Path Planning in Unstructured Environments: A Real-time Hybrid A* Implementation for Fast and Deterministic Path Generation for the KTH Research Concept Vehicle

Thesis · December 2016

DOI: 10.13140/RG.2.2.10091.49444

CITATIONS
20

READS
4,226

1 author:

Karl Kurzer
Karlsruhe Institute of Technology
7 PUBLICATIONS **98** CITATIONS

SEE PROFILE

DEGREE PROJECT IN VEHICLE ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2016

Path Planning in Unstructured Environments

A Real-time Hybrid A* Implementation for Fast and Deterministic Path Generation for the KTH Research Concept Vehicle

KARL KURZER

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Contents

List of Figures

List of Algorithms

Abstract

On the way to fully autonomously driving vehicles a multitude of challenges have to be overcome. One common problem is the navigation of the vehicle from a start pose to a goal pose in an environment that does not provide any specific structure (no preferred ways of movement). Typical examples of such environments are parking lots or construction sites; in these scenarios the vehicle needs to navigate safely around obstacles ideally using the optimal (with regard to a specified parameter) path between the start and the goal pose.

The work conducted throughout this master's thesis focuses on the development of a suitable path planning algorithm for the Research Concept Vehicle (RCV) of the Integrated Transport Research Lab (ITRL) at KTH Royal Institute of Technology, in Stockholm, Sweden.

The development of the path planner requires more than just the pure algorithm, as the code needs to be tested and respective results evaluated. In addition, the resulting algorithm needs to be wrapped in a way that it can be deployed easily and interfaced with different other systems on the research vehicle. Thus the thesis also tries to gives insights into ways of achieving real-time capabilities necessary for experimental testing as well as on how to setup a visualization environment for simulation and debugging.

Acknowledgments

As this this was written at the ITRL (Integrated Transport Research Lab) I had the opportunity to work together with a variety of people from different fields. My sincerest thanks go to my supervisors Mikael Nybacka, John Folkesson and Jonas Mårtensson for their confidence in me and the chance to work on this interesting and seminal topic.

Furthermore I want to thank Andreas Högger for his suggestion of using `C++` in combination with ROS and his continuous support with both, without ROS the integration into the RCV architecture would have been much more cumbersome. I also want to thank the invaluable asset – Rui Oliveira for always giving me the opportunity to discuss concepts, ideas and foster understanding.

Additional thanks go to Niclas Evestedt an Erik Ward, who have clarified key questions throughout this thesis and have helped with the integration of the code on the vehicle.

Special thanks go to Moritz Werling (BMW) who not only initially pointed me in the direction of the topic, but also gave valuable insights with regard to the algorithm in general. I also want to thank Marcello Cirillo (Scania), who not only mentioned important implementation details of search algorithms, but asked critical questions that improved the overall quality.

Last but not least I want to thank all people that I had the pleasure working with at ITRL and KTH in general. It has been a very pleasurable experience for me with a steep learning curve.

Part I

Introduction and Theoretical Framework

Chapter 1

Introduction

On the way to fully autonomously driving vehicles a multitude of challenges have to be overcome. A crucial part for autonomously driving vehicles is a planning system that typically encompasses different abstraction layers, such as mission, behavioral, and motion planning. While the mission planner provides a suitable route in a road network to arrive at the goal, the behavioral planner determines appropriate actions in traffic, such as changing lanes as well as stopping at intersections; the motion planner on the other hand operates on a lower level, avoiding obstacles, while progressing towards local goals [38].

An even more specialized problem is the navigation of the vehicle from a start pose to a goal pose in an environment that does not provide any specific structure; unstructured driving as opposed to structured driving (road following). Typical examples of such environments are parking lots or construction site. In these scenarios the vehicle needs to navigate safely around obstacles without any kind of path reference, such as lane markings, ideally using the optimal[1] path between the start and the goal pose.

1.1 Relevance of the Topic

Autonomous driving vehicles might be one of the most publicly discussed and researched engineering topics at the moment of this writing. While the traditional car makers seem to pick up speed only very slowly, new entrants such as Alphabet (formerly Google) Comma.ai, drive.ai, Uber in the USA as well as OXBOTICA in the UK, and ZMP, Robot Taxi and nuTonomy in Asia are accelerating the advancement in self driving car technology at a rapid pace, releasing autonomous vehicles before the decade's end. A simple query with the search term *autonomous driving* on Google Trends will underline this, revealing a huge increase in search volume over the past six years depicted in Figure 1.1 on the facing page.

In a study titled "Revolution in the Driver's Seat: The Road to Autonomous Vehicles" the Boston Consulting Group defines Tesla's Autopilot released in October 2015 the first stage of several from partial to fully autonomous driving. Furthermore their estimates are, that fully autonomous driving vehicles will

[1] with regard to a specified parameter, such as time, length, velocity, lateral acceleration, distance to obstacles, etc.

arrive by 2025. The most cited reasons by consumers in the USA to purchase an autonomous driving vehicle were an increase in safety, lower insurance premiums as well as the ability to be productive and multitask while the vehicle is driving. [31]

For the society at large the most important benefit will be the reduction of accidents, as 90 percent of the time human error accounts for these. Other benefits of self-driving cars are the reduction of space consumption necessary for parking (self-driving cars do not need space to open the doors, which accounts to roughly 15%) as reported by McKinsey & Company. [3]

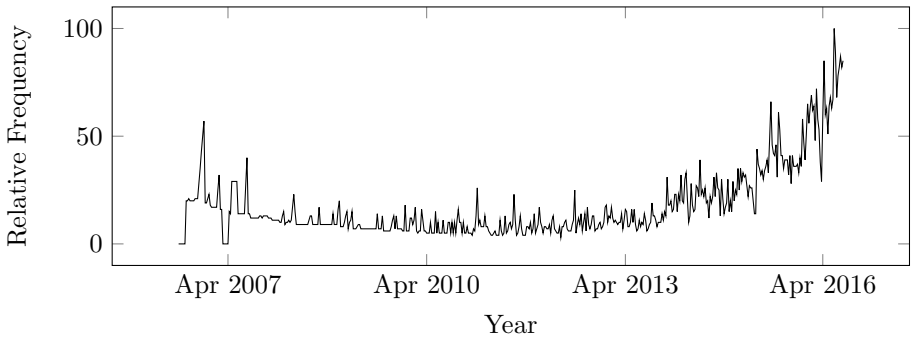


Figure 1.1: Google Trends for the query "autonomous driving"

1.2 Context of the Thesis

The Integrated Transport Research Lab (ITRL) at KTH is responding to the need for long-term multidisciplinary research cooperation to tackle the global environmental transport challenges by means of radically new and holistic technical solutions. ITRL's approach is that seamless transport services, infrastructure, novel vehicle concepts, business models and policies, all need to be tuned and optimized in chorus.

A key role for the research of new technical solutions at the ITRL is the Research Concept Vehicle (RCV), that serves as a test bed for a variety of research topics, that cover areas of the vehicle such as, design, control, perception, planning as well as systems integration.

As autonomous driving will be a part of the solution for global environmental transport challenges the aim with this and other theses is to equip the RCV step by step with the capabilities to become self-driving. Hence this thesis will be a part of a much larger project.

1.3 Problem Description

The problem that this thesis aims to solve can be summarized as follows.

Find a solution in real-time that transitions a nonholonomic vehicle collision free from a given start pose to a desired goal pose within an unstructured environment based on the input of a two dimensional obstacle map, or report the nonexistence of such a solution.

Figure 1.2 depicts the problem. The path transitions the vehicle from the start state x_s safely to the goal state x_g, while adhering to the minimal turning radius r of the vehicle and avoiding obstacles.

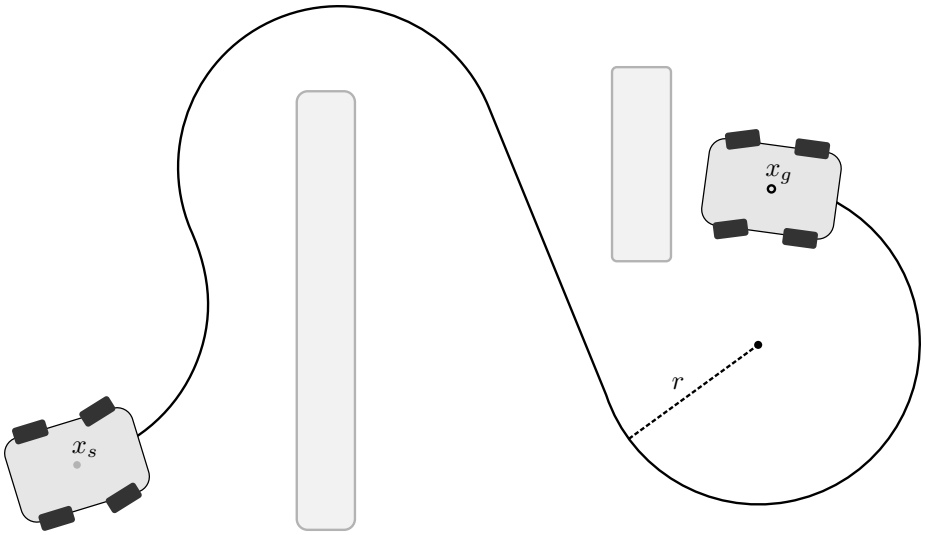
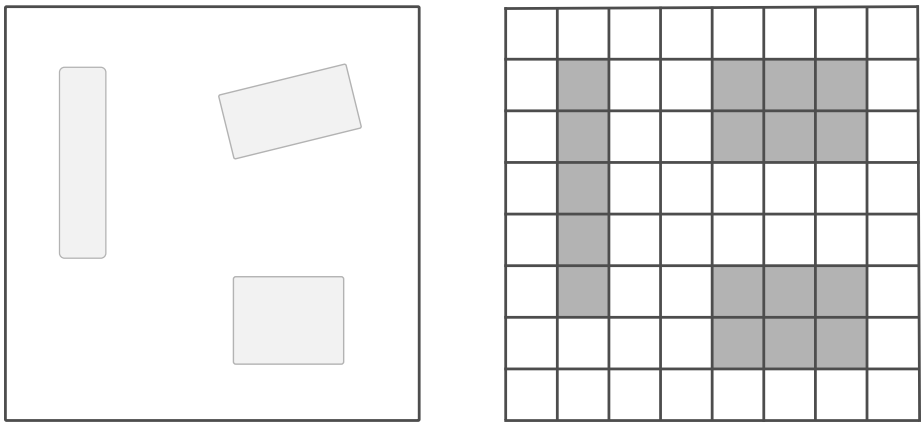


Figure 1.2: Depiction of a typical path planning problem to solve



Continuous World Binary Obstacle Grid

Figure 1.3: Obstacle grid – the input to the path planning algorithm

1.4 Scope and Aims

The points listed below describe the key requirements regarding this thesis.

1. Planning based on a local obstacle map
2. Incorporating non-holonomic constraints

3. Ensuring real-time capability
4. Performing analysis and evaluation of results
5. Integrating and driving autonomously (optional)

The input to the path planning algorithm is a grid based binary obstacle map. The second point addresses the fact that the planner should be used for vehicles that cannot turn on the spot, e.g. cars, hence the produced paths must be continuous and need to be based on a model of the vehicle. In order to use the algorithm in a car, it needs to re-plan continuously and perform collision checking. For this purpose the actual implementation needs to be as efficient as possible, thus C++ is a necessity in order to provide the re-planning frequencies required. The development shall include a critical analysis and evaluation of the algorithm as well as its results. It is the aim to deploy the software on the RCV and demonstrate its capabilities in a real-world scenario, if time and other constraints will allow for it.

The scope of the thesis is to prototypically develop as well as implement a path planning algorithm, hence this focuses on a specific solution rather than a general and will not provide a comparison between different approaches as other works might do.

1.5 Structure of the Thesis

This thesis is split into two distinct parts. The first part of this thesis will present the theoretical foundation, in order to prepare the reader for the actual implementation of the algorithm. Chapter 2 shortly introduces the vehicle platform the path planning is being developed for. Chapter 3 deals with a brief introduction into path planning, where the term is dissected and popular approaches are touched on. Chapter 4 focuses on collision detection, as it is a vital part for most path planning approaches. And Chapter 5, the last chapter of the first part dissects popular graph search algorithms, that form the basis of this work.

The second part focuses on the method and its implementation in detail. Chapter 7 uses the theoretical framework to explain the implemented hybrid A* search in detail. Chapter 8 is primarily a collection of the results as well as analysis of the same. Chapter 9 Summarizes the entirety of the thesis, while stressing the achievements of this particular implementation. And finally chapter 10 will give the reader some suggestions regarding future work that can or should be conducted.

Chapter 2

Vehicle Platform

The development of the Research Concept Vehicle started in 2012. The resulting electric two-seater is equipped with four autonomous cornering modules, that enable it to actuate each wheel independently with respect to steering angle, camber, driving or braking forces. Weighing around 400 kilogram the car can be driven one hour under normal conditions and reaches speeds of up to 70 kilometers per hour. Due to the fact that actuators are controlled by wire rather than mechanically it opens up a whole suit of new capabilities, that can improve handling, efficiency as well as safety.

The vehicle is equipped with a variety of sensors, continuously collecting data about the state of the vehicle and its environment. For the occupancy grid LIDAR sensors are used to detect obstacles and GPS, IMU, steering angle and wheel speed are used for localization.

Figure 2.1: Research Concept Vehicle of the ITRL; source tjock.se

Chapter 3

Path Planning

Fully autonomous robots need to be able to understand high-level commands, as it cannot be the goal to tell the robot how to do a certain thing, but rather what to do [23]. While autonomous robots need to be able to reason, perceive and control themselves adequately, planning plays a key role. Planning consumes the robot's knowledge about the world to provide its actuators (controllers) with appropriate actions (references) to perform the task at hand. Machine learning may still be one of the largest areas in artificial intelligence, but planning can be seen as a necessary complement to it, as in the future decisions need to be formed autonomously based on the learned [26].

This section will present and discuss a variety of different planning techniques, many of which will not only be applicable to path planning, but are powerful tools for more general problems [26].

3.1 Planning

First, one should understand the meaning of the word *planning*. Meriam Webster gives the following simple definition:"the act or process of making a plan to achieve or do something". Given this general description it becomes obvious that one can mean very different things by saying the word *planning*, the following can all be seen as acts of planning.

1. A person who intends to use public transportation to visit someone.

2. A politician who signs a bill to persuade voters.

3. A navigation system that calculates a route for a trip.

Planning in this thesis is understood as the search for a set of actions u that transitions a given start state to a desired goal state. When planning is conducted by computers one has to do it programmatically, the result is thus a *planning algorithm* that usually returns the set of actions that transitions the start to the goal state. Planning under uncertainty is not considered, such that statement number 2 is not considered in this thesis.

3.2 Path

If planning returns the set of actions that are necessary to transition from a start to a goal state, then one can say that a path consist of the entire set of actions, $u \in \mathcal{U}_{path}$ as well as the resulting states along the path $x \in X$. More specifically a path in this thesis is either the set of actions $\mathcal{U}_{path}$ that move a vehicle from a current start state to a desired goal state through the Euclidean two dimensional plane or the resulting poses $p \in \mathcal{P}$, with $X \rightarrow \mathcal{P}$.

While Latombe uses the term motion planning, Lavalle uses path planning. Their might be some slight differences in the way they can be interpreted, but in general they have been used for the same things.

3.3 Basic Problem

The problem addressed in this thesis shall be further specified and a general formulation be established. The robot is the only moving object in the world. The dynamics of the robot are not considered, hence removing any time dependencies. As collisions shall not occur they will not be modeled either.

Based on these simplifications the basic problem can be described as follows [23, 26]:

- $\mathcal{A} \subset \mathcal{W}$, the robot, is a single moving rigid object in world $\mathcal{W}$ represented in the Euclidean space as $\mathbb{R}^2$
- $\mathcal{O} \subset \mathcal{W}$, the obstacles, are stationary rigid objects in $\mathcal{W}$
- geometry, position and orientation of $\mathcal{A}$ and $\mathcal{O}$ are known a priori
- **Problem:** given a start and goal pose of $\mathcal{A} \subset \mathcal{W}$, plan a path $\mathcal{P} \subset \mathcal{W}$ denoting the set of poses so that $\mathcal{A}(p) \cap \mathcal{O} = \emptyset$ for any pose $p \in \mathcal{P}$ along the path from start to goal, terminate and report $\mathcal{P}$ or $\emptyset$, if a path has been found or no such path exists

3.4 Configuration Space

In the literature of motion planning the concept of the configuration space has been well established, as it facilitates the formulation of path planning problems, presenting various concepts with an underlying scheme yielding greater expressive power [23, 26]. Configuration spaces have been extensively used by Lozano-Pérez in the 80s for the description of spatial planning problems, e.g. the search for an appropriate space to place an object or the search for a path of an object in an unstructured environment with obstacles [23]. The search for the appropriated placement or the path an object should take, is a problem often encountered in design and manufacturing, where compactness needs to be achieved but maintainability shall not be sacrificed. An example of this is changing a headlight bulb of a car. Manufacturers are trying to build it as compact as possible, but clearly there is a need for maintainability, as a light bulb might need to be replaced from time to time. The advantage of the configuration space representation is that it reduces the problem from a rigid body to a point and thus eases the search [29].

Given the robot or agent $\mathcal{A} \subset \mathcal{W}$, where $\mathcal{W}$ is the world as the two dimensional Euclidean space $\mathbb{R}^2$ with a fixed Cartesian coordinate frame $\mathcal{F}_\mathcal{W}$, each possible configuration of $\mathcal{A}(q)$ can be described by a q of the form (x, y, θ) denoting a position along the x- and y-axis as well as an orientation in $\mathcal{F}_\mathcal{W}$ respectively. All possible configurations q form the configuration space $\mathcal{C} \subset \mathcal{W}$. Attached to the robot $\mathcal{A}$ is the frame $\mathcal{F}_\mathcal{A}$, so that $\mathcal{F}_\mathcal{W}(q) = \mathcal{F}_\mathcal{A}$, this allows the description of parts of the robot relative to $\mathcal{A}$ and not $\mathcal{W}$. The configuration space of the robot can be further divided in subsets $\mathcal{C}_{obs}$ and $\mathcal{C}_{free}$. The configuration space $\mathcal{C}_{obs} \subset \mathcal{C}$ describes the set of all q where $\mathcal{A}(q) \cup \mathcal{O} \neq \emptyset$ and hence the robot is in collision. On contrast $\mathcal{C}_{free} \subset \mathcal{C}$ is the set of all safe configurations q, $\mathcal{A}(q) \cup \mathcal{O} = \emptyset$. [23, 26]

Since the configuration space only considers the static case, the regions defined are not fully descriptive, once dynamics are introduced. In order to incorporate dynamics, analog to C the state space X can be introduced, with $X = \mathcal{C}$. In addition to $\mathcal{C}$, X allows the description of the dynamics of the robot so that with $f : x \to q$ a given state x with $(x, y, \theta, \dot{x}, \dot{y})$ can be mapped to q as (x, y, θ). Describing this will add another subset $X_{ric} \to \mathcal{C}_{X_{ric}} \subset \mathcal{C}_{free}$ to X, the region of inevitable collision. This region describes all states that will lead to a collision due to robot's dynamics. [23, 26]

Figure 3.1 illustrates the configuration space for a robot $\mathcal{A}$. The white area,$\mathcal{C}_{free}$ is safe for any configuration $\mathcal{A}(q)$. The light gray areas describe possible collisions with the dark gray areas $\mathcal{O}$ and one or more configurations of $\mathcal{A}(q)$.

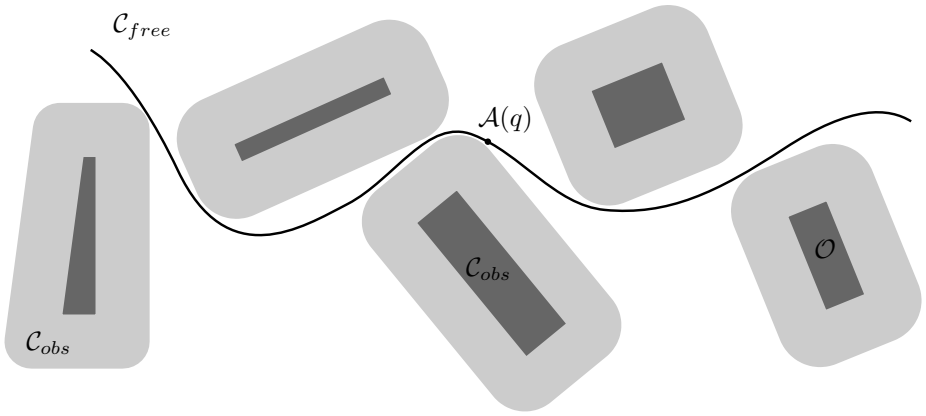


Figure 3.1: Configuration space of a robot

3.5 Popular Approaches

As with most problems, there are a variety of different methods for path planning based on the requirements of the problem to be solved. The aim of this section is not to compare methods for path planning. Each has advantages and disadvantages, as discussed in many other places, but it should rather give the reader a quick overview over different type of solutions for the same problem.

3.5.1 RRT

Rapidly-exploring random trees were first introduced in the late 90s by Lavalle and Kuffner. The goal was to create an algorithm that could be applied to general nonholonomic as well as kinodynamic planning problems, which other randomized approaches such as the randomized potential field and probabilistic road map struggled with [24]. The basic structure is exemplified in the algorithm below.

A rapidly-exploring random tree (RRT) grows by repeatedly selecting a random state in a bounded region algorithm 1.3. Starting from this random state the nearest neighbor is selected in line 4. A new state is created in line 6 by using an appropriate control action, line 5. During this phase it needs to be ensured that the resulting edge will be collision free. Finally in line 7 and 8 the tree is being grown by adding the new state and the connecting edge. Using this approach the tree explores the state space in a rapid and uniform manner as it probabilistically grows away from its root node 3.2 expanding towards larger Voronoi regions due to its randomization. Inherent to their nature RRTs are non guided and hence non optimal. [25]

An illustration of the RRT growth can be seen in Figure 3.2 on the next page for 3, 10 as well as 30 samples, depicting the fast exploration of unexplored areas.

Although random, RRTs can be biased by changing the probability distribution. Sampling in the goal region, allows for much faster convergence. Another approach is to include a measure that takes the path costs into account. [25]

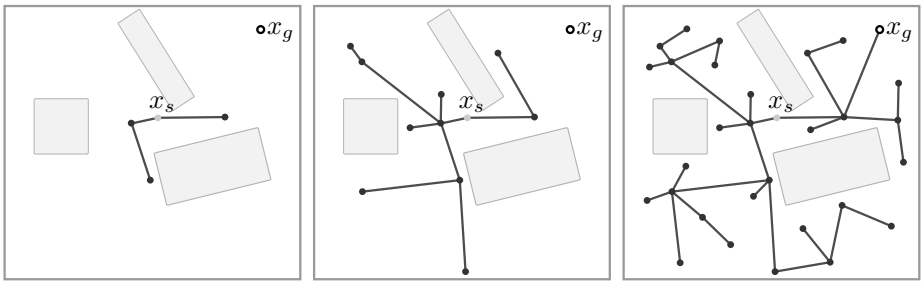
Urmson presents a heuristically-guided RRT where the probability distribution takes the size of the Voronoi region into account as well as the quality of the path so far [37].

Algorithm 1 Rapidly-exploring Random Tree

Require: $x_s \cap x_g \in X_{free}$
1: $T.init(x_s)$
2: **while** $x_{new} \neq x_g$ **do**
3: $x_{rnd} \leftarrow randomState()$
4: $x_{ngh} \leftarrow nearestNeighbor(x_{rnd})$
5: $u \leftarrow controlAction(x_{rnd}, x_{ngh})$
6: $x_{new} \leftarrow newState(x_{rnd}, x_{ngh}, u)$
7: $T.addVertex(x_{new})$
8: $T.addEdge(x_{ngh}, x_{new}, u)$
9: **end while**
10: **return** T

3.5.2 Potential Fields

Potential Fields were originally used as an online collision avoidance measure for static as well as moving obstacles using the artificial potential field concept to make the collision avoidance part of lower level of control, decreasing response time. The idea behind the artificial potential field approach can be summarized, with the robot moving in a field of forces. The goal position is an attractive, while an obstacle creates a repulsive pole [21]. The shape of the artificial potential field U represents the layout of the configuration space. Path



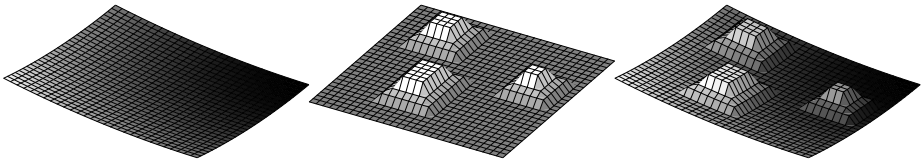
3 Samples 10 Samples 30 Samples

Figure 3.2: Rapidly-exploring random tree biased towards unexplored areas

planning is conducted in a sequential manner. In each step the artificial force $\vec{F}(q) = -\vec{\nabla}U(q)$ acting on the robot's current configuration will create a path increment towards this direction. [23]

Figure 3.3 shows the resulting potential based on a attractive goal potential (darker areas) and a repulsive obstacle potential (lighter areas). A ball placed anywhere in the resulting potential would naturally (assuming a downward force acting on the ball, such as gravity) end up in the goal location (lower right) while avoiding collisions with obstacles.

As obstacle avoidance was the original goal, potential fields come with the risk of not being able to reach the goal as the robot might get stuck in local minima. To work around this inherent characteristic one can try to formulate the potential function without local minima or incorporate techniques that allow the escape from the same, such as RRT [26]. [23]



Attractive Potential Repulsive Potential Resulting Potential

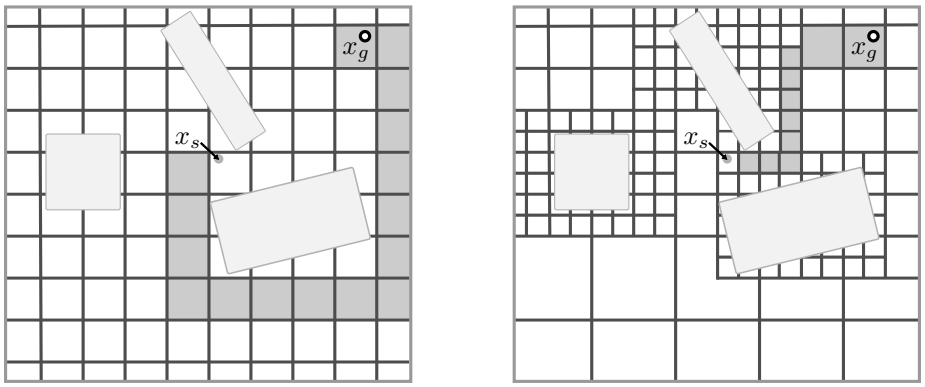
Figure 3.3: Potential Fields

3.5.3 Approximate Cell Decomposition

Path planning with approximate cell decomposition can be traced back to Brooks and Lozano-Peréz in the mid 80s. The basic idea is to divide the configuration space into rectangles with edges parallel to the axes of the space. The resulting cells will be either free, occupied or mixed, depending on the configuration space of the obstacles intersecting with the respective rectangle or not, see Figure 3.4 on the next page. The search for a path is conducted by finding a set of connected and free cells that include the start as well as the goal configuration. [6] Applicable graph search algorithms are explained in more detail in chapter 5 on page 18.

This approach does not represent the free space exactly, hence a conservative

approach must be taken in order to avoid faulty planning. Decomposing the cells in this way, although not exact, in general allows for planning methods that are easier to implement in comparison to exact cell decomposition. [23]



Constant Resolution Variable Resolution

Figure 3.4: Approximate Cell Decomposition

3.6 Differential/Kinematic Constraints

This section focuses on the differential constraints of a car-like robot. Most of the time these differential constraints are inherent to the kinematics and dynamics of the robot itself. These constraints need to be taken into account at some point, ideally during the actual path planning process ensuring the path matches the robot's constraints. If it is infeasible to consider the constraints during the planning process is infeasible one could also pass this task to the controller. Given the constraints however this will not be an easy task either. [26]

Even though a car can reach any position and orientation in the Euclidean plane, with a configuration as $q = (x, y, \theta)$, its configuration space is thus $\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1$, it cannot translate or rotate freely. A car can move forward as well as backward, but it cannot move sideways. This means that there are less possible actions than degrees of freedom, systems such as this are called underactuated, [26].

A common case for drivers of cars is to parallel park, in order to achieve a configuration q_1 that is parallel to q_0 it is necessary for a car to at least rotate and translate the same goes for a third configuration q_2 that has the same position but different orientation than q_0. [23]

The constraint that does not allow the vehicle to move sideways, but only in the direction of the heading, can be expressed as the velocity orthogonal to the the vehicles heading being equal to 0 at all times. This velocity is denoted by $v_\perp$.

$$v_\perp = \frac{\dot{x}}{\cos(\theta - \pi/2)} \tag{3.1}$$

$$v_\perp = \frac{-\dot{y}}{\sin(\theta - \pi/2)} \tag{3.2}$$

$$\frac{\dot{x}}{\cos(\theta - \pi/2)} = \frac{-\dot{y}}{\sin(\theta - \pi/2)} \tag{3.3}$$

$$\dot{x}\sin(\theta - \pi/2) + \dot{y}\cos(\theta - \pi/2) = 0 \tag{3.4}$$

The following describes the non-holonomic constraint.

$$\dot{x}\cos(\theta) - \dot{y}\sin(\theta) = 0 \tag{3.5}$$

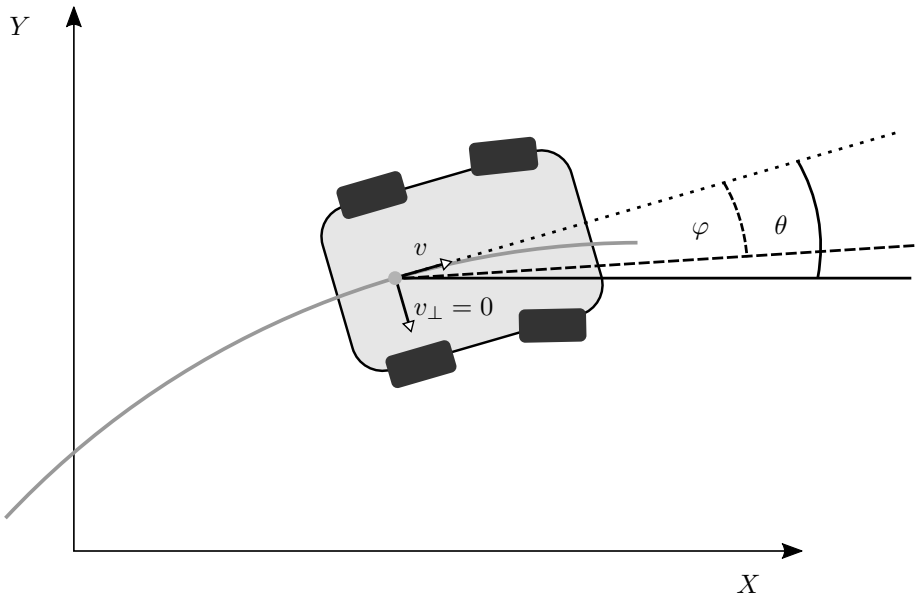


Figure 3.5: Kinematic Constraints of a car-like robot

3.6.1 Dubins Curves

In 1957 Dubins published a paper, showing that there exists an analytic solution for the minimum path length between to states $x_0, x_1 \in X$ of a particle. Such a path will have a curvature along its entirety, where the curvature r^{-1} is upper bound by a minimum turning radius r. He showed that even the most complex of these shortest paths can be represented by a using a maximum of three segments, consisting only of curves C with the upper bound curvature r^{-1} as well as straights S, CCC or CSC. [13] With this Dubins created the first analytic solution to solve path planning problem giving an upper bound curvature in constant time, applying to the non-holonomic nature of a normal car. Dubins' approach is especially useful as a heuristic for path planning in obstacle sparse environments, since it does not account for obstacles and hence paths are more likely to collide with the environment when the density of obstacles increases.

The dashed line in Figure 3.6 on the next page depicts a Dubins curve, it consists of a C, S and C segment.

3.6.2 Reeds-Shepp Curves

More than thirty years later in 1990, Reeds and Shepp solved a problem seemingly similar to the one of Dubins. They developed a solution for calculating paths of upper bound curvature under the assumption that the car could drive forwards as well as backwards. The solution with a maximum of 2 cusps (due to reversing) can be found among a set of possible paths that will never exceed 68. The minimum lenght path in the pool of possible paths is the solution. Just like Dubins Curves Reeds-Shepp curves also are made up of curved and straight segments. Due to the possibility of reversing the paths will consist at most of five segments, $CCSCC$ [34]

The solid line in Figure 3.6 depicts a Reeds-Shepp curve, it as well as the Dubins curve in this case consists of a C, S and C segment.

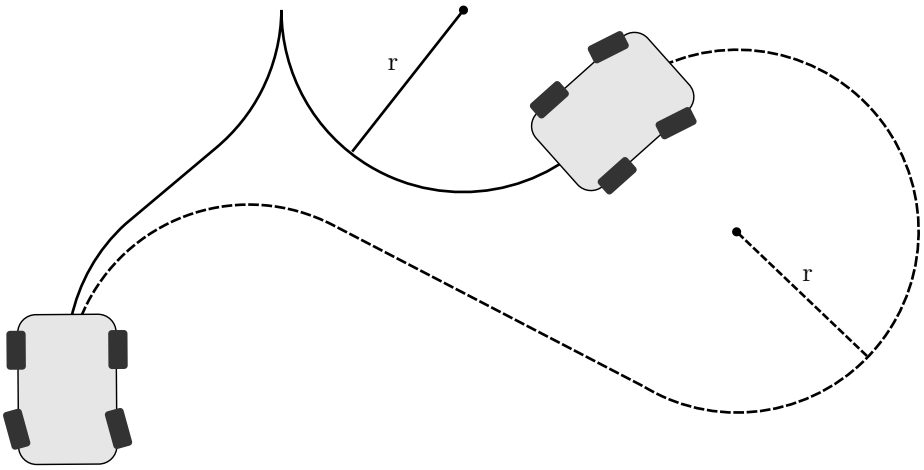


Figure 3.6: Dubins (dashed) and Reeds-Shepp (solid) curves

Chapter 4

Collision Detection

Collision detection is a basic geometric operation that is applicable to many applications such as computer games, robotics and engineering simulations [7,14, 33]. While some hard to model and computational intense planning approaches produce collision free paths by nature, others such as the ones introduced in chapter 3, require explicit collision checking along the paths they produce [26]. Collision detection as a whole is concerned with the *if*, *when* and *where* two objects collide [14], an example for a non collision free path is shown in Figure 4.1 on the next page. The following focuses primarily on the *if*. Another distinction can be made, with regard to discrete or continuous checking. While static collision detection is computationally much cheaper, it comes at the risk of tunneling, where both objects might pass each other from one time step to the next and the collision goes undetected [14].

A path $\mathcal{P}$ produced by a motion planning algorithm needs to be collision free based on the information provided, hence $\mathcal{P} \subset \mathcal{C}_{free}$. If the environment of the robot changes, so that $\mathcal{P} \cup \mathcal{C}_{obs} \neq \emptyset$ a new path needs to be computed. Whether it is beneficial to recompute paths on every update of the environment or only perform collision checking for the previous path given the change in the environment depends on the specific case.

Collision detection can be conducted in a great variety of ways. While the use of the configuration space is beneficial due to its expressive power and verbosity it might not be useful during the actual collision detection [26]. The important thing to consider is the computational cost for checking whether $q \in \mathcal{P} \wedge q \in \mathcal{C}_{free}$ is true for a given configuration q, which can be seen as a logical predicate [26]. As a path can only be considered safe, if its entirety of states is collision free, collision detection needs to be conducted along the entire length of the same [14, 26]. Two methods shall now be introduced.

4.1 Bounding Space and Hierarchies

For reasons of performance it is usually beneficial to wrap objects with bounding spaces. Bounding spaces are simple volumes for $\mathbb{R}^3$ or areas for $\mathbb{R}^2$ that encapsulate the more complex object. These allow for faster overlap tests. Depending on the accuracy needed it can be sufficient to only check these bounding spaces for collisions. [14, 26]

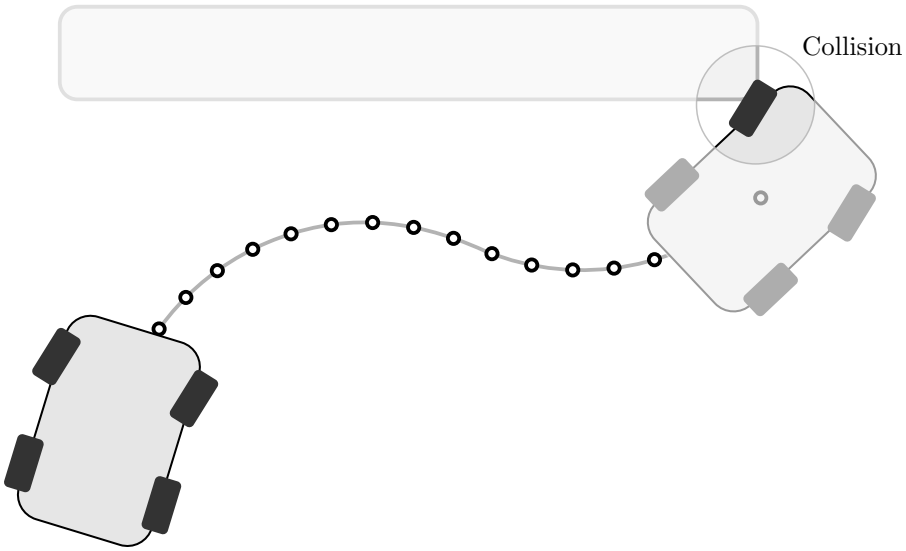


Figure 4.1: Collision detection

If higher accuracy is required, hierarchical methods can be implemented, these break up a larger complicated convex bodies into a tree. The tree represents bounding spaces that contain smaller and smaller subsets of the original object as depicted in Figure 4.2 on the facing page. This hierachy allows for more accurate geometric description of the object, while reducing the computational cost of intersection testing, since children only need to be tested if their larger parents collide. [14, 26]

Lavalle and Ericson define some criteria for the choice of appropriate bounding spaces.

- The space should fit the object as tightly as possible.
- The intersection test for two spaces should be as efficient as possible.
- The space should be easy to rotate.

Figure 4.2 on the next page illustrates two different ways bounding regions can be used. On the left a rectangular shape for approximation is used, while on the right the shape is broken down into a hierarchy with two levels using circles for approximation. For the latter a simple intersection test with obstacles using circular bounding regions consists of computing the relative distance of the centers and compute whether or not the distance is greater than die combined radii of both regions.

4.2 Spatial Occupancy Enumeration

Spatial occupancy enumeration overlays the space with a grid. This subdivision of space allows the occupancy enumeration of objects, storing an exhaustive array of grid cells covered by the respective object [14, 19]. The method is depicted in Figure 4.3 on the facing page. In $\mathbb{R}^2$ these might be squares, in

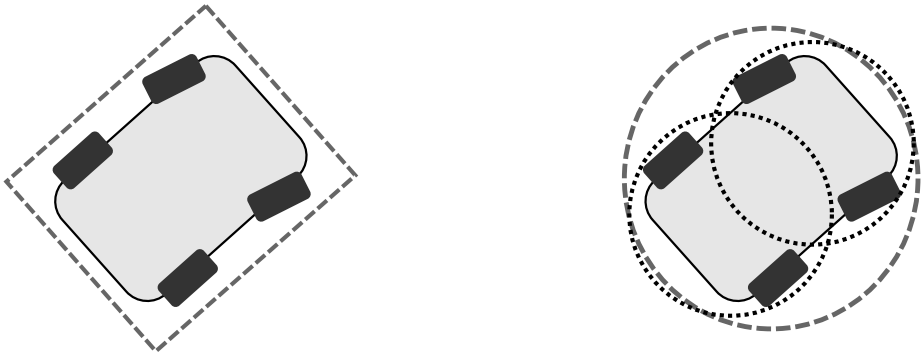


Figure 4.2: Bounding regions

$\mathbb{R}^3$ cubes. This approach offers a trivial collision check that can be conducted rapidly [14,19]. Enumerative schemes have the disadvantage that the respective spatial occupancy needs to be recomputed for each point of a given path [19]. This problem can be avoided by discretization of the orientation values, so that the occupancy enumeration can be stored in a lookup table [39]

Due to its simplicity uniform grids have been a popular choice for space subdivision, however it is important to choose the right cell size. It is important that the cell size is appropriate for the size of the objects in the environment as well as the fidelity of the sensor information. If the grid is too fine, the collision detection will take too long as many more grid cells need to be checked and the amount of sensor information per cell will be lower. If the grid is too coarse then the free space will be underestimated, making the collision check conservative and risking that the path planning algorithm will not find a solution even though one exists. In case, that a fixed cell size is an issue hierachical grids might be a solution [14].

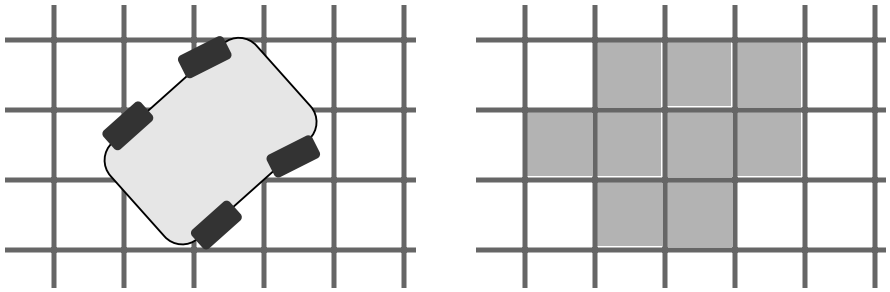


Figure 4.3: Spatial Occupancy Enumeration

Chapter 5

Graph Search

This chapter explains the basic theory necessary to understand graphs and the terminology associated with it. The later part of the chapter focuses on elaborating different graph search algorithms that are fundamental to understanding the implementation of the search algorithm in this thesis.

5.1 Fundamentals

A graph such as the one depicted in figure 5.1 consists of vertices V as well as edges E. With G being a graph $V = V(G)$, is the set of vertices of the graph and $E = E(G)$, the set of Edges. Edges connect vertices of a graph. An edge of a graph can be described by x, y, as it connects the vertices x and y. Edges that have at least one vertex in common are considered adjacent. Vertices that have at least one edge in common are called neighboring [4]

Graphs can be either directed or undirected. A directed graph has unidirectional edges, a undirected graph has bidirectional edges. In the following the word nodes and vertices might be used interchangeably.

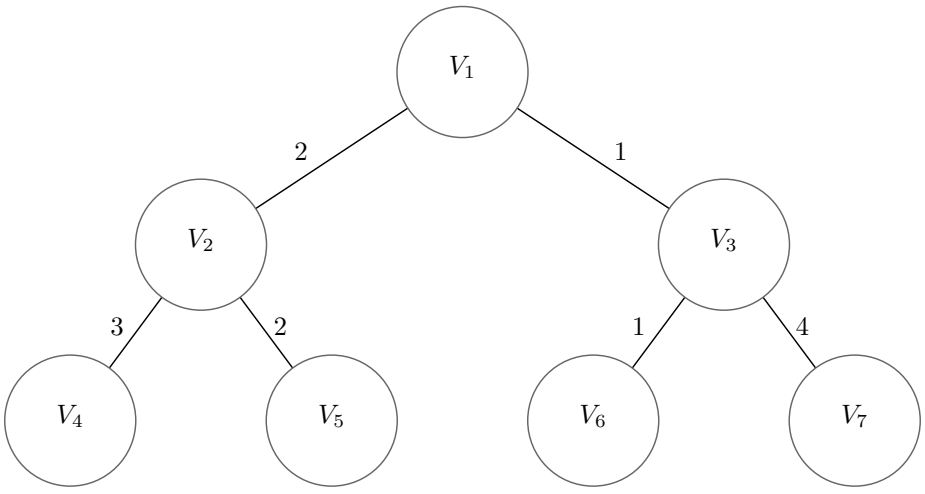


Figure 5.1: Graph theory explanation

5.1.1 State Space of a Graph

While the chapter is termed graph search one has to understand that this term can be seen as the search for a set of actions that changes the state of an object from an initial state to a desired goal state. Given this general description graph search algorithms can be applied to a great variety of problems from motion planning to artificial intelligence [26].

The following definition constitutes a general description of the state space of a graph, it is borrowed from Lavalle's famous book titled *Planning Algorithms*.

1. A nonempty *state space* $x \in X$, which is finite or countably infinite set of *states*.
2. For each *state* $x \in X$, a finite action space $U(x)$.
3. A *state transition function* f that produces a state $f(x, u) \in X$ for every $x \in X$ and $u \in U(x)$. The *state transition equation* is derived from f as $x' = f(x, u)$.
4. An *initial state* $x_s \in X$.
5. A *goal set* $X_G \subset X$.

The vertices V of the graph G can be considered the state space X of the G. Thus, each vertex holds information pertaining to a specific state. The edges E are best represented by the action space U. Where an edge $u \in U(x)$ transitions a state $x \in X$ to a state $x' \in X$ with the state transition function $f(x, u)$.

5.1.2 Open and Closed Lists

For the explanation of the following algorithms two types of lists need special attention. On the one side there is the open list O, representing the set of the search frontier, the vertices, that have not yet been expanded, but have an adjacent vertex that has been expanded, hence any vertex $v_i \in O$ is part of the frontier. On the other side there is the closed list C, representing the set of vertices that have already been expanded.

Priority Queues

Depending on the algorithm these lists need to implemented as a priority queue. A priority queue is a data structure that sorts a set by a key either from large to small or vice versa. The following operations are supported by the basic priority queue. All these actions maintain the order of the queue [36].

- insertion of a given item with a given key
- find the minimum of the keys of the items in the queue
- delete the item with the minimum key from the queue

The type of queue chosen has a considerable impact on the time complexity for the operations mentioned above. When implementing a queue it is recommended to take this into consideration, as a more complex queue might yield

considerable better performance, especially when operating on larger sets of data.[1]

5.1.3 Heuristics

In order to find an optimal path the search needs to be systematic. Various search algorithms differ most significantly in the way they expand vertices [26]. To avoid wasteful search of unpromising regions of a graph the search must be as informed as possible, only expanding nodes that have the potential to belong to the optimal path [18]. If the search uses information that leads to skipping the expansion of a specific node, hence failing to find the optimal path, admissibility is forfeited. An ideal heuristic returns the real cost of a vertex.

Heuristics are used as an aid for approximating a solution in order to address the limitations of processing power, in some cases drastically reducing the search space. A finite amount of time only allows for a finite number of calculations. Although this comes without a suprise it still is a major limitation to problems that grow exponentially with search depth. While searching a graph the search needs to decide which vertex to expand and which edge to take. Information that aims to answer this question is considered a heuristic. A heuristic might be based on some cost estimates between the current vertex and the goal vertex. [32]

A heuristic is a function that provides the necessary information that allows the algorithm to converge faster towards the goal. Only an admissible heuristic can lead to optimal results [18].

5.1.4 Optimality

One of the great advantages of graph search algorithms compared to other approaches for path planning is that many algorithms are proven to be optimal. Bellman's principle of optimality reads below.

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.* [2]

In essence any optimal solution for a problem that requires a sequence of decisions can only consist of optimal sub-solutions.

Lemma 5.1.1. *Given a weighted, directed graph $G = (V, E)$ with a cost function $g\colon E \to \mathbb{R}$ let $p = \{v_0, v_1, \ldots, v_k\}$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \{v_i, v_{i+1}, \ldots, v_j\}$ be the sub path of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.*

Proof. If we decompose the path p into $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, then we have that $g(p) = g(p_{0i}) + g(p_{ij}) + g(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with cost $g(p'_{ij}) < g(p_{ij})$. Then, $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}}$ is a path from v_0

[1] While C++ implements a basic priority queue in the *std* name space http://en.cppreference.com/w/cpp/container/priority_queue other more advanced queues (Binomial, Fibonacci, etc.) can for example easily implemented with a library such as *Boost* http://www.boost.org/doc/libs/1_60_0/doc/html/heap.html

to v_k, that has a lesser cost $g(p_{0i}) + g(p'_{ij}) + g(p_{jk})$ than $g(p)$, which contradicts the assumption that p is the shortest path from v_0 to v_k. [8] □

5.1.5 Admissibility and Consistency

Algorithms that find optimal paths on non-negative graphs are considered admissible [18]. Algorithms that are using heuristic estimates need to use heuristics that never overestimates the cost in order to be considered admissible [35]. It is clear, that the Euclidean distance between two points in R^2 is admissible since the shortest distance between two points in is a straight line.

Consistency of a heuristic implies, that the heuristic estimate for reaching the goal vertex from a vertex v must be less or equal to the estimate for v' plus the action cost, $h(v) \leq c(v, u, v') + h(v')$. This is a case of the triangle inequality, which constitutes that each side of a triangle cannot be longer than the sum of the other two sides. [35]

5.1.6 Completeness

Algorithms that find a solution if a solution exists or correctly report that there is no solution to the problem are considered complete. [26, 35]

5.2 Breadth First Search

Breadth First Search (BFS) was first developed by Moore and published by Lee in 1961 [27]. The BFS algorithm works on unweighted graphs (graphs with equal edge costs). In the original paper BFS is described by the author as "a computer model of waves expanding from a source under a form of straight-line geometry". BFS traverses a graph layer by layer, all vertices with depth k in the graph are visited before it proceeds to vertices with depth $k + 1$.

BFS does not consider edge costs, but only the number of expansions and can hence only be used on non-weighted graphs. On these graphs however BFS is complete and optimal, although the time and memory complexity is high, as the search is not guided [26, 27] .

BFS traverses all vertices V in a graph G until the algorithm terminates (e.g. due to meeting a goal condition). While the search progresses every vertex $x \in V$ changes its state from undiscovered to discovered. In order to trace the route discovered by BFS, a direction and hence a predecessor is assigned to each vertex. The start vertex will thus be the root of the resulting tree of successor. This is the key property that makes BFS a suitable candidate to solve shortest path problems [36]. Algorithm 2 depicts the structure of the search. BFS expands vertices in a *first in, first out* system, the closed list C is used to avoid unnecessary expansion of vertices that have previously been visited.

5.3 Dijkstra's or Uniform-Cost Search

While Breadth First Search delivers optimal solutions to the discrete path planning problem it does not consider edge cost and is hence in its original form only applicable to uniform cost graphs. Dijkstra's search algorithm can be seen somewhat of a refinement, even though first published two years earlier than

Algorithm 2 Breadth First Search

Require: $x_s \cap x_g \in X$
1: $O = \emptyset$
2: $C = \emptyset$
3: $Pred(x_s) \leftarrow null$
4: $O.push(x_s)$
5: **while** $O \neq \emptyset$ **do**
6: $x \leftarrow$ O.pop()
7: $C.push(x)$
8: **for** $u \in U(x)$ **do**
9: $x_{succ} \leftarrow f(x, u)$
10: **if** $x_{succ} \notin C$ **then**
11: **if** $x_{succ} \notin O$ **then**
12: $Pred(x_{succ}) \leftarrow x$
13: **if** $x_{succ} = x_g$ **then**
14: **return** x_{succ}
15: **end if**
16: $O.push(x_{succ})$
17: **end if**
18: **end if**
19: **end for**
20: **end while**
21: **return** null

BFS, in 1959. This section is also called *Uniform-Cost Search* as people such as Felner have pointed out that the original Dijkstra's algorithm was much closer to UCS than it is often conveyed in today's textbooks [15]. In his original paper Dijkstra indirectly refers to Bellman's principle of optimality, which is a needed proof for the algorithm being optimal.

Dijkstra's algorithm begins to divide all vertices into three sets, the closed set C, the open set O (implemented as a priority queue) as well as the remaining vertices. At the beginning C and O are the empty sets. After this the algorithm starts as depicted in algorithm 3 on the facing page [9]. First the start vertex x_s gets transferred to the open set. Next the while loop is entered, line 5, which either returns the goal vertex (line 9) or null if the O is the empty set. When a vertex gets expanded it is removed from O and added to C. Afterwards all edges connected to the vertex are evaluated and the vertexes they connect to. If any of these vertexes are not in C then we calculate the *cost-so-far* the same $g(x') = g(x) + l(x, u)$. Where $g(x)$ represents the *cost-so-far* for the vertex x from the start vertex and $l(x, u)$ the cost for the state transition from x to x' given the action u. If the resulting cost is lower than the current cost to reach that vertex or that vertex is not an element of O, the predecessor and the cost for that vertex will be set and the position in the priority queue will be decreased or it will added to O respectively. [8, 9, 26]

Algorithm 3 Dijkstra's Search

Require: $x_s \cap x_g \in X$
1: $O = \emptyset$
2: $C = \emptyset$
3: $Pred(x_s) \leftarrow null$
4: $O.push(x_s)$
5: **while** $O \neq \emptyset$ **do**
6: $x \leftarrow$ O.popMin()
7: $C.push(x)$
8: **if** $x = x_g$ **then**
9: **return** x
10: **else**
11: **for** $u \in U(x)$ **do**
12: $x_{succ} \leftarrow f(x, u)$
13: **if** $x_{succ} \notin C$ **then**
14: $g \leftarrow g(x) + l(x, u)$
15: **if** $x_{succ} \notin O$ **or** $g < g(x_{succ})$ **then**
16: $Pred(x_{succ}) \leftarrow x$
17: $g(x_{succ}) \leftarrow g$
18: **if** $x_{succ} \notin O$ **then**
19: $O.push(x_{succ})$
20: **else**
21: $O.decreaseKey(x_{succ})$
22: **end if**
23: **end if**
24: **end if**
25: **end for**
26: **end if**
27: **end while**
28: **return** null

5.4 A* Search

If one thinks of Dijkstra's search as a refinement of BFS then A* Search can be seen as a refinement of Dijkstra's work. While Dijkstra associated a *cost-so-far* g with each vertex of a graph to determine the next vertex to expand, A* enhances the algorithm by the use of a heuristic, allowing for much more rapid convergence under certain conditions, while still ensuring its optimality [18]. The heuristic $h(x)$ is the *cost-to-come*, based on an estimate of the cost from state x to the goal state x_g. Just as Dijkstra's algorithm A* also starts with O and C being the open and closed list respectively. Just as Dijkstra's algorithm A* the starts with a empty closed set C open set O. The search is depicted in algorithm 4 the key difference happens in line 18, here the heuristic estimate comes into play so that the cost for a state x is thus $f(x) = g(x)+h(x)$ by which the priority queue will be sorted. A standard heuristic estimate that can speed up search significantly, while maintaining admissibility and thus optimality, is the Euclidean norm for two dimensional problems.

Algorithm 4 A* Search

Require: $x_s \cap x_g \in X$
1: $O = \emptyset$
2: $C = \emptyset$
3: $Pred(x_s) \leftarrow null$
4: $O.push(x_s)$
5: **while** $O \neq \emptyset$ **do**
6: $x \leftarrow$ O.popMin()
7: $C.push(x)$
8: **if** $x = x_g$ **then**
9: **return** x
10: **else**
11: **for** $u \in U(x)$ **do**
12: $x_{succ} \leftarrow f(x, u)$
13: **if** $x_{succ} \notin C$ **then**
14: $g \leftarrow g(x) + l(x, u)$
15: **if** $x_{succ} \notin O$ **or** $g < g(x_{succ})$ **then**
16: $Pred(x_{succ}) \leftarrow x$
17: $g(x_{succ}) \leftarrow g$
18: $h(x_{succ}) \leftarrow Heuristic(x_{succ}, x_g)$
19: **if** $x_{succ} \notin O$ **then**
20: $O.push(x_{succ})$
21: **else**
22: $O.decreaseKey(x_{succ})$
23: **end if**
24: **end if**
25: **end if**
26: **end for**
27: **end if**
28: **end while**
29: **return** null

5.5 Hybrid A* Search

The hybrid A* algorithm[2] was successfully used in the DARPA Urban Challenge, a robot competition organized by the U.S. Government in 2007. In the following years insights to the algorithm were given by Dolgov et al. in [10–12,30]. The hybrid A* algorithm behaves similar to the A* algorithm. The key difference is, that state transitions happen in continuous rather than discrete space. One of the largest drawbacks of the previous approaches for path planning of non-holonomic robots is that the resulting paths are discrete and thus often not executable as changes of direction are sudden rather than smooth [12,30].

While the hybrid A* search implicitly builds the graph on a discretized grid, vertices can reach any continuous point on the grid. As a continuous search space would not be finite a discretization in form of grid cells is taken, limiting the grow of the graph. Since transitions from vertex to vertex have no predefined form it is easy to incorporate the non-holonomic nature in the state transition. The search space is usually three dimensional so that the state space X consists of x, y, θ, creating a discretized cuboid with the base representing the x, y position and the height the heading *theta* of a vertex.

Algorithm 5 on the following page outlines the steps in the hybrid A* search. Just as the ordinary A* search, it starts by defining the empty sets O and C, as well as by setting the predecessor state of the start state to *null* and pushing the start state on the open list, line 14. At line 18 the *while* loop starts, which only terminates if the open list is empty or the goal state has been reached, line 21 since a goal state might not be reached exactly the function *RoundState* in 21 is used to round both the current as well as the goal state before comparison. If the current vertex being expanded is no the goal vertex new successors will be generated for all available actions $u \in U(x)$, line 25. Is the successor is not in C, then the *cost-so-far* for the vertex are calculated. If the vertex is not in O or the *cost-so-far* are smaller than the cost for a vertex with the same index, that is also part of the O then the successor will be assigned a pointer to its predecessor, the *cost-so-far* and the *cost-to-come* will be updated. After that the vertex is pushed on the open list or the key is decreased using the new value $f(x_{succ})$. It is important to note, that even though hybrid A* is rounding the state in order to prune branches that are similar, the expansion will always happen from the actual value of the state as opposed to the rounded one.

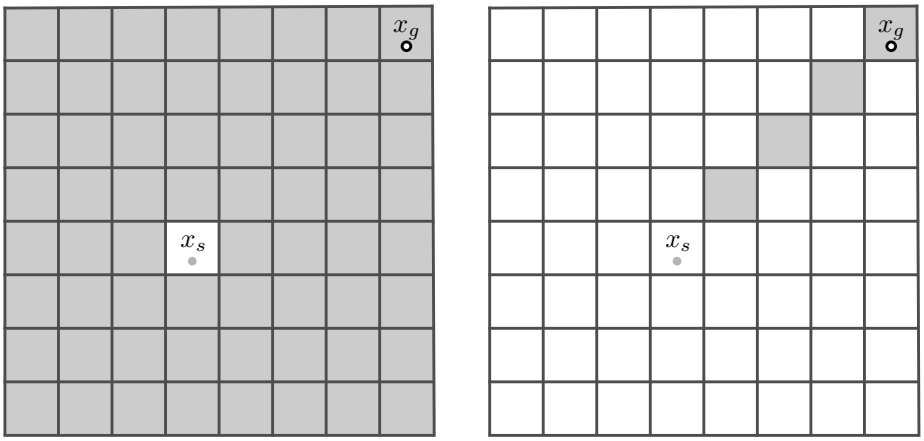
[2]The naming has not been consistent, variants are: hybrid A* and hybrid-state A*

Algorithm 5 Hybrid A* Search

function ROUNDSTATE(x)
    $x.Pos_X = \max\{m \in \mathbb{Z} \mid m \leq x.Pos_X\}$
    $x.Pos_Y = \max\{m \in \mathbb{Z} \mid m \leq x.Pos_Y\}$
    $x.Ang_\theta = \max\{m \in \mathbb{Z} \mid m \leq x.Ang_\theta\}$
    return x
end function

function EXISTS(x_{succ},$\mathcal{L}$)
    if $\{x \in \mathcal{L} \mid roundState(x) = roundState(x_{succ})\} \neq \emptyset$ **then**
        return true
    else
        return false
    end if
end function

Require: $x_s \cap x_g \in X$
$O = \emptyset$
$C = \emptyset$
$Pred(x_s) \leftarrow null$
$O.push(x_s)$
while $O \neq \emptyset$ **do**
    $x \leftarrow$ O.popMin()
    $C.push(x)$
    if $roundState(x) = roundState(x_g)$ **then**
        return x
    else
        for $u \in U(x)$ **do**
            $x_{succ} \leftarrow f(x, u)$
            if $\neg exists(x_{succ}, C)$ **then**
                $g \leftarrow g(x) + l(x, u)$
                if $\neg exists(x_{succ}, O)$ **or** $g < g(x_{succ})$ **then**
                    $Pred(x_{succ}) \leftarrow x$
                    $g(x_{succ}) \leftarrow g$
                    $h(x_{succ}) \leftarrow Heuristic(x_{succ}, x_g)$
                    if $\neg exists(x_{succ}, O)$ **then**
                        $O.push(x_{succ})$
                    else
                        $O.decreaseKey(x_{succ})$
                    end if
                end if
            end if
        end for
    end if
end while
return null



Dijkstra's Search A* Search

Figure 5.2: Dijkstra's Algorithm and A*

Part II

Method, Implementation and Results

Chapter 6

Method

The second part of this thesis presents the hybrid A* method, its implementation as well as simulation results in detail. This chapter will talk about the method used to solve the navigation problem.

The navigation of a mobile robot in the real as opposed to a discretized world, boils down to a complex, continuous-variable optimization problem. While there are a great variety of optimal and fast planners for discrete space (Dijkstra, A*), these planners tend to produce solutions that are not suitable for non-holonomic vehicles as they are not smooth and do not incorporate the vehicle constraints properly [11]. Using state lattice approaches these issues have been addressed, however this approach comes with the need of precomputing a huge amount of motion primitives to account for a multitude of different scenarios and issues of connecting the lattice appropriately [16, 17, 38]. Other approaches, such as RRT's, that are especially helpful for finding solutions in higher dimensional spaces, do produce continuous solutions but inherently come with disadvantages of being highly nondeterministic and converging towards solutions that are far from the optimum and suffer from bug trap problems [20]. The sub-optimality of RRT's has been addressed in [20], where a RRT* version that has proven asymptotic optimality is presented. Due to its basic properties, their application in path planning is only feasiable with numerous extensions to the original RRT (biasing, heuristics, etc.), Kuwata et al. demonstrated this in [22].

Building on A*, hybrid A* can be seen as the extension of an optimal, deterministic and complete algorithm. Hybrid A* is deterministic and due to the usage of admissible heuristics as well as path smoothing the solutions produced are in the neighborhood of the global optimum. The effective usage of heuristics decreases the expansion considerably and lets the search converge towards the solution quickly.

The hybrid A* planner can be split in three distinct parts. The hybrid A* search that incorporates the vehicles constraints, the heuristics that make the search well informed, allowing for fast convergence; and the path smoothing that improves the found solution using gradient descent.

For the evaluation of the developed algorithm different scenarios presented in chapter 8 on page 40 are used.

6.1 Hybrid A* Search

As described in section 5.5 on page 25 the hybrid A* (HA*) search expands vertices in continuous rather than discrete space. Even though it works in continuous space, HA* uses a discretized description of the world by pruning search branches that have similar leaf states. This is done in order to avoid growth of similar branches that add only very little to the solution, but vastly increase the size of the search graph.

A state is characterized by $\mathbf{x} = (x, y, \theta)$, where x and y denote the position and θ the heading of the vertex respectively. The action set U for a given vertex x can take any shape[1]. In order to adhere to the constraints imposed by a non-holonomic vehicle, a vertex is expanded by one of three actions; maximum steering left, maximum steering right as well as no steering. Each of this control actions is applied for a certain amount of time, resulting in an arc of a circle with a lower bound turning radius based on the vehicle constraints. This will ensure that the resulting paths are always drivable, as the actual vehicle model is used to expand vertices, even though they might result in excessive steering actions.

HA* does not take the velocity of the vehicle into account, but based on the solution of HA* an appropriate velocity profile can easily be calculated.

To incorporate the heading of the vehicle a finite three dimensional cuboid, which represents all possible states of the vehicle is used. During the expansions of vertices with the actions $u \in U(x)$ new states are generated. If a new state falls into a grid cell that is already occupied with another vertex and the new vertex has a lower *cost-so-far* the old vertex gets pruned (deleted). The search continues until a vertex reaches the goal grid cell, or all reachable cells have been reached and thus the open list is empty.

6.1.1 Vertex Expansion and Branch Pruning

The search starts with the current state of the vehicle, denoted as x_s. HA* will generate six successor vertices; three driving forward as well as three driving reverse, Figure 6.1 on page 32 depicts the expansion. The successors are generated by using arcs with the minimum turning radius of the vehicle[2]. The cost for the state transition is based on the length of the arc. Additional costs are accrued for changing driving directions, driving in reverse; and turning, as opposed to going straight. The penalty for turning as well as driving in reverse are multiplicative (depend on portion of the path turning or reversing), while the penalty for the change of driving directions is constant.

For each successor the following actions will be executed. If the successor vertex reaches a cell of the three dimensional cuboid that is not part of the closed list (meaning that cell has not yet been expanded) the evaluation continues. If the cell is not part of the open list (meaning the cell has not been reached prior by any other vertex expansion) or the *cost-so-far* from the predecessor vertex plus the cost for the vertex expansion to the successor reaching the cell is lower

[1] An opposing method is to use a state lattice, where a large amount of motion primitives connect cells always in a predefined manner.

[2] The arc length used for the expansion can be chosen arbitrarily, however a shorter length promises higher levels of resolution completeness, as the likelihood to reach each state is increasing.

than the *cost-so-far* of the vertex currently associated with that cell, then the new vertex will be assigned a pointer to its predecessor, the sum of the *cost-so-far* from its predecessor plus the cost for the expansion will be assigned to its g-value and the *cost-to-come* will be estimated using the heuristics and assigned to its h-value. If the a vertex with the same cell as the successor is on the closed or the open list and the successor's g-value is not lower, then the successor will be discarded, the branch will get pruned.

Under the assumption that the steering actions are either maximum steering right, no steering, or maximum steering left the arc length can be simply expressed as $r\left|x_\theta - x'_\theta\right|$, r being the minimum turning radius of the vehicle.

In case the arc length is shorter than the square root of the cell area, a vertex expansion can result in the successor arriving in the same cell. If this happens, it is insufficient to compare the vertices based on their *cost-so-far* as the cost of the new vertex will always be higher. Thus the comparison is based on the total estimated cost for both vertices. Since the algorithm is using consistent heuristics this leads to the effect, that the optimistic estimate will make vertices that are closer to the goal more expensive[3], hence a tie breaker is added to the predecessor to account for the consistent nature of the heuristic.Algorithm 6 illustrates this process. If the successor vertex is more expensive it will be discarded and the algorithm proceeds.

Algorithm 6 Same Cell Expansion

```
for u ∈ U(x) do
    x_succ ← f(x, u)
    if ¬exists(x_succ, C) then
        if RoundState(x) = RoundState(x_succ) then
            if f(x_succ) > f(x_x) + tieBreaker then
                delete x_succ
                continue
            end if
            Pred(x_succ) ← x
            g(x_succ) ← g
            h(x_succ) ← Heuristic(x_succ, x_g)
            if ¬exists(x_succ, O) then
                O.push(x_succ)
            else
                O.decreaseKey(x_succ)
            end if
        end if
    end if
end for
```

6.1.2 Analytical Expansion

The HA* planner sporadically calculates Dubins and Reeds Shepp curves from vertices currently being expanded to the goal. This is done partly as the ex-

[3]The total estimated cost of a vertex x is $f(x) = g(x) + h(x)$ and the successor cost $f(x) = g(x) + l(x, u) + h(x_{succ})$ using a consistent heuristic implies $h(x) \leq l(x, u) + h(x_{succ})$

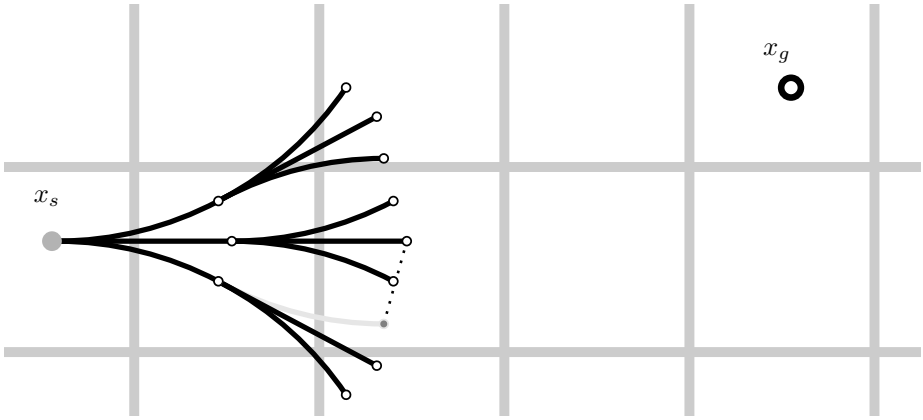


Figure 6.1: Vertex expansion and pruning

act continuous goal location is not reachable by the discretized control actions alone and in order to increase search speed. The calculated path is checked for collisions with the environment, if none exist the search terminates. In order to reduce computational load it is not beneficial to probe for these optimal point to point solutions from each vertex, but rather every n-th iteration (increasing the frequency while approaching the goal). Furthermore it is more reasonable to do so when approaching the goal or in a very obstacle sparse environment, as the likelihood for a collision is high otherwise, making this an expensive operation with little chance to payoff.

6.1.3 Collision Checking

While there are many ways to determine whether a configuration of the vehicle is collision free, $q \in \mathcal{C}_{free}$. The spatial occupancy enumeration approach presented in section 4.2 on page 16 is used for collision checking. In order to make this a viable solution it is mandatory to precompute possible configurations of the vehicle and save these in a lookup table. The advantage of this approach is that the collision checking can be conducted rapidly in constant time for any configuration of the vehicle.

As the lookup can easily be translated to a specific two dimensional cell with integer coordinates for x and y, only the different possible headings need to be taken into account. In order to compute spatial occupancy of a shape denoted by its corner points Bresenham's line algorithm can be used, but it does not correctly occupy all cells that the line between two points intersects with. For collision checking however, there needs to be certainty, thus a ray tracing algorithm described in [1] is used, that correctly marks all cells the line intersects with.

Once the basic bounding box is computed the lookup table gets filled with the spatial occupancy of the vehicle while rotating it with the θ discretization steps. In order to account for the hybrid nature of HA*, the fact, that the vehicle can reach any position within a cell, the spatial occupancy is also precomputed for 100 different positions within a grid cell.

6.2 Heuristics

While the goal is to produce drivable solutions that are approaching the optimum, it is important to make use of A* being an informed search, implementing heuristics allowing the algorithm converge quickly towards the solution. HA* is using estimates from two heuristics. As both of the heuristics are admissible the maximum of the two is chosen for any given state. The two heuristics capture very different parts of the problem, which is depicted in Figure 6.2 on page 37; the constrained heuristic incorporates the restrictions of the vehicle, ignoring the environment, while the unconstrained heuristic disregards the vehicle constraints and only accounts for obstacles.

6.2.1 Constrained Heuristic

The constrained heuristic takes the characteristics of the vehicle into account, while neglecting the environment. Suitable candidates are either Dubins or Reeds-Shepp curves. These curves introduced in section 3.6 on page 12 are the paths of minimal length with an upper bound curvature for the forward; and the forward as well as backward driving car respectively.

Since this heuristic takes the current heading as well as the turning radius into account it ensures, that the vehicle approaches the goal with the correct heading. This is especially important, when the car gets closer to the goal. For performance reasons this heuristic can be precomputed and stored in a lookup table. This is possible since it ignores obstacles and hence does not require any environment information. As it only improves the performance and not the quality of the solution a lookup table has not been implemented.

Given that both Dubins as well as Reeds-Shepp curves are minimal, this heuristic is clearly admissible.

6.2.2 Unconstrained Heuristic

The unconstrained heuristic neglects the characteristics of the vehicle and only accounts for obstacles. The estimate is based on the shortest distance between the goal node and the vertex currently being expanded. This distance is determined using the standard A* search in two dimensions (x, y position) with an Euclidean distance heuristic. The two dimensional A* search uses the current vertex as the goal vertex, and the goal vertex of the HA* search as the start vertex. This is beneficial, since the closed list of the A* search stores all shortest distances $g(x)$ to the goal and can thus be used as a lookup table, instead of initiating a new search while HA* progresses.

The unconstrained heuristic guides the vehicle away from dead ends and around u-shaped obstacles.

Since HA* can reach any point in a cell the unconstrained heuristic needs to be discounted by the absolute difference of the continuous coordinate of the current and the goal vertex.

6.3 Path Smoothing

As the paths produced by the hybrid A* algorithm are drivable, but often are made up of unnecessary steering actions it is beneficial to post process the result

with a smoother that attains a higher degree of comfort and safety [11, 12]. For this purpose a gradient descent smoother can be used that aims to minimize P consisting of the following four terms with respect to the path.

$$P = P_{obs} + P_{cur} + P_{smo} + P_{vor} \tag{6.1}$$

Each of the terms in this cost function has a special purpose that shall be explained in more detail.

6.3.1 Obstacle Term

This term penalizes collisions with obstacles. For all vertices $\mathbf{x}_i$ where $|\mathbf{x}_i - \mathbf{o}_i| \leq d_{obs}^{\vee}$ the cost P_{vor} is defined. It is based on the the distance to the next obstacle.

$$P_{obs} = w_{obs} \sum_{i=1}^{N} \sigma_{obs}(|\mathbf{x}_i - \mathbf{o}_i| - d_{obs}^{\vee}) \tag{6.2}$$

Where $\mathbf{x}_i$ is the x, y-position of a vertex on the path, $\mathbf{o}_i$ the location of the closest obstacle to $\mathbf{x}_i$. $d_{obs}^{\vee}$ acts as a threshold for the the maximum distance obstacles can affect the cost of the path. In order to penalize heavier when getting close to obstacles σ_{obs} is a quadratic penalty function. The obstacle weight w_{obs} is used to influence the impact on the change of the path.

Gradient

$$\frac{\partial \sigma_{obs}}{\partial \mathbf{x}_i} = \frac{2(|\mathbf{x}_i - \mathbf{o}_i| - d_{obs}^{\vee})\mathbf{x}_i - \mathbf{o}_i}{|\mathbf{x}_i - \mathbf{o}_i|} \tag{6.3}$$

6.3.2 Curvature Term

In order to ensure driveability the curvature term upper-bounds the instantaneous curvature of the path at every vertex. It is defined for $\frac{\Delta\phi_i}{|\Delta\mathbf{x}_i|} > \kappa_{max}$

$$P_{cur} = w_{cur} \sum_{i=1}^{N-1} \sigma_{cur} \left(\frac{\Delta\phi_i}{|\Delta\mathbf{x}_i|} - \kappa_{max} \right) \tag{6.4}$$

The displacement vector at the vertex $\mathbf{x}_i$ is defined as $\Delta\mathbf{x}_i = \mathbf{x}_i - \mathbf{x}_{i-1}$. The change in tangential angle at a vertex can be expressed by $\Delta\phi_i = \cos^{-1} \frac{\mathbf{x}_i \cdot \mathbf{x}_{i+1}}{|\mathbf{x}_{i+1}||\mathbf{x}_{i+1}|}$. The maximum allowable curvature is denoted by κ_{max}. Deviations from the maximum allowable curvature are penalized with a quadratic penalty function σ_{cur}. The curvature weight w_{cur} controls the impact on the change of the path.

Gradients

$$\frac{\partial \kappa_i}{\partial \mathbf{x}_i} = \frac{1}{|\Delta\mathbf{x}_i|} \frac{\partial \Delta\phi_i}{\partial \cos\Delta\phi_i} \frac{\partial \cos\Delta\phi_i}{\partial \mathbf{x}_i} - \frac{\Delta\phi_i}{{\Delta\mathbf{x}_i}^2} \frac{\partial \Delta\mathbf{x}_i}{\partial \mathbf{x}_i} \tag{6.5}$$

$$\frac{\partial \kappa_i}{\partial \mathbf{x}_{i-1}} = \frac{1}{|\Delta\mathbf{x}_i|} \frac{\partial \Delta\phi_i}{\partial \cos\Delta\phi_i} \frac{\partial \cos\Delta\phi_i}{\partial \mathbf{x}_{i-1}} - \frac{\Delta\phi_i}{{\Delta\mathbf{x}_i}^2} \frac{\partial \Delta\mathbf{x}_i}{\partial \mathbf{x}_{i-1}} \tag{6.6}$$

$$\frac{\partial \kappa_i}{\partial \mathbf{x}_{i+1}} = \frac{1}{|\Delta\mathbf{x}_i|} \frac{\partial \Delta\phi_i}{\partial \cos\Delta\phi_i} \frac{\partial \cos\Delta\phi_i}{\partial \mathbf{x}_{i+1}} \tag{6.7}$$

6.3.3 Smoothness Term

The smoothness term evaluates the displacement vectors between vertices. The result is that it assigns cost to vertices that are unevenly spaced as well as change direction. w_{smo} denotes the smoothness weight and hence the impact of the term on the change of the path.

$$P_{smo} = w_{smo} \sum_{i=1}^{N-1} (\Delta \mathbf{x}_{i+1} - \Delta \mathbf{x}_i)^2 \tag{6.8}$$

6.3.4 Voronoi Term

This term guides the path away from obstacles. For $d_{obs} \leq d_{vor}^{\vee}$ the cost P_{vor} is defined. It is based on the position of the node in the Voronoi field.

$$P_{vor} = w_{vor} \sum_{i=1}^{N} \left(\frac{\alpha}{\alpha + d_{obs}(x,y)} \right) \left(\frac{d_{vor}(x,y)}{d_{obs} + d_{vor}(x,y)} \right) \left(\frac{(d_{obs}(x,y) - d_{vor}^{\vee})^2}{(d_{vor}^{\vee})^2} \right) \tag{6.9}$$

The positive distance to the nearest obstacle is denoted by d_{obs}, d_{edg} is the positive distance to the nearest edge of the GVD. $d_{vor}^{\vee}$ represents the maximum distance obstacles affect the Voronoi potential. $\alpha > 0$ controls the falloff rate of the field and w_{vor}, the Voronoi weight, influences the impact on the path.

Gradients

$$\frac{\partial d_{obs}}{\partial \mathbf{x}_i} = \frac{\mathbf{x}_i - \mathbf{o}_i}{|\mathbf{x}_i - \mathbf{o}_i|} \tag{6.10}$$

$$\frac{\partial d_{edg}}{\partial \mathbf{x}_i} = \frac{\mathbf{x}_i - \mathbf{e}_i}{|\mathbf{x}_i - \mathbf{e}_i|} \tag{6.11}$$

$$\frac{\partial \rho_{vor}}{\partial d_{obs}} = \frac{\alpha d_{edg} \left(d_{obs} - d_{vor}^{\vee}\right) \left(\left(d_{edg} + 2d_{vor}^{\vee} + \alpha\right) d_{obs} + \left(d_{vor}^{\vee} + 2\alpha\right) d_{edg} + \alpha d_{vor}^{\vee} \right)}{{d_{vor}^{\vee}}^2 \left(d_{obs} + \alpha\right)^2 \left(d_{obs} + d_{edg}\right)^2} \tag{6.12}$$

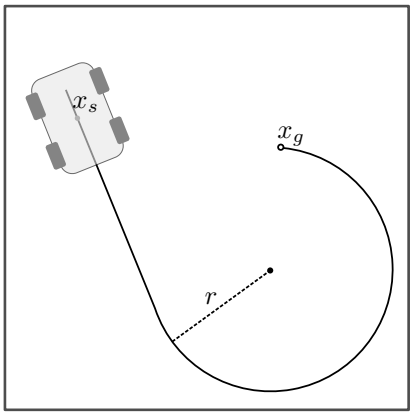
$$\frac{\partial \rho_{vor}}{\partial d_{edg}} = \frac{\alpha d_{obs} \left(d_{obs} - d_{vor}^{\vee}\right)^2}{{d_{vor}^{\vee}}^2 \left(d_{obs} + \alpha\right) \left(d_{edg} + d_{obs}\right)^2} \tag{6.13}$$

6.3.5 Gradient Descent

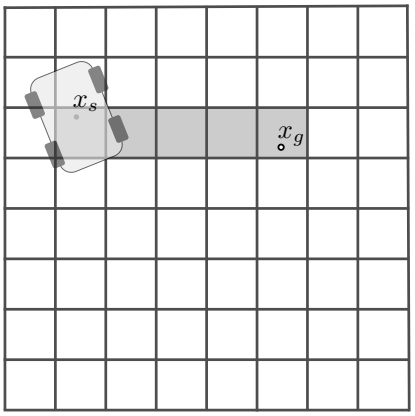
The gradient descent method is an optimization algorithm that uses the gradient of the function in order to find a local minimum. Gradient descent progresses stepwise with a step size proportional to the negative gradient of the function, $\Delta x = -\nabla f(x)$. While usual implementations use the absolute value of the gradient as a stopping criterion, a fixed number of iterations was chosen to ensure run-time consistency. [5]

Algorithm 7 Gradient Descent

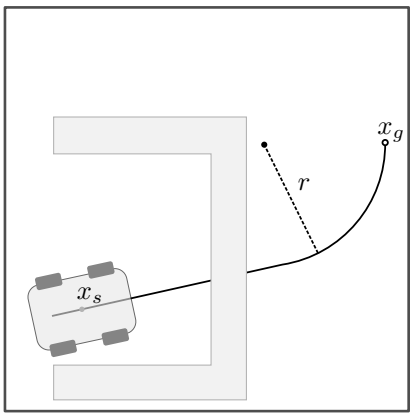
$iterations \leftarrow 1000$
$i \leftarrow 0$
while $i < iterations$ **do**
    for all $\mathbf{x} \in \mathcal{P}$ **do**
        $cor \leftarrow (0, 0)$
        $cor \leftarrow cor - obstacleTerm(\mathbf{x}_i)$
        $cor \leftarrow cor - smoothnessTerm(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{x}_{i+1})$
        $cor \leftarrow cor - curvatureTerm(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{x}_{i+1})$
        $cor \leftarrow cor - voronoiTerm(\mathbf{x}_i)$
        $\mathbf{x}_i \leftarrow \mathbf{x}_i + cor$
    end for
    $i \leftarrow i + 1$
end while
return null



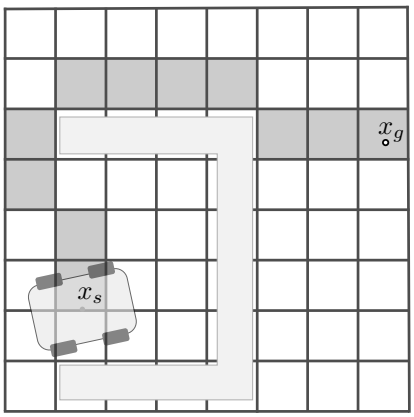
(a) The constrained heuristic accounting for the goal heading.



(b) The unconstrained heuristic heavily underestimating the cost of the path, due to the wrong heading.



(c) The constrained heuristic heavily underestimating the cost of the path, due to ignoring obstacles.



(d) The unconstrained heuristic accounting for obstacles and dead ends.

Figure 6.2: A comparison of the benefits of the constrained as well as unconstrained heuristic

Chapter 7

Implementation

The development of the algorithm is conducted in simulation. Since the algorithm only requires a binary obstacle grid as an input to compute the output, the poses of the path, the implementation behaves in the same way in simulation as in real driving experiments.

As the path planning algorithm is developed for a real vehicle, the RCV, it does require fast computation of paths. The RCV's top speed is around 12 m/s, the theoretical lower limit of the planner for paths of 30 meters length would thus be around 2.5 s for a full replanning cycle, avoiding stops due to computation. Since the length is not the only factor that determines the complexity of the search a much higher frequency needs to be attained.

The developed hybrid A* planner is able to plan a path from an initial state x_s to a final state x_g with a frequency of approximately 3–10 Hz[1] using medium grade consumer hardware (Intel Core i5-5200U 2.2 GHz), making the planner a viable option for real world driving in unstructured environments.

7.1 ROS

The algorithm is wrapped with ROS (Robot Operating System). ROS is chosen as it facilitates the interaction between different modules and hence the deployment in the actual vehicle. At its core, ROS uses a standard messaging system, that allows to publish and listen to different topics between modules that provide different functionality, so that the output of one can be easily fed into another. Another benefit of ROS are transforms. Transforms are timestamped coordinate frames that automatically conduct necessary coordinate transformations between modules publishing and listening in different coordinate frames.

7.2 Structure

Figure 7.1 on the facing page depicts the structure of the program. The inputs are the obstacle grid as well as the goal pose. During program initialization a collision lookup table as well as a obstacle distance lookup are generated.

[1] Assuming a 50 m × 100 m grid with a cell size of 1 m and heading discretization of 5°, which results in 360,000 different possible cells

Once a occupancy grid and a valid goal pose have been received the hybrid A* search begins. To calculate the constrained heuristic Reeds Shepp curves are calculated. For the unconstrained heuristic a 2D A* search is conducted. In addition to that the algorithm sporadically creates an analytical solution with Reeds Shepp curves and if it is collision free terminates and passes the found path to the smoother. In the smoother the path is optimized for distance based on closeness to obstacles as well as smoothness, using only the terms P_{obs} and P_{smo} due to time constraints and non trivial implementation issues of the Voronoi as well as curvature term. Once the gradient descent terminates the path is published via ROS to the next module–the controller.

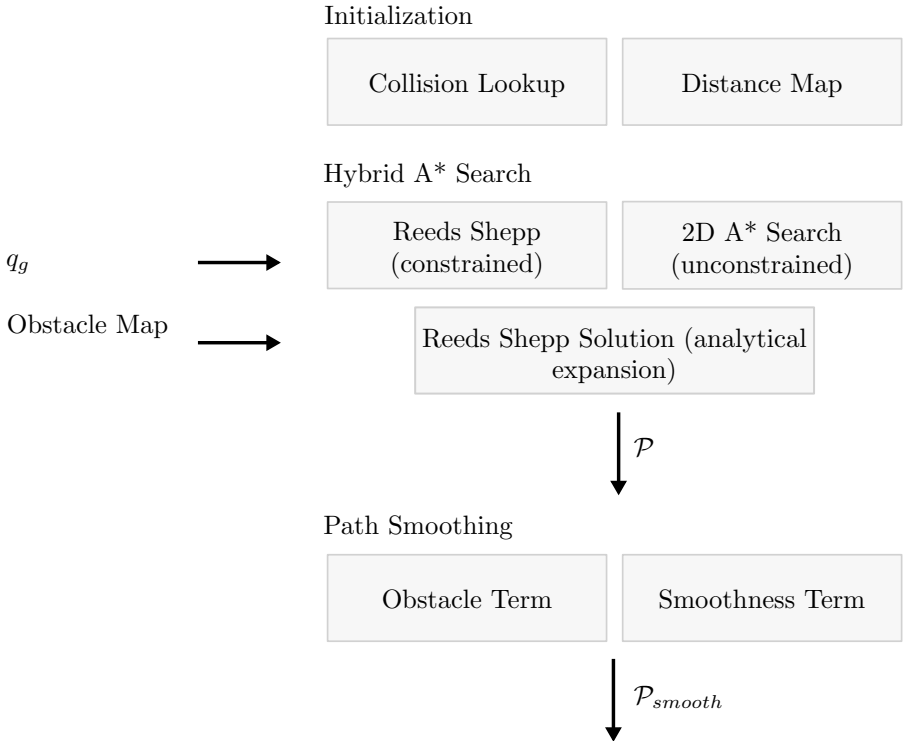


Figure 7.1: Structure of the program with its inputs and outputs

Although the current implementation is useful to conduct experiments with, it can be improved with regard to runtime considerably.

The constrained heuristic can be completely precomputed as it does not account for obstacles. For that purpose the Reeds Shepp distance from all cells in a specified area around the goal to the goal can be saved in a lookup table.

While a lookup table cannot be created before run-time for the unconstrained heuristic. The two dimensional A* search can be changed in a way, that allows for multiple queries. Currently the heuristic stores all closed cells for future queries. However, once a cell gets entered by HA* for which the two dimensional cost are unknown a new A* search from the goal to that cell gets initiated visiting cells that have already been discovered during a previous search.

Chapter 8

Results and Discussion

For the test of the algorithm different scenarios have been chosen and simulations were conducted to test the efficiency, accuracy and driveability of the solutions provided by HA*. The scenarios chosen for the simulation describe common problems path planning algorithms need to be able to overcome in order to be usable for the navigation in unstructured environments.

The following scenarios were simulated and are analyzed in the following. Each scenario is different with regard to its obstacle configuration and tests different capabilities. The term Euclidean refers to the use of an unconstrained Euclidean distance heuristic; 2D A* refers to the unconstrained shortest distance heuristic, taking obstacles into account; Dubins refers to the constrained shortest Dubins curve heuristic, limited to one driving direction; and Reeds Shepp refers to the constrained shortest Reed Shepp curve heuristic allowing for two driving directions.

Dark grey areas are free space. Black areas are obstacles. Yellow paths represent the different configurations of the vehicle while traversing the path and are used to visually check paths for collisions. Pink arrows represent a forward motion of the vehicle, while purple arrows correspond to a reverse motion. If the path is not accompanied by either pink or purple arrows then the analytical expansion has directly connected the most promising current state and the goal state. Yellow dots represent the area the 2D A* search has visited. A gradient of colors represent the relative cost estimate of each cell among all cells of the 2D A* search.

8.1 Simulation Results

8.1.1 Parking Structure

The size of the binary obstacle grid for the parking structure scenario is 50 m $\times$ 100 m. The parking structure is horizontally divided by obstacles with two small passages connecting both sides. The passages have a width of 4 meters. In the scenario a path needs to be found from the initial state x_s (in the lower half), facing north to a final state x_g (in the upper half) facing south.

Analysis

Figure 8.1a on page 45 depicts the planners result using only a Euclidean heuristic. It can be seen that the planner finds a non colliding path that is close to the shortest possible path. The planner has searched the area around the start position with a bias towards the goal position. The solution path lies in the opposite direction of the search focus.

The found path in Figure 8.1b on page 45 is identical, however the 2D A* heuristic is used. The search progresses away from the goal biased towards the passage on the left.

Figure 8.1c on page 45 shows the area explored by 2D A* and the colors represent the relative cost of each cell among all explored cells. Red areas are associated with the highest cost, blue areas with the lowest cost. When looking at the color gradient it can be seen that the found path follows the gradient very closely. Only at the beginning, when the vehicle is backing up the costs are increasing.

Discussion

Since the Euclidean heuristic only accounts for relative distance between two points and has no information with regard to the obstacles in the environment it heavily underestimates the cost to the goal and drives the search in the wrong direction expanding almost one order of magnitude more vertices than the 2D A* heuristic, which gives a clear picture of the environment. Since the 2D A* heuristics is much more informed it underestimates the true cost much less, and hence avoids wasteful expansion of unpromising vertices. The found path in Figure 8.1b on page 45 follows the cost gradient in most parts, but since 2D A* does not account for the non-holonomic nature of the vehicle, it does not account for the vehicle's need to back up out of initial parking position.

8.1.2 Obstacles

The size of the binary obstacle grid for the obstacles scenario is $50\,\mathrm{m} \times 100\,\mathrm{m}$. The environment is randomly scattered with quadratic obstacles of 1 m and 2 m width. The initial state x_s (left) as well as the goal state x_g (right) face west.

Analysis

Figure 8.2a on page 46 shows the path found using the 2D A* heuristic. The search expands vertices in a more or less straight forward manner towards the goal. In roughly the middle of the grid the analytical expansion finds a solution and connects collision free to the goal.

Adding the Dubins heuristic in Figure 8.2b on page 46 the search expands less vertices at the beginning, especially less reversing states. Afterwards the search continues towards the goal, but not using the shortest Euclidean path. Much later than 2D A* the search ends due to a successful analytical expansion.

Using the Reeds Shepp heuristic, in Figure 8.2c on page 46 the search searches a much larger amount of states in the vicinity of the initial state. While the search progresses towards the goal, it is noticeable that the search expands states in a circular fashion along the path, until the analytical expansion connects to the goal.

Discussion

Since the 2D A* heuristic does not account for the non-holonomic constrains of the vehicle it guides the search similar as a Euclidean heuristic would straight to the goal until the Dubins path of the analytical expansion corrects for the behavior. The Dubins heuristic expands nodes keeping the goal heading of the vehicle in mind, essentially restricting the search to an area along a Dubins curve. While the heuristic expands much more nodes than the 2D A* only version, this is due to the fact, that obstacles continuously block a direct connection for the path chosen by the Dubins heuristic. Only at last the analytical expansion succeeds. For the Reeds Shepp heuristic the case behaves slightly different. Since the heuristic calculates the cost under the assumption that the vehicle can drive in both directions it allows for much more reverse states to be expanded. Along the path towards the goal the Reeds Shepp heuristic continuously tries to change the heading of the car by expanding states that deviate from the path in a circular fashion.

8.1.3 Wall

The size of the binary obstacle grid for the wall scenario is 50 m × 100 m. The environment consits of a wall obstacle of 6 m width. The initial state x_s (left) faces east and the goal state x_g (right) faces north.

Analysis

Figure 8.3 on page 47 shows the path found using the Euclidean heuristic. The search progresses in a straight line towards the wall, until the analytical expansion finds a non colliding solution.

Using the 2D A* heuristic in Figure 8.3b on page 47 the search is biased to the area above the wall. The path connects with an analytical expansion.

Figure 8.3c on page 47 shows the resulting path using the Reeds Shepp heuristic. The search progresses early on to the area below the wall until the analytical expansion finds an appropriate solution.

Discussion

Since the Euclidean heuristic is oblivious to the wall, the search progresses straight towards the goal. Due to the analytical expansion the search ends early otherwise the search would have expanded cells that are located in the area top of the wall and possibly connected via a Reeds Shepp path instead of a Dubins. In the second figure, the 2D A* heuristic does account for the wall, but does not model the non-holonomic constraints appropriately, hence the search diverges to the wrong direction increasing the path length unnecessarily. The Reeds Shepp heuristic on the other hand, even though not aware of the wall, incorporates the non-holonomic characteristics and tries to approach the goal with the right heading from the start, leading to an overall much shorter path.

8.1.4 Dead End

The size of the binary obstacle grid for the dead end scenario is 50 m × 100 m. The environment consist of a large u-shaped obstacle. The initial state x_s (left)

faces east, while the goal state x_g (right) faces north.

Analysis

Figure 8.4 on page 48 shows the path found using the Euclidean heuristic. The search expands a large amount of the available cells and searches the inner part of the u-shaped obstacle before it searches outside of the same. Shortly before the search reaches the goal the analytical expansion completes the path.

The 2D A* heuristic in Figure 8.4b on page 48 barely searches the inner part of the obstacle. It quickly searches around it and creates a solution passing the lower part and then converging straight towards the goal, until the analytical expansion completes the path.

Adding the Reeds Shepp heuristic in Figure 8.3c on page 47 the search expands much less cells in the area above and below the obstacle and eventually connects via analytical expansion.

Discussion

A u-shaped obstacle or dead end is a tough hurdle to take for the Euclidean heuristic that is not aware of its surroundings. The Euclidean heuristic heavily underestimates the cost and gets stuck inside the obstacle until the cost of the solutions so far increase to a level that justifies the detour below the obstacle. While the resulting path is approaching the optimal the number of vertices expanded is unacceptable for a real-time implementation. Using the 2D A* heuristic the problem gets circumvented as it is aware of the dead end ahead. The solution quickly converges towards the goal around the obstacle. However as 2D A* does not take the heading into account the angle at which it approaches the goal is inadequate for a good analytical expansion; the result is an unnecessary circle for the correction of the heading. Taking the heading into account, the addition of the Reeds Shepp heuristic further reduces the number of vertices expanded, and thus considered optimal. While the analytical expansion finds a solution prior to leaving the obstacle behind Reeds Shepp would have approached the obstacle in the right way either way.

8.1.5 Additional Simulations

In order to test the HA* planner for turning as well as parking maneuvers two additional simulations are conducted.

Analysis

Figure 8.5 on page 49 shows HA* finding a path in a 5 m wide alley allowing the vehicle to change heading by 180 degrees. The turning is conducted in a star shape, where the vehicle reverses with the maximum steering angle right, changes to the maximum steering angle left and drives forward and repeats the process. For the parallel parking case a similar behavior can be observed as depicted by Figure 8.6 on page 49.

Discussion

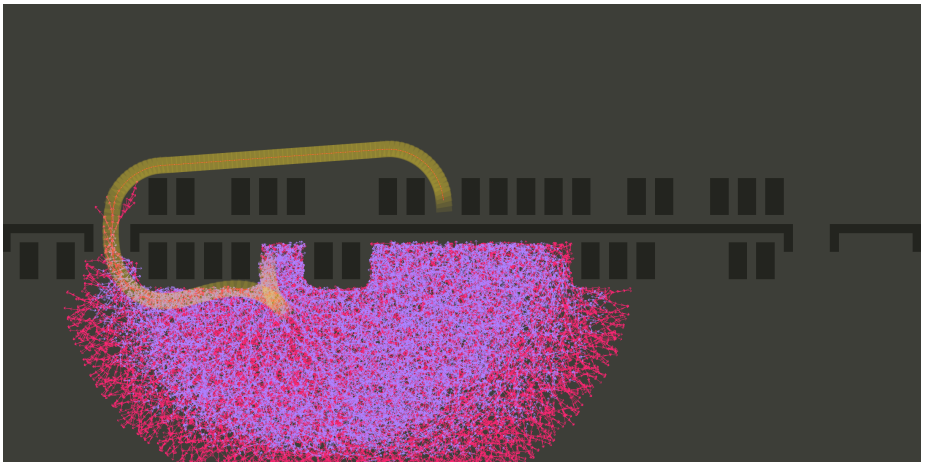
HA* does solve both problems adequately. In these cases the heuristics have no noticeable impact on the found paths, as the paths are mainly constrained by the collisions with obstacles. While the vehicle is able to turn around in the alley scenario with 5 m width, it cannot do so for an alley of 4 m width. This is mainly do to the fact that the motion primitives for the expansion of the vertices have a constant length and would continuously yield collisions, exhausting all available options quickly, without generating a solution. In the parallel parking scenario the solution can be attained as the goal position is not in the middle allowing it sufficient space in one direction to do repeated revers as well as forward expansion.

8.2 Real-world Results

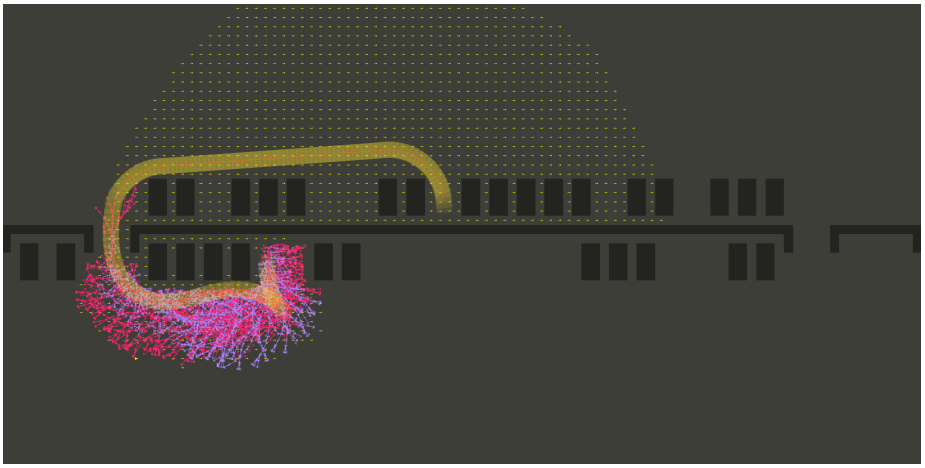
Due to a number of problems related to the RCV real-world experiments were not conducted in a organized fashion. While the planner has been used and tested on the RCV interfacing with the inputs as well as outputs, and generating satisfying paths, which a pure pursuit controller followed, it was not possible to actually setup an experiment to conduct the capabilities in real-world open space.

8.3 Conclusion

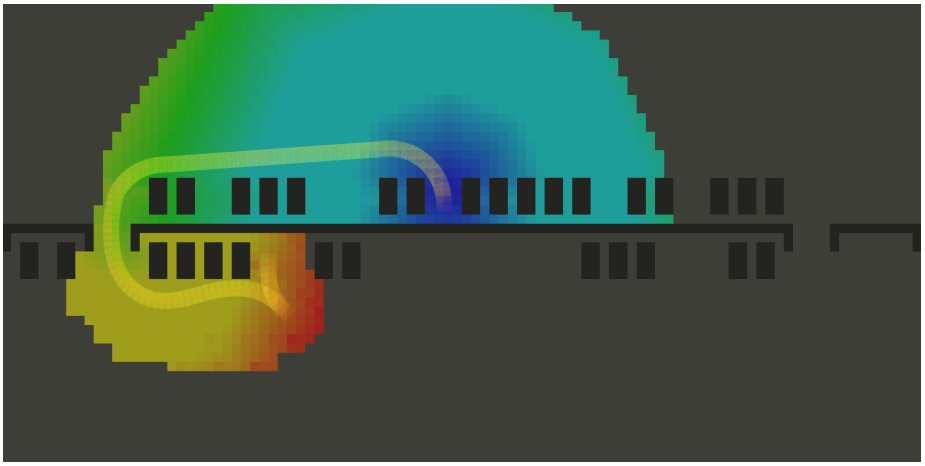
During all tested scenarios HA* has shown its ability to find a path. Since HA* accounts for the turning radius of the vehicle the paths found are drivable. With the use of well informed heuristics that model the vehicle constraints as well as capture the environment, HA* can make well informed decisions, allowing it to converge to the goal quickly. The analytical expansion heavily increases the speed of HA*, while keeping the constraints in mind. The post processing of the paths further improves the path locally. If the implementation of the curvature term would be complete, then the solution would attain the global optimum as, the weight of the smoothness term could be increased while ensuring an upper bound curvature.



(a) Euclidean, 47559 vertices, length x m

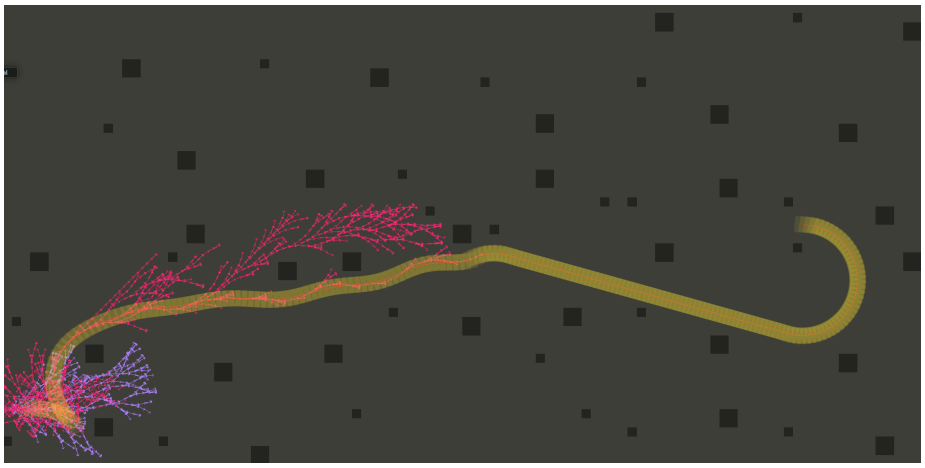


(b) 2D A* & Reeds Shepp, 4767 vertices (1517 vertices), length x m

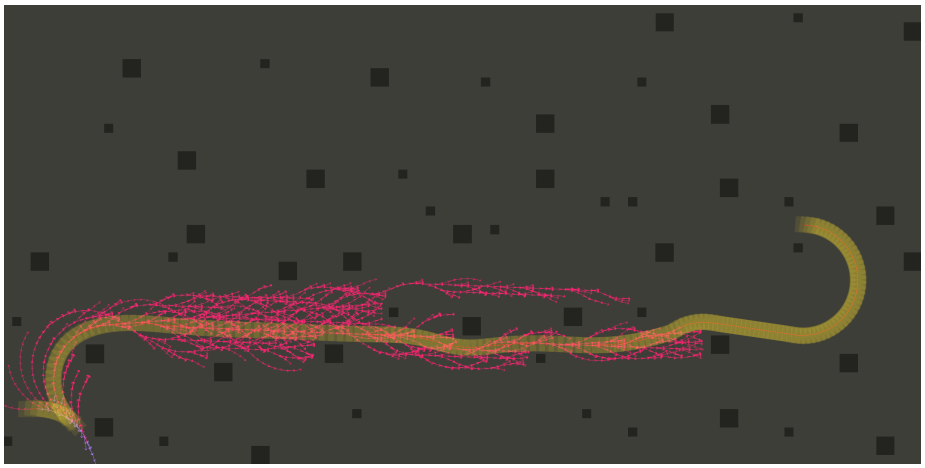


(c) 2D A* & Reeds Shepp, 4767 vertices (1517 vertices), length x m

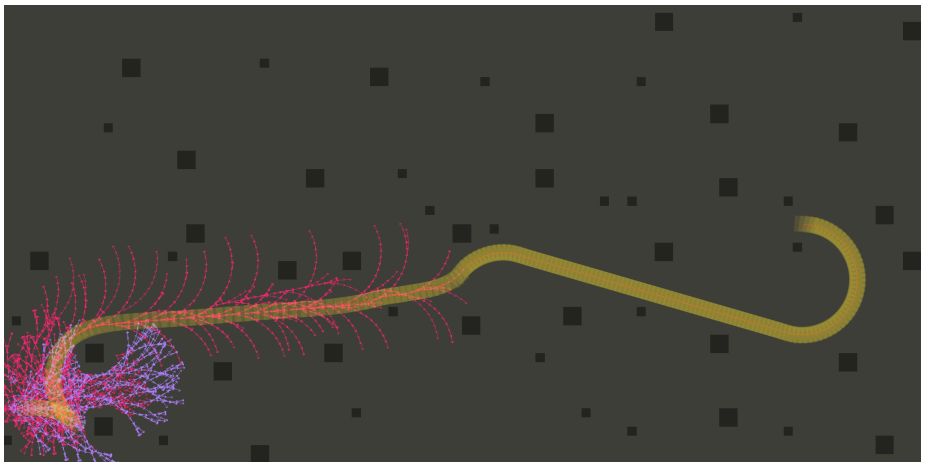
Figure 8.1: The parking structure scenario



(a) 2D A*, 2685 vertices, length 116.882 m

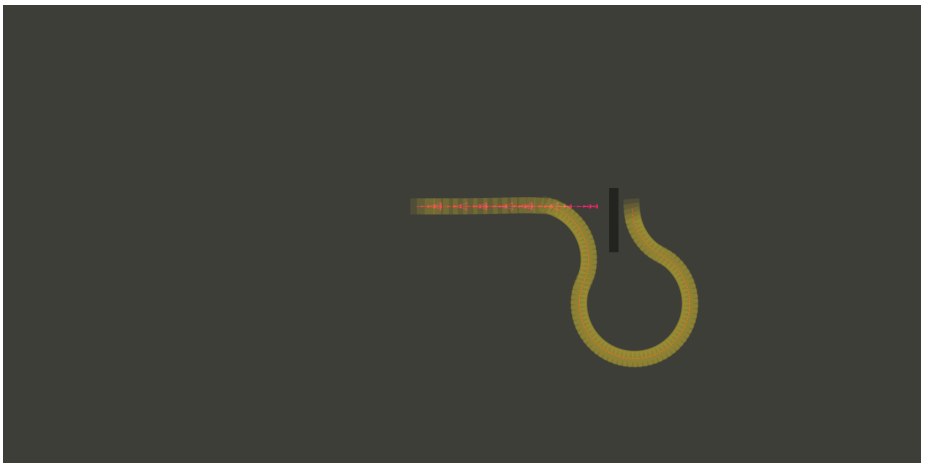


(b) 2D A* & Dubins, 3486 vertices, length 114.961 m



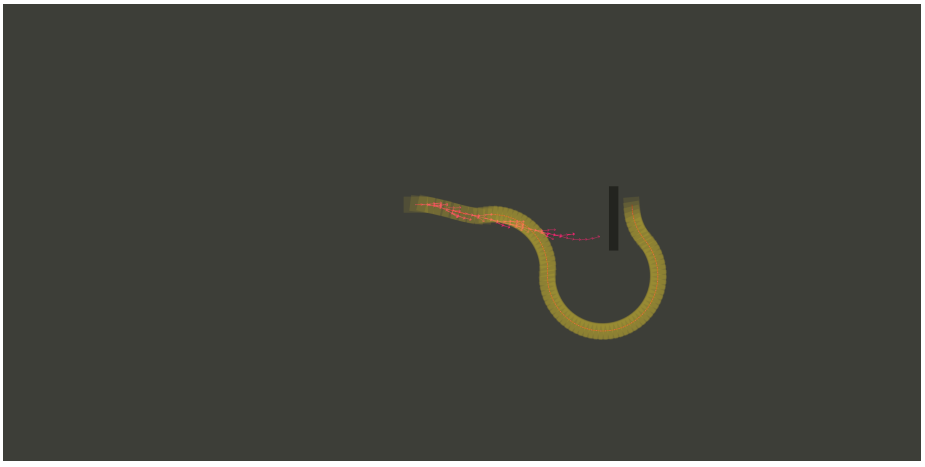
(c) 2D A* & Reeds Shepp, 4316 vertices, length 116.647 m

Figure 8.2: The obstacle scenario



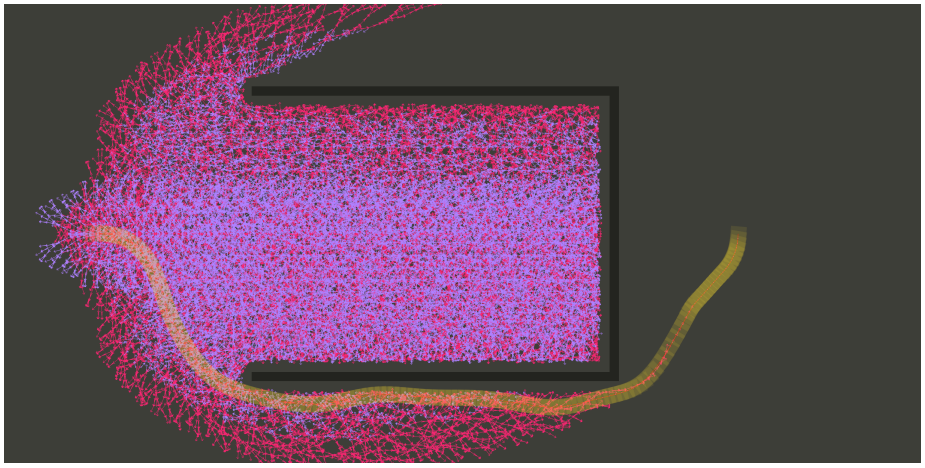
(a) Euclidean, 237 vertices, length 58.224,m

(b) 2D A*, 129 vertices, length 58.914 m

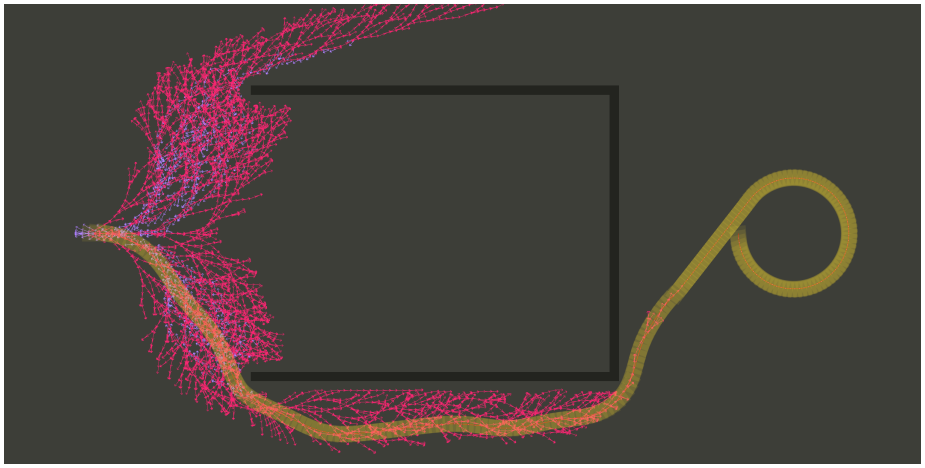


(c) Reeds Shepp, 124 vertices, length 45.780 m

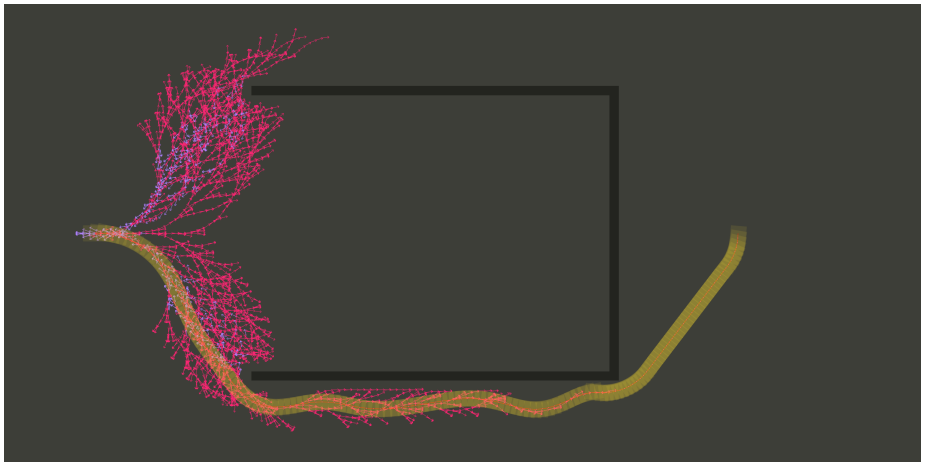
Figure 8.3: The wall scenario



(a) Euclidean, 72014 vertices, length 90.0001 m



(b) 2D A*, 8871 vertices, length 128.197 m



(c) 2D A* & Reeds Shepp, 8691 vertices, length 90.9778 m

Figure 8.4: The dead end scenario

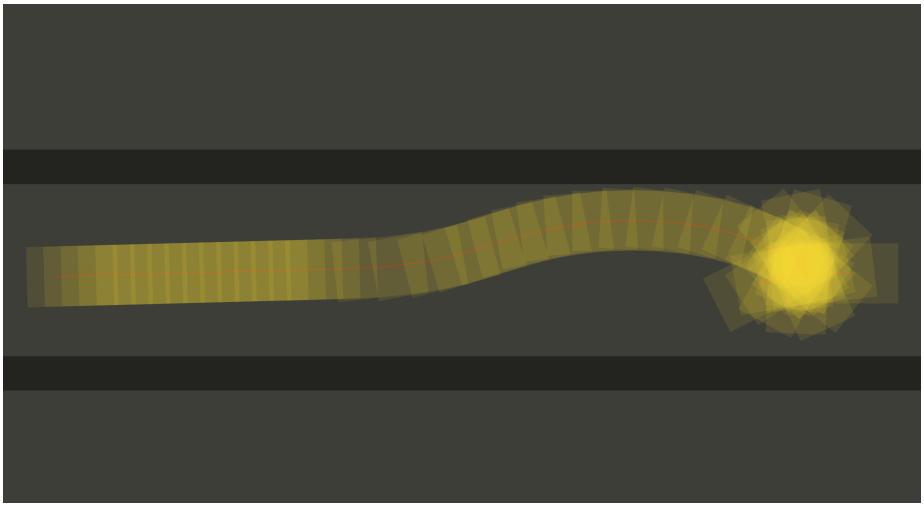


Figure 8.5: Turning on the spot for change of directions

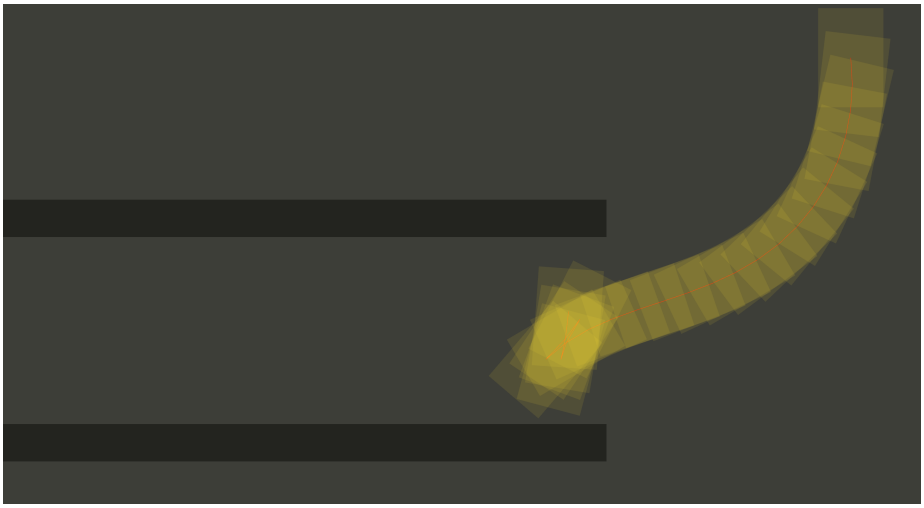


Figure 8.6: Parallel parking the car

Chapter 9

Conclusion

While the research in the area of autonomous driving continuous to grow, the DARPA Grand Challenge and especially the Urban Challenge have marked important milestones, greatly propelling the development. Hence it is not astonishing that companies heavily engaged in autonomous driving, such as Alphabet (formerly Google) or Uber recruited a large part of past participants of these challenges.

Since there are many ways to solve navigation problems of this kind the choice might seem arbitrary. While the team from CMU and winner of the DUC used a lattice planner with D* [16, 28], the MIT team used an heavily modified version of RRT both for structured as well as unstructured driving [22].

The hybrid A* algorithm is yet another approach. It is a fast planner used for planning in unstructured environments by the Standford team, developed by Dolgov, Thrun, Montemerlo (Google Self-Driving Cars), Diebel (Cedar Lake Ventures), participating with Junior in the DUC. This thesis has analyzed HA* thoroughly and developed a similar version for the KTH RCV in `C++` and ROS.

The resulting hybrid A* planner addresses the problem of finding a solution to the problem described in section 1.3 on page 3 properly. The planner models the non-holonomic nature of the vehicle in all stages of the process, vertex expansion, heuristic estimates, as well as analytic expansion. Thus, the most important characteristic of the paths is given–they are driveable.

The HA* planner solves a challenging problem in an elegant manner. HA* is fast, as it reduces the search space with the help of well informed heuristics, allowing it to converge to the goal quickly. Based on the initial solution the local smoothing attains a solution approaching the global optimum. Ideal scenarios for the planner are slow speed driving in unstructured environments. An example of that might be navigating parking lots as well as automated parking and valet parking.

The source code for this project is publicly available and can be found here, `https://github.com/karlkurzer/path_planner`; an additional documentation can be found here `http://karlkurzer.github.io/path_planner`.

Chapter 10

Future Work

While the algorithm developed in this thesis has already been successfully tested on the RCV there are still aspects that are reasonable to consider and would either improve the search speed or the quality of the overall solution. In the following the most promising improvements are shortly presented.

10.1 Variable Resolution Search

HA* uses a constant cell size and constant arc length for the vertex expansion. Usually it is desirable to increase the resolution as much as computationally possible. This will ensure greater notions of completeness as well as optimality. Completeness will be improved since a coarse resolution underestimates the free space C_{free}, making narrow passages appear not traversable without collision. A higher resolution will also create solutions that are closer to the optimum, as opposed to coarse resolutions where the solution oscillates around the optimum. In terms of computational efficiency a variable resolution search can greatly reduce the expansions necessary to converge towards the goal.

The two possibilities to make improvements in the areas of completeness as well as computational efficiency are either locally changing the arc length for the vertex expansion or the resolution of the cells. While locally changing the cell resolution requires a different data structure representing the cells it is easier to increase the arc length used for the vertex expansion. The arc length can be based on the size of the free space it expands to. A convenient representation of this can be achieved using the Voronoi diagram.

10.2 Heuristic Lookup Table

During the HA* search the constrained heuristic constantly evaluates the shortest distance from the vehicles current configuration to the goal configuration under the assumption that no obstacles exists and the car is constrained by a lower bound turning radius. As this heuristic is not dependent on sensor information it can be completely precomputed and retrieved during run-time.

10.3 Velocity Profile Generation

A reasonable addition to the HA* planner is the calculation of a velocity profile. Based on the smoothed path a reference velocity could be calculated in different ways. A simple approach would be an upper bound lateral acceleration a_y of the vehicle based on the curvature of each segment of the path. For more insights with regard to the lateral dynamics of the path at given velocities a linear single track model can then be simulated along the path.

10.4 Path Commitment and Replanning

As the vehicle is progressing towards the goal the sensors will detect new obstacles in the environment that have previously been out of range or covered by other obstacles. In order to incorporate possible changes in the environment in the path planning HA* recomputes the path to the goal with every update of the environment it receives.

This is often not necessary. Given the fact, that the sensor information in the vicinity of the car is of high quality replanning, if at all, does not need to start at the vehicles current position. A path commitment can significantly reduce the planning effort. Commitment to the path means that as new sensor information arrives the path will not be replanned at the vehicle's current position, but rather at the n-th vertex that is still in an area where the sensors reliably perceive the environment.

As planning an entire path takes considerably more time than merely checking the same for collisions one has to ask, whether it is feasible to replan a path as opposed to just check the current path for collisions given the updated model of the environment.

Another aspect are dynamic obstacles that temporarily collide with the path. The current version of HA* cannot distinguish between dynamic and static obstacles, as a dynamic obstacle, like a static obstacles, only leaves a binary footprint on the occupancy grid. This will leave the planner to believe that replanning is the right way to solve this problem. While for the static case it would be reasonable to replan based on the new information it is undesirable for dynamic obstacles since the planner does not account for the velocity of the obstacle and hence does not guarantee the path being safe.

Bibliography

[1] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. 32

[2] Richard Bellman. *Dynamic programming*. Dover Publications, Mineola, N.Y., dover ed. edition, 2003. 20

[3] Michele Bertoncello and Dominik Wee. Ten ways autonomous driving could redefine the automotive world, 2015. 3

[4] Béla Bollobás. Fundamentals, 1979. 18

[5] Stephen P. Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, Cambridge, 2004. 35

[6] Rodney A. Brooks and Tomás Lozano-Pérez. A subdivision algorithm in configuration space for findpath with rotation. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(2):224–233, 1985. 11

[7] B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *Journal of the ACM (JACM)*, 34(1):1–27, 1987. 15

[8] Thomas H. Cormen. *Introduction to algorithms*. MIT Press, Cambridge, Mass., 3rd ed. edition, 2009. 21, 22

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. 22

[10] Dmitri Dolgov and Sebastian Thrun. Autonomous driving in semi-structured environments: Mapping and planning. pages 3407–3414, 2009. 25

[11] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical search techniques in path planning for autonomous driving. 2008. 25, 29, 34

[12] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Path planning for autonomous vehicles in unknown semi-structured environments. *The International Journal of Robotics Research*, 29(5):485–501, 2010. 25, 34

[13] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79(3):497, 1957. 13

[14] Christer Ericson. *Real-time collision detection*. Morgan Kaufmann series in interactive 3D technology. Elsevier, Amsterdam and London, 2005. 15, 16, 17

[15] Ariel Felner. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. 2011. 22

[16] Dave Ferguson, Thomas M. Howard, and Maxim Likhachev. Motion planning in urban environments part i. pages 1063–1069, 2008. 29, 50

[17] Dave Ferguson, Thomas M. Howard, and Maxim Likhachev. Motion planning in urban environments part ii. pages 1070–1076, 2008. 29

[18] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. 20, 21, 24

[19] V. Hayward. Fast collision detection scheme by recursive decomposition of a manipulator workspace, 1986. 16, 17

[20] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In Yoky Matsuoka, Hugh F. Durrant-Whyte, and José Neira, editors, *Robotics*, volume 06, Cambridge, MA, 2011. MIT Press. 29

[21] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986. 10

[22] Yoshiaki Kuwata, Gaston A. Fiore, Justin Teo, Emilio Frazzoli, and Jonathan P. How. Motion planning for urban driving using rrt. 2008. 29, 50

[23] Jean-Claude Latombe. *Robot Motion Planning*, volume 124 of *The Springer International Series in Engineering and Computer Science, Robotics: Vision, Manipulation and Sensors*. Springer US, Boston, MA, 1991. 7, 8, 9, 11, 12

[24] S. M. Lavalle. Rapidly-exploring random trees: a new tool for path planning. 1998. 10

[25] S. M. Lavalle and J. J. Kuffner. Randomized kinodynamic planning. 1:473–479 vol.1, 1999. 10

[26] Steven Michael LaValle. *Planning algorithms*. Cambridge University Press, Cambridge, 2006. 7, 8, 9, 11, 12, 15, 16, 19, 20, 21, 22

[27] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961. 21

[28] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. 50

[29] T. Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, 1983. 8

[30] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008. 25

[31] Xavier Mosquet, Thomas Dauner, Nikolaus Lang, Michael Russmann, Antonella Mei-Pochtler, Rakshit Agrawal, and Florian Schmieg. Revolution in the driver's seat: The road to autonomous vehicles. 3

[32] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: symbols and search. *Communications of the ACM*, 19(3):113–126, 1976. 20

[33] M. K. Ponamgi, D. Manocha, and M. C. Lin. Incremental algorithms for collision detection between polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):51–64, 1997. 15

[34] J. A. Reeds and L. A. Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific Journal of Mathematics*, 145(2):367–393, 1990. 14

[35] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: A modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, 3rd ed. edition, 2010. 21

[36] Steven S. Skiena. *The algorithm design manual*. Springer, London, 2nd ed. edition, 2008. 19, 21

[37] C. Urmson and R. Simmons. Approaches for heuristically biasing rrt growth, 2003. 10

[38] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William "Red" Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008. 2, 29

[39] Jens Ziegler, Moritz Werling, and Jochen Schroder. *Navigating car-like robots in unstructured environments using an obstacle sensitive cost function*. IEEE, 2008. 17

TRITA -AVE 2016:41
ISSN 1651-7660