FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Spatio-Temporal Prioritized Planning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Ing. Benjamin Binder, BSc
Matrikelnummer 1226121

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Johann Blieberger
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Markus Bader

Wien, 5. Dezember 2017

Benjamin Binder	Johann Blieberger

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Spatio-Temporal Prioritized Planning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Ing. Benjamin Binder, BSc
Registration Number 1226121

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Johann Blieberger
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Markus Bader

Vienna, 5th December, 2017

Benjamin Binder | Johann Blieberger

Erklärung zur Verfassung der Arbeit

Ing. Benjamin Binder, BSc
Eulenbach 52,
3902 Vitis

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. Dezember 2017

Benjamin Binder

Kurzfassung

Mobile Roboter-Flotten werden aufgrund ihrer flexiblen Einsatzmöglichkeiten in Logistikanwendungen immer beliebter. Mit dem Einsatz von Roboter-Flotten entsteht jedoch die Notwendigkeit, die einzelnen Roboter in dieser Flotte zu koordinieren. Ein weit verbreiteter Ansatz, um diese Problemstellung zu lösen, nennt sich „Prioritized Planning“. Hierbei generiert ein Routenplaner alle Routen der Roboter sequenziell und versucht Überschneidungen mit bereits geplanten Roboter-Routen zu vermeiden.
Diese Diplomarbeit beschäftigt sich mit einem neuen Ansatz für solch einen „Prioritized Planner“. Der vorgestellte Planer erzeugt Routen für mehrere Roboter, welche zueinander synchron abgearbeitet werden, um Kollisionen und Deadlocks zu vermeiden.
Zur Koordinierung von Robotern werden meist Pixel-Karten eingesetzt. Mit steigender Pixelanzahl in den Karten wird die Planung der einzelnen Routen immer zeitintensiver. Aus diesem Grund werden für den erzeugten Planer die Pixel-Karten auf einen Suchgraphen reduziert und somit die Dauer des Planungsprozesses verringert.
Es wird ein Planer für einzelne Roboter entwickelt, welcher fähig ist, bereits geplante Routen von anderen Robotern in den Planungsprozess mit einzubinden. Diese Routen werden in den Suchgraph des Planers integriert, um so das Ergebnis des nächsten Roboters zu beeinflussen.
Weiters kann mit dem Planer die Verweildauer der einzelnen Roboter in einem Streckenabschnitt angepasst werden. Dadurch können gegebenfalls Wartezeiten integriert werden. Sollten bei der Routenplanung dennoch Kollisionen zwischen Robotern auftreten, können zusätzlich Pfad-Segmente in den Suchgraphen integriert und diese Kollisionen somit vermieden werden.
Da die erzeugten Routen des Planers zeitabhängig sind, werden diese im Anschluss an den Planungsprozess synchronisiert. Dabei werden für jeden Routenabschnitt Vorbedingungen bestimmt, welche erfüllt sein müssen, bevor ein Roboter diesen Routenabschnitt befahren darf.
Um diverse Szenarien zu lösen, die mit den derzeitig verwendeten „Prioritized Planning“-Ansätzen nicht lösbar sind, kann der Planer bei fehlgeschlagenen Planungsversuchen den Planungsprozess mit unterschiedlicher Reihenfolge und Geschwindigkeiten der Roboter wiederholen.
Für die Evaluierung des neuen Planers, wird dieser mit den aktuell verwendeten Planer-Ansätzen in einer simulierten Umgebung getestet und die Ergebnisse gegenübergestellt. Weiters werden, durch den vorgestellten Planer lösbare Szenarien gezeigt, welche durch aktuell verwendete „Prioritized Planner“ nicht lösbar sind.

Abstract

Using robot fleets have gained popularity in recent years in industrial logistics applications because of their flexible field of application. Therefore, the issue arises how to coordinate these robots. A commonly used approach to this problem is Prioritized Planning, where a coordinator plans all robot's routes sequentially avoiding already planned ones.
This thesis presents a new approach for prioritized multi robot path planning. The proposed planner works centralized and generates synchronized and deadlock-free routes for robots.
Applying search algorithms on pixel-maps is expensive, therefore, pixel-maps are reduced to search graphs using voronoi distillation.
A single robot path planner for use inside a prioritized multi robot path planner is designed. This single robot path planner constrains the creation of new routes by adding already planned ones to the used search graph. To find solutions in this graph the single robot path planner is able to take temporal and spacial constraints into account. Furthermore, potential collisions and deadlocks between robots are avoided by extending the search graph with additional segments if a collision is detected.
The global multi robot path planner includes a priority and speed rescheduler as well, to solve specific scenarios which are hard to solve using prioritized path planning. Because the routes are generated time-dependently, they are post-processed to add synchronization markers in form of preconditions for every robot. These preconditions have to be met before a robot is allowed to enter a segment.
Selected scenarios with multiple robots are used to compare the proposed planner with state of the art approaches in a simulated environment. Also scenarios are shown to be solvable with the proposed planner, which are not solvable with currently used approaches.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Motivation and Problem Statement 1
1.2 Aim of the Work 4
1.3 Methodological approach 5
1.4 Structure of the Work 6

2 State of the Art **7**
2.1 Single Robot Planning 8
2.2 Multi Robot Planning 9
2.3 Multi Robot Planning Extensions 11
2.4 Roadmapping (Voronoi Paths) 12
2.5 Summary 12

3 Approach **13**
3.1 Terminology 14
3.2 Requirements 15
3.3 Structure 17
3.4 Single Robot Route Planner (SRRP) 19
3.5 Multi Robot Route Planner Extensions 44

4 Implementation **49**
4.1 Graph Generation 51
4.2 Route Planning 56
4.3 Robot Controller 60
4.4 Test Environment 61

5 Results **63**
5.1 Experiments 64

5.2 Comparison to other planners 72

6 Conclusion **79**
6.1 Further Work 80
6.2 Implementation and Source code 80

List of Figures **81**

List of Algorithms **83**

Glossary **85**

Acronyms **87**

Bibliography **89**

CHAPTER 1

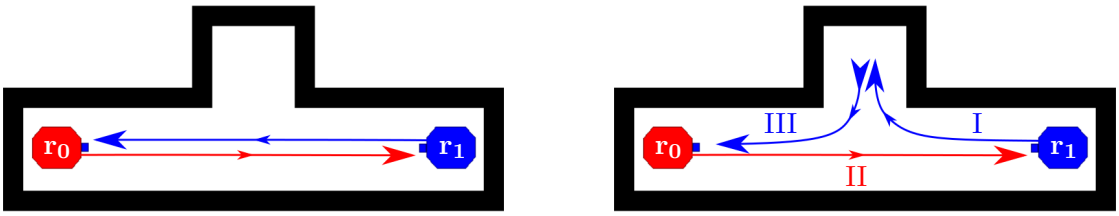
Introduction

1.1 Motivation and Problem Statement

Driverless vehicles are commonly used in warehouses and on assembly lines to transport goods [WDM07]. These vehicles are usually restricted to specific areas by using predefined tracks. The disadvantage of this approach is, that stalls can occur if obstacles are blocking these tracks. In order to avoid such stalls, one can enable vehicles to leave their tracks autonomously for a short period of time. This can be done in two ways:

- Locally, by considering only conflicting paths in a specific area.
- Globally, by replanning all paths for all vehicles.

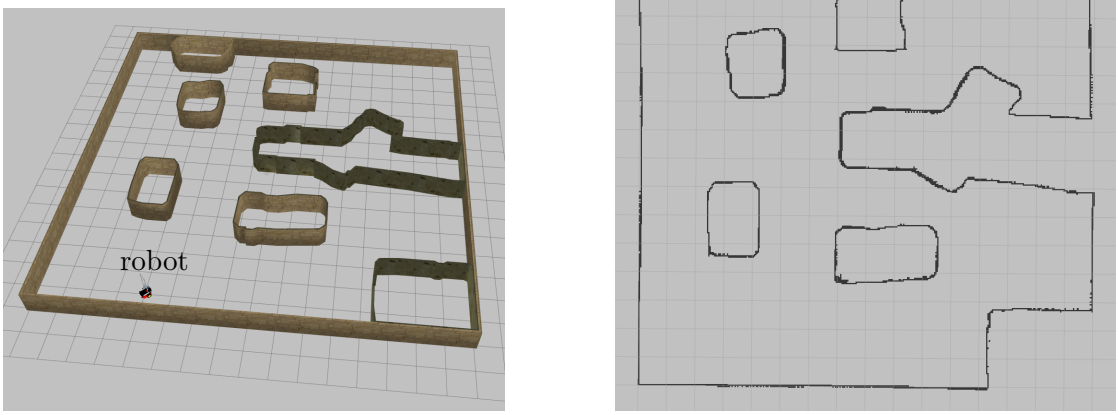
An example for a local solution to such a scenario is shown in Figure 1.1, where robot r_1 avoids robot r_0 at a crossing.



(a) The start positions of two robots, which want to switch places.

(b) The solution to the given problem, where robot r_1 has to wait at the crossing until r_0 has passed.

Figure 1.1: Example test scenario, where two robots switch places.



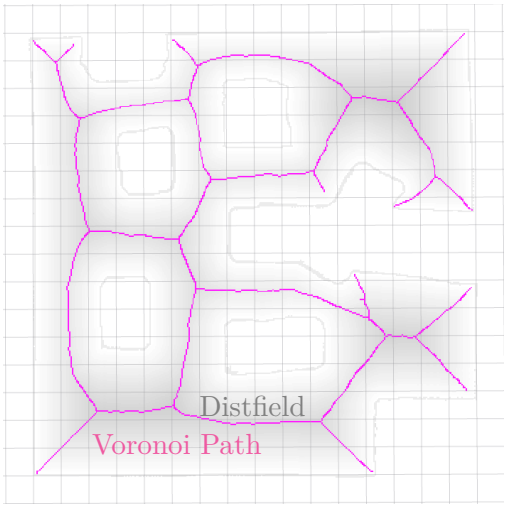
(a) The simulated environment in GazeboSim with the robot

(b) The grid map of the simulated environment

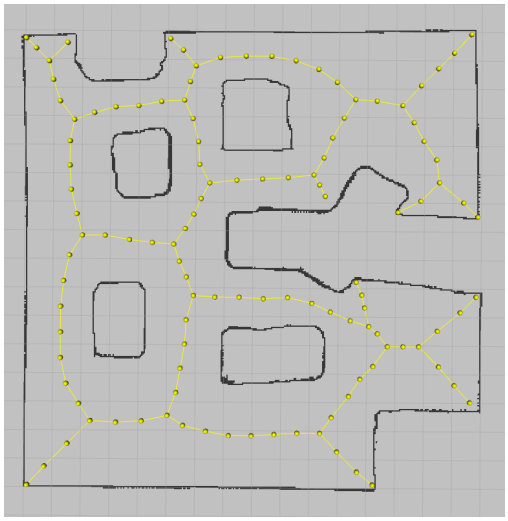
Figure 1.2: Voronoi graph generation

Paper [BRS+15] tries to fuse both approaches. It describes a well designed framework using a multi robot path planner combined with a behavior controller on every vehicle. The multi robot planner passes a set of path segments to the behavior controller of a vehicle, which decides if the robot has to move on the given path or is allowed to deviate from it using a local path planner.
This thesis presents a new approach for a Multi Robot Route Planner (MRRP) based on the ideas of [BRS+15].
The following keywords are essential to understand the ideas in this thesis:

- **Graph** refers to the search graph used by the route planner. This graph has a bijective mapping to the used environment, where each vertex is mapped to a segment of the environment.
- **Segment** is a small area of an environment with a specific shape. Every segment of an environment can be mapped to exactly one vertex of the corresponding graph.
- **Route** describes a list of segments, including space and time.
- **Path** describes a list of points located on a map to lead a robot to its goal.
- **Trajectory** is used to describe a time-dependent point-sequence leading the robot from one path point to another.



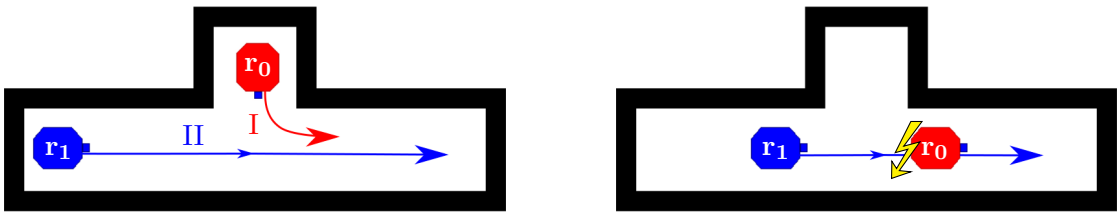
(a) Voronoi path related to the map (magenta); The Distance field (grey), which describes the minimal distance to an obstacle;



(b) The segmentation of the voronoi path from Figure 1.3a.

Figure 1.3: Voronoi graph generation

Prioritized Planning is used inside the Multi Robot Route Planner (MRRP), which sorts all robots by priority and let them plan their routes depending on the routes of higher prioritized robots [LaV06].
These planning iterations are typically done on maps represented as grid-maps, shown in Figure 1.2, where the planner can find a path and all possible deviations from it. Since paper [BRS+15] describes a behavior controller, which is capable of deviating from the given path, it is not necessary to keep detailed grid-maps. Instead, a data abstraction of the grid-map to a graph representation is sufficient to find routes, given to the behavior controller. An example for such an abstraction is a voronoi path describing the path through a map, which has at every point the maximum distance to all obstacles around (Figure 1.3). The given path can be revised further by segmenting it, which leads to a graph like the one shown in Figure 1.3b. This data abstraction helps to reduce time complexity for single robot planners, but problem cases exist, where relative execution time between robots is relevant to fulfill a given goal. Therefore, time is modeled as well in the graph representation. On this spacial-temporal-path, a planner similar to the idea of [WG12] is used, which allows "waiting-steps" in the robot's route, to wait for other robots causing potential deadlocks. Furthermore, additional strategies are used to find solutions to other problems like avoiding robots at a crossing.



(a) The start positions and the robots desired goals.

(b) A solution can't be found assuming robot r_0 has higher priority than r_1, because r_0 plans its path first and blocks r_1's trajectory.

Figure 1.4: Problem set where robot priorities matter

1.2 Aim of the Work

The scientific contribution of this work is a new approach for a MRRP. The goal of this MRRP is to find deadlock-free routes and optimize them for the use in dynamic environments. Actual challenges in multi robot planning are evaluated, to derive the structure of the MRRP. An example for such a scenario can be seen in Figure 1.1, where robots r_0 and r_1 want to switch places. Obviously, r_1 has to wait at the crossing as shown in Figure 1.1b, to let r_0 pass. Although this problem is easy to solve for a human, multi robot planners are struggling to solve such scenarios. To solve these problems, spacial and temporal locality have to be considered. In our approach this is done by introducing "spacial-temporal-graphs", describing not only spacial restrictions but also temporal ones for planning[CPA+14]. A solution to the shown issue in Figure 1.1 can be found with such a graph. Robot r_0 locks its route (II) on the graph. Robot r_1 is forced to avoid r_0's route and moves (I) to the free space at the crossing to wait there until r_0 has passed. Finally, r_1 can move to its goal (III). Furthermore, this "spacial-temporal-graph"-approach will decrease the computational time of the planner, because several areas are merged together to one edge of the graph as shown in Figure 1.3b.
In Figure 1.4 a situation can be seen, where the higher prioritized robot (r_0) blocks the route of r_1 after planning its route. This can be easily solved by switching the priorities of the robots, to let r_1 plan its route first, to force r_0 to wait. Thus, a strategy to solve such scenarios is implemented in the MRRP, which allows the planner to solve more problem sets.
After finding a valid routing table it has to be ensured that the robots maintain the given restrictions of the routing table. This can be done centralized, decentralized or decoupled. For the centralized approach a supervisor can be used to observe the robots positions and stop single robots if they offend any restriction. For the decentralized approach every robot has to communicate with each other, to maintain the selected order. Finally for the decoupled approach every robot has to follow a given velocity profile to reach its planed deadline.

1.3 Methodological approach

As mentioned in Section 1.2 the scientific contribution of this work is a new approach for prioritized path planning. This approach is implemented as ROS[1] node. To evaluate the result it is compared to other planning approaches in terms of computation time and completeness. The work is split into four parts:

1. The first task was to set up a test environment for the planner, which works on ROS with stage as simulator and RVIZ as visualization. Furthermore, a vehicle controller was programmed to follow the given routes. The test environment is described in Chapter 4.
2. Afterwards, a minimal multi robot planner for vehicles with the same radius and non intersecting routes at every point in time is created. Simultaneously, the planner is designed in a way, that it is easily expandable to the final global planner. It is intended to use a central Single Robot Route Planner (SRRP) planning all the single routes combined with a so called "path query"-class, which organizes the route allocation between the planning tasks.
3. After creating a minimal solution and the framework, the planner will be expanded to a prioritized planner, taking time into account. Furthermore a strategy to synchronize routes between robots was created.
4. The prioritized planner can now be refined to custom defined problem sets and compared to other planners.

[1] (www.ros.org)

1.4 Structure of the Work

The work is structured in six parts. In this section the motivation and the ideas covered in this thesis are explained.
In „State of the Art“ literature is described which is used for the proposed MRRP. Therefore, the framework, which forms the basis of the work is discussed, papers for single and multi robot planning are presented and some papers explaining roadmapping are shown as well.
In Chapter 3 the general ideas of the MRRP are described, by deriving the structure of the planner using papers introduced in Chapter 2. Afterwards, the structure of the used search graph is described. To show current problems in prioritized path planning, some scenarios are shown, which should be solvable by the prioritized planner. These scenarios are used to create collision resolution strategies for the MRRP to solve scenarios efficiently. Since the generated solutions of the MRRP are time dependent, an approach to execute the planned routes on the robots is discussed in the end of Chapter 3.
Chapter 4 describes the implementation of the MRRP. To these ends the test environment is described in detail. The process of generating a usable search graph for the MRRP is described, by generating voronoi paths, which are transformed into a graph. Also, an algorithm is shown, which describes the structure of the MRRP and the Robot Controller including the strategy used to synchronize routes is explained.
To evaluate the created MRRP, tests are presented in Chapter 5. At first, test cases shown in Chapter 3 are executed to verify the proper function of the MRRP. Furthermore, the resulting routes for each robot are shown and checked for their validity in the global routing table. In the second part the MRRP is compared to other planners. Thus, a simple planner with priority scheduling and a planner using waiting steps in its routes are compared to the MRRP.
In the last chapter the thesis is summarized and further work is discussed.

CHAPTER 2

State of the Art

Quite some research has been contributed to path planning in the last years [WG12] [WLW13] [CNKS15]. Most of these approaches are based on path planning for a single robot[WWW16] [CNM99]. The proposed thesis examines path planning with multiple robots on graphs trying to find a system optimal solution.
The presented approach is based on the framework proposed in [BRS+15]. This framework organizes a large number of robots in a centralized way while allowing single robots to deviate from their given trajectory. It is proposed to create a framework which consists of a centralized Automated Guided Vehicle Control System (ACS) and multiple Automated Guided Vehicles (AGVs) controlled with a behavior controller on each vehicle. The ACS contains a job planner to coordinate pending tasks and a Route Planner, which creates a global routing table for the current task schedule. The single routes of this routing table are given to the corresponding AGV. An AGV consists of a Behavior Controller, Self-Localization, and a Local Path Planner. The Behavior Controller receives routes from the ACS and selects how to follow them. Therefore, the Behavior Controller can select one of two control strategies: Following the route precisely or diverge from it locally to avoid obstacles. To diverge from the given route a Self-Localization module and a Navigation module are essential. The control task is done by using a Model Predictive Control working similarly to the Dynamic Window Approach but in a more advanced way. This framework has the major benefit of combining a global and a local planning approach resulting in a sound and flexible system.
To prevent confusion, I want to make the reader aware, that the terminology used in this chapter is the one found in the papers and not the one explained in Chapter 1.

2.1 Single Robot Planning

Single Robot Planning is discussed extensively in [LaV06], which shows different types of map representations and roadmaps for planning. Furthermore, it shows the Dijkstra and A-Star algorithm, which are used for many single robot planning approaches.
Mapping on a maximum distance roadmap is explained in the paper [WWW16], which describes a heuristic for planning trajectories using generalized Voronoi diagrams GVDs to speed up the planning process. In comparison to Euclidean heuristics, this approach avoids local minima and for this reason shows, a significant speedup of the algorithm.
A simple algorithm to avoid obstacles and other robots is the Optimal Reciprocal Collision Avoidance (ORCA) algorithm proposed in [vdBGLM11]. This algorithm calculates so-called velocity obstacles, which are describing velocities leading to a collision in the next t seconds. The robot selects the best velocity out of its velocity space, avoiding the ones described by the velocity obstacles. In this case, the best velocity means the velocity which leads the robot as fast as possible to its target. This algorithm works well for planning locally but has problems to find a global plan in the presence of local minima.
Another set of algorithms is called bug algorithms. A survey of these algorithms is shown in paper [GKM10]. These algorithms including simple strategies like following a wall, to find their goal. The benefit of these algorithms is that they don't need to know any kind of map because they rely only on their sensors. One of these algorithms is described in [KRR98], which has two behaviors for following walls and moving towards obstacles. This helps the algorithm to find the shortest path locally.

2.2 Multi Robot Planning

A solution and a proof to a complete algorithm, which is able to find a solution for multi robot planning can be found in [LB11]. This algorithm defines primitive operations to move the robots towards their goals. The basic operations are push and swap, which enable a robot to push another one in any direction or to swap place with it if they are in an appropriate spot for switching. These two operations are enough to prove the completeness of the algorithm. However, a few other operations have to be added to produce short and clean paths.
Another complete algorithm is proposed in paper [PCM08]. This algorithm is split into four phases and uses a minimal spanning tree of the search graph to compute the solution. In the first phase, every robot has to find a leaf of the spanning tree. If this is possible and one leaf is empty, which means there have to be at least $n + 1$ leafs for n robots, the algorithm is guaranteed to find a solution.
In the second phase, the robots are ordered such that every robot is on a subtree of its goal node. If two robots are on the same subtree, the robots are ordered by the depth of their goal in the tree.
In step three every robot is moved to its goal and in step four the generated paths are optimized to find a concurrent plan.
This idea has the advantage of completeness, but is not optimal and produces, longer path lengths compared to prioritized planners like the one explained in [BBT02].
In [ELP86] and in [LaV06] the idea of prioritized planning is proposed, which splits a multi robot planning problem into multiple single robot planning problems to decrease complexity. Furthermore, [LaV06] proposes the velocity tuning method to solve a planning problem with multiple moving obstacles for a robot by keeping the planned path and tuning the velocities until all collisions are resolved.
Paper [CNKS15] introduces a revised prioritized planning algorithm, which is complete for certain structures of the environment. Furthermore, it presents an asynchronous decentralized approach for the proposed algorithm. The idea behind this algorithm is to never overrun goal points of higher prioritized robots and to avoid start-points of lower prioritized robots. If there is an environment, which guarantees such paths for every vehicle, the planner will always find a valid plan. The decentralized version of the algorithm starts the planning algorithm on every robot. After finishing the planning task, the trajectories are checked for conflicts. If there are conflicts the lower priority robot has to plan its trajectory again until every robot has found a valid trajectory.
Paper [WLW13] suggests a strategy for coordinating a large number of robots using roadmaps generated using a voronoi-distillation. Such a voronoi path is a fully connected path with the maximum distance to every obstacle in the vicinity. It introduces a so-called electric circuit based path planning (ECPP) algorithm, which works on GVDs. Therefore, a path segment has a specific resistance, which describes the number of robots and the thickness of the path. Now, the robot acts like current in an electrical system and tries to find the path with least resistance. Thereby, robots are well distributed over all existing paths. But the ECPP algorithm will not provide collision-free paths but minimizes the

possibility of "traffic jam".
In paper [CPA+14] a whole framework for planning and synchronizing robot trajectories is presented. The framework consists of a motion planner, a motion coordinator, a vehicle executive, a trajectory smoother and a motion controller. The motion planner generates an initial trajectory using a lattice-based planner. The motion coordinator tries to solve all conflicts for overlapping paths by altering temporal and spacial constraints. If the motion coordinator fails, the motion planner is triggered to replan the trajectories. After finding conflict-free trajectories, the trajectory smoother is used to refine the trajectory for the vehicles considering accelerations and turn radii. Finally, the vehicle executive selects a trajectory and updates temporal constraints when a vehicle misses a deadline. The motion controller is used to move the vehicle along a given trajectory.
The paper [JN01] presents a collision resolution strategy to avoid deadlocks with robots locally. It describes an algorithm, which lets every robot plan its trajectory without knowledge of others. In the case a robot moves along its path and detects a robot in its vicinity, it starts to communicate with it. After a successful connection, the robots can exchange their trajectories and check them for potential collisions. If a collisions is detected each robot gets a priority for this scenario. The robots are allowed to move in the order of their priorities through the collision. An algorithm checks if this priority order causes a deadlock, where two robots block their paths mutually, and tries to resolve it if there is one. If this coordination fails, single robots are requested to replan their trajectory until a valid solution is found. The benefit of this approach is that any planning algorithm can be used to plan and replan the robots trajectories. Furthermore, no global communication is needed. Because potential deadlocks are resolved locally, non-optimal trajectories can be found.

2.3 Multi Robot Planning Extensions

A enhanced version of an A*-algorithm is represented in the paper [WG12]. It is called Spatial-Temporal-A*-algorithm and lets the planner choose for every step between moving in any direction or waiting on spot. To apply a Spatial-Temporal-A*-algorithm the data structure used to create planes has to be enhanced by a third dimension. This means each pixel of the map used for planning is extended by a list to save the time a robot spends on it. This approach decreases the computational complexity by keeping a high success rate for the algorithm. This rate increases even more if an adaptive priority assignment strategy is used.
Papers [BBT01] and [BBT02] are discussing techniques to select priority orders for robots to improve the number of solvable scenarios and shorten the overall path length of a plan. The paper [BBT01] tries to achieve this by initializing the priorities randomly and using a hill-climbing search to exchange priorities. This significantly decreases the overall path length and the number of solvable problems.
The second paper ([BBT02]) tries to find constraints for priorities. For example, if the target location of a robot R_a lies too close to an optimal trajectory of another robot R_b, robot R_b has to plan its trajectory first. This reduces the number of exchanged priorities and therefore the number of planning iterations.
In paper [Rya07], a strategy is presented to analyze a graph for special subgraphs to simplify it. These subgraphs are merged to one vertex of the graph, which presents three methods to access the new graph:

- *ENTER*: Tests if a robot can enter the subgraph.
- *EXIT*: Tests if a robot can exit a subgraph.
- *TERMINATE*: Tests if a robot can move to its goal vertex.

This approach performs well in structures containing long hallways and roads because vertices connected in long straight lines occur with high probability.
Generation of the underlying graph for planing will be presented next.

2.4 Roadmapping (Voronoi Paths)

Since the generation of voronoi graphs is needed in this thesis some papers to generate such graphs are reviewed as well. An idea how to generate voronoi paths is presented in [WLW13], which uses a distance transform algorithm from [FYS03] and an improved version of [ZS84], which is used to skeletonize a distance-transformed image. The disadvantage of the approach in [ZS84] is, that it needs multiple iterations for one image which slows down the skeletonizing process. [NAU06] comes up with a fast algorithm to calculate the skeleton of an image by following the ridge of a distance transform map. Therefore, the algorithm calculates maximum-, minimum- and saddle-points of a distance-transformed image. Because these points are all local minima and maxima on the ridge, the algorithm only has to expand from every found point to the highest neighbor until an already expanded point is found. This decreases the number of iterations compared to [ZS84] and therefore the computation time.

2.5 Summary

In this chapter, multiple approaches to the topic of path planning are explained. Basically, two different groups of approaches can be identified:

1. Centralized approaches
2. Decentralized approaches

Centralized ones ([LB11], [PCM08],...) are in general complete and (at least theoretically) able to find optimal solutions, but their computational complexity grows exponentially with the number of robots. Decentralized approaches ([LaV06], [CNKS15], [WG12],...) are in general incomplete, but their computational complexity scales better compared to centralized ones.
Furthermore, various approaches to improve the results of decentralized planners are shown ([WG12], [BBT02],...). Adding these improvements to a planner increases its performance, but the planner is still not able to solve key scenarios important for the proposed planner.
To these ends, a new approach for a Prioritized Planner is described in the following chapter.

CHAPTER 3

Approach

In this chapter the approach and algorithm of the MRRP are developed. First the terminology used in this thesis is introduced. The requirements to the MRRP are concluded in Section 3.2. Section 3.3 describes the chosen internal structure of the MRRP. Sections 3.4 and 3.5 are describing the algorithm used by the MRRP, where Section 3.4 describes the adaption of the Single Robot Route Planner (SRRP) and Section 3.5 describes possible extensions to the MRRP.

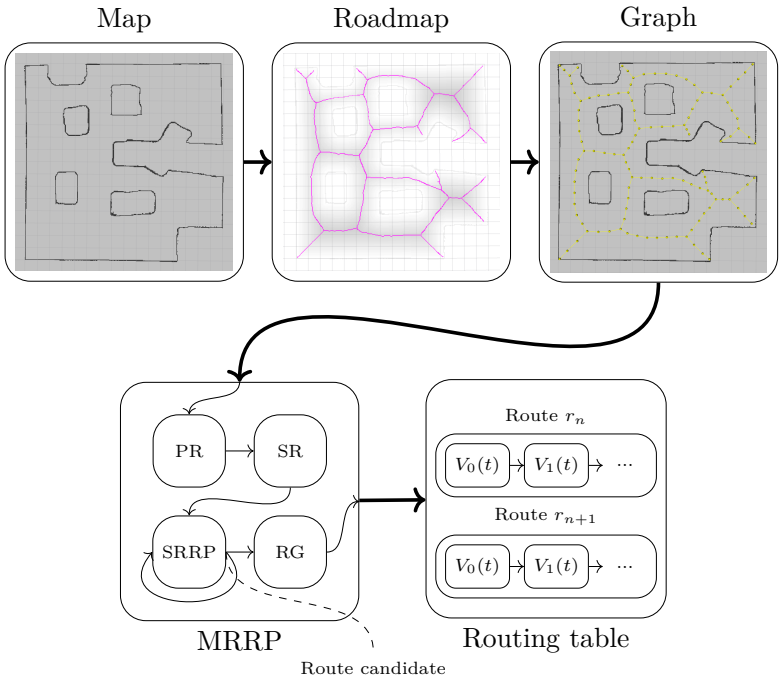


Figure 3.1: The overall picture of the Route Planner

3.1 Terminology

In Figure 3.1 the overall picture of the routing table generation is shown, to clarify the terminology used in this thesis. As one can see the map is shown in the top left corner. This map is a grid-map representing static obstacles in an environment. An algorithm is used to convert this map into a roadmap, which is based on a voronoi path. This roadmap is processed further to generate a graph, which can be used by the MRRP. The MRRP consists of the Priority Rescheduler (PR), the Speed Rescheduler (SR), the SRRP and the Route Generator (RG). The Priority Rescheduler (PR) and the Route Generator (RG) are only shown for completeness because they pass the graph combined with a corresponding schedule to the SRRP. The SRRP generates for each robot a route candidate. The reason the SRRP generates route candidates and not routes is because they are not synchronized and not part of a routing table. Therefore, the RG receives a list of route candidates and generates the final routing table consisting of a list of synchronized routes.
These algorithms used to generate the specific structures mentioned are described in Chapters 3 and 4.
The following glossary explains the most important terms and definitions:

- **Graph** refers to the search graph used by the route planner. The graph has a bijective mapping to the used environment, where each vertex is mapped to a segment of the environment.
- **Segment** is a small area of an environment with a specific shape. Every segment of an environment can be mapped to exactly one vertex of the corresponding graph.
- **Routing table** is a synchronized list of routes.
- **Route** describes a list of segments, including space and time, which are part of a routing table.
- **Route candidate** describes any list of vertices, including space and time.

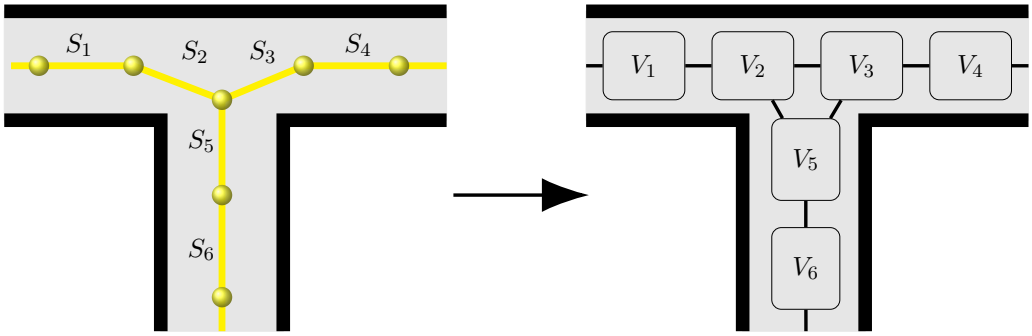
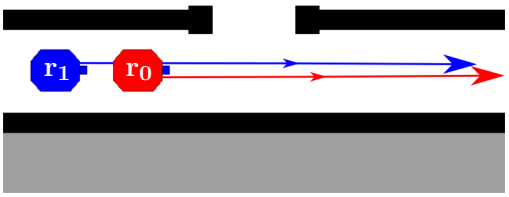


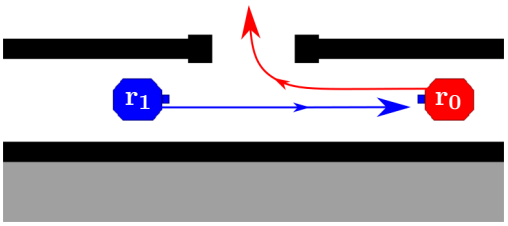
Figure 3.2: A graph transformation from any roadmap consisting of subsegments to the search graph used by the MRRP. ($S_x \rightarrow V_x$)

3.2 Requirements

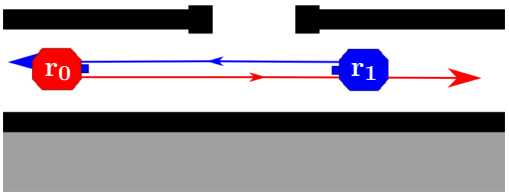
As mentioned in Chapter 1, the MRRP is based on the framework proposed in [BRS+15]. This work describes an ACS, which coordinates a number of AGVs centralized.
The ACS consists of a job Planner, arranging the single goals of the robots and a Route Planner, which has to find a number of vertices leading all AGVs to their goals. The proposed MRRP is used as Route Planner of the ACS. An AGV consists of a Behavior Controller, a Self-Localization, Sensors, a Motor Controller and a Navigation module. These modules enable the AGV to avoid dynamic or static obstacles locally.
Therefore, it can be assumed, that if there exists a path p_x through a segment s_x the AGV will find it. This means, that the MRRP only has to consider a search graph $G(V, E)$ without knowing the exact structure of the environment. In G a vertex represents a path segment with a specific length and width. The length is needed to find the shortest route candidate using a search algorithm and the width to check if a robot r_x can move through v_x. The edges E are undirected and representing the connection between two neighboring nodes. In Figure 3.2 a roadmap transformed into a valid search graph for the MRRP can be seen.
On this graph, the MRRP should be able to find solutions for different scenarios. In order to make the MRRP work properly in tight environments, specific showcases are considered, which should be solvable using the MRRP. These cases are shown in Figure 3.3. Figure 3.3a shows two robots moving sequentially, Figure 3.3b shows a robot waiting on another one before moving out of a room, Figure 3.3c shows two robots switching places, Figures 3.3d and 3.3e are showing scenarios where a robot blocks another one's path and Figure 3.3f shows a scenario where one robot pushes another one, sitting on its goal, away to move into the room. These scenarios should work for multiple robots and in any combination.
Since the generated route has temporal dependencies between robots, it is not guaranteed, that the path is deadlock-free if robots have no communication to each other. Therefore, a strategy to synchronize the robots using the given routing table is necessary.



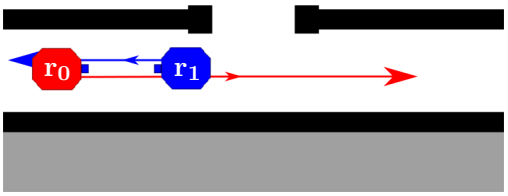
(a) **Scenario Sequential:** Robot r_1 follows r_0.



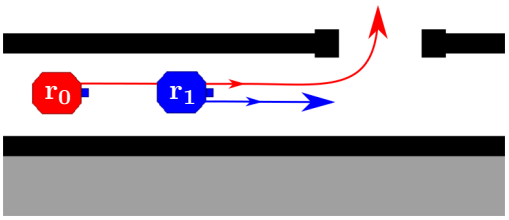
(b) **Scenario Wait:** Robot r_1 has to wait until r_0 enters the room



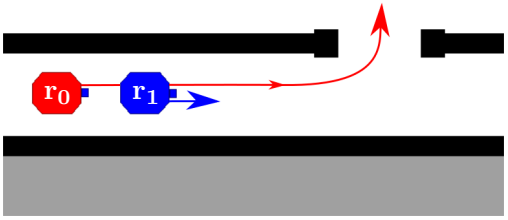
(c) **Scenario Avoid at Crossing:** To switch positions robot r_1 has to avoid r_0 in the door above the hallway



(d) **Scenario Avoid at Start:** Robot r_1 sits on its start position and wants to move to the left. Therefore, it has to move backwards and let r_0 pass.



(e) **Scenario Avoid at Goal**: Robot r_1 has already reached its goal but has to move back and forth again to avoid r_0 moving into the door on top of the hallway.



(f) **Scenario Push:** Robot r_1 has already reached its goal but has to move back and forth again to avoid r_0 moving into the door on top of the hallway. The difference to 3.3e is that the crossing is not directly at the robots goal and therefore r_1 has to search for the crossing.

Figure 3.3: Test cases used for creating the SRRP

3.3 Structure

With the requirements mentioned in Section 3.2 in mind, a planning strategy for the MRRP is selected. [LaV06] identified two types of methods used to address such planning problems:

1. Centralized Methods
2. Decentralized Methods

The centralized method describes one in which all vehicles are delineated as one system with multiple degrees of freedom. The papers [WG12], [LB11] and [PCM08] describe centralized approaches for solving the planning problem. Centralized approaches are in general complete and (at least theoretically) able to find optimal solutions, but their computational complexity grows exponentially with the number of robots [BBT02].
Decentralized methods, however, are generally incomplete, but scale better for a larger number of robots in terms of computational complexity. The decentralized approach is described in [LaV06] and used in many other papers like [BBT02], [WG12] and [CNKS15]. The MRRP should work for a large number of robots, therefore, a decentralized approach is chosen. However, the basic approach for decentralized methods is not able to find a solution to any of the given scenarios shown in Figure 3.3. Non-solvable examples are shown in [LaV06] and [Mar06], which match the scenarios in Figure 3.3c (Scenario Avoid at Crossing). Therefore, the prioritized planning approach used for the MRRP is revised by changing the SRRP and using a priority scheduling strategy similar to [BBT02] and a newly created velocity scheduling strategy explained below.
Since prioritized planning splits multi robot planning problem into multiple single ones, the structure of the MRRP is basically a Single Robot Route Planner (SRRP) which is invoked multiple times. To avoid overlapping route candidates a Route Coordinator, which is aware of already planned route candidates, is integrated into the SRRP. Furthermore, strategies to resolve the given scenarios from Figure 3.3 are explained in Section 3.4. To these ends, a Multi Robot Collision Resolver (MRCR) is integrated into the SRRP to solve problems locally by waiting and avoiding other robots. Additionally, a priority-rescheduler and a speed-rescheduler are integrated into the MRRP to achieve deadlock-free routes when robot priority and speed matters (Section 3.5).
To keep the found routing table deadlock-free, while executing it, a synchronization strategy between the robots has to be used. The Route Generator explained in Section 4.2.1 refines the given routes by adding dependencies and synchronization points to the given route candidate.
In Figure 3.4 the structure of the MRRP is shown. The MRRP receives a graph and a set of goals to initiate the planning task. The Priority Rescheduler and the Speed Rescheduler use random priority schemes for the first planning attempt, which are given to the SRRP. The SRRP consists of a Potential Expander the Route Coordinator and the MRCR. The Potential Expander elects with every iteration of the SRRP a vertex to expand and gives it to the Route Coordinator. The Route Coordinator checks if the

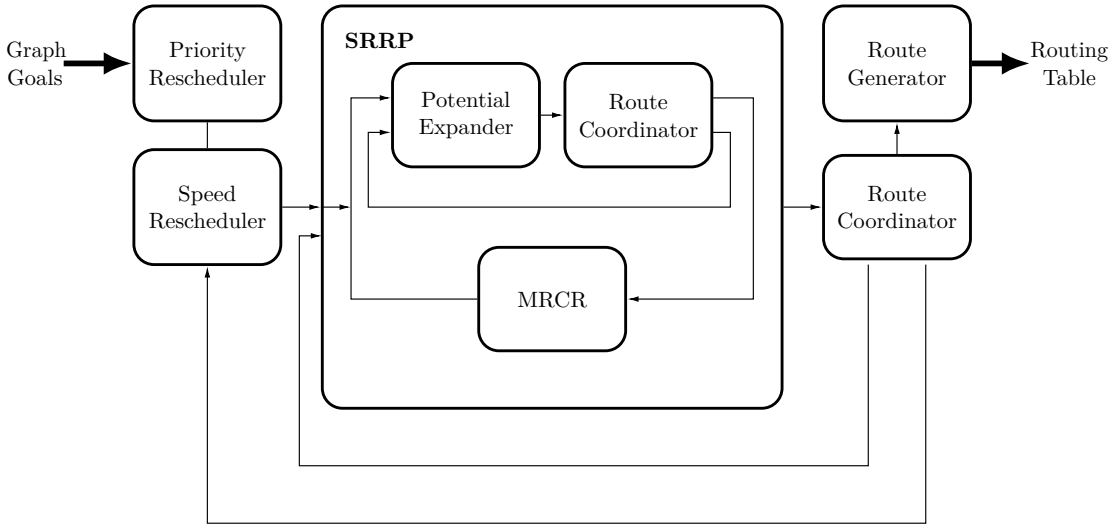


Figure 3.4: The structure of the MRRP

vertex and the assigned potential to it is conflicting with other route candidates. If the expansion is valid the Potential Expander continues to expand the next vertex. If the expansion is invalid the MRCR gets invoked to try to extend the search graph in a way that the Potential Expander finds a route candidate avoiding this collision and gives this graph to the Potential Expander again. If the goal of the route candidate is found the route candidate is given to the Route Coordinator to check it for its validity and added to a list of route candidates. Afterwards the SRRP is invoked with the next robot in the priority list. If the generated route candidate is invalid the Route Coordinator invokes the Speed Rescheduler to find a new speed schedule, which might solve the problem. If the Speed Rescheduler fails to find such a schedule the Priority Rescheduler (PR) has to reschedule the priorities and start the planning process for all robots again. If the SRRP succeeds to find a route candidate for every robot the Route Coordinator gives the list of route candidates to the route generator, which prepares the path for execution on a robot, by synchronizing them. This list of synchronized routes is called routing table. Because the Route Coordinator has to check and save route candidates it is needed inside the SRRP to constrain the expansions of the potential calculator and outside of the SRRP, to save and coordinate all found route candidates.

3.4 Single Robot Route Planner (SRRP)

In this section the approach for the SRRP included in the MRRP is described. The SRRP is mainly responsible for finding subsolutions to the introduced scenarios in Figure 3.3. Therefore, these scenarios are used to explain different parts of the approach.
[LaV06] describes different approaches to find a specific goal in a graph. These approaches are breadth and depth-first search, the Dijkstra algorithm, the A-Star algorithm and best-first search. For the MRRP the A-Star algorithm is utilized, because of optimality and the lowest time complexity compared to the other approaches.

Algorithm 3.1: A-Star Algorithm

```
   Input: Graph G, SourceVertex v_s, GoalVertex v_g
 1 priority Queue Q()
 2 foreach vertices v in Graph G do
 3 |   if v ≠ v_s then
 4 |   |   v.SetPotential (∞)
 5 |   |   v.SetPredecessor (∅)
 6 |   end
 7 |   Q.Add (v, ∞)
 8 end
 9 v_s.SetPotential (0)
10 Q.SetPredecessor (v_s, 0)
11 while Q not empty do
12 |   vertex v = Q.PopElement ()
13 |   if IsGoal (v) then
14 |   |   return G
15 |   end
16 |   foreach Neighbor v_n of v do
17 |   |   if v.GetPotential () + GetCosts (v, v_n) < v_n.GetPotential () then
18 |   |   |   v_n.SetPotential (v.GetPotential () + GetCosts (v, v_n))
19 |   |   |   v_n.SetPredecessor (v)
20 |   |   |   Q.SetPriority (v_n, v_n.GetPotential () + CalcHeuristic (v_n))
21 |   |   end
22 |   end
23 end
```

3.4.1 Single Robot Planning Algorithm

The A-Star algorithm is used in graph theory to determine if a goal vertex is reachable from a defined start vertex and what are the minimum costs to reach this goal. Therefore, the graph has to define movement costs between two vertices. The A-Star algorithm is shown in Algorithm 3.1. At the start of the algorithm, every vertex of the search graph G is assigned with an infinite potential and no predecessor vertex. The variable potential saves the overall costs for moving from the start vertex to the current vertex. The variable predecessor marks the vertex, right before the current vertex on the shortest path. Furthermore, all vertices are added to a sorted queue with priority infinity. Since the start vertex has to be expanded first and the shortest path from start to start has zero costs, the potential and the priority are set to zero.
Now the algorithm iterates over all elements in the queue by priority and tries to find the goal. Therefore, the element with the highest priority is selected first and expanded. Expand means that all neighbors of the selected vertex are updated with the minimum of their current potential and the selected vertices potential plus the costs from the selected vertex to the neighbor. The priority of such a vertex is calculated by using the potential plus the estimated distance to the goal, defined by a proper heuristic. Such a heuristic has to be admissible to reach optimality using the A-Star algorithm. Admissible means that the heuristic must not overestimate the costs to the goal vertex. Such an admissible heuristic would be the Euclidean distance for example. Since the single robot planner has to find the shortest path between two vertices and the A-Star algorithm only assigns potentials to the vertices, every time a potential is updated, the predecessor is updated as well with the selected vertex, to be able to find the shortest path by backtracking from the found goal vertex to the start. In Algorithm 3.1 two functions are used, which are assumed to be existent:

- `float` **`CalcCosts`**`(start vertex, goal vertex)`
  Calculates the cost between two vertices. This costs are dependent on the goal the algorithm has to achieve. In path planning these costs are often the path-length or the travel time between two vertices.

- `float` **`CalcHeuristic`**`(vertex)`
  Estimates the costs needed to travel from the given vertex to the goal. It is important that the real costs are smaller equal to the value of GetPotential.

Since this algorithm is created for single robot path planning, a few additions have to be made for using it in multi robot path planning. Firstly the SRRP has to be aware of already planned route candidates. To these ends, a Route Coordinator (Section 3.4.2) is introduced, which saves all planned route candidates and constrains newly planned ones. This Route Coordinator presents a method for saving an already planned route candidate and a method for checking if a vertex is occupied at a specific point in time. Therefore, a vertex's potential is only updated if it is not occupied by another robot. This addition can be seen in Algorithm 3.2 line 17. This is sufficient for a minimal working solution for a multi robot path planner, but since the MRRP should be able to solve complex

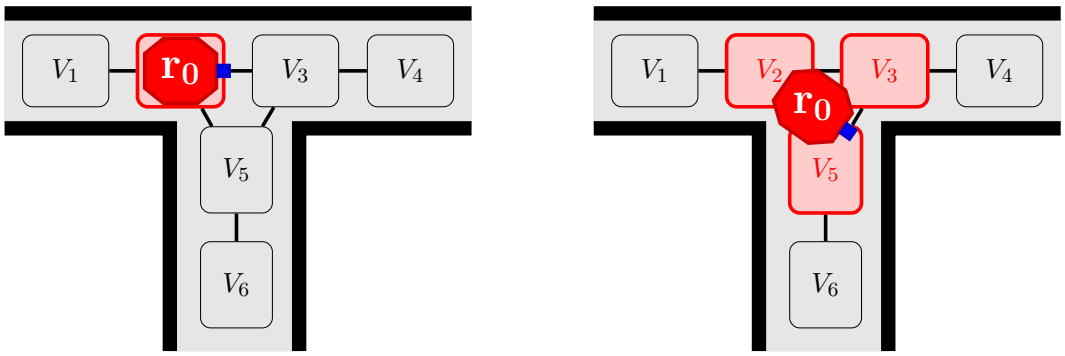
problems like waiting on other robots and avoiding other robots, another addition is included. Therefore, a module, which is able to resolve potential collisions, called MRCR (Section 3.4.3) is added. It checks if a vertex is rejected because of a robot collision and tries to bypass the collision by extending the search graph. It returns a list of possible resolutions to the SRRP, which have to be added to the queue. The integration into the algorithm is shown in Algorithm 3.2 line 23.

Algorithm 3.2: Adapted A-Star Algorithm

```
   Input: Graph G, SourceVertex v_s, GoalVertex v_g
 1 priority Queue Q()
 2 foreach vertices v in Graph G do
 3 |   if v ≠ v_s then
 4 |   |   v.SetPotential (∞)
 5 |   |   v.SetPredecessor (∅)
 6 |   end
 7 |   Q.add(v, ∞)
 8 end
 9 v_s.SetPotential (0)
10 Q.SetPriority (v_s, 0)
11 while Q not empty do
12 |   vertex v = Q.PopElement ()
13 |   if IsGoal (v) then
14 |   |   return G
15 |   end
16 |   foreach Neighbor v_n of v do
17 |   |   if RouteCoordinator.CheckVertex (v,v_n) then
18 |   |   |   if v.GetPotential () + GetCosts (v, v_n) < v_n.GetPotential ()
   |   |   |    then
19 |   |   |   |   v_n.SetPotential (v.GetPotential () + GetCosts (v, v_n))
20 |   |   |   |   v_n.SetPredecessor (v)
21 |   |   |   |   Q.SetPriority (v_n, v_n.GetPotential () + CalcHeuristic (v_n))
22 |   |   |   end
23 |   |   else
24 |   |   |   set of vertices V_r = CollisionResolver.Resolve (v, v_n, G)
25 |   |   |   foreach v_r in V_r do
26 |   |   |   |   Q.add(v_r, v_r.GetPotential () + CalcHeuristic (v_r))
27 |   |   |   end
28 |   |   end
29 |   end
30 end
```

timestep / *robot*	t_n	t_{n+1}	t_{n+2}
robot r_0	V_2	V_2, V_5	V_2, V_3, V_5
robot r_1	V_{13}	V_{13}	V_{14}
robot r_2	V_{12}	V_{12}	V_{13}

Figure 3.5: The data structure for saving the robot routes. t_n saves the vertices occupied for the specific time slice.



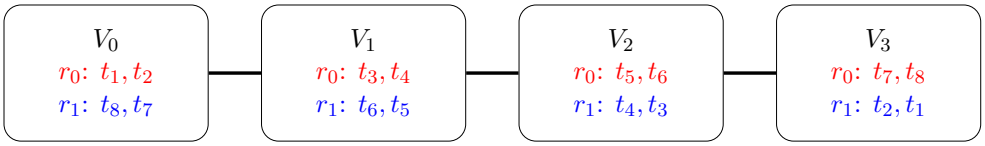
(a) Robot occupies only one vertex at a point in time

(b) Robot occupies three vertices at a point in time, while moving through the crossing

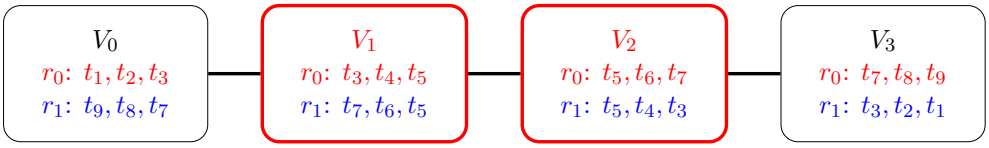
Figure 3.6: Segment occupation of a moving robot

3.4.2 Route Coordinator

The Route Coordinator is responsible for saving already planned route candidates from the SRRP and constraining the generation of new ones. To save these route candidates different data structures can be used. For a simple path planner, one can save generated paths spatially, which has the disadvantage that a specific space is reserved for only one robot. Another solution would be, to save space and direction of the robot's movement and allow robots to move on the same path if their direction matches. This approach would solve our scenario from Figure 3.3a (Scenario Sequential), but would cause problems, for other scenarios from Figure 3.3, where time has to be considered. The idea to solve this issues is to use a timed data structure for saving route candidates given by the SRRP.
This data structure is shown in Figure 3.5. It saves for each robot the locked vertices at every point in time, which gives the SRRP the possibility, to include waiting steps into the planner for solving temporal planning problems as well. Furthermore, the Route Coordinator can plan route candidates, which can go forth and back on one vertex to avoid other robots. The data structure has also to be able to save more than one vertex for every robot and time step because by the given data-structure it can happen, that



(a) Scenario, where robots r_0 and r_1 will find a invalid route candidate.



(b) Solution to the problem in 3.7a by using a slight time overlap for each vertex. Therefore, a collision is detected in V_1 and V_2

Figure 3.7: A problem where two robots will switch places without detecting a collision. It is assumed that every vertex is two "time-steps" long.

one robot occupies more than one vertex if it moves for example through a crossing. Such a case is depicted in Figure 3.6. When using time for saving robot positions another problem arises. If two robots are moving in opposite directions and would collide in a specific spot, it can happen that the Route Coordinator is not able to recognize the collision if both robots switching their vertices at the same time, depicted in Figure 3.7a. To solve this problem, the vertices of a route candidate are saved with a slight time overlap shown in Figure 3.7b.
To integrate the Route Coordinator into the SRRP, it presents two methods:

- `bool` **`CheckVertex`**`(start vertex, goal vertex)`
  This method checks a given vertex at a specific point in time against all already saved route candidates in the Route Coordinator for collisions. It returns success or failure containing the colliding robot.
- `bool` **`SaveRoute`** `(route)`
  Checks a newly planned route candidate and saves it internally for constraining other ones.

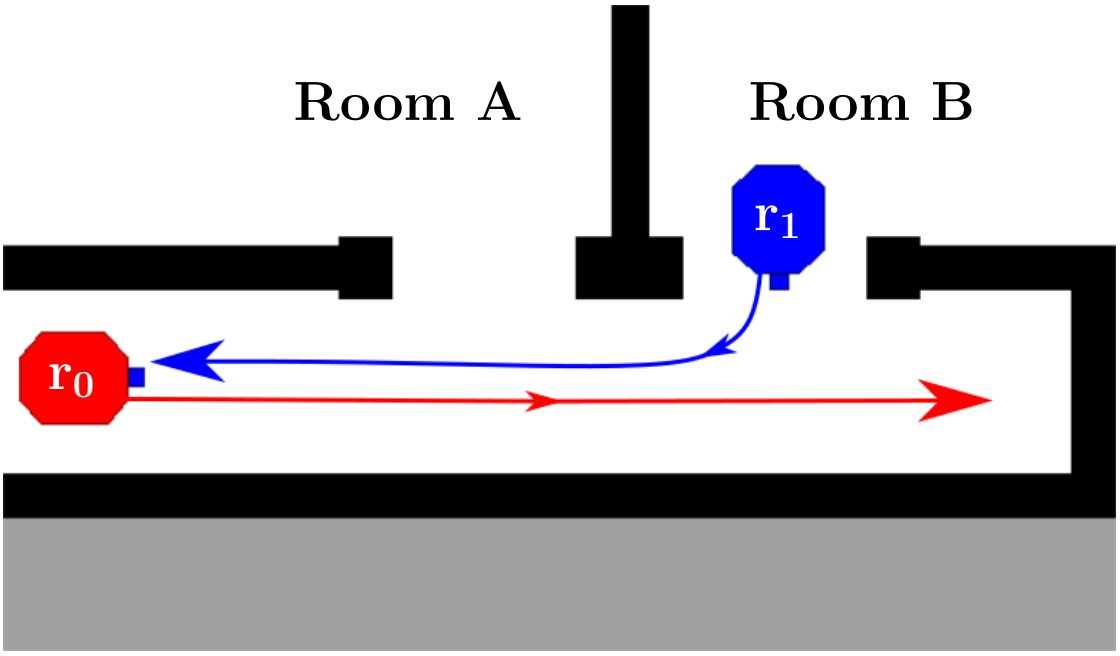


Figure 3.8: A scenario where robot r_1 has to avoid r_0

3.4.3 Multi Robot Collision Resolver (MRCR)

As one can see in Figures 3.3b (Scenario Wait), 3.3c (Scenario Avoid at Crossing), 3.3d (Scenario Avoid at Start), 3.3e (Scenario Avoid at Goal), and 3.3f (Scenario Push), taking time into account is not sufficient to solve these scenarios. Therefore, waiting steps and redundant vertices have to be added to the search graph. The MRCR is used to generate such waiting steps and vertices using three different algorithms described below:

- Backtracking Algorithm (BTA)
- Avoid Robot Algorithms (AVRA)
- Push Robot Algorithm (PRA)

The challenge for designing the MRCR is to create a solution, which does not remove possible solutions from the SRRP and still allows the MRRP to find the shortest route candidate from backtracking the potentials generated by the SRRP.
Assume the scenario from Figure 3.8. Therefore, robot r_1 wants to follow the blue line. Obviously, there are two solutions:

1. Robot r_1 carries on moving and avoids r_0 in Room A
2. Robot r_1 waits in Room B and lets r_0 pass first

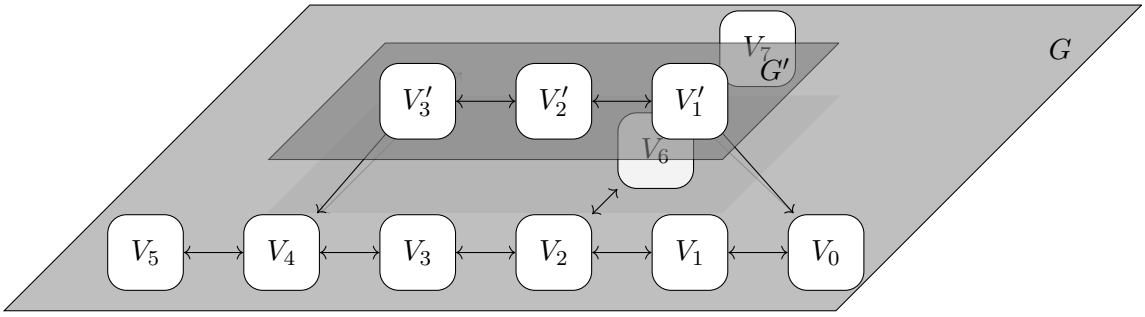


Figure 3.9: An example for a three dimensional graph extension. Vertices $V'_{1,2,3}$ are already assigned with different potentials compared to their parent nodes ($V_{1,2,3}$).

Both of these solutions lead to a valid behavior of the robot, but the problem, how to consider both solutions to the scenario arises. If we don't allow a robot to wait, only solution one will be found. This can lead to problems if waiting in the left crossing is the only solution, which can happen when another higher prioritized robot has already planned its route candidate. Furthermore, the planner has the possibility to select the least time-consuming solution. If we allow the robot to wait, the planner has somehow to save the calculated potentials for the alternative vertices without influencing the original route candidate. A simple solution, therefore, is to save a list with different potentials, but this makes the construction of a working traceback complicated. Furthermore, route candidates where a robot avoids another one will produce multiple entries in the list, which can lead to problems. The used idea for the MRRP is to temporarily extend the graph, by copying vertices.
In Figure 3.9 such a scenario is shown. G describes the search graph, which is extended with G' to resolve a collision. Therefore, the first robot's route candidate is already planned, leading the robot from V_5 to V_7. For the second robot two scenarios are kept in mind:

1. The robot moves from V_0 to V_7
2. The robot moves from V_0 to V_5

While resolving the problem, the MRCR has no idea in which branch the goal vertex is placed. Therefore, both branches have to be considered. The solution for scenario one would be to move from V_0 to V_7 in front of the first robot on $G = (V, E)$. The second solution would be to wait on V_1 until robot one has passed and carry on moving to V_5. Obviously, the solution to scenario one is found anyway. For scenario two, the second robot collides with the first one in V_3. Assume the MRCR has found a resolution, therefore, by waiting in V_1 and afterwards moving to V_3. Lets call the found vertices the collision set V_c. In order to guarantee both scenarios a new subgraph $G' = (V', E')$ is introduced.

Furthermore, a new Joint Graph is created for the planning process, which is generated as follows:

1. Copy all vertices involved in the collision to G'.
   $S \in V_c \leftrightarrow S' \in V'$
   Where S' is the copy of S

2. Map all edges to the new graph as follows:
   $\forall V_x, V_y \in V : (V_x, V_y) \in E \wedge V'_x \in V' \wedge V'_y \in V' \rightarrow (V'_x, V'_y) \in E'$
   Where V'_x is the copy of V_x

3. Create a joint graph $G_j = (V_j, E_j)$:
   $G_j = G \cup G'$

4. connect all edges of margin vertices with their original neighbor:
   $\forall V'_x, V'_y \in V' : (V_x, V_y) \in E \wedge d^+(V'_x) \leq 1 \wedge V'_y \notin V' \rightarrow (V'_x, V_y) \in E_j$
   Where $d^+(V_x)$ is the out-degree of V_x

After creating G_j critical potentials and predecessors, from vertices in G' have to be assigned. These are all vertices:

1. V'_x, which are used for waiting tasks

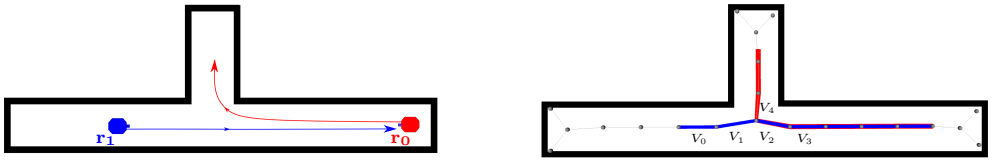
2. V'_y incident to edges connecting G' with G, where the vertex $V_y \in G$ has already a potential assigned

3. V'_z in between V'_x and V'_y

V'_x is needed because the time a robot needs to spend on the vertex exceeds the time it would need to move through the vertex. V'_y needed because there are only edges from G' to the base graph G to not change its structure. This means if we would let the Potential Expander continue planning on V_y instead on its successor V'_x, it will not find a successor and discards this resolution. V'_z is needed because if the Potential Expander finds a vertex with its potential assigned it will fail to expand this vertex. Therefore, the Potential Expander has to continue expanding on the last vertex with an assigned potential to find the intended resolution strategy, which is returned to the Potential Expander. Since there are only connections from G' to G, G' is removed after resetting the graph vertex's predecessors and for the next planning iteration, G_j consists only of G. To the SRRP the MRCR presents one method:

- `list<vertex> **Resolve**(vertex, vertex, graph)`
  Tries to find a resolution to a marked collision on the graph and returns the end vertex of the resolution, which the Potential Expander adds to its queue. Since for different scenarios multiple collision resolution strategies are possible the resolve method returns a list of these vertices.

In the following sections the used strategies for resolving collisions are explained. Therefore, the introduced scenarios in Figure 3.3 are used to explain these strategies.



(a) The start positions of the robots and the desired paths by the corresponding arrows.

(b) The solution to the given Problem with critical vertices marked ($V_0 - V_4$), which are used for the Backtracking Algorithm (BTA)

Figure 3.10: Example Testcase which needs the BTA (Solution shown in Figure 3.11)

Backtracking Algorithm (BTA)

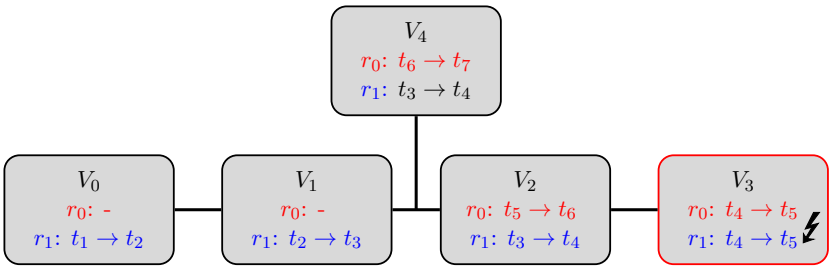
In Figure 3.3b (Scenario Wait) a scenario is shown, where a robot has to wait for another one. To solve this scenario, waiting steps can be inserted into r_1's route candidate. Because of the Route Coordinator's data structure, the SRRP is also able to take time into account. Thereby the problem arises, where and when to insert waiting steps. One approach would be to consider waiting steps for every expansion, like the Spatio-Temporal A* algorithm proposed in paper [WG12]. This has the major disadvantage that, the time complexity of the planning problem is increased by one dimension through adding time. To keep time complexity low, the MRCR approaches this problem by backtracking. Assume the scenario depicted in Figure 3.3b (Scenario Wait). Robot r_0 is planned first and follows the red arrow. If the SRRP expands the route candidate for r_1 along the blue line, it detects a collision with r_0 in the hallway. To resolve this collision the MRCR is triggered, which backtracks the route candidate until the first vertex, distinct from the r_0's route candidate is found. This vertex is assigned with the time, r_0 reaches the first vertex on its route candidate distinct from r_1's. From this point, the SRRP can carry on expanding r_1's route candidate without finding the previous collision.
How to integrate this approach into the MRCR is shown in Algorithm 3.3. The shown method is called recursively until a valid solution or the start vertex of the expansion is found. Therefore, the algorithm calculates the potential of the colliding robot on the collision vertex P_c, to assign this potential to the new copy of the vertex. If V_a is not the start vertex of the expansion it is tracked back once, by copying V_a to the new vertex to expand V'_e and the predecessor of V_a to the new actual vertex V'_a. Furthermore, the Potential of V'_e is reset and the successor of the vertex is saved to enable the potential calculator to expand this vertex again. For V'_a the Potential is set to P_c and its successor to V'_e. If it is the start vertex tracking back is not possible. Therefore, V_a is copied to V'_a and the potential is set to P_c for V'_a. After creating the new vertices, it is checked if the newly added vertex V'_a has a valid potential to continue expanding. If not and a new collision is detected, V'_a and V'_e are backtracked again. Obviously, if V_a is the start vertex further backtracking is not possible and NULL is returned. In the shown algorithm Resolve is called in line 26. This method calls all resolve methods (BTA, Avoid Robot at Crossing Algorithm (AVRCA), Avoid Robot at Start Algorithm (AVRSA), Avoid Robot

Algorithm 3.3: BTRA

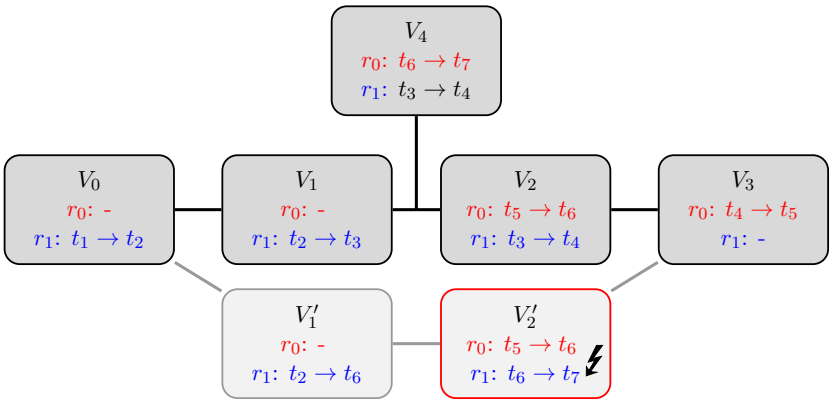
Input: pointer(*acutal_vertex* V_a), pointer(*vertex_to_expand* V_e), *colliding_robot* R_c

Output: pointer(*found_vertex* V_f) or NULL

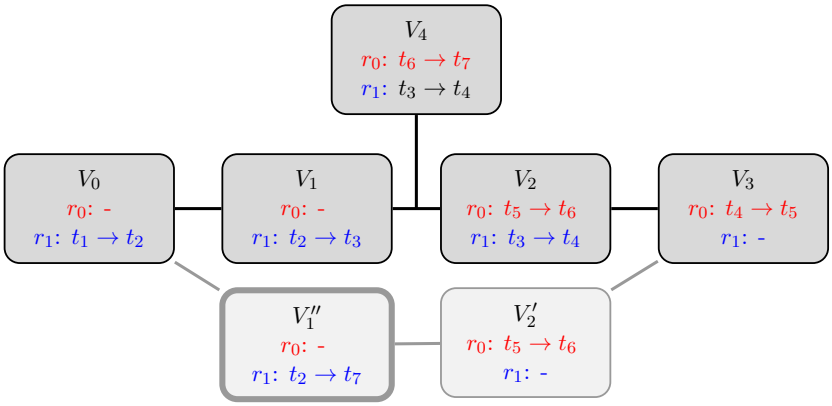
```
1  P_c = CalculatePotentialOfCollision (V_e, R_c)
2  if P_c == NULL then
3  |  return NULL
4  end
5  P_ca = CalculatePotentialOfCollision (V_a, R_c)
6  if V_a is start_vertex ∨ P_ca == NULL then
7  |  V'_e = V_e
8  |  V'_a = Copy (V_a)
9  |  V'_a.SetPotential ( P_c)
10 |  V'_a.SetCollidingRobot ( R_c)
11 |  V'_a.SaveSuccessor ( V'_e)
12 else
13 |  V'_e = Copy (V_a)
14 |  V'_e.SetPotential ( -1)
15 |  V'_e.SaveSuccessor ( V_e)
16 |
17 |  V'_a = Copy (V_a.GetPredecessor ())
18 |  V'_a.SetPotential ( P_c)
19 |  V'_a.SaveSuccessor (V'_e)
20 end
21
22 if IsNodeFreeUntil (V'_a, P_c, out R_c) then
23 |  return V'_a
24 else
25 |  if  R_c is not NULL ∧ V_a is not start_vertex then
26 |  |  return  Resolve (V'_a, V'_e, R_c)
27 |  end
28 end
29 return NULL
```



(a) The initial robot Collision on V_3



(b) The first backtracking iteration creating V_1' and V_2', which leads to another collision



(c) The last iteration of the BTA by setting V_1'' to the appropriate time step.

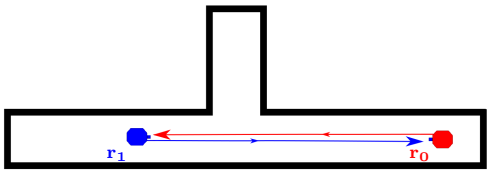
Figure 3.11: The implementation of the BTA

at Goal Algorithm (AVRGA), Push Robot Algorithm (PRA)) sequentially to resolve a collision.
An example for this algorithm is shown in Figures 3.10 and 3.11. Figure 3.10 shows the start positions of the two robots and their final route candidates. Additionally, critical vertices used by the MRCR are marked with V_x. In Figure 3.11 the procedure of resolving this collision is shown. In this scenario, one can see the four vertices used by the MRCR already assigned with their given time steps. Robot r_0's time steps are fixed because this route candidate is already found and added to the Route Coordinator. Time steps drawn in black are expanded by the Potential Expander but not necessary for the actual solution. As mentioned in 3.4.2 each vertex is saved with a time overlap to the next one. In Figure 3.11a the situation, where the planner fails to expand a route candidate, can be seen. At this point the MRCR gets triggered with $V_2 \rightarrow V_3$. The resolver tracks back onto vertex V_1. This vertex gets copied and the potential for waiting on this vertex is set. V_2' is copied from V_2. Its potential is kept empty because at this point the final value of its potential is not known. In the next iteration a new collision is found in V_2' shown in Figure 3.11b. Therefore, the MRCR recognizes, that V_1 is only occupied from robot one and sets the priority to the value robot zero leaves the collision vertex. As one can see V_1'' is the vertex, which is finally used for waiting until robot zero has passed. Therefore, the Potential Expander can continue expanding on this vertex and will find a solution for this problem.

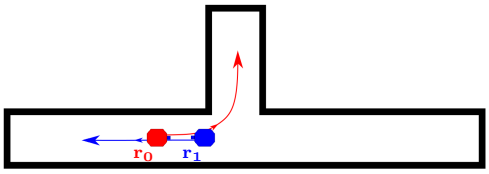
Avoid Robot Algorithms (AVRA)

In Figure 3.3c (Scenario Avoid at Crossing) a scenario can be seen where both robots want to switch places. It is assumed, that the start point of robot r_0 is the goal point of r_1 and vice versa. Furthermore, it is assumed, that r_0's route candidate is already planned. If the SRRP consists only of the Route Coordinator and the BTA, the planner will fail because it will end backtracking in the start point of r_1. Obviously, this scenario can be solved by letting r_1 move into the door above the hallway and back on its path. Furthermore, in Figure 3.3d (Scenario Avoid at Start) a scenario can be seen, where robot r_1 sitting on its start point is blocking r_0's route candidate. This scenario can be solved by moving r_1 in the opposite direction until it has passed the door. Afterwards, r_0 can move into the door above and r_1 can finally follow its route candidate toward the goal. In Figure 3.3e (Scenario Avoid at Goal) a scenario is shown, where robot r_1 sits on its goal and blocks r_0's route candidate. This scenario can be solved similar to the one of Figure 3.3d (Scenario Avoid at Start), by moving r_1 through the crossing and let it wait there.
All of these scenarios can be solved by allowing the MRCR to not only backtrack the route candidate but also enable him to expand into branches of crossings and back to create scenarios like the one in Figure 3.3b (Scenario Wait). The MRCR presents, therefore, three different algorithms to avoid higher prioritized robots:

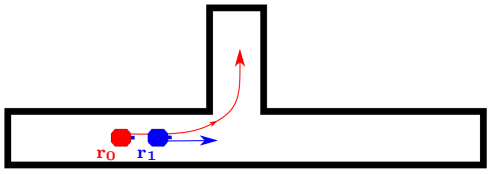
1. **AVRCA (Figure 3.3c / Scenario Avoid at crossing)**

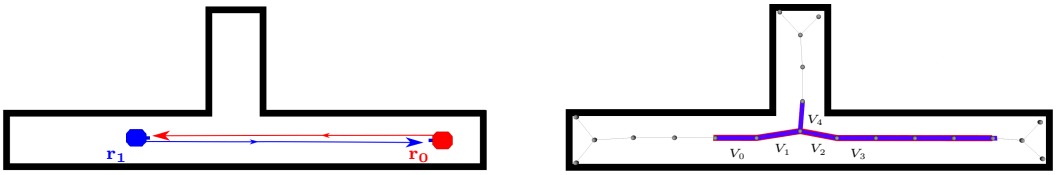


2. **AVRSA (Figure 3.3d / Scenario Avoid at Start)**



3. **AVRGA (Figure 3.3e / Scenario Avoid at Goal)**





(a) The start positions of the robots and the desired paths by the corresponding arrows.

(b) The solution to the given Problem with critical vertices marked ($V_0 - V_4$), which are used for the avoid crossing resolution strategy

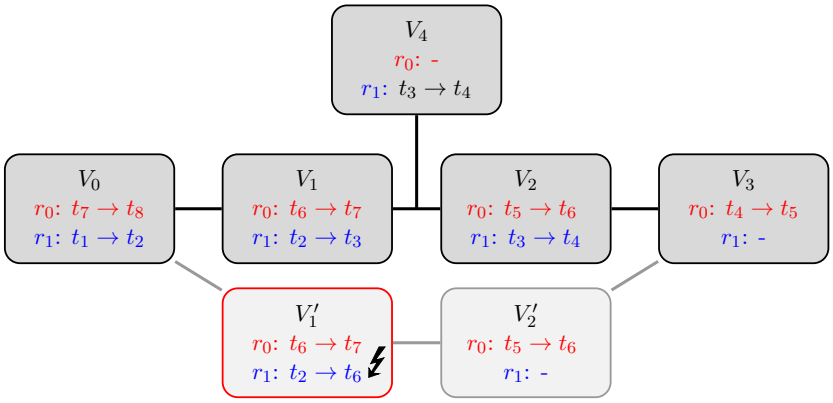
Figure 3.12: Example Testcase which needs the AVRCA (Solution shown in Figure 3.13)

For the **AVRCA** strategy this will look like as follows. The SRRP finds a collision between robots r_1 and r_0 and uses the BTA. After the MRCR reaches the door (which is seen as crossing in the graph), it will try to wait on the upper branch of the crossing and simultaneously continue backtracking. In this scenario, the BTA will fail, but waiting in the upper branch of the crossing will succeed and is added to the route candidate. The time until the robot sits on this branch is the time, where r_0 leaves the vertex, which is connected to the waiting branch. This creates the same scenario as shown in Figure 3.3b (Scenario Wait) after the BTA is applied, which can be solved by expanding towards r_1's goal.
The algorithm, therefore, is shown in Algorithm 3.4. This algorithm is called after every backtracking iteration. At first, the potential of the colliding robot is calculated, which is the time until a robot has to wait on another vertex to be allowed to move through this one. If in between V_a and V_e is a crossing the AVRCA can be applied. Therefore, each vertex of the crossing except V_a and V_e are candidates for the avoidance strategy. Now for each candidate, a copy of it is produced and V_e and V'_a are set as successor and predecessor to build the connection to the base search graph G. Furthermore, the potential is set to P_c or to the expansion costs of V'_x if they exceed P_c. If this route candidate is valid V'_x is added to the list of found vertices V_f. If this route candidate contains a collision with another robot the PRA is called, which is explained in Section Push Robot Algorithm (PRA). This method basically tries to expand towards any other free vertex and back to V'_x. If a solution to this problem is found it is again returned to the SRRP.
Like in Section 3.4.3 an example for such a resolution is shown in Figures 3.12 and 3.13. Figure 3.12 shows the start positions of the two robots and their final route candidates and in Figure 3.13 the procedure of resolving this collision is shown. Again r_0's time steps are fixed. Time steps drawn in black are expanded by the Potential Expander but not necessary for the actual solution.
In Figure 3.13a, the state, where the MRCR has backtracked once, is shown. To resolve the new collision detected in V'_2, Vertex V'_4 is added to the graph to present a valid route candidate for the Potential Expander. This is shown in Figure 3.13b, where V'_4 is returned to the SRRP, to continue expanding.

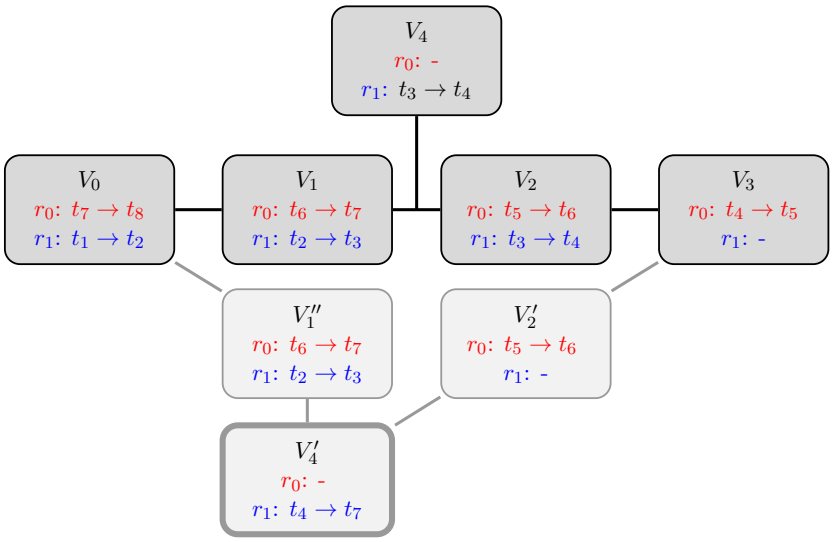
Algorithm 3.4: AVRCA

Input: pointer(*acutal_vertex* V_a), pointer(*vertex_to_expand* V_e), *colliding_robot* R_c
Output: *list of found_vertices* V_f

```
1  P_c = CalculatePotentialOfCollision (V_e, R_c)
2  if P_c == NULL then
3  |  return NULL
4  end
5  if crossingBetween(V_a,V_e) then
6  |  V'_a = Copy (V_a)
7  |  V_ap = V_a.GetPredecessor ()
8  |  V'_a.SetPotential (V_ap.GetPotential() + CalcCosts (V_ap, V_a))
9  |  foreach vertices V_x in crossing except V_a and V_e do
10 |  |  V'_x = Copy (V_x)
11 |  |  V'_x.SaveSuccessor ( V_e)
12 |  |  V'_x.SavePredecessor (V'_a)
13 |  |  V'_x.SetPotential (Max (P_c, V_a.GetPotential () + CalcCosts (V_a, V'_x)))
14 |  |
15 |  |  if IsNodeFreeUntil (V'_x, V'_x.GetPotential (), out R_c) then
16 |  |  |  V_f.Add (V'_x)
17 |  |  else
18 |  |  |  if R_c is not NULL then
19 |  |  |  |  V_f.Add (PRA (V_a, V'_x, R_c))
20 |  |  |  end
21 |  |  end
22 |  end
23 end
24
25 return V_f
```

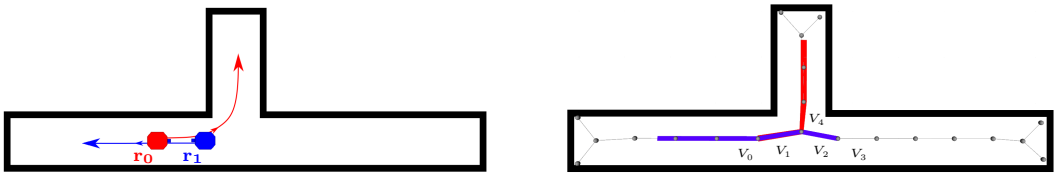


(a) The backtracking attempt equal to 3.11b, which leads to another collision



(b) The avoidance resolution by copying V_4 and setting the appropriate potential. Simultaneously the BTA will continue until one of both has found a solution.

Figure 3.13: The implementation of the AVRCA



(a) The start positions of the robots and the desired paths by the corresponding arrows.

(b) The solution to the given Problem with critical vertices marked ($V_0 - V_4$), which are used for the AVRSA

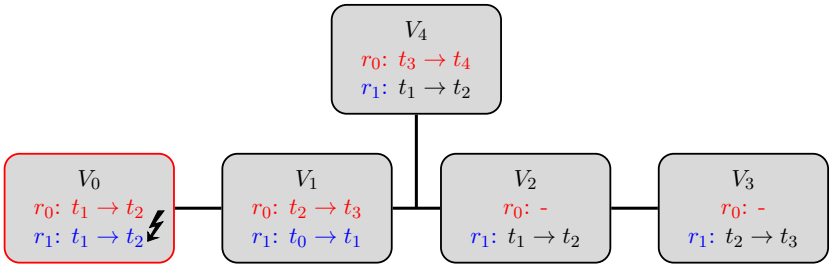
Figure 3.14: Example Testcase which needs the AVRSA (Solution shown in Figure 3.15)

The **AVRSA** strategy is used if a robot sits on its start point and blocks a higher prioritized robots route candidate. In Figure 3.3d (Scenario Avoid at Start) such a scenario is shown. The Potential Expander detects a collision between robots r_1 and r_0. Backtracking is not possible because r_1 already sits on its start position. To resolve this problem the MRCR expands into all branches of the crossing to the right of r_1 and lets r_1 wait until r_0 has passed. Obviously, in the scenario, the expansion into the upper branch of the crossing will fail, but the one in the right branch will succeed and the planner will find a valid solution.
The algorithm used for this scenario is shown in Algorithm 3.5. If the actual base vertex for the expansion V_a is the start vertex, the algorithm iterates over all neighbors on the opposite site of V_e. This means if V_e is in the list of predecessors, the successors are taken and vice versa. For each V_x in this neighbor-set V_a and V_x are copied. The successors and predecessors are set, to create a new start sequence in the following direction: $V_a \rightarrow V_x' \rightarrow V_a' \rightarrow V_e$. Furthermore, the potential of V_x' is set to P_c or to the expansion costs of V_x' if they exceed P_c. If this route candidate is valid V_x' is added to V_f. If not the PRA, shown in Section Push Robot Algorithm (PRA), is used to find another waiting spot.
In Figures 3.14 and 3.15 an example to this scenario is shown. Figure 3.12 shows the start positions of the two robots and their final route candidates and in Figure 3.13 the procedure of resolving this collision is shown. Again r_0's time steps are fixed. Time steps drawn in black are expanded by the Potential Expander but not necessary for the actual solution.
In Figure 3.15a, the collision is shown, where the Potential Expander tries to expand into V_0. This can be resolved by copying V_2 to V_2' and expanding into it. Furthermore, V_1 is copied to V_1' to present a vertex, which allows the Potential Expander to find a route candidate back. Additionally, V_4' is created and checked if an expansion into V_4' and back is possible. Obviously, this fails because r_0 will overrun r_2 when waiting in V_4. The newly created graph can be seen in Figure 3.15b.

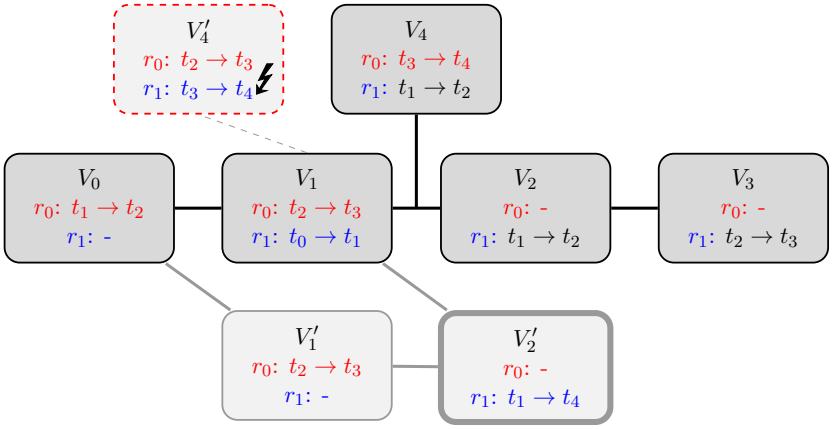
Algorithm 3.5: AVRSA

Input: pointer(*acutal_vertex* V_a), pointer(*vertex_to_expand* V_e), *colliding_robot* R_c
Output: *list of found_vertices* V_f

```
1  P_c = CalculatePotentialOfCollision (V_e, R_c)
2  if P_c == NULL then
3  |  return NULL
4  end
5  if V_a is start then
6  |  foreach vertices V_x in neighbours which not contain V_e do
7  |  |  V'_a = Copy (V_a)
8  |  |  V'_x = Copy (V_x)
9  |  |  V'_x.SaveSuccessor ( V'_a)
10 |  |  V'_a.SaveSuccessor ( V_e)
11 |  |  V'_x.SavePredecessor ( V_a)
12 |  |  V'_a.SavePredecessor ( V'_x)
13 |  |  V'_x.SetPotential (Max (P_c, V_a.GetPotential () + CalcCosts (V_a, V'_x)))
14 |  |
15 |  |  if IsNodeFreeUntil (V'_x, GetPotential (V_x), out R_c) then
16 |  |  |  V_f.Add (V'_x)
17 |  |  else
18 |  |  |  if R_c is not NULL then
19 |  |  |  |  V_f.Add ( PRA (V_a, V'_x, R_c))
20 |  |  |  end
21 |  |  end
22 |  end
23 end
24 return V_f
```

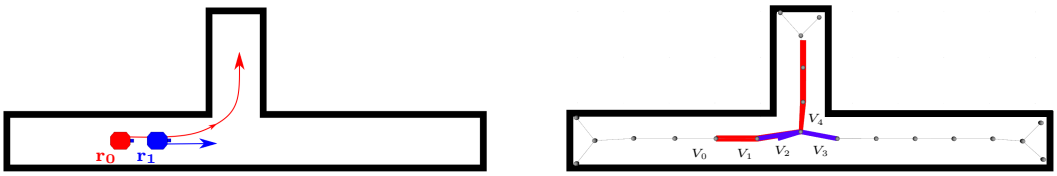


(a) Shown the expansion attempt into V_0, which leads to a collision



(b) The collision resolution by adding a new start sequence $V_1 \to V_2' \to V_1'$ to avoid robot r_0 route. V_4' is also expanded but fails due to a collision with r_0.

Figure 3.15: The implementation of the AVRSA



(a) The start positions of the robots and the desired paths by the corresponding arrows.

(b) The solution to the given Problem with critical vertices marked ($V_0 - V_4$), which are used for the AVRGA

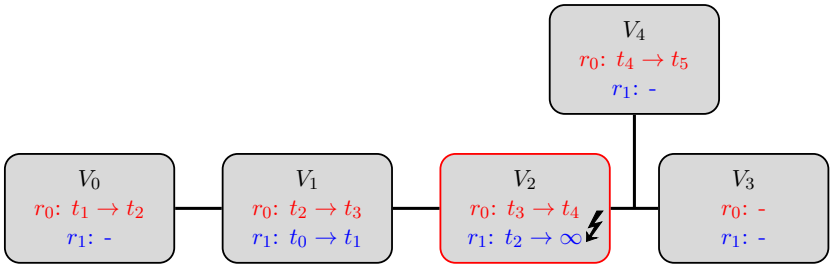
Figure 3.16: Testcase which needs the AVRGA (Solution shown in Figure 3.17)

The **AVRGA** strategy is used if a robot has already reached its goal and blocks a higher prioritized robots route candidate. In Figure 3.3e (Scenario Avoid at Goal) such a scenario can be seen. Robot r_1 moves towards its goal and stops expanding because the goal of r_1 is already found. This blocks robot r_0 and the planner will fail to find a route candidate. Therefore, r_1 is only allowed to stop on its goal if it is not occupied in the future. To wait on another vertex for this point in time the backtracking strategy can be used, but if r_1 starts in front of r_0 this will fail as well. Similar to the Avoid Robot at Start algorithm this can be solved as well by expanding into the next crossing to avoid r_0 there.
The algorithm, therefore, is shown in 3.6. If the vertex the planner wants to expand to (V_e) is the goal vertex the planner adds a new goal sequence. To these ends, the MRCR expands into the neighborhood, which not contains V_a. For each vertex V_x in this neighborhood a new goal sequence is created by copying V_e and V_x. V_e is the goal vertex and is copied once for moving through it (V_e'). V_x' is the vertex, where the robot tries to wait and assigned with the potential P_c or the expansion costs of V_x' if they exceed P_c. The Predecessors and Successors are set to fit the given goal sequence: $V_e' \rightarrow V_x' \rightarrow V_e$. If the robot is not able to wait in this vertex the PRA, shown in Section Push Robot Algorithm (PRA), is used to find another waiting spot. All results are saved into the list V_f and returned to the SRRP.
In Figures 3.16 and 3.17 an example to this scenario is shown. Figure 3.12 shows the start positions of the two robots and their final route candidates and in Figure 3.13 the procedure of resolving this collision is shown. Again r_0's time steps are fixed. Time steps drawn in black are expanded by the Potential Expander but not necessary for the actual solution.
In Figure 3.17a, the collision is shown, where r_1 tries to stay on its goal in V_2. In Figure 3.17b the resolution is shown, by adding the new goal sequence $V_2' \rightarrow V_3' \rightarrow V_2$. Another solution, which is tested by the algorithm is the sequence $V_2' \rightarrow V_4' \rightarrow V_2$, but this will fail due to a collision between r_0 moving upwards and r_1 waiting on V_4.

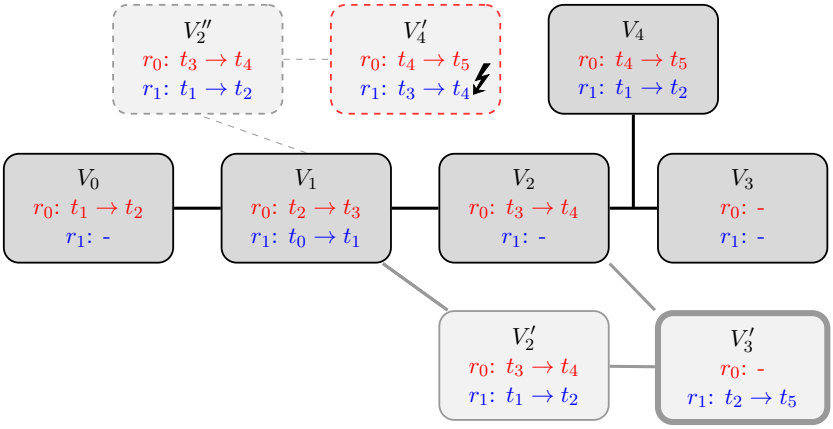
Algorithm 3.6: AVRGA

Input: pointer(*acutal_vertex* V_a), pointer(*vertex_to_expand* V_e), *colliding_robot* R_c
Output: *list of found_vertices* V_f

```
1  P_c = CalculatePotentialOfCollision (V_e, R_c)
2  if P_c == NULL then
3  |  return NULL
4  end
5  if V_e is goal then
6  |  foreach Vertices V_x in neighbours which not contain V_a do
7  |  |  V'_e = Copy (V_e)
8  |  |  V'_x = Copy (V_x)
9  |  |  V'_e.SavePredecessor ( V_a)
10 |  |  V'_x.SavePredecessor ( V'_e)
11 |  |  V'_e.SaveSuccessor ( V'_x)
12 |  |  V'_x.SaveSuccessor ( V_e)
13 |  |  V'_e.SetPotential ( V_a.GetPotential () + CalcCosts (V_a,V'_e))
14 |  |  V'_x.SetPotential (Max (P_c, V'_e.GetPotential () + CalcCosts (V'_e, V'_x)))
15 |  |
16 |  |  if IsNodeFreeUntil (V'_x, GetPotential (V_x), out R_c) then
17 |  |  |  V_f.Add (V'_x)
18 |  |  else
19 |  |  |  if R_c is not NULL then
20 |  |  |  |  V_f.Add (PRA (V_e, V'_x, R_c))
21 |  |  |  end
22 |  |  end
23 |  end
24 end
25 return V_f
```

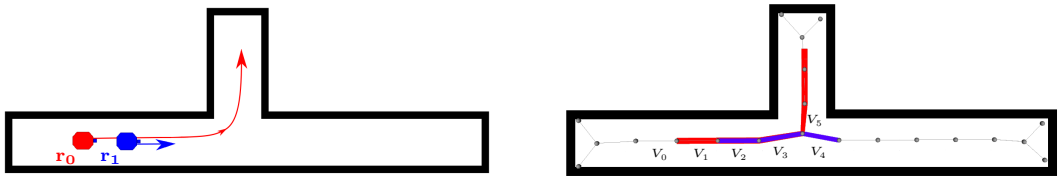


(a) The expansion of robot r_1 finding a collision in V_2, because it is not allowed to wait forever in this vertex.



(b) The solution for the given problem generated from the MRCR. Therefore a route arises, which moves the robot through the goal and back again. Additionally the route $V_2'' \to V_4' \to V_2$ is also created. Obviously there is a collision in V_4' between robots r_0 and r_1. Therefore this route is rejected by the MRCR.

Figure 3.17: The solution for the scenario shown in 3.16.



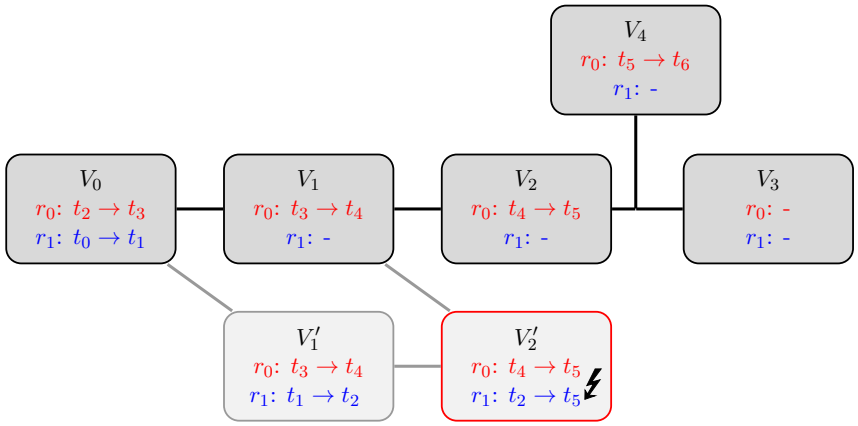
(a) The start positions of the robots and the desired paths by the corresponding arrows.

(b) The solution to the given Problem with critical vertices marked ($V_0 - V_4$), which are used for the PRA

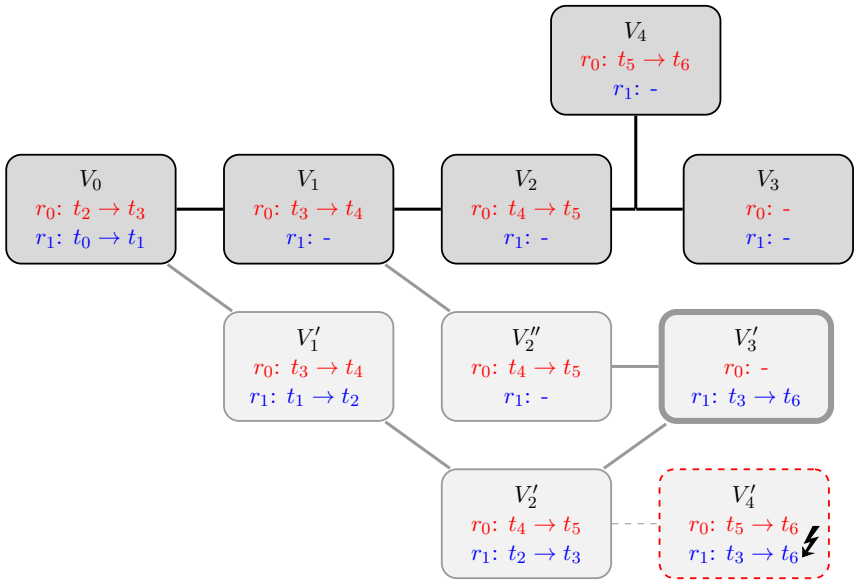
Figure 3.18: Example Testcase which needs the PRA (Solution shown in Figure 3.19)

Push Robot Algorithm (PRA)

The Backtracking Algorithm (BTA) and the Avoid Robot Algorithms (AVRA), solve problems where the robot passes a crossing or sits directly at a crossing. But many scenarios are not exactly like these ones. For example, Figure 3.3e (Scenario Avoid at Goal) shows a scenario solvable by the explained avoid robot at goal algorithm. If this scenario is slightly changed and r_1's goal is not exactly at the crossing but one vertex away this strategy fails because no free branch of a crossing is close to wait. Such a scenario is shown in Figure 3.3f (Scenario Push). The solution, therefore, is to allow the MRCR to "push" waiting vertices in the moving direction of the higher prioritized robot. For the scenario from Figure 3.3f (Scenario Push) the MRCR uses the avoid robot at goal algorithm at first. Obviously, a collision is detected again because robot r_0 moves through the same vertex. Now the MRCR is allowed to expand another vertex into all branches right of this vertex. In this scenario, a free vertex at the crossing is found and the MRCR assigns the found one with the time r_1 has to wait until r_0 has passed. Additionally, the MRCR has to add vertices for moving back from the waiting vertex.
The algorithm, therefore, is shown in 3.7. At first, the Potential of the colliding robot in V_e is calculated. If V_e is a wait_vertex, which means a vertex generated by any of the Avoid Robot Algorithms (AVRA) or by the PRA, the strategy can be applied. To these ends, V_e is set to the potential of V_a plus the costs for moving through V_a. For each V_x in the neighbor set, which not contains V_a, the following operations are executed. V_e is copied to create a vertex for the planner to find the route candidate back to the vertex in the base graph G and V_x is copied as new wait_vertex. Similar to the AVRGA the potential for the wait_vertex V'_x is set with the maximum of P_c and the cost for moving through V'_x. If this V'_x has a valid potential, which causes no collisions with other robots it is added to V_f. If there is a collision found in V'_x the PRA is called recursively.
In Figures 3.16 and 3.17 an example to this scenario is shown. Figure 3.12 shows the start positions of the two robots and their final route candidates and in Figure 3.13 the procedure of resolving this collision is shown. Again r_0's steps are fixed. Time steps drawn in black are expanded by the Potential Expander but not necessary for the actual solution.



(a) A collision, while the robot tries to use the AVRGA strategy



(b) The PRA by adding V_4 and V_3 to the resolution graph. V_4 is rejected because this vertex leads to a collision between robots r_0 and r_1.

Figure 3.19: The implementation of the PRA

Algorithm 3.7: PRA

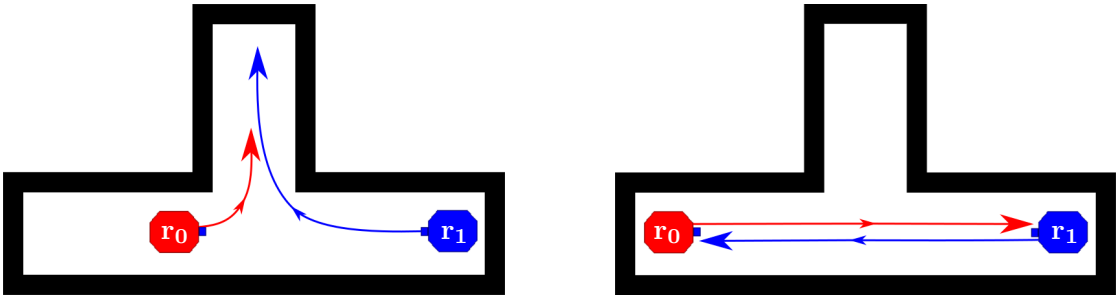
Input: pointer(*acutal_vertex* V_a), pointer(*vertex_to_expand* V_e), *colliding_robot* R_c
Output: *list of found_vertices* V_f

```
1  P_c = CalculatePotentialOfCollision (V_e, R_c)
2  if P_c == NULL then
3  |  return NULL
4  end
5  if V_e is wait_vertex then
6  |  V_e.SetPotential ( V_a.GetPotential + CalcCosts (V_a,V_e))
7  |  foreach vertex V_x in neighbours which not contain V_a do
8  |  |  V'_e = Copy (V_e)
9  |  |  V'_x = Copy (V_x)
10 |  |  V'_x.SavePredecessor ( V_e)
11 |  |  V'_e.SavePredecessor ( V'_x)
12 |  |  V'_x.SaveSuccessor ( V'_e)
13 |  |  V'_e.SaveSuccessor ( V_a)
14 |  |  V'_x.SetPotential ( Max (P_c, V'_e.GetPotential () + CalcCosts (V_a, V'_x)))
15 |  end
16 |
17 |  if IsNodeFreeUntil (V'_x, V'_x.GetPotential (), out R_c) then
18 |  |  V_f.Add (V'_x)
19 |  else
20 |  |  if R_c is not NULL then
21 |  |  |  V_f.Add (PRA (V_e, V'_x, R_c))
22 |  |  end
23 |  end
24 end
25 return V_f
```

In Figure 3.19a the first attempt to resolve a collision by the AVRGA is shown, which causes a collision in the wait_vertex V_2'. To try to find another solution the MRCR uses PRA. Therefore, V_2' is copied to V_2'' and V_2' is used to move to the new wait_vertex V_3', which is set with the potential the robot has to wait there. Simultaneously V_4 is copied as well and checked for a solution, which fails because of a collision when r_1 wants to wait in V_4'. The solution is shown in Figure 3.19b.

3.5 Multi Robot Route Planner Extensions

In Section 3.4 the used SRRP is explained, which enables the MRRP to solve scenarios like the ones shown in Figure 3.3, but there are still cases, which lead to problems, when planning with multiple robots. Two such scenarios are shown in Figure 3.20. To solve these scenarios two extensions to the MRRP are added explained in Sections 3.5.1 and 3.5.2.



(a) A test case where priority matters. If robot r_0 has higher priority than r_1 the MRRP fail because r_0 blocks r_1's route candidate. If r_1 has higher priority it will find a route candidate to its goal and r_0 has to adapt to r_1's route candidate.

(b) A scenario, where priority scheduling fails because no robot is fast enough to move into the spot on the top to avoid the other one. (Assumed both have the same maximum speed)

Figure 3.20: Problem sets, where the SRRP fail.

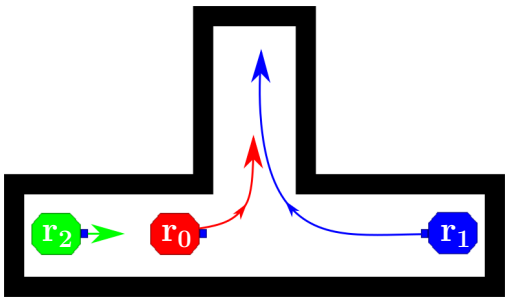


Figure 3.21: In this scenario it is unnecessary to exchange r_2's priority with any other robot

3.5.1 Priority Rescheduling

The scenario, shown in Figure 3.20a shows two robots, where robot r_0 has higher priority than r_1. Therefore, r_0 can plan its route candidate ignoring r_1's route candidate, which leads to the situation, that r_0 has planned its route candidate onto its goal, but blocks r_1's route candidate. If r_1 would have higher priority than r_0, the planner can plan the route candidate of r_1 before the route candidate of r_0 and the collision from the first priority scheme would not exist. Therefore, r_1 plans its route candidate in a straight line and r_0 is forced to wait at the crossing until r_1 has passed, which can be solved by the SRRP.
In the papers [BBT01] and [BBT02] an approach for solving such scenarios is explained, by trying to find valid priority schemes for decoupled planning. Therefore, it is proposed to assign all robots priorities randomly in the first iteration of the algorithm. After receiving the first result from the SRRP, two robots priorities are exchanged randomly and the SRRP is triggered again until a solution is found. This means if there is a priority scheme, which solves the given scenario this strategy will find it eventually. Furthermore, the algorithm can be run further after finding a successful priority scheme until all schemes are tested or until a custom defined number of iterations to improve the overall route length of the routing table.

Algorithm 3.8: Get Priority Schedule

Input: already tested schedules V_l, pointer to RouteCoordinator P_c, failed robot r_i
Output: new schedule s_p or NULL

```
1  if r_i == NULL then
2  |  r_i = P_c.GetRobotWithMostOveralCollisions ()
3  end
4  sorted list of robots R_l = P_c.GetRobotListSortedByNrOfCollision (r_i)
5  foreach robot r in R_l do
6  |  current schedule s_p = V_l.GetLastEntry ()
7  |  s_p = FlipRobots (s_p, r_i, r)
8  |  if (V_l not contains s_p) then
9  |  |  V_l.Add(s_p)
10 |  |  return s_p
11 |  end
12 end
13 return NULL
```

Because the algorithm exchanges priority schemes randomly, "inefficient priority exchanges" can happen. An example, therefore, is shown in Figure 3.21. In this scenario, robot r_0 has the highest priority followed by r_1 and r_2. This assignment lets the SRRP fail planning all route candidates because r_0 blocks r_1's route candidate. Thus, it is not useful to exchange r_2's priority with any other robot. To avoid such "inefficient priority exchanges" one can count the number of unresolvable collisions between two robots and

decide on this number, which priorities to exchange. Therefore, the SRRP has to count all collision resolution calls, which are answered with an empty list of vertices. If no routing table is found from the SRRP, the MRRP can exchange the two priorities with the most unresolved collisions. Thus r_2 from the scenario in Figure 3.21, is ignored when the MRRP decides to exchange priorities. If there is no such collision the MRRP can take the number of collisions with a resolution into account, to find the route candidate with the most conflicts. If there is no collision of any type the route candidate length is optimal if a routing table is found or there is no solution at all to this problem. The reason, therefore, is that if there are no collisions between robots, every robot can be planned using a single robot planner, which will either succeed for all robots on its shortest route candidate or fail for at least one.
The algorithm used for priority rescheduling is shown in Algorithm 3.8. Therefore, three methods are assumed to be known:

- `List<Robots>` **`GetRobotListSortedByNrOfCollision`** `(robot)`
  This method returns a list sorted by unresolvable collisions. If two robots have the same amount of unresolvable collisions with r_i the one with more overall collisions is higher placed in the list. If a robot has no collision of any type it is not part of this list.

- `robots` **`GetRobotWithMostOveralCollisions`** `()`
  This method returns the robot, which has the most overall collisions. This would be the robot with the greatest sum of *GetRobotListSortedByNrOfCollision*.

- `schedule` **`FlipRobots`** `(schedule, robot, robot)`
  Exchanges two robots in a priority schedule.

3.5.2 Speed Rescheduling

Priority scheduling solves problems like the one from Figure 3.20a, but scenarios exist, where two robots have the same distance to a crossing like that shown in Figure 3.20b. Obviously, the solution to this scenario is to let one of both robots avoid the other robot at the crossing. But in this scenario where both robots have the same distance to the crossing the AVRCA will fail because independent of the priority scheme the lower prioritized robot is too slow to move into its waiting spot.
To find a solution for such a problem, the higher prioritized robot has to wait a short period of time for the lower prioritized one. This can be done through decreasing the higher prioritized robots speed. Assume robot r_0 has the highest priority but half of its maximum speed. What happens is that r_0's route candidate needs twice as much time to execute, but thereby r_1 has enough time to use the AVRCA for avoiding r_0, which solves the scenario.
When using this approach two issues arise. The first issue is, that the time calculated for a route candidate cannot be generated out of the potential field directly because of robots with reduced speed using this speed for the planning process of the whole route candidate. This issue can be resolved by following the route candidates for all robots simultaneously and assigning the least potential value valid to a vertex or synchronize robots independent of their maximum speed, which is explained in 4.2.1.

Algorithm 3.9: Get Speed Schedule

Input: already tested schedules per robot V_l, pointer to RouteCoordinator P_c, failed robot r_i
Output: new schedule s_p or NULL

```
1  if r_i == NULL then
2  |  return NULL
3  end
4  sorted list of robots R_l = P_c.GetRobotListSortedByNrOfCollision (r_i)
5  foreach robot r in R_l do
6  |  current schedule s_p = V_l.GetLastEntry ()
7  |  s_p = DecreaseSpeed (s_p, r, δ_v)
8  |  if IsScheduleValid (s_p, V_l[r_i]) then
9  |  |  V_l.Add(s_p)
10 |  |  return s_p
11 |  end
12 end
13 return NULL
```

The second issue, for speed scheduling, is that it is not trivial to find a speed schedule, which leads to a valid solution. Thus the speeds of the robots have to be adapted if the SRRP fails to find a plan, similar to the priority rescheduling strategy. Like used for the priority rescheduling strategy collisions between robots are counted, while assigning

the search graph with priorities. If a routing table fails, the robot, which has the most collisions with the robot for which the SRRP failed to find a plan, gets its maximum speed reduced at δv. Every robots route candidate with lower or equal priority to this robot is planned again. Furthermore, the number of times a robots maximum speed can be decreased has to be limited because if a robot is selected, which will not solve the problem, but has the most collisions with the failing robot, the planner will end up in an infinite loop. The algorithm used for speed rescheduling is shown in 3.9. Therefore, three methods are assumed to be known:

- `List<Robots>` **`GetRobotListSortedByNrOfCollision`** `(robot)`
  This method returns a list sorted by unresolvable collisions. If two robots have the same amount of unresolvable collisions with r_i the one with more overall collisions is higher placed in the list. If a robot has no collision of any type it is not part of this list.

- `bool` **`IsScheduleValid`** `(schedule, history of schedules)`
  Checks if a given Schedule is valid depending on the History for a given robot. For a robot r it is only allowed to decrease the speed n times for a collision with a specific robot r_i.

- `schedule` **`DecreaseSpeed`** `(schedule, robot, speed)`
  Decreases the robots speed in a given schedule.

CHAPTER 4

Implementation

This chapter describes the implementation of the MRRP including additional algorithms and the setup used for testing. The framework is implemented using ROS. ROS is a framework which provides a communication interface to simplify writing robot software. It is organized in nodes and topics, where a node represents a program, which is able to communicate with other ones over topics. A topic is a communication channel, which allows a node to publish messages over a TCP-connection. Other nodes can subscribe to these topics and receive the published messages. Furthermore, ROS presents multiple tools for visualization, maintenance and debugging.
In order to cover the implementation this chapter is structured in four parts:

- **Graph Generation**
  Containing a Voronoi Path Generator (4.1.1), a Path Segmentation node (4.1.2), to generate a graph out of a given roadmap and a path saver/server node, to save and publish precomputed maps
- **Path Planning**
  Containing the MRRP, which implements the scientific contribution explained in Chapter 3. However, some of the additional helper functions used are explained in this chapter to complete the MRRP.
- **Robot Controller**
  Containing a robot motion controller and a state observer to synchronize robots.
- **Test Environment**
  Implemented in Robot Operationg System (ROS), including Stage and RVIZ to simulate the robots. Stage is a simulation software, which can be used with ROS to simulate mobile robots and RVIZ is a visualization tool, which presents an interface for publishing and visualizing topics.

In Figure 4.1 the test environment of the MRRP is shown, containing the topics used for communication between the nodes. The topics contain the following data:

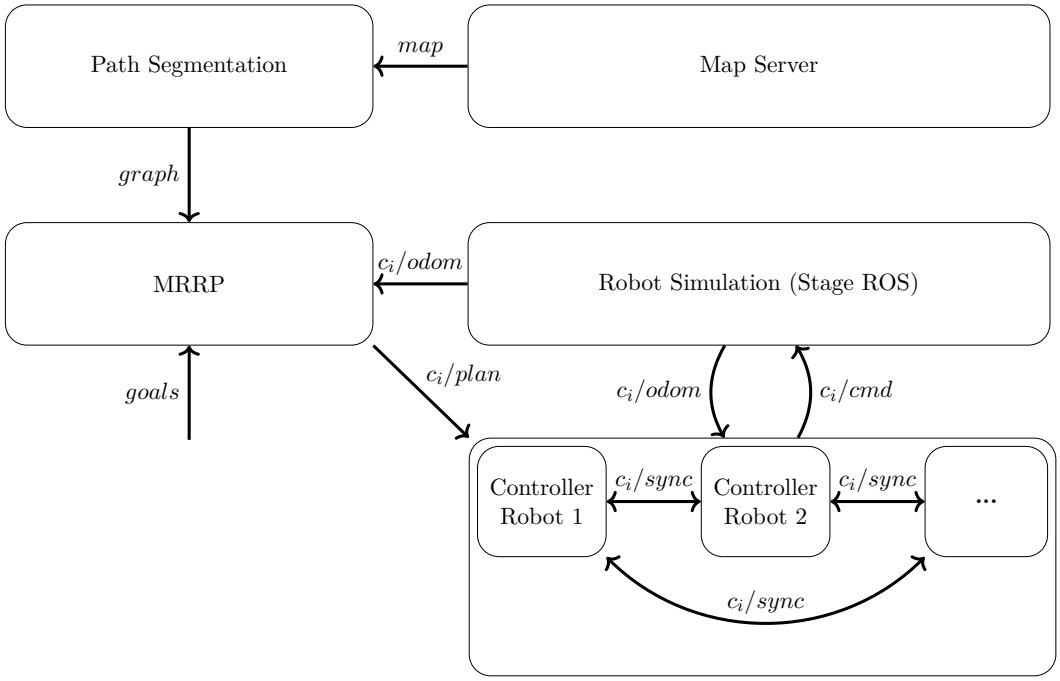


Figure 4.1: The test environment consisting of Path Server, Path Segmentation, the MRRP, simulation environment and multiple controllers (c_i) for each vehicle.

- **map** *(grid_map_msgs/GridMap)*
  Contains a distance transform and a roadmap represented as pixel map.

- **graph** *(voronoi_segmentation/VoronoiGraph)*
  Contains the used search graph by the MRRP

- **odom** *(nav_msgs/Odometry)*
  Contains the odometry data of a robot $(x, y, z, q_x, q_y, q_z, q_w)$

- **plan** *(tuw_multi_robot_utils/SegmentPath)*
  the routing table containing a route to follow for each robot represented as segments with start, end and preconditions for synchronization

- **sync** *(tuw_multi_robot_utils/PathPrecondition)*
  Contains the synchronization messages for each robot to guarantee a deadlock free execution of the plan

- **cmd** (geometry_msgs/Twist)
  Contains the control commands for a robot

4.1 Graph Generation

In this section, the generation of the graph used by the MRRP is explained.
Since our planner uses a search graph instead of a pixel map an algorithm to find a data abstraction from a pixel map has to be used.
The idea, therefore, is to use roadmaps described in [LaV06] to create a data abstraction. An approach, hereby, is shown in [WWW16], which describes voronoi paths used for planning. The benefit of voronoi paths is, that they describe a fully connected path with line-of-sight contact to every pixel on the map. Furthermore, these lines describe the path with the maximum distance to obstacles at every point. Section 4.1.1 describes the creation of such paths.
To build a usable graph for the MRRP the voronoi paths have to be converted into a voronoi graph containing multiple segments of a specific length. This process is described in Section 4.1.2.
To improve the execution time of the MRRP, these nodes are used to precompute the maps. However, they can be used at execution time as well, to be prepared for new or changing environments.

4.1.1 Voronoi Path Generator

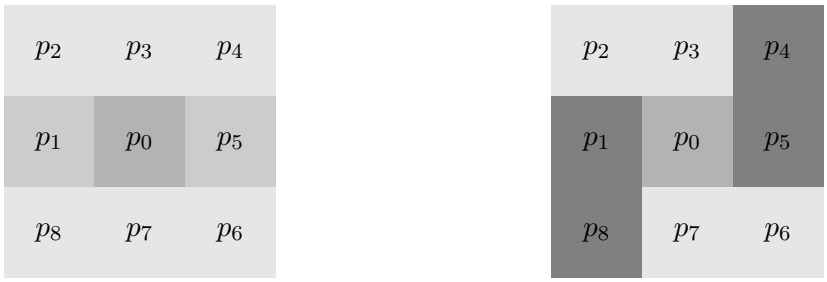
To generate the mentioned voronoi paths, two steps are typically needed. The first one is to create a distance transform map, which shows the minimum distance to all obstacles for every pixel of a map. The second one is to thin the distance transform map, computing the pixels following the ridge of the distance field.

Distance Transform Algorithm

Basically Distance Transform algorithms are scanning all pixels multiple times and setting the minimal distance for everyone. An algorithm for such an operation is proposed in [Bor86]. This algorithm is implemented in the OpenCV library and is used by the MRRP. After the transformation we have a map with all foreground pixels marked with zero and all other pixels marked with the distance to the closest foreground pixel.

Thinning Algorithm

This distance map can be used to generate the voronoi path. Therefore, a thinning algorithm is used. A very popular one is the one proposed in [ZS84]. However, there are images, where this algorithm does not find the maximum of the distance field depending on the distance transform algorithm used. Thus another approach is used for the thinning algorithm. This algorithm can be found in [NAU06], which computes all start-, maximum-, saddle-, and double-saddle points of a distance field (shown in Figure 4.2). Afterwards, it expands from every found point to the next higher one until an already expanded point is found.
There are some cases in which the algorithm finds a two-pixel thick path, which is not intended for the voronoi path. An example case is shown in Figure 4.3. These problems



(a) A maximum point example. Therefore all neighbor pixels have to be smaller-equal to the center point

(b) A saddle point example. Therefore the upper and lower neighbors are smaller and the left and right ones are higher.

Figure 4.2: Examples for maximum- and saddle-points. (Lighter colors describing lower and darker ones higher distance values.)

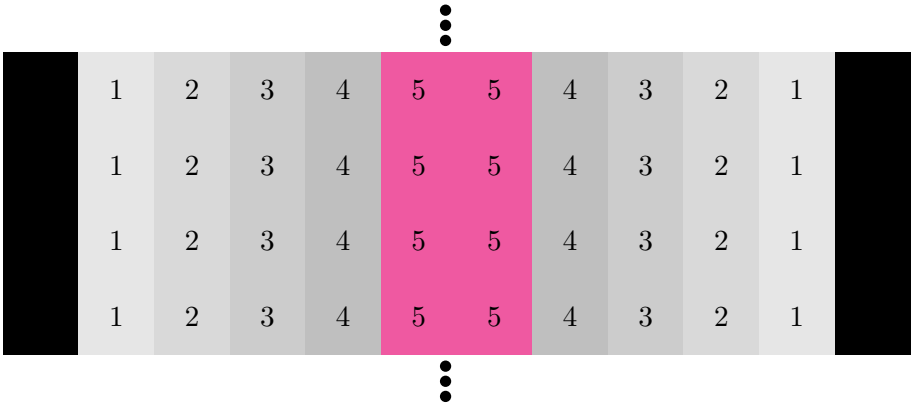


Figure 4.3: Shown a case where the final image of the algorithm from [NAU06] is more than one pixel thick

appear if there are corridors with an even number of pixels in width. To avoid such paths with two pixels thickness the Thinning algorithm from [ZS84] is used to thin the path, even more. Therefore, this algorithm finds a path on the ridge of the distance field, because the original structure was already on the ridge before. However, this algorithm normally needs several iterations to find a solution. In this case, it does not need more than two iterations to find the solution, because the original structure is not thicker than two pixels.
These two generated maps are given to the path segmentation to generate the final graph vertices used by the MRRP.

4.1.2 Segmentation

To generate the graph out of the given path and the distance field an algorithm is used to find crossings in the given data structure. To visualize the algorithm a crossing in a given pixel map is shown as example in Figure 4.4. The segmentation algorithm is shown in Algorithm 4.1

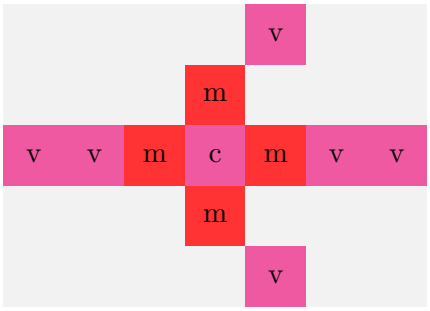


Figure 4.4: A crossing where many pixels have more than two neighbors. Shown the voronoi path (magenta; v) and the margin pixels (red; m) neighboring a non-margin pixel (less than two neighbors)

Algorithm 4.1: Path Segmentation

Input: pixel map containing the path m_p, maximum segment length l_s
Output: list of segments representing the graph G

```
1  list of crossings c_l
2  foreach non marked pixel p in m_p do
3      if p is not background pixel then
4          if GetPathNeighbours (p) > 2 then
5              crossing c = ExpandCrossing (p)
6              c_l.Add (c)
7          end
8      end
9  end
10 foreach crossing c in c_l do
11     foreach non marked margin pixel p_m in c do
12         list of points P_s = ExpandPath (p_m)
13         list of segments S = GetSegmentListWithMaximumLength (P_s, l_s)
14         c.AddToCrossingAndUpdateNeighbours (S)
15     end
16 end
```

Figure 4.5: Shown in magenta the path (v); in red the margin pixels (m) of the crossing; in green the non-crossing pixels to expand; the Dijkstra expander starts in p_1 and expands until a margin pixel is found, which is not p_1; the algorithm will end up in p_2 and save the segment from $(p_1 \rightarrow p_2)$

It is known from the given path, that it has a maximum thickness of one pixel. This means, that every non-crossing pixel, also named path pixels, has less or equal than two neighbors. In Figure 4.4 these non-crossing pixels are marked with v. This means also that all crossing pixels marked with m and c have to have more than two neighbors.
The used algorithm iterates over all pixels in the map and checks them for the number of neighbors, which are non-background pixels. If such a pixel is found a Dijkstra expansion is used to find all linked crossing pixels with no path pixel in between. In Figure 4.4 this would be all pixels marked with m and c.
After finding the set of these pixels, all pixels having a neighbor, are marked as margin pixels, marked with m in Figure 4.4. The center (c) of this crossing is the mean of the x and y values of all margin pixels. After marking a crossing the algorithm continues iterating over the left pixels of the map.

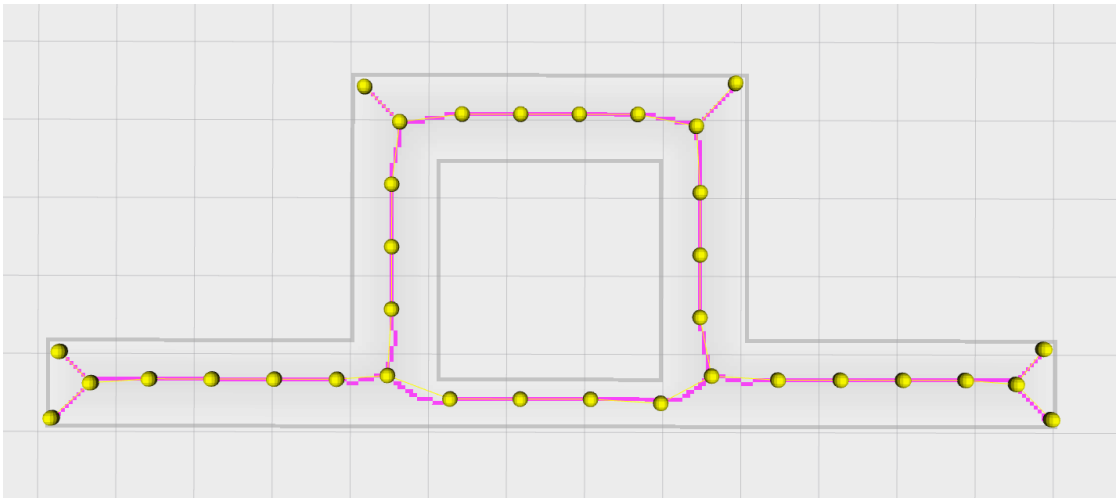


Figure 4.6: Shown the segmentation of a path in yellow and in magenta the given path

After detecting all crossings and their contained margin points, a Dijkstra expansion can

be done to find the segments of the Graph, by expanding between the margin pixels. Therefore the Dijkstra algorithm is only allowed to expand on path pixels to force it, to expand away from a crossing. The algorithm terminates if another margin pixel is found, which is not the starting one. Since the path is only one pixel thick and all crossings are detected, there is only one margin pixel the expander can find and therefore, the segment is unique. The segment expansion is shown in Figure 4.5.
To get a graph using only segments not exceeding a maximum length the found segment can be split into multiple ones, where all segments have the same length. These found segments are saved in the segment data structure, which contains *length*, *width*, a list of *predecessors* and a list of *successors*. The minimum distance of such a segment is found by taking the minimum value in the distance map of all pixels contained in a segment. The length of the segment is calculated by using the pixels coordinates. For setting the neighbors all found segments are linked to crossings and updated with every new segment linked to the crossing.
An example for a generated graph can be found in Figure 4.6, where the voronoi path is shown in magenta and the graph segments in yellow.

4.2 Route Planning

In this section, the implementation of the MRRP is explained. This algorithm is split into three parts:
1. **Calculate Route Candidates** (Algorithm 4.2),
receiving start and goal Segments as well as a speed and priority schedule and returning the found list of route candidates containing all assigned potentials.

Algorithm 4.2: Calculate Route candidates

Input: search graph G; start vertices v_s; goal vertices v_g; speed Schedule S_s; priority Schedule S_p
Output: list of robot route candidates P_r or NULL

```
1  foreach Robot r_i do
2  |  graph with potentials G_p = CalculatePotentials (S_p[r_i], S_s[r_i], v_s[r_i], v_g[r_i])
3  |  if G_p == NULL then
4  |  |  return NULL;
5  |  end
6  |  route P_i = Traceback (G_p, v_s[r_i], v_g[r_i])
7  |  if P_i == NULL then
8  |  |  return NULL;
9  |  end
10 |  P_r[i] = P_i
11 end
12 return P_r
```

2. **Calculate Schedules** (Algorithm 4.3),
receiving start and goal Segments and returning the found list of route candidates containing all assigned potentials.

Algorithm 4.3: Calculate Schedules

Input: search graph G; start vertices v_s; goal vertices v_g
Output: list of robot route candidates P_r or NULL

```
1 while Priority Scheduler.GetSchedule (out Priority Schedule S_p) do
2 |  while Speed Scheduler.GetSchedule (out Speed Schedule S_s) do
3 |  |  P_r = CalculatePaths (G, S_s, S_p, v_s, v_g)
4 |  |  if P_r != NULL then
5 |  |  |  return P_r
6 |  |  end
7 |  end
8 end
9 return NULL
```

3. **Create Route candidate** (Algorithm 4.4),
receiving start and goal positions and returning a synchronized routing table to all robots. While the algorithms for Calculate route candidates and Calculate Schedules

Algorithm 4.4: Create Route candidate

Input: search graph G; start positions p_s; goal positions p_g
Output: route P_l or NULL

```
1  start vertices v_s= FindPoseSegments (p_s)
2  if v_s == NULL then
3  |  return NULL
4  end
5  goal vertices v_g = FindPoseSegments (p_g)
6  if v_g == NULL then
7  |  return NULL
8  end
9
10 P_r = CalculateSchedules (G, s_s, s_g)
11 if P_r == NULL then
12 |  return NULL
13 end
14
15 P_l = GeneratePathWithPreconditions (P_r)
16 return P_l
```

are described in Chapter 3, the algorithm Create Route candidate is described in this chapter. To these ends, the **FindPoseSegments** algorithm is described in Section "Find Pose Vertices" and the **GeneratePathWithPreconditions** algorithm is described in 4.2.1.

Find Pose Vertices

Since start and goal positions are published as odometry message, but the MRRP works on a graph, the published positions have to be assigned to the closest segments. Because the used search graph matches the given map, one can iterate over all segments and check if the distance to the center line of the segment is smaller than the width of it. The first segment found in this way is used as start/goal position. After a route candidates is found an additional segment is added, which connects the last segment of the route candidates with the real goal position.

4.2.1 Route Generator (RG)

Since the routing table generated by the MRRP is time-dependent, a robot cannot execute this routes without any information from other robots. For example in the scenario shown in Figure 3.3b, the blue robot has to wait until the red robot has passed the crossing. The MRRP includes, therefore, two approaches to synchronize the robots.

- A Velocity Profile, which has to be executed exactly
- Segment Constraints, which constrains robots to wait for others on a segment

Velocity Profile

Algorithm 4.5: MovePath

Input: list of routes P, Route Coordinator P_C, actual robot r_i, step nr n_s
Output: list of routes P

```
1  actual route p = P[r_i]
2  if n_s < 0 then
3  |  n_s = p.size()
4  end
5  last time t_l=0
6  for i:0...n_s do
7  |  t_l = t_l + p[i].GetLength () / r_i.GetSpeed ()
8  |  foreach robot r in P_C.GetEarlierRobotsOnVertex (p[i], r_i) do
9  |  |  step nr on other route n_r = GetPathStep (r, p[i]) + 1
10 |  |  MovePath (P, P_C, r, n_r)
11 |  |  t_l = Max (t_l, P[r][n_r])
12 |  end
13 |  p[i].SetTime (t_l)
14 end
15 return P
```

To synchronize all robots a velocity profile can be generated, which has to be exactly executed from each robot to guarantee a deadlock-free movement. This has the benefit, that no communication is needed, but problems arise if the model of the robot is not exactly known or if the robot moves in dynamic environments.
However, such a Velocity Profile is generated by the MRRP by simulating the routes execution. Therefore, the Velocity Profile Generator (VPG) iterates over a robot's route candidate, until he finds a segment S_c, which passes another robot earlier than the current active robot. This can be seen by comparing the assigned potentials to the route candidate. If such a route candidate is found the VPG switches to this robot and executes this route candidate until another segment S'_c is found or the robot reaches the segment S_c. Therefore, the algorithm to find a Velocity profile for one robots route is shown in

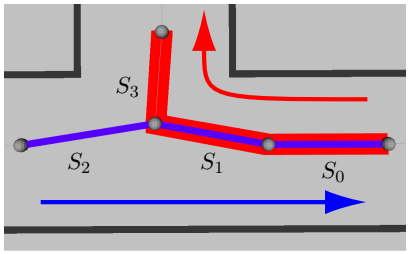
Algorithm 4.5. This algorithm is executed for each route in the list of routes P. In this algorithm, two other algorithms are used:

- `list<robot>` **`GetEarlierRobotsOnVertex`**`(vertex v, robot r)`
  The Route Coordinator is aware of all routes and presents a call, which returns all robot passing a specific segment earlier than a given robot.
- `int` **`GetPathStep`**`(robot r, vertex v)`
  This algorithm returns the step number of a segment contained in a robot's route.

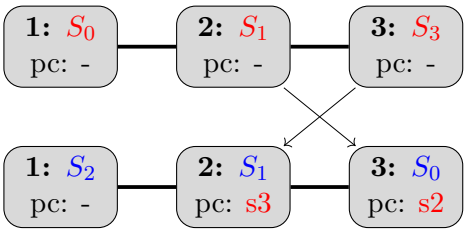
These velocities are only set for single segments and probably changes for each segment. To get a smooth velocity profile, the VPG iterates over each found velocity profile selecting each segment up to the first one, where a robot has to wait for another one. For this set of segments, its average speed is applied to each segment. After finding a set of segments the VPG starts collecting the next set until it has reached the end of the route.

Segment Constraints

To give a robot more freedom to move, one can save constraints for every segment, which has to be satisfied before a robot is allowed to enter the segment. This has the benefit, that every robot can select the velocity appropriate to its environment without risking a deadlock. To execute these segments with constraints every robot has to be connected to each other robot to read the actual robot state.
To these ends, every Segment in each route is assigned with constraints containing the robot-id, of the constraining robot R_c and a step-id, which has to be reached from R_c. This id is the number of segments the robot R_c has to have passed in its route before a robot can pass this segment. In Figure 4.7 two routes and the segment constraints can be seen. For each route only constraints of earlier robots are taken into account. To execute such a route, every robot publishes the current number of segments executed. A robot which needs a clearance for a segment checks the current executed segment number of the constraining robot to know if the segment is released.



(a) In this scenario, the blue robot has to wait for the red one. Furthermore, the segments of the route are shown. The red robot wants to follow the red route starting from the right. The blue robot wants to follow the blue line starting from the left.



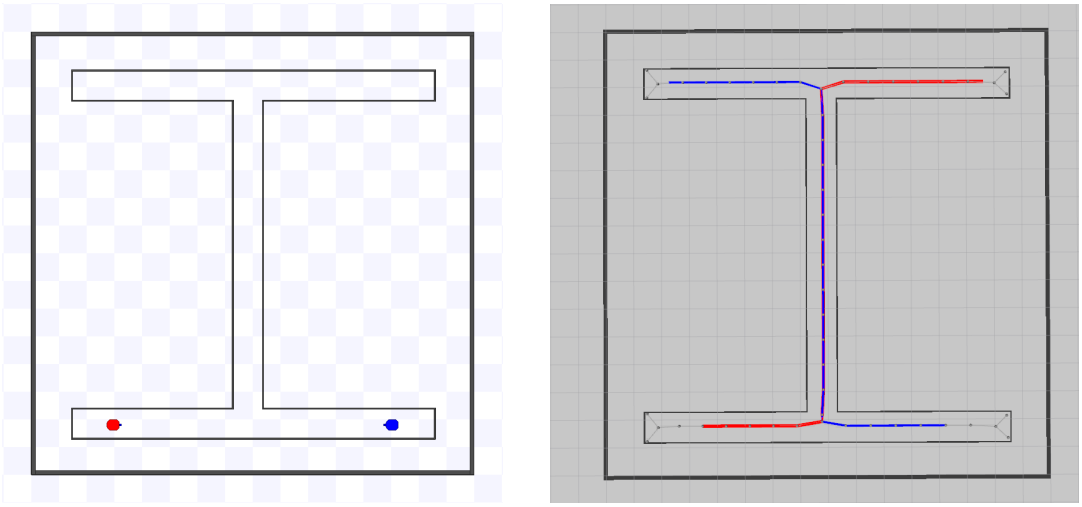
(b) Above the route for the red robot and below the route for the blue one. The arrows are marking the constraints for the blue robot. Obviously, one could remove the precondition for S_0, because the blue robot cant reach S_0 before the red one has reached step three.

Figure 4.7: Example for segment constraints

4.3 Robot Controller

For the simulation a robot controller is created, which takes a routing table as described in Section Segment Constraints as input. Every routing table consists of multiple segments, with preconditions. The robot controller subscribes to every robots sync message and to the current robots odometry message.
The controller continuously iterates over all other robots sync states to determine if it is allowed to move the robot to the next segment in the given routing table. If the robot is allowed to move to its next segment, a PID-control is utilized to calculate linear and angular command messages and publish them to the robot's motor driver. After receiving the targeted segment, its own sync message is increased by one step.
However, since standard robot controllers normally are not able to receive such route messages with preconditions a robot synchronizer is created, which receives a synced route from the MRRP and publishes a valid route to the given robot controller. This route starts from the current position of the robot and ends in the last segment, where the robot is allowed to move to, depending on the other robots sync messages. A state observer, receives the current robots odometry pose and updates with it the current sync position of the route, to allow other robots to move.

4.4 Test Environment



(a) The robot simulation environment Stage (b) The ROS-visualization tool (RVIZ)

Figure 4.8: The test environment

For simulating multiple robots the robot simulation **Stage** is used. Stage (Figure 4.8a) is a 2D simulation environment for multiple vehicles, which allows users to integrate sensors and actors like laser range-finders, grippers and bumpers. Therefore, a configuration file has to be written, which defines the shape, the sensors and the motion model of the vehicles. Furthermore, the environment can be configured by using a (black-white) image. For the integration in ROS, stage publishes odometry data and sensor readings and subscribes to a command topic for each robot. The Path Planner uses the odometry data to determine the robots start position for planning. The controller uses the command topic to move the robot, and the subscribed topics to determine the position.
To visualize the output of the MRRP and the test environment, the **Ros-visualization tool (RVIZ)** is used, shown in Figure 4.8b). This tool presents an interface for visualizing and publishing ROS topics.

CHAPTER 5

Results

This chapter shows, different tests with the MRRP. All tests are done in a Simulation environment called Stage. This has the advantage, that more tests can be done in a shorter time period compared to a real world test environment. Because the MRRP has no feedback loop from a robot, which can influence the routing task, testing the planner in real world would not have any benefit compared to the simulation.
In Section 5.1 a number of special scenarios with two to three robots are tested, to show which scenarios the planner can solve and when the planner fails to solve a scenario. The used environment for these tests is shown in 5.1.
In Section 5.2, the MRRP is tested in different environments, with different configurations, to compare the included modules to each other.

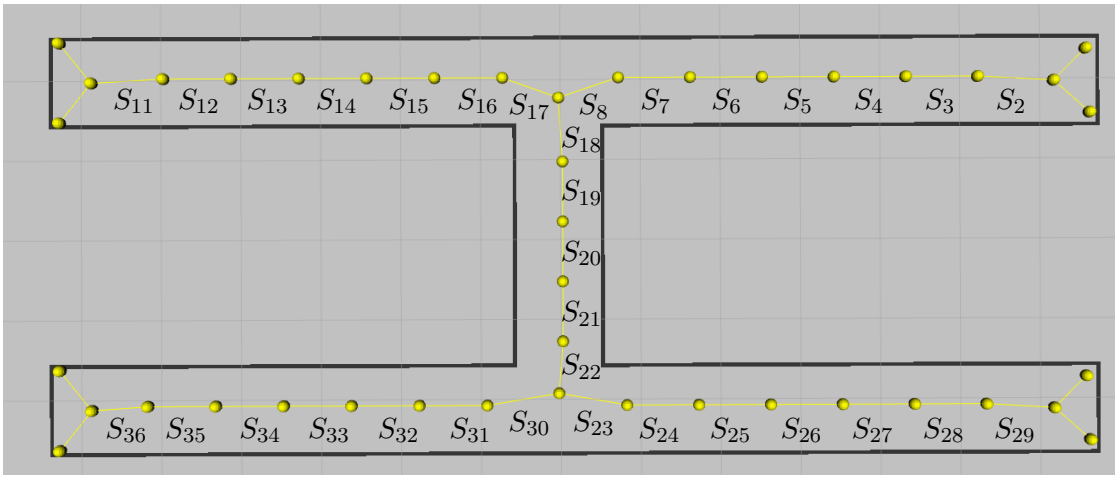
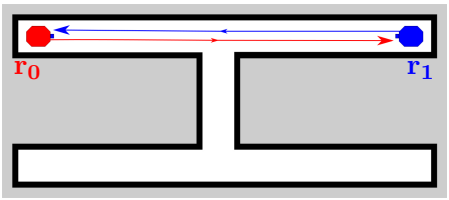


Figure 5.1: The test environment, with all generated graph nodes (vertices) shown in yellow and marked with S_x.

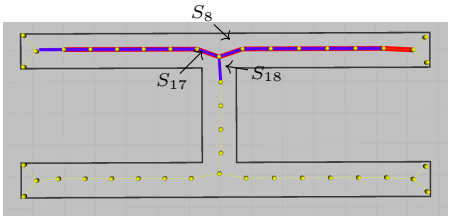
5.1 Experiments

For testing the MRRP the environment shown in 5.1 is used. To illustrate the function of the MRRP four test cases with up to three robots are used.

5.1.1 Switch



(a) A scenario, where the red and the blue robot want to switch places. The red robot wants to follow the red arrow to the right and the blue one wants to move to the red ones start position



(b) The solution to the scenario from Figure 5.3a, where the blue robot waits for the red one at the crossing.

robot zero (r_0)	robot one (r_1)
0 : **S11** [-]	0 : **S2** [-]
1 : **S12** [-]	1 : **S3** [-]
2 : **S13** [-]	2 : **S4** [-]
3 : **S14** [-]	3 : **S5** [-]
4 : **S15** [-]	4 : **S6** [-]
5 : **S16** [-]	5 : **S7** [-]
	6 : **S8** [-]
	7 : **S18** [-]
6 : **S17** [r_1 past step 7]	8 : **S19** [-]
7 : **S8** [r_1 past step 7]	9 : **S18** [r_0 past step 7]
8 : **S7** [r_1 past step 5]	10: **S17** [r_0 past step 7]
9 : **S6** [r_1 past step 4]	11: **S16** [r_0 past step 5]
10: **S5** [r_1 past step 3]	12: **S15** [r_0 past step 4]
11: **S4** [r_1 past step 2]	13: **S14** [r_0 past step 3]
12: **S3** [r_1 past step 1]	14: **S13** [r_0 past step 2]
13: **S2** [r_1 past step 0]	15: **S12** [r_0 past step 1]
	16: **S11** [r_0 past step 0]

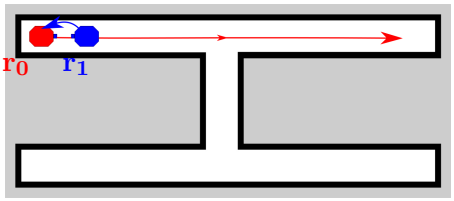
(c) The routing table found by the MRRP for both robots including segment order and preconditions.

Figure 5.2: A test scenario for the MRRP

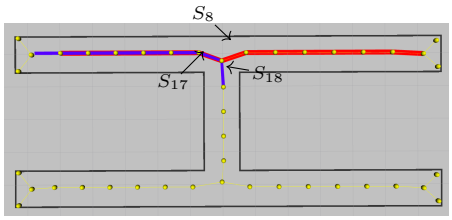
In Figure 5.2, a basic test case from Figure 3.3 is shown, where the red and the blue robot want to switch places. For solving this scenario the SRRP fails in the first attempt and the speed rescheduler decreases the speed for the robot with priority one which is, in this case, the red robot. Afterwards, the SRRP finds a solution by letting the blue robot wait at the crossing for the red one and generates the routes from Figure 5.2c. As one can see the red robot executes its route until step five and waits afterwards for the blue robot until it has reached step seven, which is the save spot at the crossing to wait for the red robots step seven. In step six and seven of the red robot, the precondition states to wait for the blue robots step seven in both steps. The reason, therefore, is that the crossing segments S_8, S_{17} and S_{18} are overlapping and have to be locked simultaneously.

5.1.2 Push

Another test case from Figure 3.3 is shown in Figure 5.3. There, the red robot has the highest priority and wants to move to the right. The blue robot has to avoid the red one by moving back to the crossing and let it pass. The result of the MRRP is shown in 5.3c. As one can see the red robot has constraints to wait until the blue robot has vanished from its route from segment S_{12} to S_8. Afterwards, the blue robot has to wait until the red one has left its future route to move back (segments S_{18} to S_{16}). Again, segments S_8, S_{17} and S_{18} are locked simultaneously because they are overlapping.
Since these test cases should work by design it is not surprising that they are valid, but it is getting more interesting when testing with multiple robots.



(a) A scenario, where the red and the blue robot want to switch places.



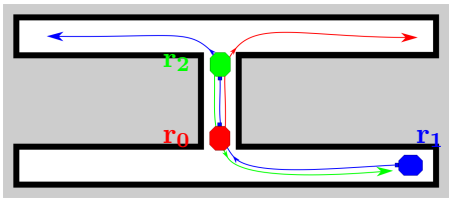
(b) The solution to the scenario from Figure 5.3a, where the blue robot waits for the red one at the crossing.

robot zero (r_0)	robot one (r_1)
0 : **S11** [-]	0 : **S12** [-]
1 : **S12** [r_1 past step 0]	1 : **S13** [-]
2 : **S13** [r_1 past step 1]	2 : **S14** [-]
3 : **S14** [r_1 past step 2]	3 : **S15** [-]
4 : **S15** [r_1 past step 3]	4 : **S16** [-]
5 : **S16** [r_1 past step 4]	5 : **S17** [-]
	6 : **S18** [-]
6 : **S17** [r_1 past step 6]	7 : **S19** [-]
7 : **S8** [r_1 past step 6]	
	8 : **S18** [r_0 past step 7]
8 : **S7** [-]	9 : **S17** [r_0 past step 7]
9 : **S6** [-]	10: **S16** [r_0 past step 5]
10: **S5** [-]	11: **S15** [r_0 past step 4]
11: **S4** [-]	12: **S14** [r_0 past step 3]
12: **S3** [-]	13: **S13** [r_0 past step 2]
13: **S2** [-]	14: **S12** [r_0 past step 1]
	16: **S11** [r_0 past step 0]

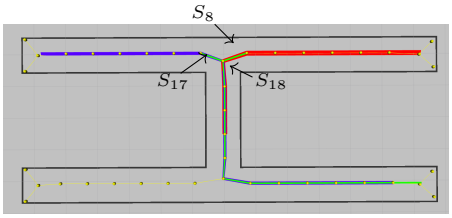
(c) The routing table found by the MRRP for both robots including segment order and preconditions.

Figure 5.3: A test scenario for the MRRP

5.1.3 Multiple robots



(a) A scenario with multiple robots creating multiple potential collisions for the SRRP.



(b) The solution to the scenario from Figure 5.4a.

robot zero (r_0)	robot one (r_1)	robot two (r_2)
0 : **S21** [-]	0 : **S29** [-]	0 : **S18** [-]
1 : **S20** [-]	1 : **S28** [-]	1 : **S17** [-]
2 : **S19** [-]	2 : **S27** [-]	2 : **S16** [-]
3 : **S18** [r_2 past step 1]	3 : **S26** [-]	
4 : **S8** [r_2 past step 1]	4 : **S25** [-]	
5 : **S7** [-]	5 : **S24** [-]	3 : **S17** [r_0 past step 4]
6 : **S6** [-]	6 : **S23** [-]	4 : **S8** [r_0 past step 4]
7 : **S5** [-]	7 : **S22** [-]	5 : **S7** [r_0 past step 5]
8 : **S4** [-]	8 : **S21** [r_0 past step 0]	
9 : **S3** [-]	9 : **S20** [r_0 past step 1]	
	10: **S19** [r_0 past step 2]	
	11: **S18** [r_0/r_2 past step 4/4]	
	12: **S17** [r_0/r_2 past step 4/4]	
	13: **S16** [r_2 past step 2]	6 : **S8** [r_0/r_1 past step 4/12]
	14: **S15** [-]	7 : **S18** [r_0/r_1 past step 4/12]
	15: **S14** [-]	8 : **S19** [r_0/r_1 past step 2/10]
	16: **S13** [-]	9 : **S20** [r_0/r_1 past step 1/9]
	17: **S12** [-]	10: **S21** [r_0/r_1 past step 0/8]
	18: **S11** [-]	11: **S22** [r_1 past step 7]
		12: **S23** [r_1 past step 7]
		13: **S24** [r_1 past step 5]
		14: **S25** [r_1 past step 4]
		15: **S26** [r_1 past step 3]
		16: **S27** [r_1 past step 2]
		17: **S28** [r_1 past step 1]
		18: **S29** [r_1 past step 0]

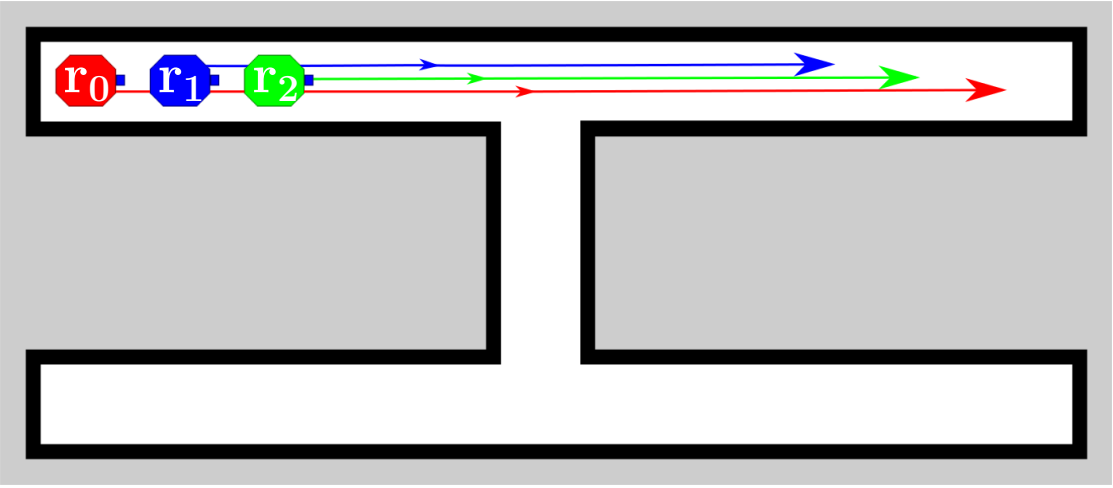
(c) The routing table found by the MRRP for all robots including segment order and preconditions.

Figure 5.4: A test scenario for the MRRP

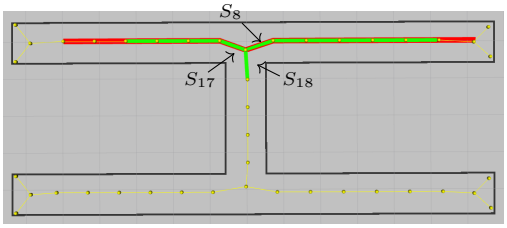
In Figure 5.4 a test case is shown, where three robots want to switch places. This

scenario is solvable as well using the MRRP, because enough spots for the robot to avoid each other are present. Here it can be seen, how the MRRP works using multiple resolution strategies. For planning the red and the blue robots routes the MRRP has no constraints because both routes are not overlapping in time and space simultaneously. When planning the green robot with the lowest priority multiple potential collisions occur, which have to be avoided. At first, the red robot pushes the green robot towards its start point to the next crossing. When the SRRP plans the route of the green robot back to its start point, a potential collision occur between the blue and the green one. Therefore, the SRRP backtracks to the last crossing and avoids the blue robot's route there. After avoiding both robots the green route can be planned back to the start and towards its goal. The MRRPs output to these routes is shown in Figure 5.4c.

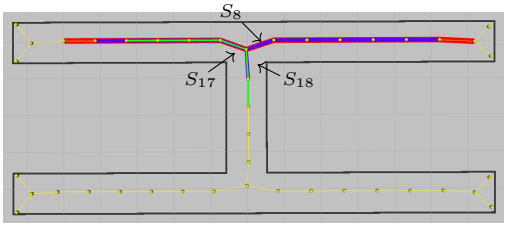
5.1.4 Limitations



(a) A scenario, where all priority and speed schedules fail with the MRRP



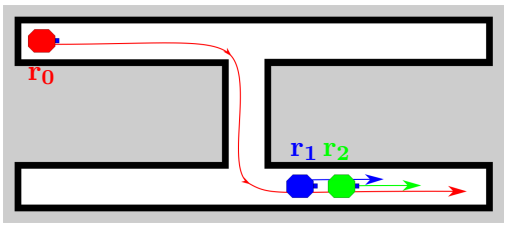
(b) The first attempt, where the red robot has the highest priority, the green one the second highest and the blue one the lowest. As one can see this fails after planning the green robots route.



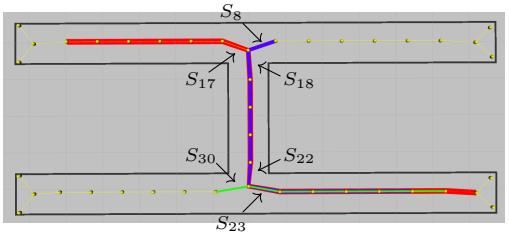
(c) Shown the second attempt, where the green robot has the lowest and the red one the highest priority. This schedule fails when planning the greens route because there is no valid route from the waiting spot at the crossing to its goal.

Figure 5.5: A scenario where the MRRP fails.

The test cases shown in Figures 5.2, 5.3, and 5.4 where all valid to illustrate the function of the MRRP. In Figure 5.5 a test case is shown, where the MRRP fails. In this scenario, the robots are starting as shown in 5.2a. As one can see, not every priority schedule will work for this test case. Therefore, the red robot has to have the highest priority because it wants to move from the most left to the most right point on the graph. For the remaining robot priorities, both orders can be tested. If the green robot gets the second highest priority it will avoid the red robot at the crossing in between its start and goal position. But when planning the blue robots route no spot is left to avoid the red robots route because the green robot blocks the crossing. This is shown in Figure 5.5b. Since this fails, one can try to exchange the blue and the green robot's priorities, shown in Figure 5.5c. Therefore, the blue one avoids the red robot's route at the crossing. The



(a) A scenario for the MRRP, where the blue and the green robot have to avoid the red one at the crossings.



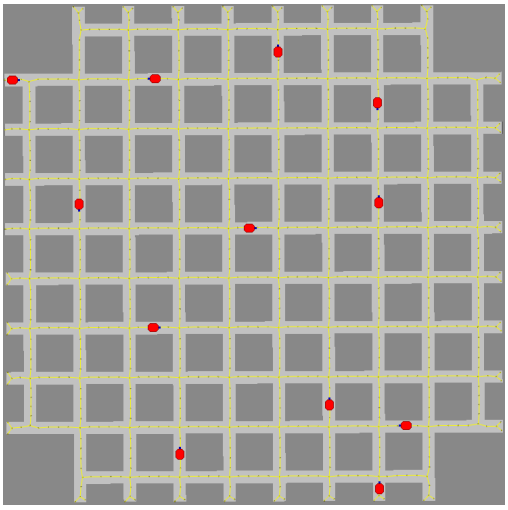
(b) The found solution to the scenario in 5.6a.

robot zero (r_0)	robot one (r_1)	robot two (r_2)
0 : **S11** [-]	0 : **S24** [-]	0 : **S25** [-]
1 : **S12** [-]	1 : **S23** [-]	1 : **S24** [r_1 past step 0]
2 : **S13** [-]	2 : **S22** [-]	
3 : **S14** [-]	3 : **S21** [-]	2 : **S23** [r_1 past step 2]
4 : **S15** [-]	4 : **S20** [-]	3 : **S30** [r_1 past step 2]
5 : **S16** [-]	5 : **S19** [-]	4 : **S31** [-]
	6 : **S18** [-]	
	7 : **S8** [-]	
6 : **S17** [r_1 past step 7]	8 : **S7** [-]	
7 : **S18** [r_1 past step 7]		
8 : **S19** [r_1 past step 5]	9 : **S8** [r_0 past step 7]	
9 : **S20** [r_1 past step 4]	10: **S18** [r_0 past step 7]	
10: **S21** [r_1 past step 3]	11: **S19** [r_0 past step 8]	
11: **S22** [r_1/r_2 past step 2/3]	12: **S20** [r_0 past step 9]	
12: **S23** [r_1/r_2 past step 2/3]	13: **S21** [r_0 past step 10]	
13: **S24** [r_1/r_2 past step 0/1]		5 : **S30** [r_0/r_1 past step 12/2]
14: **S25** [r_2 past step 0]		6 : **S23** [r_0/r_1 past step 12/2]
15: **S26** [-]	14: **S22** [r_0/r_2 past step 12/6]	7 : **S24** [r_0/r_1 past step 13/0]
16: **S27** [-]	15: **S23** [r_0/r_2 past step 12/6]	8 : **S25** [r_0 past step 14]
17: **S28** [-]	16: **S24** [r_0/r_2 past step 13/7]	9 : **S26** [r_0 past step 15]
18: **S29** [-]	17: **S25** [r_0/r_2 past step 14/8]	10: **S27** [r_0 past step 16]
	18: **S26** [r_0/r_2 past step 15/9]	11 : **S28** [r_0 past step 17]
	19: **S27** [r_0/r_2 past step 16/10]	

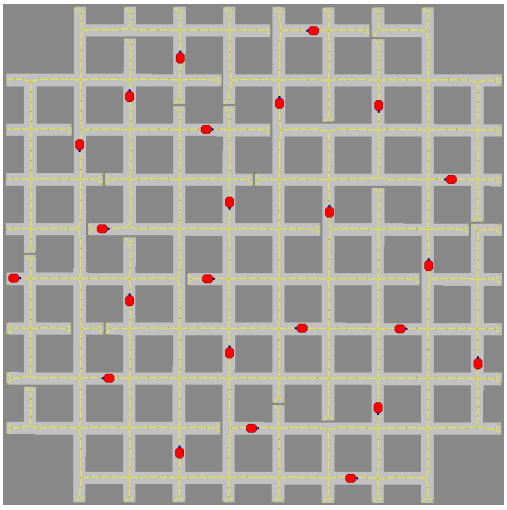
(c) The routing table found by the MRRP for all robots including segment order and predictions.

Figure 5.6: A test scenario for the MRRP

route of the green robot has to avoid the red and the blue robots route. In the first step, this is possible because the green robot starts in front of both other robots and can avoid the red and the blue one, one segment after the crossing (S_{19} in Figure 5.1). But after this point, no option for the green robot is left because the blue robot blocks its route to the goal and therefore, the routing table fails again.
This leads to an assumption about the planner, which states that the planner can fail if there are less than $n-1$ crossings on the route of the involved robot with the highest priority. Where n is the number of robots involved in the conflict and a crossing counts as n_b-2 crossings, where n_b is the number of branches. In this case, the blue and the green robot have one crossing in between the blues start and goal, but since the red robot is also involved there have to be three crossings in on the red robot's route.
On the counter side this means if there are more than $n-1$ crossings on the route of the robot with the highest priority, the planner will find a solution. For example if the goal positions of the robots would be in one of the lower two hallways of the environment (e.g.: S_{27}, S_{28}, S_{29} in Figure 5.1) this planner finds a solution because there are two crossing in between the red robot's route, which allows every lower prioritized robot to avoid the red robot's route.
In Figure 5.6a a scenario is shown, which depicts how the MRRP solves scenarios, where conflicting robots are in between the last crossing of the highest prioritized route and its goal. At first the SRRP plans the red robot's route. Afterwards, the MRRP expands towards the blue robot's goal, which detects a potential collision at its goal and tracks back to the start. Because at the blue robot's start position still a potential collision exists the avoid robot at start algorithm tries to find a spot to avoid the red robot and finds it at one of the two crossings in the environment. Finally the SRRP plans a route for the green robot, which works similar to the blue robot, by waiting at a free crossing. The resulting routes for this routing table are shown in Figures 5.6c and 5.6b.



(a) The test environments for the MRRP



(b) The second test environments for the MRRP, which is a basically the environment shown in 5.7a with randomly added walls.

Figure 5.7: The used environments for testing the MRRP

5.2 Comparison to other planners

To verify the gain of the MRRP compared to other planners, multiple tests are done by randomly selecting start and goal positions in a given map for varying numbers of robots. These randomly generated positions are saved and executed for every planner in the same order. The two environments shown in Figures 5.7a and 5.7b, are used for testing the MRRP. This test environments are kept similar to be able to use the same start and goal positions on both environments as well.

5.2.1 Prioritized Planning

In the first step, the modules of the SRRP are tested and compared to each other, by using three different configurations of the planner without speed and priority scheduling. The first configuration is the planner without any collision resolution strategy, only taking time into account. The second configuration is the simple planner including the BTA and the third one is the full SRRP including the BTA, the AVRA and the PRA.
Furthermore, the single robot planner combined with priority rescheduling is compared to the MRRP and the SRRP used in the first test run. All tests are executed with four, eight fourteen, eighteen and twenty-four robots. For all test runs the average of the longest route length, the average planning time and the percentage of solved test cases are compared. For testing the modules of the SRRP only the percentage of solved scenarios are shown in Figure 5.8, because all other stats are only changing marginally. For the first set of test runs one can see in Figure 5.8, that allowing the planner to use waiting steps increases the number of solved scenarios by 60 percent. The results can be improved further by eight percent by including the AVRA and the PRA. Since the environment shown in Figure 5.7a is a simple one without dead ends the gain increases for more complex environments further, shown in Figure 5.9.

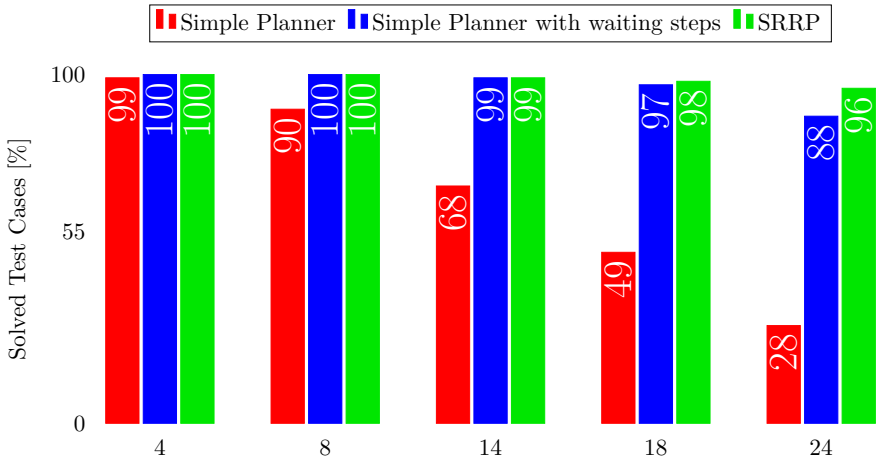


Figure 5.8: The percentage of solved scenarios in environment one for different single robot planner.

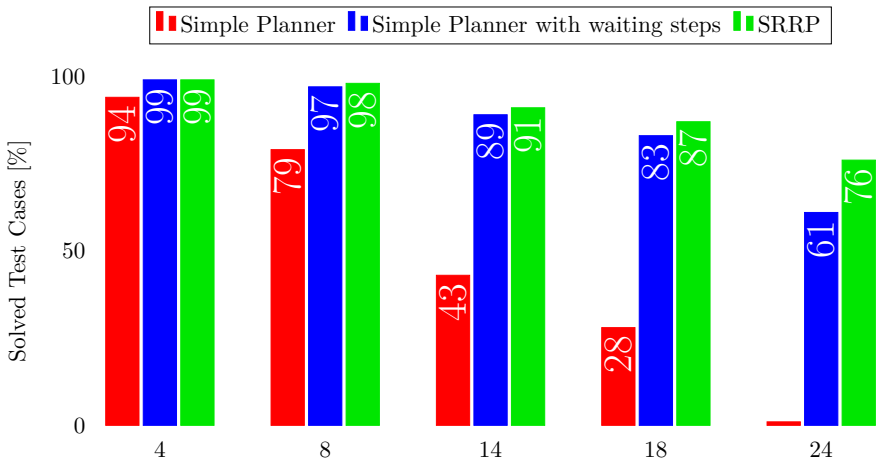


Figure 5.9: The percentage of solved scenarios in environment two for different single robot planner.

5.2.2 Prioritized Planning with Extensions

To verify the SRRP to other strategies like a simple planner combined with priority rescheduling some more test cases are done. Because every rescheduling strategy needs multiple planning iterations, the average execution time for finding a routing table is compared as well. In Figure 5.10, 5.11 and 5.12 the comparison of the simple planner combined with priority rescheduling, the SRRP without any rescheduling strategy and the full MRRP are compared by the number of solvable test cases and execution time of the planner. Furthermore, the average route length of the longest route in a routing table is compared. As one can see from Figure 5.10 priority rescheduling combined with a simple

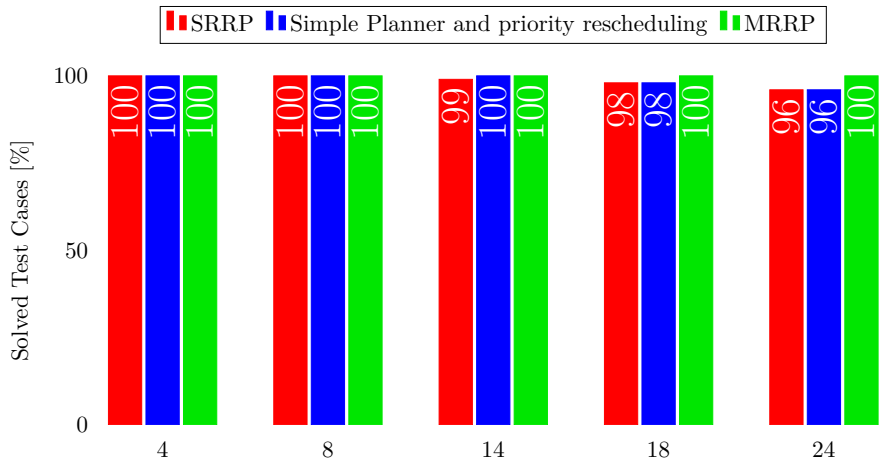


Figure 5.10: The percentage of solved scenarios in environment one for different multi robot planner.

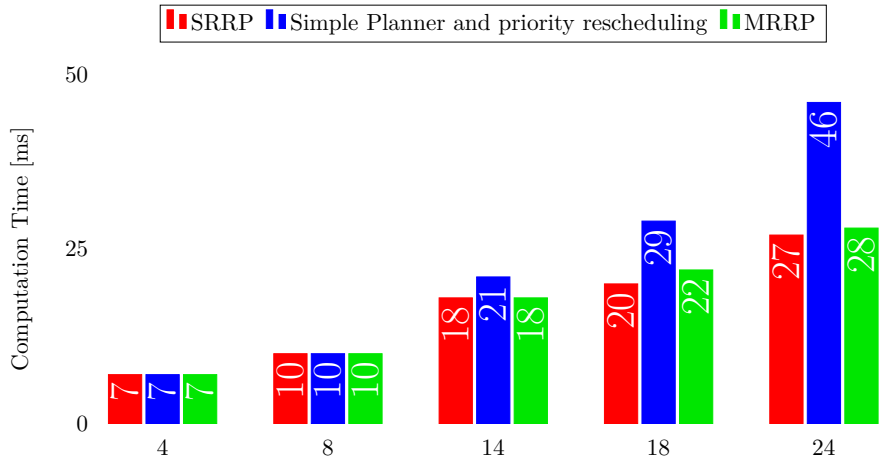


Figure 5.11: The average computation time of the solved scenarios in environment one for different single robot planner.

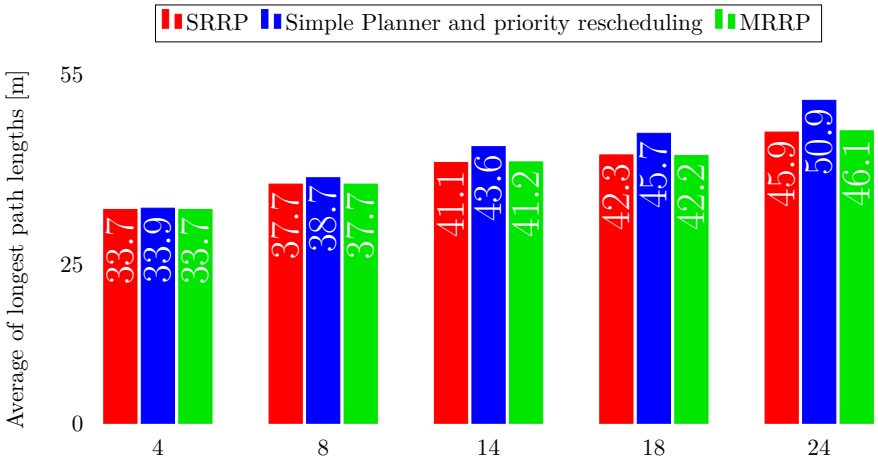


Figure 5.12: Compared the longest path lengths of solved scenarios averaged.

planner works slightly better than the SRRP alone in terms of test case coverage, but is worse in terms of execution time and overall route length, seen in Figures 5.11 and 5.12. For a low number of robots, this strategy is as good as the other ones, but route length and execution time are growing with larger numbers of robots. The reason, for the increased planning time, is the increasing number of rescheduling iterations with growing numbers of robots, where each iteration triggers the planner again. The longer route lengths originate from the fact that collisions between robots have to be avoided in a large scale by using the simple planner, but the SRRP can solve these collisions at crossings in the vicinity of them.

The MRRP combines the best of both worlds and uses basically the SRRP to solve problems and triggers the priority scheduling approach combined with a speed scheduling approach when the SRRP fails. This has the benefit of low execution times and route lengths comparable to the SRRP as well as a high test case coverage. As one can see in Figure 5.10 the MRRP manages to find a solution for all scenarios. Since the MRRP

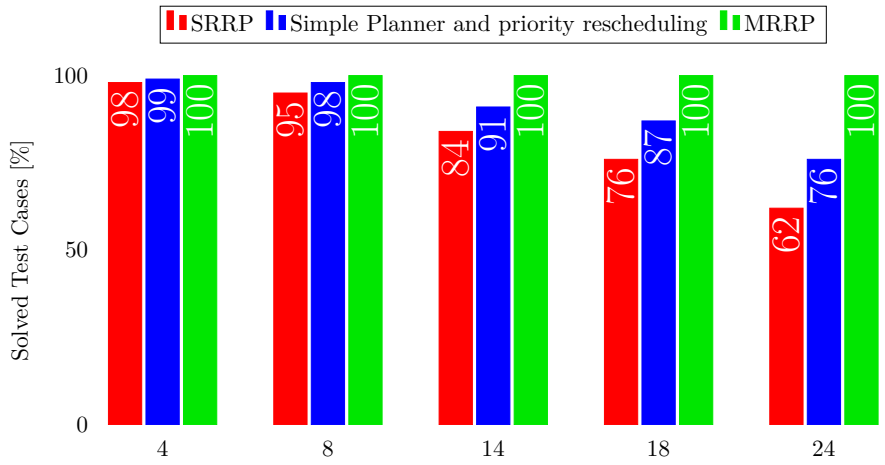


Figure 5.13: The percentage of solved scenarios in environment two for different single robot planner.

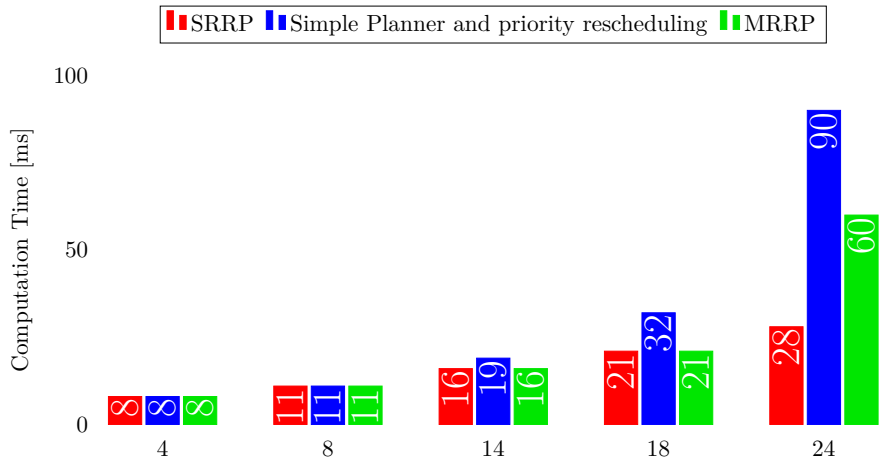


Figure 5.14: The average computation time of the solved scenarios in environment two for different single robot planner.

is designed to shine in complex environments further test runs are executed with a more complex environment, by adding walls randomly to the environment shown in Figure 5.7a. This altered environment can be seen in Figure 5.7b. The results can be seen in Figure 5.13, 5.14 and 5.15. Again test case coverage, average execution time

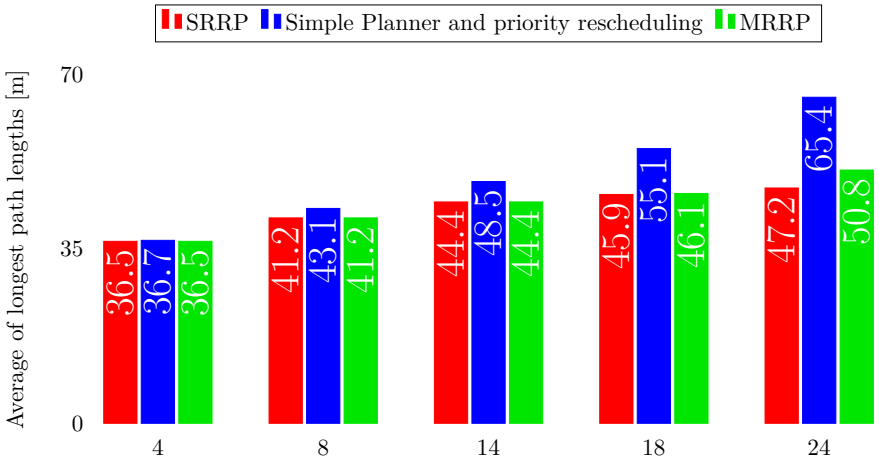


Figure 5.15: Compared the longest path length of solved scenarios averaged.

and average route lengths are compared. In this test environment, the SRRP has better results compared to the simple planner with the priority scheduler because there is a big gap in between test case coverage of the simple planner and the SRRP, which is shown in Figure 5.11. Therefore many replanning iterations have to be done by the priority rescheduler which increases the execution time. As one can see the MRRP has a fairly large average execution time for the test run with 24 robots as well. This high number originates from multiple test cases where everyone of them needed more than 300ms to find a plan, because of a large number of replanning approaches. All of these test cases where only solvable by the MRRP. Calculating the execution time without these test cases will result in 35ms.
Generally speaking, the MRRP finds more solution than priority rescheduling with a simple planner in less time, because of the ability from the SRRP to solve collisions in the local vicinity of them. Furthermore, the priority and the speed rescheduling strategy improve the results further to combine the low execution time of the SRRP with the ability to find solvable priority schemes for a better test coverage. What can be seen as well is, that the MRRP shines in more complex environments, because of the SRRPs collision resolution strategies.

CHAPTER 6

Conclusion

The aim of this work was to find a new approach for multi robot path planning. The planner has to generate deadlock free routes for a large number of robots. Therefore, this thesis presents an approach based on Prioritized Planning. The paper describes a multi robot path planner containing a single robot path planner, a collision resolver, a Route Coordinator, a priority and speed rescheduler to find a solution to a given problem on a graph describing space and time.
The single robot path planner is able to avoid moving obstacles by inserting waiting steps and additional path segments in a trajectory. These alternative trajectories are integrated into a Dijkstra algorithm by extending the search graph temporarily, with a solution generated by the collision resolver.
The Route Coordinator saves already planned robot trajectories as moving obstacles in the graph to constrain the generation of a new one.
The collision resolver is created to analyze potential collisions between a robot and a moving obstacle, found by the single robot path planner and find a resolution, therefore. This resolution is temporarily added to the search graph to enable the path planner to find a route for the conflicting robot.
To solve scenarios non-solvable by prioritized planning alone a speed rescheduler and a priority rescheduler are proposed, which try to find valid speed and priority schemes for a prioritized planning approach by considering potential collisions found while planning a robot's route.
It is shown, that the planner outperforms currently used strategies to find solutions to a number of given scenarios considering time, space and number of solved instances.

6.1 Further Work

In Chapter 5 a test case is shown, where MRRP fails to find a solution. In this case, no solution using a standard prioritized planning approach can be found. Because the green and the blue robot are blocking themselves mutually. An idea to solve this would be to split trajectories on certain points to have different parts of the trajectory, which can be prioritized differently, what would solve this problem. Therefore the blue robot would plan its trajectory to the crossing, where he has to wait for the red one. On this waiting segment, the route can be split and the green robot's trajectory can be planned, which allows him to find its goal. After planning the green robot's route the blue one can finish its route. It remains to check, when and where to split routes and if this strategy is feasible.
Furthermore the potential calculator only uses the time a robot needs to pass the segment. An idea for an improvement there would be to try to update this potential calculator with live data and statistical data from other robots to improve the MRRP behavior.
Another idea would be to optimize routes for different goals like the lowest execution time for a routing table. An idea therefore is to iterate over all trajectories and replan single ones to improve the result.
The speed rescheduler works well as a supplement to the solve specific test cases. Therefore, it would be an interesting topic to examine if there is a well working heuristic for the speed scheduler to replace the priority scheduler or other similar approaches like waiting.
As one can see there are lots of topics to investigate further for the MRRP, which maybe can help to find a fast and complete approach for prioritized planning.

6.2 Implementation and Source code

The created MRRP is successfully used in a research project, to coordinate an autonomous robot fleet. Since this project uses ROS as framework the planner is implemented as ROS node as well and will be available on GitHub.

List of Figures

1.1 Example test scenario, where two robots switch places. 1
1.2 Voronoi graph generation 2
1.3 Voronoi graph generation 3
1.4 Problem set where robot priorities matter 4

3.1 The overall picture of the Route Planner 13
3.2 A graph transformation from any roadmap consisting of subsegments to the search graph used by the MRRP. ($S_x \rightarrow V_x$) 15
3.3 Test cases used for creating the SRRP 16
3.4 The structure of the MRRP 18
3.5 The data structure for saving the robot routes. t_n saves the vertices occupied for the specific time slice. 22
3.6 Segment occupation of a moving robot 22
3.7 A problem where two robots will switch places without detecting a collision. It is assumed that every vertex is two "time-steps" long. 23
3.8 A scenario where robot r_1 has to avoid r_0 24
3.9 An example for a three dimensional graph extension. Vertices $V'_{1,2,3}$ are already assigned with different potentials compared to their parent nodes ($V_{1,2,3}$). 25
3.10 Example Testcase which needs the BTA (Solution shown in Figure 3.11) . 27
3.11 The implementation of the BTA 29
3.12 Example Testcase which needs the AVRCA (Solution shown in Figure 3.13) 32
3.13 The implementation of the AVRCA 34
3.14 Example Testcase which needs the AVRSA (Solution shown in Figure 3.15) 35
3.15 The implementation of the AVRSA 37
3.16 Testcase which needs the AVRGA (Solution shown in Figure 3.17) 38
3.17 The solution for the scenario shown in 3.16. 40
3.18 Example Testcase which needs the PRA (Solution shown in Figure 3.19) . . 41
3.19 The implementation of the PRA 42
3.20 Problem sets, where the SRRP fail. 44
3.21 In this scenario it is unnecessary to exchange r_2's priority with any other robot 44

4.1 The test environment consisting of Path Server, Path Segmentation, the MRRP, simulation environment and multiple controllers (c_i) for each vehicle. 50
4.2 Examples for maximum- and saddle-points. (Lighter colors describing lower and darker ones higher distance values.) 52
4.3 Shown a case where the final image of the algorithm from [NAU06] is more than one pixel thick 52
4.4 A crossing where many pixels have more than two neighbors. Shown the voronoi path (magenta; v) and the margin pixels (red; m) neighboring a non-margin pixel (less than two neighbors) 53
4.5 Shown in magenta the path (v); in red the margin pixels (m) of the crossing; in green the non-crossing pixels to expand; the Dijkstra expander starts in p_1 and expands until a margin pixel is found, which is not p_1; the algorithm will end up in p_2 and save the segment from $(p_1 \rightarrow p_2)$ 54
4.6 Shown the segmentation of a path in yellow and in magenta the given path 54
4.7 Example for segment constraints 60
4.8 The test environment 61

5.1 The test environment, with all generated graph nodes (vertices) shown in yellow and marked with S_x. 63
5.2 A test scenario for the MRRP 64
5.3 A test scenario for the MRRP 66
5.4 A test scenario for the MRRP 67
5.5 A scenario where the MRRP fails. 69
5.6 A test scenario for the MRRP 70
5.7 The used environments for testing the MRRP 72
5.8 The percentage of solved scenarios in environment one for different single robot planner. 74
5.9 The percentage of solved scenarios in environment two for different single robot planner. 74
5.10 The percentage of solved scenarios in environment one for different multi robot planner. 75
5.11 The average computation time of the solved scenarios in environment one for different single robot planner. 75
5.12 Compared the longest path lengths of solved scenarios averaged. 76
5.13 The percentage of solved scenarios in environment two for different single robot planner. 77
5.14 The average computation time of the solved scenarios in environment two for different single robot planner. 77
5.15 Compared the longest path length of solved scenarios averaged. 78

List of Algorithms

3.1 A-Star Algorithm 19
3.2 Adapted A-Star Algorithm 21
3.3 BTRA 28
3.4 AVRCA 33
3.5 AVRSA 36
3.6 AVRGA 39
3.7 PRA 43
3.8 Get Priority Schedule 45
3.9 Get Speed Schedule 47

4.1 Path Segmentation 53
4.2 Calculate Route candidates 56
4.3 Calculate Schedules 56
4.4 Create Route candidate 57
4.5 MovePath 58

Glossary

graph Refers to the search graph used by the route planner. The graph has a bijective mapping to the used environment where each vertex is mapped to a segment of the environment.. 2–4, 13, 14, 17, 25, 81

path Describes a list of points located on a map to lead a robot to its goal.. 1–3, 15, 20, 22, 51, 85

roadmap A data abstraction of a grid-map.. 13–15, 49, 50, 81, 85

route Describes a list of segments, including space and time, which are part of a routing table.. 2–7, 14, 15, 17, 22, 37, 40, 45, 50, 58–60, 65, 66, 68, 69, 71, 73, 75–81, 85

route candidate Describes any list of vertices, including space and time.. 13–15, 17, 18, 20, 22–25, 27, 30–32, 35, 38, 41, 44–48, 56–58, 83

routing table A list of synchronized routes describing a plan for a robot fleet.. 4, 6, 7, 13–15, 17, 18, 45, 46, 48, 50, 57, 58, 60, 64, 66, 67, 70, 71, 75, 80, 85

segment A small area of an environment with a specific shape. Every segment of an environment can be mapped to exactly one vertex of the corresponding graph.. 2, 14

trajectory Describes a time-dependent point-sequence leading the robot from one path point to another. 2

voronoi path A roadmap which describe a fully connected path with line-of-sight contact to every pixel on the map.. 3, 14

Acronyms

ACS Automated Guided Vehicle Control System. 7, 15

AGV Automated Guided Vehicle. 7, 15

AVRA Avoid Robot Algorithms. 24, 31, 41, 73

AVRCA Avoid Robot at Crossing Algorithm. 27, 31, 32, 34, 47, 81

AVRGA Avoid Robot at Goal Algorithm. 27, 31, 38, 41–43, 81

AVRSA Avoid Robot at Start Algorithm. 27, 31, 35, 37, 81

BTA Backtracking Algorithm. 24, 27, 29, 31, 32, 34, 41, 73, 81

MRCR Multi Robot Collision Resolver. 17, 18, 21, 24–27, 30–32, 35, 38, 40, 41, 43

MRRP Multi Robot Route Planner. xi, 2–4, 6, 13–15, 17–20, 24, 25, 44–47, 49–52, 56–58, 60, 61, 63, 64, 66–73, 75, 77, 78, 80–82

ORCA Optimal Reciprocal Collision Avoidance. 8

PE Potential Expander. 17, 18, 26, 30, 32, 35, 38, 41

PR Priority Rescheduler. 13, 14, 17, 18

PRA Push Robot Algorithm. 24, 30, 32, 35, 38, 41–43, 73, 81

RC Route Coordinator. 17, 18, 20, 22, 23, 27, 30, 31, 59, 79

RG Route Generator. 13, 14, 17, 18, 58

ROS Robot Operationg System. 5, 49, 61

SR Speed Rescheduler. 13, 14, 17, 18

SRRP Single Robot Route Planner. xi, 5, 13, 14, 16–27, 29, 31–33, 35, 37–39, 41, 43–48, 65, 67, 68, 71, 73, 75–78, 81

VPG Velocity Profile Generator. 58, 59

Bibliography

[BBT01] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Optimizing Schedules for Prioritized Path Planning of Multi-Robot Systems. In *ICRA*, 2001.

[BBT02] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous Systems*, 41(2–3):89 – 99, 2002. Ninth International Symposium on Intelligent Robotic Systems.

[Bor86] Gunilla Borgefors. Distance Transformations in Digital Images. *Comput. Vision Graph. Image Process.*, 34(3):344–371, June 1986.

[BRS+15] M. Bader, A. Richtsfeld, M. Suchi, G. Todoran, W. Holl, and M. Vincze. Balancing Centralised Control with Vehicle Autonomy in AGV Systems for Industrial Acceptance. In *Proceeding of the Eleventh International Conference on Autonomic and Autonomous Systems (ICAS 2015)*, 2015.

[CNKS15] M. Cap, P. Novak, A. Kleiner, and M. Selecky. Prioritized Planning Algorithms for Trajectory Coordination of Multiple Mobile Robots. *IEEE Transactions on Automation Science and Engineering*, 12(3):835–849, July 2015.

[CNM99] M. Crneković, B. Novaković, and D. Majetić. Mobile Robot Path Planning in 2D Using Network of Equidistant Path. *Journal of computing and information technology*, Vol.7(2):123–135, June 1999.

[CPA+14] Marcello Cirillo, Federico Pecora, Henrik Andreasson, Tansel Uras, and Sven Koenig. Integrated Motion Planning and Coordination for Industrial Vehicles. In *Proceedings of the Twenty-Fourth International Conferenc on International Conference on Automated Planning and Scheduling*, ICAPS'14, pages 463–471. AAAI Press, 2014.

[ELP86] Michael Erdmann and Tomas Lozano-Perez. On Multiple Moving Objects. *Algorithmica*, 2:1419–1424, 1986.

[FYS03] Yi-Ta Wu Frank Y. Shih. Fast Euclidean distance transformation in two scans using a 3x3 neighborhood. *Computer Vision and Image Understanding (1077-3142)*, 2003.

[GKM10] C. Goerzen, Z. Kong, and B. Mettler. A Survey of Motion Planning Algorithms from the Perspective of Autonomous UAV Guidance. *J. Intell. Robotics Syst.*, 57(1-4):65–100, January 2010.

[JN01] Markus Jäger and Bernhard Nebel. Decentralized collision avoidance, deadlock detection, and deadlock resolution for multiple mobile robots. In *IROS*, 2001.

[KRR98] Ishay Kamon, Elon Rimon, and Ehud Rivlin. TangentBug: A Range-Sensor-Based Navigation Algorithm. *The International Journal of Robotics Research*, 17(9):934–953, 1998.

[LaV06] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006.

[LB11] R. Luna and K. E. Bekris. Efficient and complete centralized multi-robot path planning. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3268–3275, Sept 2011.

[Mar06] F. M. Marchese. Multiple Mobile Robots Path-Planning with MCA. In *International Conference on Autonomic and Autonomous Systems (ICAS'06)*, pages 56–56, July 2006.

[NAU06] A. Nedzved, S. Ablameyko, and S. Uchida. Gray-scale thinning by using a pseudo-distance map. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 2, pages 239–242, 2006.

[PCM08] M. Peasgood, C. M. Clark, and J. McPhee. A Complete and Scalable Strategy for Coordinating Multiple Robots Within Roadmaps. *IEEE Transactions on Robotics*, 24(2):283–292, April 2008.

[Rya07] Malcolm Ryan. Graph Decomposition for Efficient Multi-robot Path Planning. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2003–2008, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[vdBGLM11] Jur van den Berg, Stephen J. Guy, Ming Lin, and Dinesh Manocha. *Reciprocal n-Body Collision Avoidance*, pages 3–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[WDM07] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. In *Proceedings of the 19th National Conference on Innovative Applications of*

Artificial Intelligence - Volume 2, IAAI'07, pages 1752–1759. AAAI Press, 2007.

[WG12] Wenjie Wang and Wooi-Boon Goh. Multi-robot Path Planning with the Spatio-temporal A* Algorithm and Its Variants. In *Proceedings of the 10th International Conference on Advanced Agent Technology*, AAMAS'11, pages 313–329, Berlin, Heidelberg, 2012.

[WLW13] Qi Wang, Marco Langerwisch, and Bernardo Wagner. Wide Range Global Path Planning for a Large Number of Networked Mobile Robots based on Generalized Voronoi Diagrams. *IFAC Proceedings Volumes*, 46(29):107 – 112, 2013.

[WWW16] Qi Wang, Markus Wulfmeier, and Bernardo Wagner. Voronoi-Based Heuristic for Nonholonomic Search-Based Path Planning. In Emanuele Menegatti, Nathan Michael, Karsten Berns, and Hiroaki Yamaguchi, editors, *Intelligent Autonomous Systems 13*, volume 302 of *Advances in Intelligent Systems and Computing*, pages 445–458. Springer International Publishing, 2016.

[ZS84] T. Y. Zhang and C. Y. Suen. A Fast Parallel Algorithm for Thinning Digital Patterns. *Commun. ACM*, 27(3):236–239, March 1984.