C_2) = \sum_{i=0}^{n} d(v_{1i}, v_{2i})$$
Chapter 3

Sequential Context-Aware Route Planning

3.1 Context-Aware Route Planning

The classical shortest path planning expects that if a node $v$ lies on the shortest path from $s$ to $t$, then the shortest path to $v$ can be expanded to shortest path to $t$. However, this approach may run into difficulties in certain scenarios. Fig. 3.1 shows one such scenario. The agent $A$ has his goal node in node 4 and the agent $B$ has his goal in node 0. Consider that agent $B$ is already planned. In this case agent $A$ cannot be planned because partial path to node 1 cannot be further expanded towards goal because it would cause collision with agent $B$. What is required of agent $A$ is to move to node 2, wait for agent $B$ to pass and then move towards goal node 4. However, this is not possible in classical shortest planning, but is possible for the algorithm described in this section.

The main idea of the Context-Aware Route Planning (CARP) algorithm [22] is that it considers only partial plans leading to the free time window on a resource as opposed to a classical planning approach that considers partial plans to whole nodes. If the partial plan arrives to resource $r$ at time $t$ which lies in the free time window $w$ then every other partial plan that arrives to the same time window on the resource $r$ in time $t' > t$ can be simulated by waiting in resource $r$ from $t$ to $t'$. This approach allows agent $A$ in the example on Figure 3.1 to move to node 2, wait for agent $B$ to pass to his goal and then move directly to goal node 4.

The main algorithm performs a search through the free time window graph in a similar way to A*. The algorithm keeps track of open partial plans with their values $f = g + h$ where $g$ is the actual time cost of the partial plan and $h$ is heuristic estimate of a cost of a plan to goal resource from the end of the partial plan. Our implementation assumes that each edge is traversed in one unit of time which enables the heuristic estimate
Figure 3.1: Example problem where classical planning approach can not find solution. Agent A has node 4 and agent B has his goal in node 0. Agents are planned in B,A order.

Algorithm 1: The CARP algorithm

1 if $\exists w [w \in W \mid t \in w_{entry} \land r_1 = resource(w)]$ then
2 mark(w,open)
3 entryTime(w) ← t
4 end
5 while open $\neq \emptyset$ do
6 $w ← \arg\min_{w' \in open} f(w')$
7 mark(w,closed)
8 $r ← resource(w)$
9 if $r = r_2$ then
10 return followBackPoints(w)
11 end
12 $t_{exit} ← g(w) = entryTime(w) + d(resource(w))$
13 for $w' \in \{\rho(r,t_{exit}) \setminus closed\}$ do
14 $t_{entry} ← \max(t_{exit}, start(w'))$
15 if $t_{entry} < entryTime(w')$ then
16 backpointer(w') ← w
17 entryTime(w') ← $t_{entry}$
18 mark(w',open)
19 end
20 end
21 end
22 return null
to be the Euclidean distance to the goal node. The search process can be seen in Algorithm 1. The first step is to check whether there exists a time window \( w \) on a resource \( r \) such that \( t \in w_{\text{entry}} \) (line 1). In case no such window exists then no plan exists and thus \( \text{null} \) is returned. If such window exists it is marked as open and a time \( t \) is marked as an entry time into the window \( w \) (lines 2-3). On line 5 a partial plan with the minimum cost \( f(w) = g(w) + h(w) \) is selected and marked as closed (line 6). If a resource \( r \) that is associated with the window \( w \) is the goal resource \( r_2 \) then the shortest path to \( r_2 \) has been found and it is returned through following back pointers. If the heuristic used to estimate \( h \) is consistent then no other partial plan on the open list have higher cost and expansion of these partial plans would never create a plan with lower cost. If a resource \( r \) is not the goal resource then an exit time \( t_{\text{exit}} \) from the window \( w \) is found as \( \text{entryTime}(w) + d(r) \). Once the exit time \( t_{\text{exit}} \) is found then the algorithm iterates over all reachable time windows \( w' \in \rho(r, t_{\text{exit}}) \) where \( \rho(r, t_{\text{exit}}) \) is a set of all reachable time windows from \( w \) and earliest exit time \( t_{\text{exit}} \). For each of these windows an entry time \( t_{\text{entry}} \) is found as a maximum of \( t_{\text{exit}} \) and a start of window \( \text{start}(w') \). If the entry time \( t_{\text{entry}} \) is smaller than the entry time to \( w' \) then \( w' \) is marked as open and added to the open list as well as update the entry time into \( w' \) to \( t_{\text{entry}} \) and back pointer to \( w \). In case where no plan to the goal \( r_2 \) exists the algorithm returns \( \text{null} \).

At the start of the algorithm all resources start with one available free time window \([0, \infty)\). After finding a plan for each agent the free time window graph is updated by removing the time intervals used in the plan of previous agent.

### 3.2 Sequential algorithm

As mentioned, the proposed algorithm employs CARP [22]. It was chosen because of its ability to plan sequentially one robot at a time while keeping the information about movement of other robots in a compact and easy to update manner. More specifically, CARP uses the infrastructure described above to find paths in a resource graph, where each node of the infrastructure keeps information about its capacity and occupancy in given time windows. This allows it to use a modified A* algorithm that finds a free time window in the start resource corresponding to a start node and attempts to find a path through these time windows to a free time window in the resource that corresponds to the goal node.

The proposed algorithm aims to generate a trajectory for a robot \( a_k \) assuming that trajectories for \( k - 1 \) robots are already planned which can possibly lead to modification of those planned trajectories. The main idea is to iteratively build a set of robots whose trajectories mostly influence an optimal trajectory of \( a_k \), see Algorithm 2. The algorithm maintains two
structures:

- $\mathcal{N}$ – a set of robots in the neighborhood of $a_k$ which is initially set to contain $a_k$ (line 3), and

- $A$ – a sequence of robots not in $\mathcal{N}$. This sequence initially stores the order in which trajectories of the robots $a_1 \ldots a_{k-1}$ were generated (line 1).

Moreover, cost of the best solution that has been found so far is set to a high number.

**Algorithm 2: Plan update**

Input: $G = (V, E)$ – a graph

- $\mathcal{T} = \{t_i\}_{i=1}^{k-1}$ – already planned trajectories
- $V_S$ – start position
- $V_G$ – goal position
- $M$ – size of a neighborhood

Output: $\mathcal{T}^{new} = \{t_i\}_{i=1}^{k}$ – updated set of trajectories

1. $A \leftarrow \langle a_1, a_{k-1} \rangle$
2. $t_k \leftarrow$ shortest path from $V_S$ to $V_G$ in $G$
3. $\mathcal{N} \leftarrow \{a_k\}$
4. $best \leftarrow \infty$
5. $M - 1$ times do
   6. $a_b = \text{argmin}_{a_i \in A} d(a_i, \mathcal{N})$
   7. $A \leftarrow A \setminus \{a_b\}$
   8. $\mathcal{N} \leftarrow \mathcal{N} \cup \{a_b\}$
   9. $\mathcal{C} \leftarrow \text{CARP}(G, A)$
   10. foreach $\pi \in \Pi(\mathcal{N})$ do
       11. $\mathcal{P} \leftarrow \text{CARP}(G, \pi, \mathcal{C})$
       12. $c \leftarrow \text{cost}(\mathcal{C} \cup \mathcal{P})$
       13. if $c < best$ then
           14. $best \leftarrow c$
           15. $\mathcal{T}^{new} \leftarrow \mathcal{C} \cup \mathcal{P}$
       end
   end
17. return $\mathcal{T}^{new}$

New robots are iteratively added to the neighborhood in the loop starting at line 5. The robot $a_b \in A$ which minimizes the distance to the neighborhood is found at each iteration first, where the distance of a robot to a set is defined as the distance of the robot to the closest robot in the set. The
distance between two robots is then determined as the average Euclidean distance of robots’ positions at discrete time steps:

\[ d(a_i, a_j) = \frac{\sum_{\tau=\tau_S}^{\tau_G} |t_i(\tau), t_j(\tau)|}{\tau_G - \tau_S}, \]

where \( t_i(\tau) \) is a position of \( a_i \) at time \( \tau \) if it follows the trajectory \( t_i \), \( t_j(\tau) \) is a position of \( a_j \) at time \( \tau \) if it follows the trajectory \( t_j \), and \( \langle \tau_S, \tau_G \rangle \) is a time interval when \( a_i \) or \( a_j \) moves. Note that trajectories of robots in \( A \) are initially taken from the input set \( T \) while an initial trajectory of \( a_k \) is determined as the shortest path between its start and goal positions on \( G \) making use of the \( A^* \) algorithm (line 2). These initial trajectories are updated as soon as new plans are found at the next steps of the algorithm.

The found closest robot \( a_k \) is then removed from the sequence \( A \) (line 7), added to a set of neighbors \( N \) (line 7) and new trajectories for \( A \) are computed by the CARP algorithm from the scratch (line 9) as demonstrated in Fig. 3.2.

![Figure 3.2: Adding an robot into a set of neighbors.](image)

All possible permutations of robots in \( N \) are considered next (line 10). A set of trajectories \( P \) is determined by the CARP algorithm taking into account trajectories \( C \) for each such permutation (line 11). This is realized by running CARP for \( N \) on a resource graph with free windows generated by CARP when computing \( C \) at line 9. The set \( P \) is then added to \( C \), and the cost of this solution is computed (line 12) and compared with the best solution found till now (line 13). If the new solution is better than the currently best, it is stored together with its cost (lines 14 and 15). The best-found solution is finally reported at line 18.

Calculation of computational complexity of the proposed algorithm is based on the fact that CARP for \( n \) robots comprises \( n \) calls of \( A^* \). We call CARP \( M - 1 \) times at line 9 gradually for \( k - 2, k - 3, \ldots, k - M - 1 \) robots which leads to \( \frac{M}{2} (2k - M - 3) \) calls of \( A^* \). Similarly, CARP at line 11 is called \( \sum_{N=2}^{M} N! \) times which leads to \( \sum_{N=2}^{M} NN! \) calls of \( A^* \).
The total number of A* calls can be significantly reduced in two ways. Firstly, when calling CARP at line 9 after removal of $a_b$ not all trajectories have to be recomputed. We can instead preserve trajectories of robots which were in $A$ before $a_b$ as they are not influenced by $a_b$. Only trajectories of robots behind $a_b$ in $A$ have to be recomputed which leads to a reduction of A* calls by 50% in average.

The second reduction is similar. If the permutations are generated in a lexicographic order at line 10 then two consecutive permutations have typically a big joint head as depicted in Fig. 3.3. Plans of robots in that head can be preserved while recomputation has to be done only for the rest. Assuming neighborhood size $|\mathcal{N}| = 4$, instead of calling $A^* 4 \times 4! = 96$ times, only 64 calls is performed which is 67%. The reduction is even greater for $|\mathcal{N}| = 5$: 325 calls instead of 600 which is 54%, while only 45% (1956 instead of 4320) A* calls are needed for $|\mathcal{N}| = 5$.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 3.3: First permutations of five elements in the lexicographic order. The yellow plans can preserved.

### 3.3 Experiments

The experiments were performed on a computer equipped with Intel Xeon E5-2690. The maps the experiments were performed on were created to show how the proposed algorithm performs depending on the density of the given graph, specifically the number of edges. The set of 21 maps was generated by creating a minimum spanning tree of a $20 \times 20$ grid map and then iteratively adding a given number of original edges to it until the original grid was recreated. Furthermore, 500 different random assignments were generated for a fleet of 100 robots by randomly sampling start and goal nodes for each robot. Each of these assignments was tested on all 21 maps.

The goal of the experiments was to test how the proposed algorithm scales with the number of edges in the graph as robots are sequentially added to the system. For comparison we ran the original CARP algorithm with no change to priorities of the robots, i.e., priorities were set randomly.
Because this approach proved to have a high failure rate, we introduced two variants that after each planning attempt randomly shuffled the robot order 10 and 100 times (CARP10, CARP100 respectively) and tried planning again from scratch. The best plan regarding the sum of the number actions of individual robots was considered as the result. We also added CARP with the longest first heuristic to determine robot priorities as introduced in [29] to the comparison as LF. The proposed algorithm was run in several variants that differ in the parameter $M$ specifying the size of the neighborhood. For the parameters 4, 5, 6 (Proposed_4, Proposed_5, Proposed_6 respectively) all the permutations of the neighborhood were considered. For the parameter 10 (Proposed_10) a 150 different neighborhood permutations were chosen randomly to decrease computational complexity.

Orders for all 100 robots were presented sequentially to all algorithms on all maps and assignments in the first experiment. Failure rate (Fig. 3.5a) and the number of actions of all robots were observed (Fig. 3.5b). In the second experiment, the algorithms were sequentially given one robot at a time at each iteration for all 500 assignments on a map shown in Fig. 3.4 with the goal of showing how much time it takes to generate a plan for each algorithm after adding robot into the system.

The results for the failure rate of the algorithms can be seen in Figure 3.5a. It can be seen that the proposed algorithm has a fail rate in between
Table 3.1: Times of addition for 50th and 100th robot

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time for 50th robot [ms]</th>
<th>Time for 100th robot [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed_4</td>
<td>3.23</td>
<td>7.6</td>
</tr>
<tr>
<td>Proposed_5</td>
<td>10.86</td>
<td>19.48</td>
</tr>
<tr>
<td>Proposed_6</td>
<td>50.58</td>
<td>75.81</td>
</tr>
<tr>
<td>Proposed_10</td>
<td>43.84</td>
<td>54.10</td>
</tr>
<tr>
<td>CARP</td>
<td>0.63</td>
<td>1.60</td>
</tr>
<tr>
<td>CARP10</td>
<td>5.29</td>
<td>12.15</td>
</tr>
<tr>
<td>CARP100</td>
<td>52.08</td>
<td>126.57</td>
</tr>
<tr>
<td>LF</td>
<td>0.85</td>
<td>2.50</td>
</tr>
</tbody>
</table>

the fail rates of CARP and CARP10.

The Figure 3.5b shows the average overall quality of the plan from the first experiment measured as the total number of actions of all robots. The graph shows that all versions of the proposed algorithm perform similarly to each other with Proposed_10 having the best results. Compared to the basic CARP algorithm the proposed algorithm has up to 65 less number of total actions performed across all robots in Proposed_6 variant (on the 10th map) and up to 35 fewer actions for Proposed_6 and Proposed_10 on the original grid map. Moreover, all variants of the proposed algorithm outperform CARP10 and are at least comparable to CARP100 which is much more time-consuming. It can also be noticed that LF generates worst results. It is not much surprising as it was designed to optimize a makespan.

The results of the second experiment are presented in Fig. 3.5c where a time required to plan trajectories for all 100 robots is presented and in Fig. 3.5d which shows the required time to find a trajectory for k-th robot considering trajectories of robots 1...k−1 are already planned. It is evident that CARP100 is the slowest of all tested algorithms with Proposed_6 as the second slowest. It is worth noticing that for Proposed_6 the time of testing all permutations of the neighborhood took longer than replanning of the rest of the robots. The table 3.1 shows the actual times to plan 50th and 100th robot in this experiment.

Finally, we also studied scalability of the algorithms by performing experiment for 200 and 300 robots. The map in the experiment was gradually build in the same way as in the previous case, but the size of the supporting grid was increased to 100×100 nodes. Besides already mentioned algorithms, a simplified version of the proposed approach (Direct) is compared in which CARP is not run as soon as a new robot is added to a set of neighbors at line 9 in Alg. 2. Instead, CARP for the sequence $A$ is run once, when the whole set $\mathcal{N}$ is determined. Similarly, permutations are computed only for the whole size of the set of neighbors.

The results are depicted in Fig. 3.6. It can be seen that shapes of all
curves are similar to those presented for 100 robots. We can thus state that performance of the proposed approach does not dramatically change with an increasing number of robots in comparison to the original CARP algorithm.

From the results as a whole we can see that Proposed_4 and Proposed_5 show the best ratio between the quality of the found plan and the time required to compute it. Additionally, if the time requirements for planning are not as tight, it is possible to run a more demanding version of the algorithm to increase the quality of the solution. To increase the overall success rate of the algorithm, it is possible to combine the proposed algorithm with a version that has a higher success rate, such as CARP100 or even higher in case the proposed algorithm does not find a solution.

Figure 3.5: Comparison of the proposed approach with CARP.
Figure 3.6: Comparison of the proposed approach with CARP for a fleet of 200 robots (left) and 300 robots (right).
Chapter 4

Improved Discrete Multi-Robot Rapidly Exploring Random Tree

4.1 Discrete Multi-Robot Rapidly Exploring Random Tree

A discrete multi-robot rapidly-exploring random tree (MRdRR T)[30] is a modification of the RRT algorithm for pathfinding in an implicitly given graphs embedded in a high-dimensional Euclidean space.

Just like RRT, the MRdRR T grows a tree $\mathcal{T}$ rooted in the vertex $s$ representing start positions of the robots in a composite configuration space $\mathbb{R}^d$ by iteratively adding new points to the tree while also trying to connect to the goal configuration $t$ without violating any constraints, e.g. collision with the environment. The growth is achieved by randomly sampling a point $u$ in the composite configuration space and then extending the tree towards this sample. Note that vertices newly added to the tree are taken from $G$: given a sample $u$ and the node $v \in V$ nearest to it, the best neighbor $v' \in V$ has to be found. To generate neighbor nodes of already visited nodes MRdRR T uses a technique called oracle. Without loss of generality consider that $G$ is embedded in $[0, 1]^d$. For two points $v, v' \in [0, 1]^d$ the $\rho(v, v')$ denotes a ray that begins in $v$ and goes through $v'$. $\angle_v(v', v'')$ given three points $v, v', v'' \in [0, 1]^d$ denotes the (smaller) angle between $\rho(v, v')$ and $\rho(v, v'')$. The way the oracle is used is given sample point $u$ it returns the neighbor $v'$ of $v$ such that the angle between rays $\rho(u, v')$ and $\rho(v, v')$ is minimized. This can be defined as

$$O_D(v, u) := \arg\min_{v' \in V} \{ \angle_v(u, v') | (v, v') \in E \}.$$  

It is possible that the tree will if given sufficient time, eventually reach
during the expansion phase. However it is unlikely for larger problems. MRdRRT, therefore, attempts to connect the newly added node with \( t \) employing so-called local connector which is successful for restricted problems only, but fast, so it can be run often.

4.2 Proposed improvements

Although the original MRdRRT can solve path-finding problems for several robots, the realization of its particular steps is inefficient, which disqualifies it to deal with complex scenarios containing tens of robots. The authors of MRdRRT present experimental results with up to 10 robots and mention that their algorithm is unable to solve problems with a significantly larger number of robots. We, therefore, introduce several improvements to the original version.

The original expansion phase generates random samples from a bounding box of \( G \) which is inefficient in maps with tight spaces as it would not allow robots to stand still as their next action and would not find a solution in situations where standing still was required for one of the robots. Moreover, the majority of generated points is far from a solution leading to a relatively huge growth of the tree over the configuration space and thus to the high computational complexity of the algorithm. Instead of generating a point from \( \mathbb{R}^d \), we find shortest paths for every robot separately in the preprocessing phase and after that we compose a sample only from points \( q \) for which \( \text{dist}(s_i, q) + \text{dist}(q, t_i) \leq \text{dist}(s_i, t_i) + \Delta \), where \( \text{dist} \) is a distance of two points, \( s_i \) and \( t_i \) are start and goal positions of \( i \)-th robot, and \( \Delta > 0 \) is a defined constant threshold.

The original oracle generates a sample and checks it for collisions, which is inefficient as many samples are discarded. Our version iterates over positions of all robots \( v_i \) and tries to generate a new step \( v_i' \) for each of them towards the sample point \( u_i \) while avoiding collisions and also minimizing the angle \( \angle(u_i, v_i, v_i') \) by keeping a list of collision configurations that need to be avoided.

Another proposed improvement is the use of the CARP algorithm \[22\] as a local connector as well as a random shuffling of the order in which CARP attempts to plan trajectories of individual agents to their desired locations. This algorithm creates a free time window graph on which agents find the shortest paths one by one while updating the free time window graph with their paths so that collisions are avoided.

The last set of modifications to the algorithm is the addition of steps inspired by RRT* algorithm \[32\] which include the new rewiring step and modification of expansion step, see Alg. 3. At the start of the algorithm, the tree is initialized with the node that contains the initial configuration of agents (line 1). The main loop of the algorithm then starts with the newly modified expansion phase (line 3). After a new node is added the rewiring
Algorithm 3: Improved MRdRRT algorithm

1. $T$.init $(s)$
2. while true do
3.     EXPAND $(T)$
4.     REWIRE $(T, v')$
5.     $P \leftarrow CONNECT\_TO\_TARGET (T, t)$
6.     if not_empty$(P)$ then
7.         return RETRIEVE\_PATH $(T, P)$
8.     end
9. end

The step is called (line 4) that attempts to revise the structure of the tree to improve path length to the root. The algorithm finally tries to connect the newly added node with the goal configuration. If it succeeds the algorithm returns the found path.

The change to the expansion phase, Alg. 4, consists of connecting the new node $v'$ to a node already in the tree $T$ that minimizes the distance traveled from the initial configuration $s$. The additional step of the expansion phase in the original RRT* consists of checking nodes in the radius around the new node $v'$ for the best predecessor and then connecting $v'$ to it. However, in the multi-agent discrete scenario (Alg. 4) the computational requirements to perform a similar task are much higher because it would require to run a local connector method on each node in the radius and then perform the distance to root check. The expansion phase was thus modified so that it employs nearest neighbor search instead of radius (line 2). The key difference is that in the first step of expansion the random sample $u$ (line 1) is generated, but after that the new node $v'$ is not created from the nearest neighbor of $u$. Instead, $N$ nearest neighbors of $u$ are iterated over (lines 6-11) and a new node $v'$ is generated from each of them using the oracle $O_D$, but not added into the tree. Each $v'$ is checked for the distance traveled through the tree $T$ towards the root $s$ and only a node that minimizes this distance is connected to its corresponding predecessor.

The rewiring step of RRT* locally revises a structure of $T$ by checking whether nodes within the radius $r$ around a newly added node $v'$ have the distance traveled towards the root node shorter when they had $v'$ as their predecessor. This step was modified for the use in a multi-agent discrete case (Alg. 5) by omitting the radius and iterating over $N$ nearest neighbors of $v'$ instead. Because these neighboring configurations $c$ might not be direct neighbors of $v'$ in the composite graph $G$, the local connector is used to obtain a path between these two nodes (line 3). If the local connector fails to find the path, the neighbor is immediately skipped (lines 4-5). In the case the local connectors successes in finding a path $p$ between $v'$ and $c$ it
Algorithm 4: Improved MRdRRT EXPAND($T, r$)

1. $u \leftarrow \text{RANDOM\_SAMPLE}()$
2. $NNs \leftarrow \text{getNearestNeighbours}(u)$
3. $v'_{\text{pred}} = -1$
4. $d_{\text{best}} = \infty$
5. $v'_{\text{best}} = \emptyset$
6. for $c \in NNs$
   7.     $v' \leftarrow O_D(c, u)$ if $l_T(c) + \delta(c, v') < d_{\text{best}}$ then
   8.         $d_{\text{best}} = l_T(c) + \delta(c, v')$
   9.         $v'_{\text{pred}} = c$
 10.     $v'_{\text{best}} = v'$
 11. end
12. end
13. $T$.add_vertex($v'_{\text{best}}$)
14. $T$.add_edge($v'_{\text{pred}}, v'_{\text{best}}$)

is checked whether the length of the path from the root to $v'$ concatenated with the path $p$ and the node $c$ is shorter than the distance traveled through $T$ from the root to $c$ (lines 5-7). If it is shorter, then all nodes of $p$ are added to $T$. The first node of $p$ is connected as the successor of $v'$ and the last node of $p$ is chosen as a new predecessor of $c$. An example of the rewiring step is displayed in Fig. 4.1.

Algorithm 5: REWIRE($T, v'$)

1. $NNs \leftarrow \text{getNearestNeighbours}(v')$
2. for $c \in NNs$
   3.     $p \leftarrow \text{LOCAL\_CONNECTOR}(v', c)$
   4.     if $p \leftarrow \emptyset$ then
   5.         continue
   6. end
7.     $n \leftarrow \text{LastNode}(p)$
8.     if $l_T(v') + l(p) + \delta(n, c) < l_T(c)$ then
9.         $T$.add($p$)
10.        $c$.$\text{predecessor} = n$
11. end
12. end
4.3 Experiments

Performance of the proposed method has been evaluated, and comparison with the CARP algorithm [22] has been done. The experiments were performed on two sets of artificially created maps and assignments with the aim to compare the quality of obtained results and runtime as well as the reliability of both algorithms.

This first set of maps was created to demonstrate how both algorithms perform depending on the density of the given graph and the number of cycles in it. The set was created by generating a random spanning tree of a $20 \times 20$ grid map followed by the creation of additional maps by iteratively adding a fixed number of original edges into the spanning tree. The experiments were thus carried out on the set consisting of 11 maps ranging from a spanning tree to the full grid. Furthermore, 100 different assignments for a fleet of 100 robots were created by randomly sampling start and goal nodes for each agent and each such assignment was tested on all of these maps. The CARP algorithm was tested by giving it the limit of 1, 10, 100, 1000 attempts to find a solution where the order in which agents were planned was randomly
Figure 4.2: Comparison of the proposed approach (dRRT) with CARP.
Figure 4.3: Map generation procedure.

Figure 4.4: Example of a generated map.

shuffled for each attempt.

The results of this experiment can be seen in Fig. 4.2. The first thing to notice in Fig. 4.2a is that the proposed approach shows a much higher success rate even on the spanning tree, where it had its single failure. Contrary, CARP had 100% failure rate on the spanning tree when given only 1 attempt and 39% when given 1000 attempts. It can be seen on Fig. 4.2c and Fig. 4.2d that once the number of edges in the graph reaches 1090, the algorithms behave very similarly in terms of runtime and the needed number of iterations to find the plan. The proposed method provides a slightly higher median of a number of steps of the resulting plan as can be seen in Fig. 4.2b. This can be attributed to the fact that the median is calculated from a higher number of successful plans compared to CARP algorithm.

The second set of maps was created specifically together with assignments so that the problems would be impossible to solve for the CARP algorithm.
The maps and assignments were randomly generated by the following process:

1. Create a basic problem that is impossible to solve for the CARP algorithm depicted in Fig. 4.3a. Arrows indicate the starting and goal positions of robots A and B on the graph. CARP fails because the agents need to swap their positions while having the same distance to the only node they can use to avoid each other. Because CARP plans agents sequentially one by one while ignoring the subsequent agents, no ordering of these agents can solve this issue.

2. Pick random node that has only one edge associated with it.

3. Either add 2 nodes A and B to either side of this node if possible along with corresponding assignment of 2 agents – The first agent going from A to B and the second one from B to A. The example of this step can be seen in Fig. 4.3b. The alternative method to expand the map is to connect the same structure to it as in the Step 1 together with the same type of assignment, the example of which can be seen in Fig. 4.3c.

4. Repeat Steps 2 and 3 until the map of a required size is generated.

The example of a fully generated map following the previous steps can be seen in Fig. 4.4.

The second set of experiments was carried out on the second set of maps with the aim to illustrate the behavior of the proposed algorithm on assignments that CARP algorithm can not solve. The total of 400 different combinations of a map and assignment were generated: 100 each for 10, 20, 30 and 40 agents. The results of this experiment can be seen in Fig. 4.5. The setup numbers 1 to 4 correspond to the number of agents 10 to 40 respectively.

For up to 30 agents the success rate is 100% while it is decreased to 95% for 40 robots. Regarding the computational time results, the algorithm takes approximately 1 second to calculate the paths for each agent in assignments that are impossible to solve for CARP algorithm for up to 30 agents even with relatively complicated assignments.
Figure 4.5: Results of the proposed approach on assignments which CARP is unable to solve.
Chapter 5

Parallel Push and Rotate

5.1 Original Push and Rotate algorithm

The Push and Swap algorithm [33] 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 [19]. This algorithm is proved to be complete for setups with two or 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 is that it does not take into account Kornhauser’s [34] 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 [19]. 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.

5.1.1 Push

The push operation attempts to move the agent $a_1$ from the 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 Fig. 5.1, 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.

5.1.2 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 Fig. 5.2). 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.
5.1.3 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 Fig. 5.3) 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 Fig. 5.3 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.

5.2 Proposed algorithm

5.2.1 Warehouse-related requirements

Several challenges appear when designing an algorithm for real continuous environment usage instead for a discrete world of graphs. On the other hand, one can employ several simplifications in comparison to general graph algorithms assuming specific properties of a graph describing a typical warehouse graph. 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.
Non-constant time of movement between nodes

The non-constant time of movement from a node to another node is one of the main differences from standard graph algorithms. 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.

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 Fig. 5.4. 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.

**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 5.2.2 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.

**Simplifications**

The real warehouse environment increases the complexity of the algorithm significantly, however the graph of the warehouse has certain properties 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 the 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 [19] 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; new operation replan is introduced instead 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 the number of robots explained in Section 5.3.1.

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 the 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. It can happen that the buffer would get too small during the movement of the robots, the movement would have to be paused in this situation.

5.2.2 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 a 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 consists of three parts: the single robot path planning phase, the initial trajectory generation, and the robot maneuvering phase (Algorithm 6). In the single robot path planning phase, shortest path is calculated for each robot from its start node to its goal node making use A* algorithm. In the second phase, the trajectories following the paths are generated using the robot model. The robot maneuvering phase simulates robot’s movement following calculated trajectories and uses operations stop, push and replan to modify the trajectories in case of conflict. Each robot has its planned trajectory $J_r$ 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 lengths of the shortest paths to their destinations. The temporary priority is incremented by the push operation and reset to the value of the main priority. The stop operation tries to resolve the conflict by stopping one of the robots, the push operation pushes robot with lower priority.

**Single robot path planning phase**

The lines 1 to 8 of the Algorithm 6 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* [35]. 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 $H(n_s)$ is used to sort nodes in the open list (a list of not yet explored nodes). In the proposed algorithm, three costs for long spline 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
Algorithm 6: 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. $\text{trajectories} \leftarrow \text{empty vector}$
2. $\text{paths} \leftarrow \text{empty vector}$
3. $\text{forall } r \in R$ do
   4. if Robot $r$ has rack then
      5. $P \leftarrow \text{shortest\_path}(r, G_r)$
   6. else
      7. $P \leftarrow \text{shortest\_path}(r, G)$
   8. end
9. $J_r \leftarrow \text{generate\_trajectories}(P, G, L)$
10. $\text{trajectories} \leftarrow \text{trajectories} + J_r$
11. $\text{paths} \leftarrow \text{paths} + P$
12. end
13. $\text{J\_sol} \leftarrow \text{solver}(\text{trajectories}, \text{paths}, G, R)$

travel through two default edges, but longer than traveling through 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 $0 < C_s < \frac{C_l}{2}$, 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.

**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 come into a conflict in moments that were not captured by the samples; thus the algorithm would not detect it. It is reasonable to have at least ten samples for each edge. To make sure, the sampling frequency $F_s$ is sufficient, it should meet the condition $F_s > \frac{l_{\text{min}}}{v_{\text{max}}} \times 10$, where $l_{\text{min}}$ is the minimum length of an
edge and \( v_{\text{max}} \) is the maximal robot velocity. The sample time \( T_s = \frac{1}{f_s} \) is usually used in the algorithm.

### Algorithm 7: Solver algorithm

**Data:** generated trajectories \( J \)

**Result:** Trajectories \( J \), modified to be collision-free.

```plaintext
solved ← false
2 \textbf{while} !solved \textbf{do}
3 \quad \textbf{if} conflict_detect() \textbf{then}
4 \quad \quad resolve_crash(crash)
5 \quad \textbf{if} check_solved() \textbf{then}
6 \quad \quad solved ← true
7 \quad \textbf{else}
8 \quad \quad resolve_priority_reset()
9 \quad \quad resolve_replan()
10 \quad \quad state_shift(1)
11 \textbf{end}
12 \textbf{end}
```

**Robot maneuvering phase**

This phase is described by Algorithm 7. 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 9. 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 invoke certain operations, their purpose is described further in the operations description.

The *Conflict detect* (Algorithm 8) searches through 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 9 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 10) 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 Fig. 5.5b, 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 (Fig. 5.5a), the stop operation helps to 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 to a dead-lock situation. In Fig. 5.6b 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 when it would help.

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

Figure 5.5: Examples showing situations when the stop operation is possible and when it is not.
Algorithm 8: 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 9: 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.

Stop

The idea of the stop operation (Algorithm 11) is very simple – to stop one robot so 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 (Fig. 5.6a).

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
Algorithm 10: 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. \( r_{\text{high\_priority}} \leftarrow r_1 \)
3. \( r_{\text{low\_priority}} \leftarrow r_2 \)
4. else
5. \( r_{\text{high\_priority}} \leftarrow r_2 \)
6. \( r_{\text{low\_priority}} \leftarrow r_1 \)
7. end

8. if \( r_1 \) is finished or \( r_2 \) is finished then
9. return \( \text{Replan\_operation}(r_{\text{high\_priority}}, r_{\text{low\_priority}}) \)
10. if can\_be\_stopped then
11. return \( \text{Stop\_operation}(r_{\text{high\_priority}}, r_{\text{low\_priority}}) \)
12. return \( \text{Push\_operation}(r_{\text{high\_priority}}, r_{\text{low\_priority}}) \)

(a) The stop operation will be always successful.  
(b) Deadlock example

Figure 5.6: Examples of stop operation situations.

is the next step, it is checked during the state shift (Algorithm 7, line 10) if the edge that the robot \( r_{\text{stop}} \) 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 \( J_{r_{\text{stop}}} \) (line 4).

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
Algorithm 11: 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 $J_{\_wait}$

---

Figure 5.7: Comparison of the original and new push operation on simple case.

movement of the agents even when there is no conflict (section 5.1.1). 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 Fig. 5.7a show the operation had to be carried consecutively twice in order to let the agent $a_1$ through. 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 (Fig. 5.8).

The new push (Algorithm 12) 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 Fig. 5.9. The operation selects the robot that can be pushed with respect to the directed edges.
Figure 5.8: 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 operation needs to find the closest node that is not in the path of the robot $r_{go}$. This search needs a list 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 through 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 [36] 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 its planned trajectory to be later replaced with the push trajectory (line 10).

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 7 (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. The trajectories are added together to form a new trajectory for the robot $r_{push}$.

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 $r_{go}$
Figure 5.9: 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.

is at the node. However, the state was shifted back before thus the state must be shifted by $(\text{steps}_\text{shift} - t_{\text{back\_to\_node}})$, where $t_{\text{back\_to\_node}}$ is the number of steps from robot $r_{\text{go}}$ 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 $J_{\text{wait}}$ for the robot $r_{\text{go}}$ with minimal wait of $(\text{steps}_\text{shift}t_{\text{back\_to\_node}})$ steps to ensure that the robot $r_{\text{push}}$ 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 $J_{\text{wait}}$ sequence into the $J_{r_{\text{go}}}$ trajectory right after the current step (line 17) and the operation is complete.

Replan

The replan operation (Algorithm 13) is used when one of the robots is finished (Fig. 5.10). 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.

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 $\text{nodes\_to\_avoid}$ (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
Algorithm 12: 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. $[\text{steps}_\text{shift}, t_{now}] \leftarrow$ 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 $(\text{steps}_\text{shift} - t_{back\_to\_node})$.
16. $J_{wait} \leftarrow$ Generate wait sequence with at least
    $(t_{back\_to\_node} - \text{steps}_\text{shift})$.
17. Update $J_{r_{go}}$ with $J_{wait}$.

To replace it further with the new replanned one (line 6). Similarly as in Subsection 13, 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 performing 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 4 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 13, one can simply assume
Figure 5.10: Both operations *stop* and *push* would fail, because the robot $r_1$ is finished and cannot be moved. This situation requires the *replan* operation.

that $r_{\text{push}} = r_1 = r_2$. The algorithm still benefits from the avoidance of the finished robots saving future *replan* operation calls.

**Algorithm 13: replan operation**

**Data:** Robots that crashed $r_1$ and $r_2$, robot model $L$, graph $G$

**Result:** Replan the trajectory $J_r$ of the non-finished robot $r_x$.

1. $r_x \leftarrow$ The robot ($r_1$ or $r_2$) that is not finished.
2. $\text{nodes}_\text{to}_\text{avoid} \leftarrow$ Nodes occupied by all finished robots.
3. $t_{\text{now}} \leftarrow$ 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. $J_{r_x} \leftarrow J_{r_x}(j_0, \ldots, j(t_{\text{now}}))$.
7. $P \leftarrow$ Calculate the shortest path from $n_c$ to $n_g$ avoiding nodes in $\text{nodes}_\text{to}_\text{avoid}$.
8. $J_{\text{replan}} \leftarrow$ Generate trajectory using path $P$ and the robot model $L$.
9. $J_{r_x} \leftarrow J_{r_x} \cup J_{\text{replan}}$
10. temporary priority of $r_x \leftarrow$ main priority of $r_x$.

5.3 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.
5.3.1 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 Fig. 5.11a, 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 (Fig. 5.11). 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.
5.3.2 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 document are shown in Section 5.4.

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.

5.4 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 which performed equally and sometimes even faster than the PC setup.

During the experiments, the warehouse map displayed in Fig. 5.12 was used. A task for 50 robots with various distances from the start node to the goal node was created. 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.

5.5 Execution delay

One of the main advantages of the algorithm is discussed in Subsection 5.3.2. 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 Fig. 5.13 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.

5.6 Two approaches comparison

The ability of the algorithm to add robots to the plan during the calculation was discussed in Subsection 5.3.2. 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 runtime and solution time impact, are compared to the standard approach which calculates all robots at the same time.

5.6.1 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. Fig. 5.14, 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.

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 graphs 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
Figure 5.13: Real-time during the state time of the algorithm during the calculation of the sequential approach.

Figure 5.14: The number of conflicts for the sequential and standard approaches.

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.

5.6.2 Calculation time comparison

Perhaps the most significant impact of the standard approach is on the solution time. Running the algorithm multiple times through 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 Fig. 5.15 (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.

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.

Fig. 5.16 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 so-
5.6.3 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 Fig. 5.17a 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 through 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 shortened 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.

When the number of robots exceeds 31, the robot with the longest trajectory is affected differently by the two approaches. In conclusion, the effect of the sequential approach on the solution time is not significant.
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. Similarly to the previous experiments, the difference starts to be significant in the 30 and more robots range and it diverges fast.

The ratio of the sum of trajectory steps for all robots provided by the planner in comparison to the lowest threshold is shown in Fig. 5.17b. 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 Fig. 5.14 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 Fig. 5.17a suggest that the longest trajectory for 40 robots is about 5 minutes. This represents the worst case scenario.
Bibliography


