Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Sondre Bø

Motion planning for terrain vehicles

Path generation with radial-constrained A* and trajectory optimization

Master's thesis in Cybernetics and Robotics
Supervisor: Marius Thoresen, Kristin Y. Pettersen
June 2020

NTNU
Norwegian University of Science and Technology

FFI Forsvarets forskningsinstitutt
Norwegian Defence Research Establishment

Sondre Bø

Motion planning for terrain vehicles

Path generation with radial-constrained A* and trajectory optimization

Master’s thesis in Cybernetics and Robotics
Supervisor: Marius Thoresen, Kristin Y. Pettersen
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Problem description

The formulation of the problem description from the specialization project is maintained with some modifications [2].

Motion planning is the process of finding local paths that a specified vehicle can follow based on what it senses from the environment. Traditional motion planning methods usually find the shortest routes to avoid obstacles. For terrain vehicles, the surroundings are more complex, and the world is not always divided into obstacle/no-obstacle. The optimal path is not always the shortest, but may be the one with the smallest climb, minimal roughness or the one with least vegetation, or the route that is considered as the safest path.

The problem which will be elaborated in this report is finding optimal motion planning for a non-holonomic vehicle in terrain with respect to terrain characteristics. By using a grid to represent the environment, different terrain characteristics can be included, and a cost can be made for each cell in the grid. A* is an effective method to find the optimal path, but a vehicle is unable to follow a path generated in a grid. However, the trajectory can be made feasible by optimizing the path to yield the vehicle's driving behaviour. This report will present relevant literature for path planning for non-holonomic vehicle and optimization. Further, it will be looked into how these methods can be combined to perform motion planning in terrain.

Abstract

To perform non-holonomic motion planning in terrain is a challenging task. The optimal path in terrain may not be shortest in length, but the one with the smallest climb, least vegetation, or the route considered the safest. The terrain is often represented as a grid with traversability values. Finding the shortest path in a grid can be done with algorithms like A*, but the paths are not feasible for a car-like vehicle to follow due to too abrupt turns. The path planner needs to create a path that is driveable for a non-holonomic vehicle, by modifying an existing path-search algorithm or/and post-process the path by optimization.

This paper presents earlier work on terrain traversability, non-holonomic navigation, and trajectory optimization which are inspirations of this thesis' path planner. The path planner finds the path in a costmap provided by FFI, where each cell has a value that represents the traversability degree of this area. By dilating the costmap and given a start- and a goal-location, the radial-constrained A* algorithm is used to find a discrete- and safe-path that respects the turning radius of a non-holonomic vehicle. Further on, the discrete path is optimized by gradient descent optimization. The optimization uses the gradient of a cost function that smooths the path while maintaining the curvature and cost-efficiency of the path. If the stopping criterion of the gradient descent optimization is met, the initial discrete path will have been transformed into a smooth, cheap, and driveable path.

Sammendrag

Å planlegge en optimal rute for et bil-lignende terrengkjøretøy byr på forskjellige utfordringer. Den optimale bane kan ikke alltid anses som den korteste, men kan være den ruten med minst stigning, mest klaring eller det tryggeste veivalget. Disse terrengegenskapene kan lagres i et 2d-kart som representerer farbarheten i et visst område, og kartet kan videre brukes i et banesøk. Ved å behandle kartet som en graf, kan populære graf-søk algoritmer som A* brukes til å finne den optimale ruten. Et terrengkjøretøy vil få problemer ved å direkte følge A* sin rute, da ruten vil være diskret og bestå av svinger som kjøretøyet ikke kan følge. Baneplanleggeren må da respektere kjøretøyets begrensninger i planleggingsfasen.

Denne rapporten presenterer relevant litteratur som har påvirket utviklingen av ruteplanleggeren. Materialer innen navigering i terreng, ruteplanlegging for bil-lignende kjøretøy og rute-optimalisering vil bli introdusert. Ruteplanleggeren finner en bane i et kostkart som er mottatt fra FFI, der hver rute i kartet representer farbarheten i det respektive området. Videre, hvor et strukket (*dilated*) kostkart og start- og sluttpunkt er gitt, brukes det en radius-begrenset A* algoritme til å finne en diskret og trygg rute gjennom terrenget, hvor svingradiusen til kjøretøyet er tatt hensyn til. Den diskrete ruten blir deretter glattet med *gradient descent* optimering. Optimeringsalgoritmen bruker gradienter fra en kostfunksjon som glatter en diskret bane, samtidig som den tar vare på krumningen og det billige banevalget gjennom optimeringen. Avslutningsvis, når stoppkriteriene for *gradient descent* optimeringen er tilfredsstilt, vil den diskrete ruten ha blitt forandret til en glatt, billig og kjørbar bane for et terrengkjøretøy.

Contents

Problem description **i**

Abstract **ii**

Sammendrag **iii**

Preface **xvi**

1 Introduction **1**

1.1 Background 1

1.2 Motivation 2

1.3 Literature review 3

1.3.1 Terrain navigation 3

1.3.2 Non-holonomic navigation 5

1.3.3 Trajectory optimization 6

1.4 Assumptions 7

1.5 Contributions 7

1.6 Outline 8

2 Theory **9**

2.1 Traditional motion planning 9

2.2 Graph search algorithms 12

2.2.1 Breadth-first search 14

2.2.2 Dijkstra's algorithm 16

2.2.3 BFS vs. Dijkstra 18

2.2.4 The A* algorithm 18

2.2.5 Dijkstra's vs A* 20

2.3 Non-holonomic planning 21

2.3.1 Kinematics for a simple car 22

2.3.2 Turning radius in grid 23

2.3.3 Hybrid A* search 25

2.4 Image processing 27

2.4.1 Grayscale dilation 27

2.4.2 Averaging 28

2.4.3 Gradient field 29

2.5 Optimization 32

2.5.1 Mathematical formulation of an optimization problem 33

2.5.2 Global and local optimization 34

2.5.3 Gradient-based trajectory optimization 36

2.5.4 Gradient descent optimization 37

2.5.5 Conjugate gradient 38

3 Method **41**

3.1 Data and map representation 42

3.1.1 Polaris Ranger XP900 - Vehicle specifications 44

3.2 A* implementation 46

3.2.1 Traversability cost 49

3.2.2 Heuristic 50

3.2.3 Map dilation 51

3.2.4 Node selection based on minimum turning radius 54

3.3 Trajectory optimization 58

3.3.1 Smoothing term 59

3.3.2 Cost term 61

3.3.3 Curvature term 62

3.3.4 Gradient descent optimization 64

3.4 Overview of the motion planner 66

4 Results and discussion 67

4.1 Discrete radial-constrained A* algorithm 68

4.1.1 Run-time 72

4.1.2 Bug with radial-constraint 74

4.2 Cost optimization 76

4.3 Optimizing the radial-constrained A* 81

4.3.1 Mass simulation 88

4.3.2 Selection of weights 91

4.3.3 Stopping criterion 92

4.4 Three DOF start state with radial-constrained A* 96

4.5 Further discussion 99

5 Conclusions and future work 103

5.0.1 Further work 104

References 105

List of Tables

3.1 Ranger CP 900 Specifications 45

4.1 Table for map1 70

4.2 Table for map2 71

4.3 Table for map3 71

4.4 Table for map4 71

4.5 Table of run-time and memory-consumption in the algorithms. *Iterations* are the number of operations in the *Closed*-set, which are nodes added and re-evaluations of nodes. 73

4.6 Table of the total cost, length and cost per length. 78

4.7 Parameters for optimization on four different maps. 82

4.8 Table that shows the results from the mass simulation of three types of optimization on one path. The average is from all successful simulation and the number of feasible paths is stated. 89

4.9 Table for figure 4.18. Shows run-time for optimization. 95

List of Figures

1.1 Olav, an unmanned ground vehicle 2

2.1 A 2D world with a robot navigating along a planned path among obstacles. 11

2.2 From map to graph 13

2.3 From 2D-grid to graphs 13

2.4 Shortest path from start node to the goal node. 14

2.5 BFS search with four neighbors 15

2.6 Shows how to reverse back to the start node from every location. Inspired by [8]. 16

2.7 Shows the backtracking of the lowest-cost path and the cost to each cell. Movement cost to white cell = 1. Movement cost to green cell = 6. 17

2.8 Expanded(*Closed-set)* and visited(Open-set) nodes for uninformed and heuristic search in open landscape 21

2.9 Illustrates the steering for a car-like robot 22

2.10 Illustrates the steering problem for a car-like robot in a grid 23

2.11 Turning radius from trajectory 24

2.12 Shows the different consideration of adjacent nodes in regular A*(left) and Hybrid A*(right) 26

2.13 Illustrates grayscale dilation with a maximum function from input- to output array with a kernel of size 3x3 28

2.14 Illustrates averaging from input to output array with filter of size 3x3 29

2.15 The horizontal gradient are calculated in a image array. 30

2.16 The vertical gradient are calculated in a image array. 30

2.17 Shows how a vector field can be created from x- and y-gradients. . . 31

2.18 Illustrates the global and local minimums of an objective function f . 35

2.19 The gradient descent method finding a global minimum from an initial pose. 38

3.1 How OLAV perceives the vegetation 43

3.2 Map of the terrain 44

3.3 Illustrates the spatial occupancy for a vehicle in a grid. The occupancy is circle-shaped to consider all angles the vehicle can have at this location. 45

3.4 From map to graph 47

3.5 Shows a solution path of A* from a start- to a goal-location 49

3.6 Discrete cost illustration 50

3.7 Illustrated segments of solution paths from A*. The orange-shaded area illustrates the width of the vehicle if it was to drive this path. . . 52

3.8 Solution path from A* in a dilated map, with a kernel diameter of 14 cells. 53

3.9 Solution paths from A* in a dilated- and original-costmap. The paths are shaded with the vehicles width of 7 cell lengths. 54

3.10 Four scenarios of simulating the turning radius of three-point circles. Two of the points are the orange cells, the third represents the *next* node to be evaluated. 57

3.11 Shows different stages of smoothing 61

3.12 Shows a flowchart of the path planning process conducted in this thesis. The blue-dotted box to the left is the path planner, finding a discrete and cheap path that respects non-holonomic steering limitations. The blue-dotted box to the right is the trajectory optimizer, which uses a cost function that yields the vehicle's driving behavior. The gradient of this cost function is calculated and used in gradient descent optimization, outputting a smooth and driveable path. 66

4.1 **Orange**: Regular A* path. **Blue**: Path from A* at dilated map. **Green**: Path with radial constriction in A* at dilated map. 69

4.2 Shows the environment where the three paths are created. The orange path uses the map from 4.2a as input-graph, while the two other paths use the map from 4.2b as input. 70

4.3 *Map 2*. Shows the search area for the different path found for map 2 (4.1b), i.e. the expanded nodes with associated cost from the *Closed*-set. 73

4.4 Illustrates the color-coding for figure 4.3 74

4.5 Shows an enhanced section of the radial-constrained path from 4.1d . 74

4.6 Three-point-circles created with every four nodes to illustrate large-scale turning radius 75

4.7 Zoom-in of a narrow passage from figure 4.1c 76

4.8 Shows the evolution of a cost-optimized path 77

4.9 Shows monitoring of total cost in the optimization of the section presented in figure 4.8. The color-coding represent the corresponding path from figure 4.8 79

4.10 Shows an averaged section of an expensive area to the left, and the corresponding gradient field to the right. The section is withdrawn from figure 4.8. 80

4.11 *Map 1.* The **blue** path is only smoothed and deviates from the discrete path by cutting corners. The **orange** path is also smoothed but deviates from the blue path where the cost gets expensive. The **green** path is smoothed and cost-optimized, but consider the curvature to not exceed the minimum turning radius of the vehicle. 83

4.12 *Map 2.* Three types of optimization of a discrete path. The three continuous paths are in in the neighborhood of the discrete path . . . 84

4.13 *Map 3.* It can be seen that the initial guess is good, avoiding high-cost areas like the **orange** and **green** path does. The **blue** path is nearly converging to a straight line, not worrying about the higher-cost areas. 85

4.14 *Map 4.* The **green** and **orange** following the discrete path tightly. One exception is in one of the most distinct curve, where the **orange** and **green** are taking a longer turn to a cheaper area than the discrete path. The **orange** path is violating the minimum turning radius doing this, but the **green** path has a curvature term that penalizes too-small turning radius. 86

4.15 The cost per length of the optimized paths relative to the cost per length of the discrete path. 90

4.16 Illustrates nodes clustering in bright/low-cost area. w_s and w_c is set to 0.2 while w_o=0.9. α = 0.1 and the duration is 500 iterations. 91

4.17 Shows a path's cost for each iteration during optimization. Starting at 2700 and converging in about 2400. Max iterations was 1000 93

4.18 Shows three optimized paths.Two paths which are optimized until driveable, and one path which are optimized with a duration of 1000 iterations. 94

4.19 Shows the different paths generated at approximately the same start location with different start angles relative to the world frame. 97

4.20 Shows a heuristic map of the radial constrained search from subfigure 4.19c. The path reaches the brightest spots cheapest, and require the highest total cost to reach the black areas. 98

Preface

This master's thesis is submitted as a part of the requirements for the master degree at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The work presented in this thesis has been carried out under the supervision of Ph.D. candidate Marius Thoresen from FFI, Norwegian Defence Research Establishement, and Prof. Kristin Y. Pettersen at the Department of Engineering Cybernetics, NTNU.

This master's thesis is a continuation of a specialization project I conducted during autumn 2019. As this report is not published, the important background theory and methods from the project report will be restated in full throughout this report to provide the best reading experience. Below, a complete list of the material included from the specialization project is listed.

- Problem description, with some modifications at the last paragraph
- Chapter 1: Parts of 1.1 and the last three paragraphs from subsection 1.3.1.
- Chapter 2: Parts of section 2.1, and some parts from subsection 2.2.4
- Chapter 3: The first paragraph of section 3.1

The main focus in the specialization project was to compare graph-search run-time between three different types of modification to a traversability-degree costmap. The comparison was conducted with horizontal and vertical search, versus horizontal, vertical and diagonal search. As a conclusion, the best search was horizontal and vertical search in an original costmap, both in run-time and cost-efficiency.

At the start of this project, FFI provided a sensor recording of an unmanned ground vehicle performing motion planning from a start position to a goal. This recording was further utilized to extract necessary information for analyzing and developing.

Unless otherwise stated, all figures and illustrations have been created by the author.

Sondre Bø

Trondheim, June 2020

Chapter 1

Introduction

This introductory chapter will briefly provide background and motivation for the problem to be solved. A literature review summarized existing relevant knowledge around the approach for the problem and established the foundation of the later path planner. The scope of the work is then defined through some assumptions. Finally, the contributions of the thesis are defined and elaborated.

1.1 Background

In this section, the background of the problem to be solved will be presented. Some of the materials in this section are from the specialization project Motion Planning for Terrain Vehicles (2019)[2], and is included for the completeness of the thesis.

FFI has for years been doing research on autonomous vehicles and drones. Lately, the research has also been including unmanned ground vehicles, which have great potential to perform military tasks better, faster, and more efficiently. By introducing unmanned vehicles, people can be removed from dangerous tasks which increases the safety of military operations.

The car industry has made great advancements in developing self-driving cars, and the armed forces can benefit from these developments. By adapting the self-driving vehicle to drive in terrain, the vehicle can be sent out by soldiers to inspect e.g., what triggered an alarm in an unknown area. This will also make it possible for the armed forces to defend bases more efficiently in the future [7].

FFI is currently developing an autonomous vehicle with the goal of demonstrating autonomous driving in terrain. For a few years, FFI has been using a Polaris Range 900XP ATV called OLAV as their base vehicle platform (see figure 1.1). This platform is rigged with sensors to detect and interpret the terrain to perform motion planning.

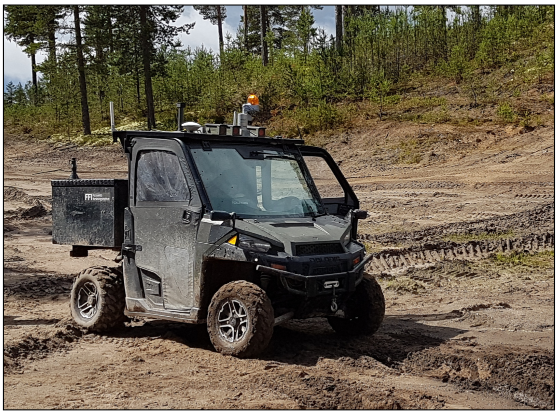


Figure 1.1: Olav, an unmanned ground vehicle

1.2 Motivation

The base vehicle platform by FFI have been using a method based on Hybrid-A* as their path planner. This algorithm is using a tree-search with motion primitives directly at the sensor-data to search through the terrain. However, the search-tree for Hybrid-A* is expanding exponentially with the distance searched, affecting the planner's run-time and limits the planning-horizon for each planning sequence.

As a result of the limitations to FFI's current path-planner, they want to explore other optional planners. This thesis is going to investigate an alternative motion planner which uses the informed A* search algorithm to find a discrete path in the terrain, and then post-process the path to get it smooth.

1.3 Literature review

A wide selection of methods has been developed to perform motion planning in terrain. For each approach, different techniques have been used to evaluate the environment, finding the shortest path, or optimizing an existing path. This section will present some of the key inspirators in the development of this thesis. The literature review will focus on how the A* algorithm, or modifications of A*'s are used in terrain and for non-holonomic path planning. Material which is optimizing paths to be smooth and driveable will also be presented.

1.3.1 Terrain navigation

In this subsection, a literature review of navigation in terrain will be presented. The last three paragraphs are from the specialization project Motion Planning for Terrain Vehicles (2019)[2], and is included for the completeness of the thesis.

NASA's Mars Exploration Rovers depends on a high-level of autonomy in search of evidence of past water activity on Mars [3]. The rovers are navigating autonomously throughout the terrain with an algorithm called GESTALT (Grid-based Estimation of Surface Traversability) [11]. The GESTALT algorithm uses stereo cameras to evaluate the terrain, storing the information as a local terrain model in a grid-based goodness map. Each grid cell is assigned a goodness value, which indicates the cost of traversal. The Rovers' navigation system uses Field D* to generate a path between two locations [6], which aims to minimize the cost of traversing this path based on the goodness map. The Field D* is a computational effective interpolation-based path- and re-planning algorithm for generating low-cost paths trough non-uniform grids.

DARPA grand challenge was arranged in 2004 and 2005 to spur innovation to unmanned ground vehicles capable of navigating in off-road terrain [32]. As the competitors received the approximate global route that the robot should take, the path planner's primary role was local obstacle avoidance. The robot led by Stanford University created its own base trajectory, which is a smoothed version of the global route the team received before the competition. The smoothing was done by least-squares optimization, where the focus was to minimize the curvature and still keep the distance to the original global path to a minimum. As the optimized path was still piece-wise linear, it was made differentiable with cubic spline interpolation before it could safely be used as a trajectory [32].

[15] focuses on a system that can accurately traverse off-road environments in high-speed autonomously. Their solution is to analyze the off-road terrain using a 3D lidar, determine potential hazards, and then plan a safe route around it. The route is planned by an arc-based planner that simulates and evaluates arcs based on traversability. The arc with the best score is kept, and the UGV has a local path to follow.

[34] investigates a geometrical approach to identify the roughness in terrain using the terrain elevations from a point cloud generated using a 3D camera. This terrain identification is mainly done to perform high-speed navigation decisions for UGV in rough unknown terrain.

[12] presents a global path planning strategy for UGV from an aerial scan of the environment. A set of algorithms is processed on the aerial scan to produce a 2D

costmap, which is based on prior traversability. Further on, the costmap is used as an input to a D* path planner.

1.3.2 Non-holonomic navigation

DARPA urban challenge was a competition where robotic vehicles had to navigate in urban environments autonomously [5]. This competition motivated Stanford to develop a path-planning algorithm that generates smooth paths in an unknown environment. This approach is based on two steps, where the first use Hybrid State A*-search to find a minimum-cost path where the non-holonomic driving behavior of the robot is considered. The second step locally improves the quality of Hybrid A*'s solution by performing numeric non-linear optimization, which perturbs the path to a local, or global optimum [5].

[17] focuses on navigating an unmanned surface vehicle formation accurately and effectively using path planners to generate optimal trajectories. The USVs are restricted by motion constraints, hence, the resulting trajectory has limitations in the allowed curvature and need to provide practical collision maneuvers. [17] presents an algorithm called *angle-guidance fast marching square*(AFMS) to generate the trajectories compliant to the USV's restrictions. The AFMS path planning algorithm uses FMS as the base algorithm but creates a guidance-range in the planning space to consider the USV's motion constraints. The guidance range consist of two different sectors, one *turning range* sector and one *obstacle range* sector. It is desired that the path should remain within the turning range sector, and the obstacle range sector should act as an obstacle.

Grids are commonly used as spatial models in robot navigation. To find a safe path for the robot, a graph can be constructed from the two-dimensional model of the environment. As the graph is constructed from a grid, algorithms like A* and Dijkstra can be executed to find the shortest path. However, these kinds of graph-search algorithms rarely consider the agent's dynamic constraints, which may result in a path with too sharp turns. LIAN (limited-angle) is a heuristic search algorithm that generates smooth paths in a grid-based environment while addressing the angle-constraint problem for non-holonomic vehicles[35]. The LIAN algorithm has similarities with A* but evaluates the potential successors in the path with a midpoint algorithm to identify valid cells based on a maximum angle of an alternation.

1.3.3 Trajectory optimization

[37] presents CHOMP (covariant Hamiltonian optimization for motion planning), which is a trajectory optimization technique for motion planning used in high-dimensional spaces. This article focuses on using a functional gradient technique to iteratively improve the smoothness and obstacle distance for the initial trajectory. The CHOMP algorithm can be used to locally optimize feasible trajectories, solving motion planning queries and converging to low-cost trajectories even when initialized with infeasible trajectory. The approach for CHOMP's motion planning is built on two assumptions: "Gradient information is often available and can be computed inexpensively" and "Trajectory optimization should be invariant to parametrization".

[25] uses a sequential convex procedure to incorporate collision avoidance into trajectory optimization to solve motion planning problems. This approach focuses on planning problems with many degrees of freedom, from 7- and up to 28 DOF. [25] uses a two-step optimization algorithm to create a feasible motion plan from scratch, where the first step is numerical optimization with sequential convex optimization. The second step for [25]'s optimization algorithm is obstacle avoidance, where the signed distance is computed using convex-convex collision detection, and the continuous-time safety of a trajectory is ensured. This approach yielded higher quality paths in lower run-time than other motion planners that use trajectory optimization, including the CHOMP algorithm.

1.4 Assumptions

FFI has provided a 65-seconds recording of their autonomous vehicle performing motion planning in terrain. This recording contains all the information the vehicle needed to perform the navigation. During the development of the path-planner in this thesis, only the provided data from the recording is used in the development and test-phase of the path planner. There is no guarantee that the path planner algorithm from this project will work on other types of data and a physical vehicle.

The thesis' scope is the path planning process on one sequence only. The vehicle's position is accessible in a costmap at all times, and the path planning starts when a goal location is received. It is assumed that the goal location is placed in a detected spot within the costmap, and generating temporary way-points is not needed. The path planning ends when a driveable- and smoothed-path in the costmap is found.

To measure the performance between the paths, a total cost from the costmap is calculated. The developer of this thesis has no indications on how well a vehicle will perform at a certain total cost, as the paths was not tested on a physical vehicle. Even though, it is assumed that a lower total cost-number is a better choice.

1.5 Contributions

The main contributions of the work presented in this thesis are as follows:

- A comparison between the most popular discrete graph-search algorithms
- The development of a path planner that uses a dilated costmap based on terrain characteristics. This map is used to find a discrete and safe path that respects a minimum turning radius for a non-holonomic vehicle.

  - Inspired by [3] that finds a low-cost path with Field D* through a non-uniform grid.
  - The radial-constraint limitation of the search directly in the grid is inspired by [35] which restrain the search by an angle.

- The design of a cost-function that yields the desired driving behaviour for a non-holonomic vehicle.
  - Inspired by [5] that uses a cost-function to optimize a Hybrid A* path generated in a binary environment.

- Pre-processing of a costmap by averaging to generate a gradient field which is further used in a cost-term in an objective function.

- Utilizing gradient descent optimization to smooth a discrete path, while maintaining the curvature and cost-efficiency of a path.
  - Inspired by [5] that uses conjugate gradient to smooth a path to a local- or global-minimum.
  - The gradient descent implementation is also motivated by [14]

- A study evaluating the performance of the discrete radial-constrained path and trajectory optimization of the discrete solution path.

1.6 Outline

The report is organized as follows. In Chapter 2, relevant theory is presented for non-holonomic path planning, image processing and optimization. Chapter 3 presents the implementation of A*, and how this was modified to find safe path for non-holonomic vehicles. Chapter 3 also presents the development of the optimization technique to smooth a discrete path, while maintaining the curvature and cheap total cost. Chapter 4 perform tests that examine the different aspects of the algorithm. Finally, chapter 5 concludes the work and share thoughts on future works for this algorithm.

Chapter 2

Theory

This chapter will provide the necessary theory to understand the concept and approaches behind the motion planning process for this master thesis. Firstly, traditional motion planning will be elaborated. Then some graph search algorithms will be discussed. Furthermore, some image processing techniques will be presented. Lastly, essential information about optimization will be introduced.

2.1 Traditional motion planning

In this section, some theory of traditional motion planning will be stated. Some of the materials in this section are from the specialization project Motion Planning for Terrain Vehicles (2019)[2], and is included for the completeness of the thesis.

Motion planning refers to the robot's motion in a 2D- or 3D world that contains obstacles. A path is a sequence of commands for the robot to move safely to the goal state without colliding with obstacles.

When planning the trajectories for the robot to a goal state, it is important to represent the robot's world correctly. This world is usually a geometric model of a system of bodies in space and can be defined either as boundary representation or a solid representation. The first step to define the geometric model is to either define a 2D- or 3D world $\mathcal{W}$. If it is a 2D world, $\mathcal{W} = \mathbb{R}^2$, and if it is a 3D world, $\mathcal{W} = \mathbb{R}^3$. Representing a more complicated world with more than three dimensions is not necessary as this simple world generally contains two kinds of entities[16]:

- **Obstacles**: Parts of the world $\mathcal{W}$ that are permanently occupied, e.g., trees or walls.

- **Robots**: Bodies that are modeled geometrically and can move via a planned path.

These two entities are considered as closed subsets of the world $\mathcal{W}$. We let the obstacle region O denote the set of all points in $\mathcal{W}$ that exist in one or more obstacles; hence, $O \subseteq \mathcal{W}$. The 2D world is often represented in a grid where the occupied cells are the subset O, and the rest of the world is free cells[16]. Figure 2.1 shows an example of a world $\mathcal{W}$.

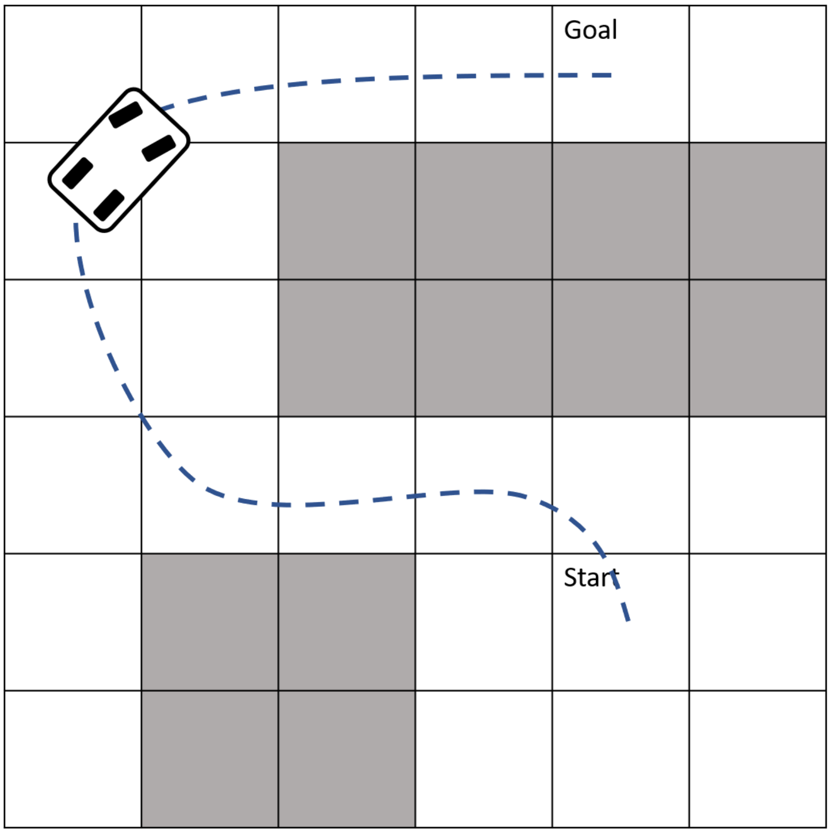


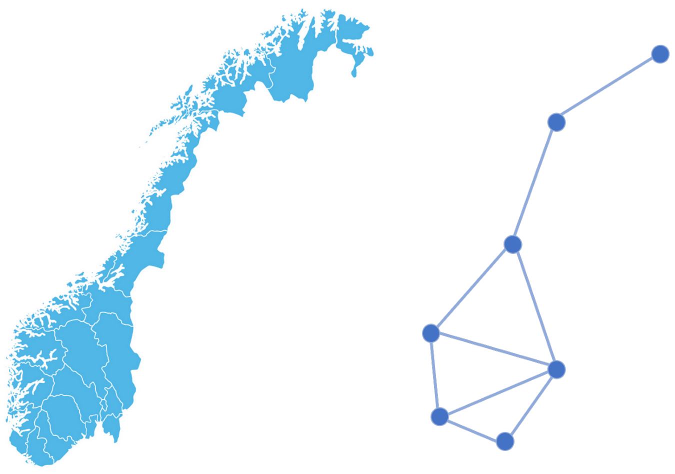
Figure 2.1: A 2D world with a robot navigating along a planned path among obstacles.
.

Dividing the world into free and occupied cells works well when the robot is operating in simple environments as indoor or on open roads [16]. These worlds are considered simple because the degree of traversability is the same wherever the robot move, except when moving into obstacles. The objective of a motion plan in these types of worlds usually is to find the path with least *cost*. In a simple world, the cost for a path planner would be the path's total distance, however, in a more complex world e.g. terrain, other factors than distance may be relevant for the planner's decisions. To simulate different terrain like hills, degree of vegetation, or different frictions to surfaces, weights can be distributed to the free cells in the world, $\mathcal{O} \subsetneq \mathcal{W}$/the search space. These weighted cells will represent the degree of traversability, and together form a costmap describing the terrain. The advantage by using a costmap for navigation is that the solution path will be the path with the least cost, and then be the safest, easiest, and best path, as the degrees of driveability are estimated on the user's criterion. The *optimal motion plan* is to find the optimal solution for a path-planning problem by finding the cheapest path in a costmap or minimizing a cost-function f for a path [16].

2.2 Graph search algorithms

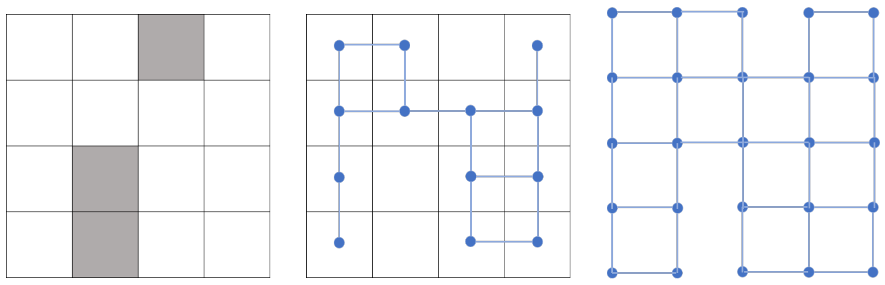
The world $\mathcal{W}$ is usually represented as a grid environment, either as a binary grid with obstacle/no-obstacle or as a weighed map, as mentioned in section 2.1. By constructing a graph from the two-dimensional map-environment, graph search algorithms can be applied to search for paths through the map[31]. The best route through the map is the most cost-efficient route, and a good algorithm to find this path is to use the A* algorithm. This algorithm utilizes effective properties from both breadth-first Search and Dijkstra's Algorithm to reach its goal.

When graph search algorithms are used to find a path, they need a graph as an input, as well as start- and stop-node locations in the graph, and the algorithm returns a path if the search is successful. The input graph, the map, is a construction of nodes and edges. The nodes represent locations, and the edges connect the locations. Depending on the problems agenda, the edges in the graph can be both with and without weights.



(a) Graph representing a map (b) Map of Norway [26]

Figure 2.2: From map to graph

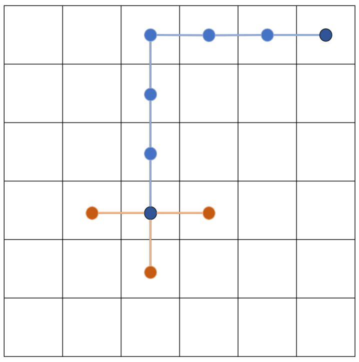


(a) Obstacle-map (b) Nodes located mid-cell (c) Nodes located in corner

Figure 2.3: From 2D-grid to graphs

There are many different ways to represent a graph as a map. An example of such a graph can be a map of a country (figure 2.2 of Norway), where the nodes can represent large cities, and the highroads can be edges. Weights can be added to the graph to represent distances or speed-limits. Another map-type is to use a two-dimensional grid as a map 2.3. This graph is structured like a grid, where all nodes are structured to have equal distances to each other, and the edges represent how to get from one node to another. The nodes placement in the grid can either be located in the center of the cells (subfigure 2.3b) , or in the corners (subfigure 2.3c).

There are two ways to represent a node's children/neighbors in a two-dimensional grid environment, either with four neighbors (horizontal and vertical search) or eight neighbors (including diagonal search). Depending on the neighbor representation, a search algorithm will in many cases find two different solution paths. In the case with four neighbors (subfigure 2.4a), there are several optimal solutions which include the shortest route to the goal node, which are seven steps. In the case with eight neighbors (subfigure 2.4b), there is only one optimal solution, including four steps.


(a) Four neighbours to the start node

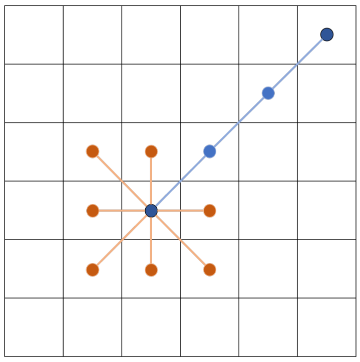
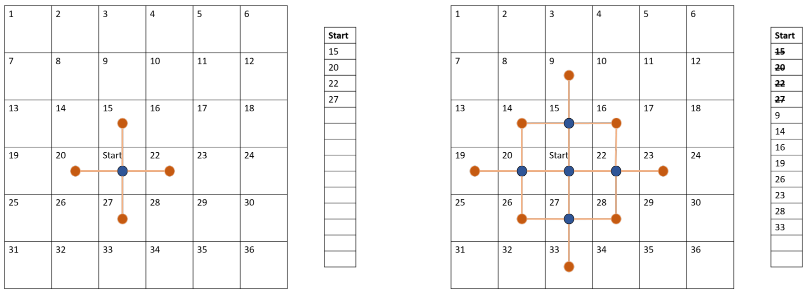

(b) Eight neighbours to the start node

Figure 2.4: Shortest path from start node to the goal node.

2.2.1 Breadth-first search

The BFS algorithm is an uninformed graph traversal algorithm that is used to find the shortest path in a graph. The algorithm uses a queue data structure that follows the First-In-First-Out(FIFO) principle [4]. Starting to explore a start node, the algorithm puts all of the start node's adjacent nodes in the queue, and then continue to explore the next one in the queue, i.e. one of the start node's neighbors. The algorithm then puts all adjacent nodes of this neighbor that are not already visited to the queue. This process is repeated until the whole graph is explored. Since BFS uses a FIFO queue to explore the graph, the search will grow uniformly and is therefore referred to as a breadth-first search. Figure 2.5 illustrates the BFS algorithm during search a grid.



(a) BFS search depth = 1

(b) BFS search depth = 2

Figure 2.5: BFS search with four neighbors

The BFS algorithm does not directly construct a path. It can be modified to keep track of where the visited nodes came from for every location visited. By adding a table that keeps track of where the visited node was expanded from, the shortest path can be reconstructed from every location to the start location [8]. If BFS is used to find a path to a single destination, the algorithm can be modified to allow early exit. The early exit function terminates the search if the expanded node is the goal node. Figure 2.6 shows the shortest path from a goal node to a start node.

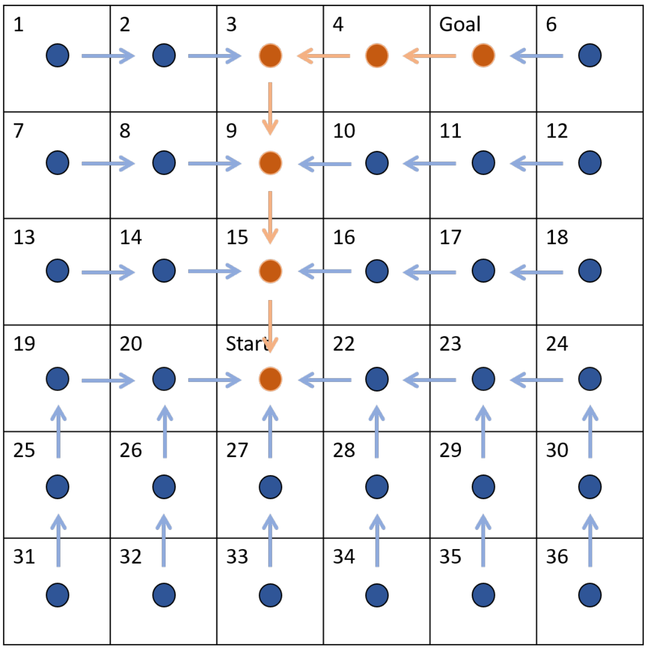


Figure 2.6: Shows how to reverse back to the start node from every location. Inspired by [8].

2.2.2 Dijkstra's algorithm

Graph traversal algorithms have the purpose to always find the shortest path in the graph. The definition of *shortest* may vary and can be evaluated by e.g. distance, safety, time or difficulty. To balance the priority between these elements, the edges between the nodes can be weighed. These weights will then represent the cost to move between the nodes in the graph.

Dijkstra is an algorithm to solve a single-source shortest-path problem on a weighted graph [4]. This algorithm is based on the breadth-first search but uses a priority queue instead of a FIFO queue. This allows exploring paths with the lowest cost first, instead of exploring all directions equally as in BFS. When the algorithm is visiting nodes, it stores information in two tables: one for where the new node was visited from, and one that stores the cost to get to this node. If a node's neighbor is already visited, the algorithm checks if the new cost will be less than the already stored cost to this node, if it is, the cost table and came-from table gets updated with new values [4]. Figure 2.7 shows the lowest cost path found with Dijkstra's algorithm, where moving costs are 1 through white cells, and 6 through the green cells.

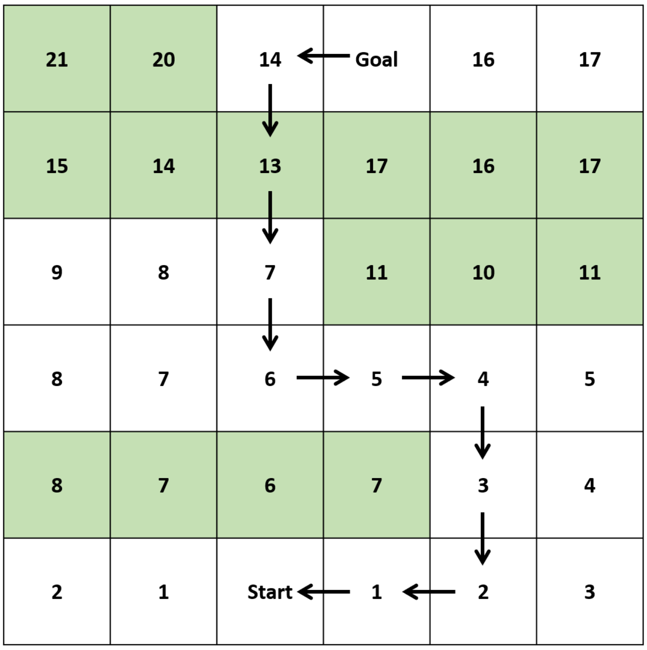


Figure 2.7: Shows the backtracking of the lowest-cost path and the cost to each cell. Movement cost to white cell = 1. Movement cost to green cell = 6.

2.2.3 BFS vs. Dijkstra

There is not much difference between the breadth-first search and Dijkstra's algorithm. If the cost in the graph is uniform, BFS and Dijkstra will search in the same manner and find the same optimal solution. Both algorithms are also uninformed, they can start the search without prior knowledge, e.g. location to the goal, and still find a solution path if it exists. Since both algorithms cover a large area in their search with breadth-first, it is beneficial to use them in a multiple-target search. The disadvantage of searching a large area is that it requires a lot of memory, thus the running time can be large in big graphs [29].

The queues are the component which distinguishes these algorithms. With BFS's FIFO queue, the search finishes one depth layer at a time before moving to the next layer. BFS defines the shortest path with the number of depth-layers it needs to traverse from the start- to the goal node. With Dijkstra's priority queue, the visited nodes with the lowest cost are placed first in the queue. Thus, when Dijkstra's search pops the goal node from the priority queue, the shortest path is found. Dijkstra's definition of the shortest path is the route with the lowest total cost from start- to the goal-node. BFS and Dijkstra's algorithm will find the same path in a uniform weighted graph, but different shortest paths in a weighted graph.

2.2.4 The A* algorithm

In this subsection, some theory about the A algorithm will be stated. Some of the materials in this subsection are from the specialization project Motion Planning for Terrain Vehicles (2019)[2], and is included for the completeness of the thesis.*

Dijkstra is an algorithm that finds the shortest path solely based on the cost. The algorithm searches for the goal in a breadth-first manner, using maybe more memory than necessary.

The A* algorithm is a modification of Dijkstra, which uses a heuristic as a guide in the search. By introducing a heuristic for the evaluation, the algorithm can prioritize paths that are cheap and appear to lead closer to the goal [8], which will save both time and space. This is mainly done from a function which evaluates a node *n* in the search space:

$$f(n) = g(n) + h(n) \tag{2.1}$$

Where

- *g(n)* represents the cost of getting from the start node to node *n*.
- *h(n)* represents the heuristic, an estimate of the distance from node *n* to the goal.
- *f(n)* represents the total expected cost of a solution path going from the start node, through the node *n*, to a goal.

By using the *g*, *h* and *f* values, A* will be directed towards the goal and will find the shortest possible route. Starting with a node *n*, the search is done iteratively by evaluating *n*'s children with the cost function *f(n)*. The evaluated nodes are stored together with the cost information in a priority queue. The search then continues with the node with the smallest cost in the priority queue. This process continues until the goal node is retrieved from the priority queue.

2.2.4.1 Heuristic

The A* algorithm will only be guaranteed to find the optimal path if the heuristic function is admissible, which means that as long as the heuristic function does not overestimate the distance to the goal, the solution path will be the optimal path [30]. In a 2d path planning, two types of heuristics are often used. The L1 norm, often known as the Manhattan distance:

$$||x||_1 = \sum_i |x_i| = |x_1| + |x_2| + ... + |x_i| \tag{2.2}$$

and the L2 norm, often known as the Euclidean distance:

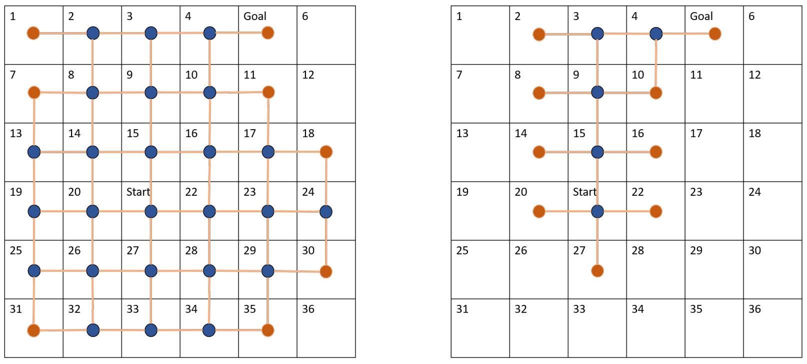
$$||x||_2 = \sqrt{\sum_i x_i^2} = \sqrt{x_1^2 + x_2^2 + ... + x_i^2} \tag{2.3}$$

A* can be used to solve many types of planning problems and with different cost-functions and heuristics, e.g., solve an 8-puzzle game, moving a sofa through tight spaces [16], or find a traversable path for terrain vehicles.

2.2.5 Dijkstra's vs A*

What differs between A* and Dijkstra's algorithm is the heuristic which is used in A*. A* search is a guided or informed search, it uses an (admissible) heuristic to predict the cost to the goal, and consider the nodes which seem to lead closer to the goal. This extension may save the search a lot of memory, thus lower the run-time in the right circumstances [28].

For instance, in an open landscape where the cost does not differ much Dijkstra's approach would have to evaluate approximately the same amount of nodes as BFS, while A* approach can greedily follow the heuristic, guiding the search towards the goal, see figure 2.8a. The heuristic used in subfigure 2.8b is the Manhattan distance. Both subfigures expand the lowest node number first if the cost/distance is equal.



(a) Uninformed (b) Informed/Heuristic

Figure 2.8: Expanded(*Closed-set)* and visited(Open-set) nodes for uninformed and heuristic search in open landscape

In another scenario where the cost differs much, or there is a lot of obstacles, A*'s heuristic may not be the biggest help. The path which seems to lead close to the goal might be blocked, or be more expensive than a longer route. At the worst, A* and Dijkstra will have the same run-time. In other words, A* will never have higher run-time than Dijkstra but may have significantly lower run-time in some scenarios. Both algorithms will eventually return the optimal solution path if it exists, (if A*'s heuristic is admissible).

2.3 Non-holonomic planning

One of the main challenges that appear when representing a map as a grid is that the real-world is continuous, while the positioning of the nodes in the grid is discrete. Since the locations of the nodes in the grid are predetermined, the alternative routes for the path-planners are limited. Discrete path planning algorithms often find paths with too abrupt turns which is not feasible for a robot to directly follow. Therefore, either a continuous path planner is needed, e.g. Hybrid A* search [22], or smoothing algorithms have to be applied on the discrete path. Either way, the cars kinematics need to be accounted for in generation of the solution path.

2.3.1 Kinematics for a simple car

A car-like robot with four wheels is restricted in how they move. As the back wheels can not slide, the robot's maneuvering depends on the steering angle and the distance between the axles. This type of vehicle has three degrees of freedom, the position of x and y, and the angle between the world frame and body frame of the car [16].

The resulting trajectory of a path planning algorithm may be feasible for one vehicle but infeasible for another. This depends on the vehicle's kinematic, what the maximum steering angle or minimum turning radius is for the vehicle. The curves in the resulting trajectory can not have a smaller radius than the vehicle's minimum turning radius, if this is the case, the resulting trajectory would be infeasible for this vehicle.

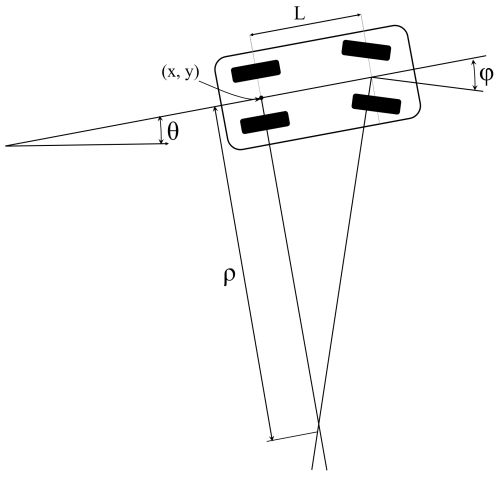


Figure 2.9: Illustrates the steering for a car-like robot

.

Denote ϕ for the steering angle at the front axle, and the length L between the front- and back-axle at the vehicle. If ϕ is constant while the robot is driving, the vehicle will drive in a circular motion with a turning radius of ρ. The distance between the intersection between the two lines from the axles, and the center of the vehicle is the turning radius at that given time, see figure 2.9 for illustration. From trigonometry the relation between the turning radius ρ and the steering angle ϕ is:

$$\tan(\phi) = \frac{L}{\rho} \tag{2.4}$$

where L is the distance between the axles [16]. Figure 2.10 illustrates an infeasible path for a car with turning radius ρ. The path is infeasible because the vehicle can not track the path due to too abrupt steering angle.

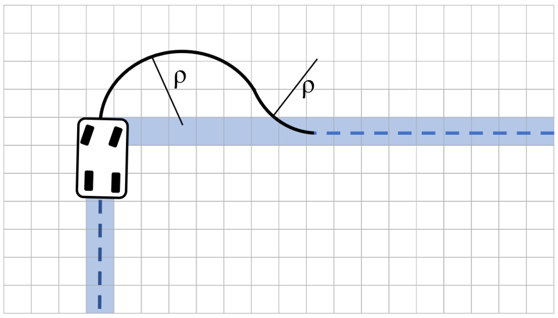


Figure 2.10: Illustrates the steering problem for a car-like robot in a grid

.

2.3.2 Turning radius in grid

To find out if the resulting path is feasible for a vehicle or not, the turning radius can be calculated directly from the path. Pick three points on the trajectory, and a circle with radius ρ can be created which passes through all three points [13]. The circle equation is stated below:

$$(x - x_c)^2 + (y - y_c)^2 - \rho^2 = 0 \tag{2.5}$$

where (x_c, y_c) is the center of the circle. By using the circle equation 2.5 for all three points, there will be three equations with three common unknowns; ρ, x_c and y_c. The idea of extracting the turning radius from the path is illustrated in figure 2.11.

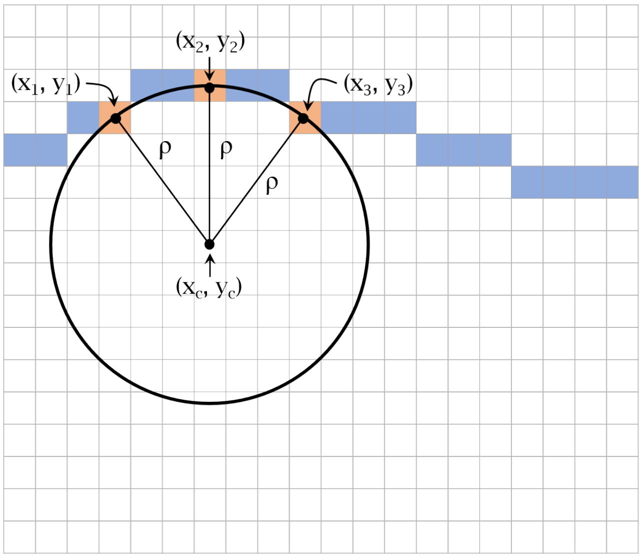


Figure 2.11: Turning radius from trajectory

.

$$(x_1 - x_c)^2 + (y_1 - y_c)^2 - \rho^2 = 0 \tag{2.6a}$$
$$(x_2 - x_c)^2 + (y_2 - y_c)^2 - \rho^2 = 0 \tag{2.6b}$$
$$(x_3 - x_c)^2 + (y_3 - y_c)^2 - \rho^2 = 0 \tag{2.6c}$$

Two linear equations can be created to find the center (x_c, y_c) and ρ expressed with (x_1, y_1), (x_2, y_2) and (x_3, y_3). These equations can be created by subtracting equation 2.6a from equation 2.6b and equation 2.6a from equation 2.6c.

$$(x_2^2 + y_2^2) - (x_1^2 + y_1^2) + 2x_c(x_1 - x_2) + 2y_c(y_1 - y_2) = 0$$
$$(x_3^2 + y_3^2) - (x_1^2 + y_1^2) + 2x_c(x_1 - x_3) + 2y_c(y_1 - y_3) = 0$$

By isolating y_c and solving for x_c we get:

$$x_c = \frac{(y_3 - y_1)(x_2^2 + y_2^2 - x_1^2 - y_1^2)) + (y_1 - y_2)(x_3^2 + y_3^2 - x_1^2 - y_1^2)}{2((x_1 - x_3)(y_2 - y_1) + (x_1 - x_2)(y_1 - y_3))} \tag{2.7}$$

$$y_c = \frac{x_2^2 + y_2^2 - x_1^2 - y_1^2 + x_c(x_1 - x_2)}{2(y_2 - y_1)} \tag{2.8}$$

$$\rho = \sqrt{(x_1 - x_c)^2 + (y_1 - y_c)^2} \tag{2.9}$$

2.3.3 Hybrid A* search

BFS, Dijkstra and A* are all used to compute paths in discrete state spaces. As most robots and vehicles navigate in continuous state spaces, a new path planner which takes account of the non-holonomic constraints and continuous nature for the vehicle is needed to produce a smooth path. Hybrid A* is an efficient path-planner algorithm that was used by Stanford's racing team in the DARPA Urban Challenge in 2007. This algorithm produces a continuous path in the discrete grid which is directly driveable for the robot [5].

Hybrid A* is a modified version of the regular A* star search. While A* is only allowed to visit a specific location in the adjacent node, e.g. center or corner of cells, Hybrid A* can treat each cell as continuous, and visit every location within the adjacent cells during the search [22]. When Hybrid A* considers the neighboring cells, it considers the non-holonomic nature of the vehicle and adds only the nodes that the robot is able to reach. The neighboring nodes are evaluated by pre-calculated motion primitives, to determine reachable states for the non-holonomic vehicle. By using the motion primitives, arcs, the hybrid approach guarantees that the paths found is traversable. Further, the paths found are evaluated by a cost function, in the same manner as discrete A*. The most cost-efficient path is chosen and is guaranteed to be drivable by the Hybrid A* algorithm. The selected path is, however, not guaranteed to be cost-optimal, as many possible cells may have been pruned. But the solution might be in the neighborhood of the global optimum, allowing to arrive at the global optimum solution by using gradient descent to improve the path [5].

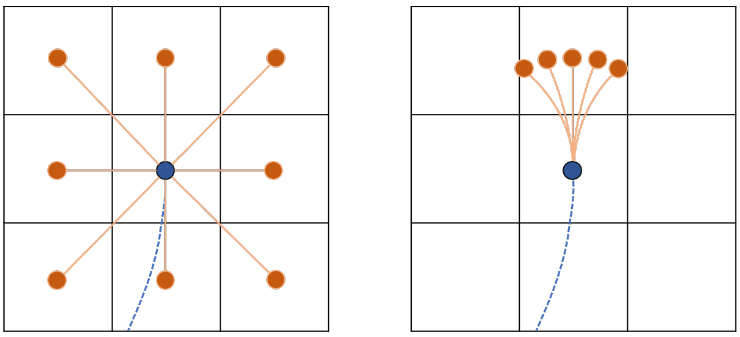


Figure 2.12: Shows the different consideration of adjacent nodes in regular A*(left) and Hybrid A*(right)

.

2.4 Image processing

Graph search algorithms have the purpose to find the best path in a given graph. The optimal path highly depends on how the graph is represented. As mentioned earlier, a graph can be generated from a map that is stored as a two-dimensional array. By manipulating the array, the graph can be changed, and there will be a new optimal solution path for the graph problem.

The manipulation can be done by image processing since the array contains numbers as in a regular grayscale or RBG images [18]. How the image is processed depends on what the definition of *optimal* for the problem is, or what the desired driving behavior is for the path-following car.

2.4.1 Grayscale dilation

Grayscale dilation is a morphological operation that uses a structuring element(or a kernel) to an input array and creating an output array of the same size. The value of each pixel in the output array is decided by the neighborhood of the corresponding pixel in the input array. The structuring element strides over the input array, checks all values inside the array, and assigns either the *maximum* or *minimum* pixel value from the neighborhood to the corresponding pixel in the output array [9].

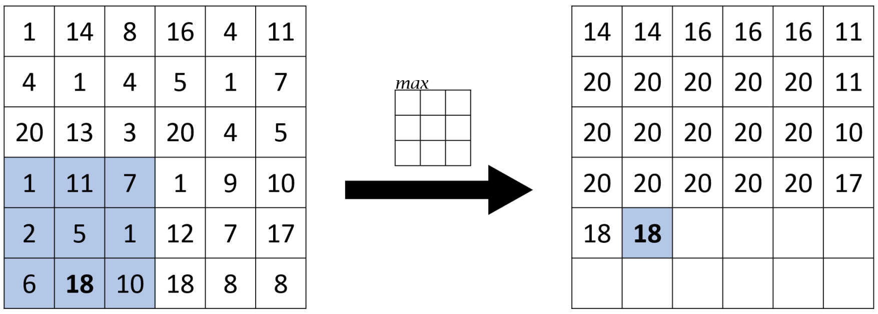


Figure 2.13: Illustrates grayscale dilation with a maximum function from input- to output array with a kernel of size 3x3

Dilation is used to enlarge or enhance important elements in a picture. For an input map to a path planning problem, this can be useful to enhance obstacles and high-cost areas.

2.4.2 Averaging

The grayscale dilation enhances either the highest or lowest pixel value for a pixel's neighborhood in an image. As this is highly effective in enhancing objects, important information is lost in the process. The dilation assigns only the maximum or the minimum in a neighborhood, hence all other pixels in this neighborhood are filtered.

Averaging can be used to gather information about a pixel's neighborhood. The averaging filter takes the average of all pixels of a neighborhood in the input array and assigns the average value to the corresponding pixels in the output array[21]. The size of the neighborhood is defined as the filter's dimension. The process is illustrated in figure 2.14.

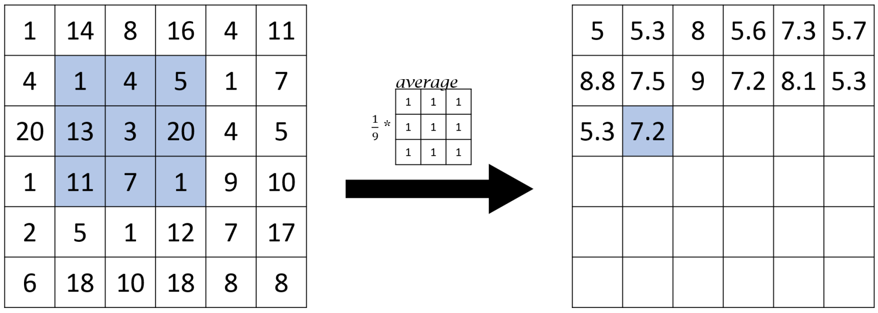


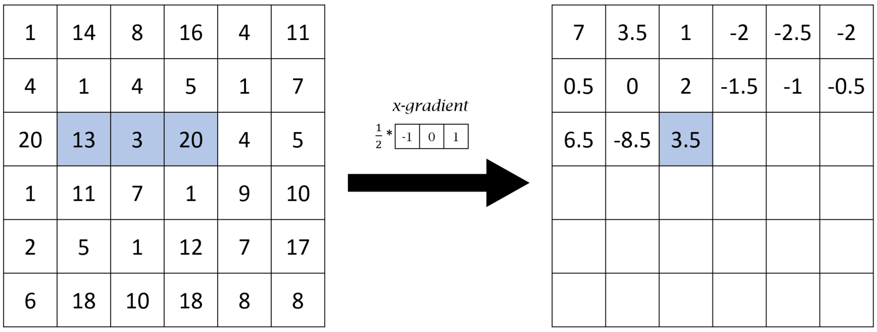
Figure 2.14: Illustrates averaging from input to output array with filter of size 3x3
.

Image information is also lost by performing averaging. Information about single pixels disappears, but are replaced with information about that pixel's neighborhood.

2.4.3 Gradient field

The gradient can be calculated from an image to mathematically visualize the differences in an image. The gradient for an image outputs two kinds of arrays, one array for vertical differences, and one array for horizontal differences [27]. The two arrays can be fused to create a gradient field of the map. The gradient field has the same dimension as the input array, but the cells contain vectors instead of pixel values. These vectors point towards the area which increases the most over that pixel, and the size of the vector informs about how big the difference is.

The gradient of x and y is calculated by convolution. For the x gradient, a kernel with size 1x3 is used and convolved over the input array. The horizontal difference is calculated over a pixel and divided by two, then the new value is assigned to the corresponding pixel in the output array. The process is illustrated in figure 2.15.

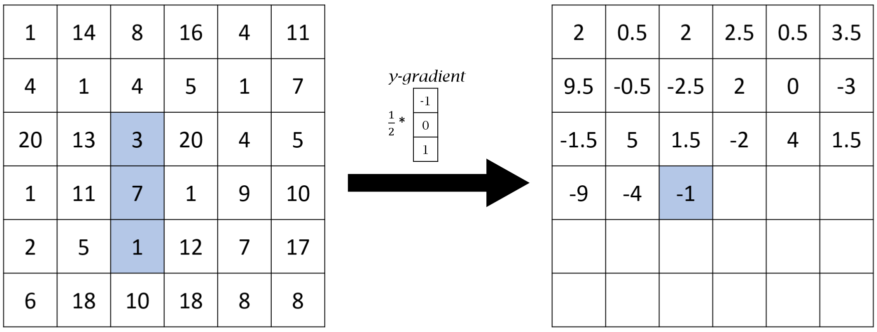


1	14	8	16	4	11
4	1	4	5	1	7
20	13	3	20	4	5
1	11	7	1	9	10
2	5	1	12	7	17
6	18	10	18	8	8

7	3.5	1	-2	-2.5	-2
0.5	0	2	-1.5	-1	-0.5
6.5	-8.5	3.5			

Figure 2.15: The horizontal gradient are calculated in a image array.
.

The y gradient is calculated in the same manner as the x gradient, except the kernel has a dimension of 3x1 and the vertical difference is calculated over the pixels. The process of calculating the vertical gradient is shown in figure 2.16.



1	14	8	16	4	11
4	1	4	5	1	7
20	13	3	20	4	5
1	11	7	1	9	10
2	5	1	12	7	17
6	18	10	18	8	8

2	0.5	2	2.5	0.5	3.5
9.5	-0.5	-2.5	2	0	-3
-1.5	5	1.5	-2	4	1.5
-9	-4	-1			

Figure 2.16: The vertical gradient are calculated in a image array.
.

The two arrays of gradients can be fused, and visualized with vectors, see figure 2.17.

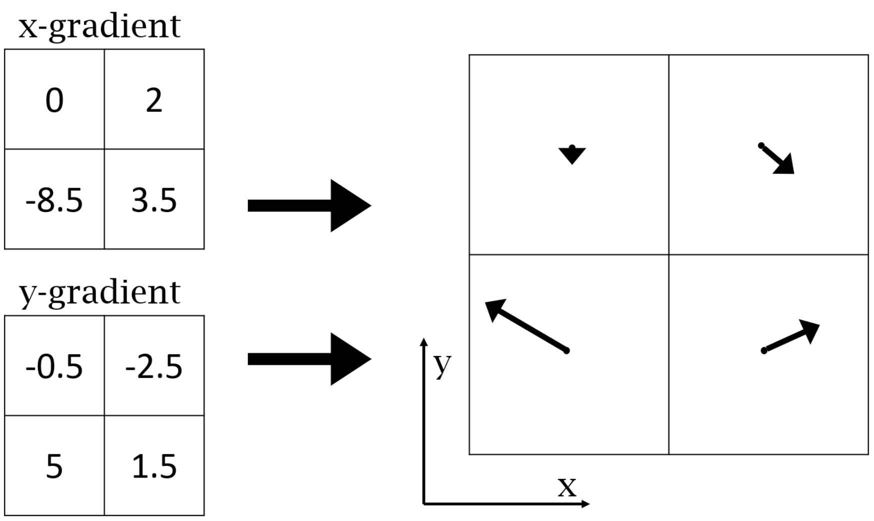


Figure 2.17: Shows how a vector field can be created from x- and y-gradients.

.

The gradient field are convenient to visualize differences in the map, and then use as a guide towards high- or low-cost areas for path planners.

2.5 Optimization

It was mentioned in section 2.3 that a smoothing algorithm could be applied on a discrete path to obtain a feasible path. The smoothing can be done by optimization, where the position of the nodes is being optimized. A definition of optimization is [33]:

> *"Optimization is an act, process, or methodology of making something (such as design, system, or decision) as fully perfect, functional, or effective as possible."*

Optimization is a widely used tool in the analysis and decision science of physical systems. To use optimization as a tool, an *objective* needs to be identified which can be used as a quantitative measure of the system under the study. This *objective* can vary from being e.g. time, profit, cost, length or any function that can be represented by a single number. The *objective* is defined as certain characteristics of the system called *parameters*, *unknowns* or *variables*. The goal of the optimization problem is to find *variables* that optimize the *objective*. In some optimization problems the *variables* are *constrained*, e.g. demand from stores, capacity of production or steering angle for a vehicle. The whole process of identifying the *objective*, *variables*, and *constraints* for the problem is called *modeling*. Depending on the problem and *model*, different optimizing algorithms can be utilized to find the solution[20].

An optimizing algorithm can be used in a path planning process. The objective function will then be a cost-function consisting of terms that define smoothness or cost in the path. The constraints in the optimization problem can be the vehicle's constraints, which the most obvious one is the steering angle.

2.5.1 Mathematical formulation of an optimization problem

"Mathematically speaking, optimization is the minimization or maximization of a function subject to constraints on its variables". [20] uses the following notation for the optimization model:

- x is the vector of *variables*, also called *unknowns* or *parameters*. E.g., x is the path, which is describe by a sequence of nodes.
- f is the objective function, a (scalar) function of x that we want to maximize or minimize. E.g., the cost function which yields the path desired driving behaviour.
- c_i are *constraint* functions, which are scalar functions of x that define certain equations and inequalities that the unknown vector x must satisfy. E.g. the equation of turning radius or steering angle of the vehicle.

Where the optimization problem can be written as:

$$\min_{x \in R^n} f(x) \qquad \text{subject to} \quad c_i(x) = 0, \quad i \in \mathcal{E} \tag{2.10a}$$

$$c_i(x) \geq 0, \quad i \in \mathcal{I} \tag{2.10b}$$

Where $\mathcal{E}$ and $\mathcal{I}$ are sets of indices for equality and inequality constraints.

The objective function is defined from the variables, and are supposed to be maximized or minimized to find the solution. The constraints are also functions of the variables and are a set of equality- and inequality-equations that need to be satisfied in order to find a feasible solution. If the objective- and constraint-functions are all linear, the type of problem is known as a linear programming problem. If the objective function or some of the constraints are nonlinear, the problem is a nonlinear program [20].

2.5.2 Global and local optimization

A formal definition from [20] of the global minimizer of f is:

> A point x^* is a *global minimizer* if $f(x^*) \leq f(x)$ for all x

where x ranges over all over $\mathbb{R}^n$. A formal definition from [20] of the local minimizer of f is:

> A point x^* is a *local minimizer* if there is a neighborhood $\mathcal{N}$ of x^* such that $f(x^*) \leq f(x)$ for all $x \in \mathcal{N}$

where x^* is the vector which holds the optimal solution for the objective function [20]. Figure 2.18 illustrates the global minimum for all $\mathbb{R}^n$ and three local minimums of three separate neighborhoods $\mathcal{N}$ of x^*_{LM}.

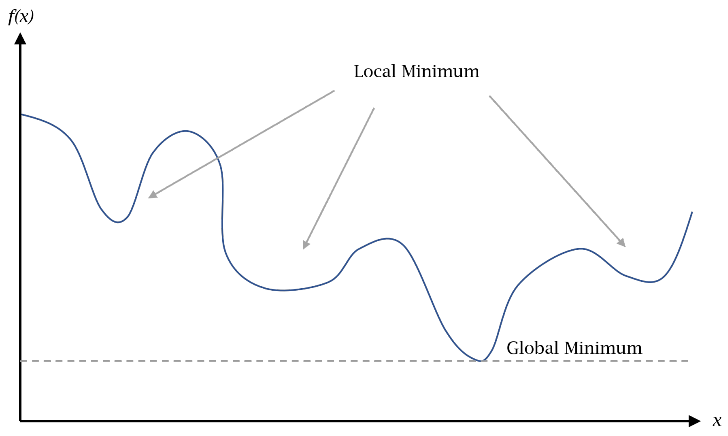


Figure 2.18: Illustrates the global and local minimums of an objective function f
.

When utilizing an optimizing algorithm on a model, one often prefers to find a global solution. The global solution is the lowest/highest function value among all feasible points. The local solution is the point where the objective function is smaller/higher than all other feasible points in the neighborhood $\mathcal{N}$. Many nonlinear optimizing algorithms seek only the local solution, because the global solution may be hard to recognize and even harder to find. This is not the case for linear programming problems because the local solutions are also global solution[20].

Finding the solution in optimization problems is done iteratively. The optimization algorithms start with an initial guess with variables and depending on the algorithm chosen, new variables are estimated for the next steps until the iterations terminate, hopefully at the solution. The objective function, constraints, and first- and second-gradients are all important factors of deciding the next variables. Some algorithms also use information gathered from previous steps to estimate further [20].

The choice of initial guess is essential, especially for a nonlinear optimization problem like path planning. If the optimization is used to smooth a path, the initial guess will be the discrete path that is found by the graph search algorithm. If the graph search finds a solution which is not optimal, the optimization algorithm will most likely iterate to a local solution, and not the global solution.

2.5.3 Gradient-based trajectory optimization

Assume one of the mentioned graph search algorithms returns a feasible path in a path planning problem. In some cases, it is not advantageous or practical to include all constraints for the model in the graph search algorithm. Trajectory optimizing addresses the problem of perturbing the trajectory while satisfying a set of constraints to improve the quality of the path. The methods of trajectory optimization fall under the nonlinear programming class [16].

To be able to use numerical computation methods to optimize the trajectory, the generated trajectories are specified in terms of parameter space. The optimization can then be compared to an incremental search in the parameter space while satisfying all constraints. The direction of motion in the perturbations of the trajectories is decided from the gradient of the objective function in the parameter space while respecting the constraints. Hence, the optimization of trajectories has similarities to Newton's method and gradient descent methods, where both methods are finding the next search direction based on the gradient- and second-gradient of the objective function. What differs between Newton's method and gradient descent is how they are selecting the step size of the search [16].

2.5.4 Gradient descent optimization

Gradient descent is an optimizing algorithm which produces a minimizing sequence $x^{(k)}, k = 1\ldots, n$ where

$$x^{(k+1)} = x^{(k)} + t^{(k)} \Delta x^{(k)} \tag{2.11}$$

and $t^{(k)} > 0$ (except when $x^{(k)}$ is optimal)[1]. Δx is called the step- or search direction, and $k = 0, 1, ..., n$ denote the iteration number of the optimization. $t^{(k)}$ is called the step size- or length at iteration k. Since the gradient descent method is a descent method, it can be said that

$$f(x^{(k+1)}) < f(x^{(k)}), \tag{2.12}$$

except when $x^{(k)}$ is optimal [1]. Equation 2.12 states in words that the next iteration in *gradient descent method* should always have lower value than the existing point. The gradient descent algorithm is as following:

Algorithm 1 *Gradient descent method*

```
Require: a starting point $x \in \mathbf{dom}\, f$
  repeat
    1. $\Delta x := -\nabla f(x)$
    2. Line search. Choose step size t via exact or backtracking line search.
    3. Update. $x := x + t\Delta x$
  until stopping criterion is satisfied
```

In the gradient descent method, Δx is chosen as the negative gradient of the objective function $-\nabla f(x)$. The step size t is either calculated from the *exact line search* or *backtracking line search*[1]. When the gradient and step size are calculated, the next point can be found from the minimizing sequence described in equation 2.11.

This process repeats until the stopping criteria are satisfied. A common stopping criteria is to use $||\nabla(f(x))||_2 \leq \eta$, where η is small and positive, but other criterions can also be used[1]. In other words, the step direction are heading downward the slope the objective function are creating until it reaches a valley [24]. How the gradient descent method is reaching the optimum x^* in f is illustrated in figure 2.19.

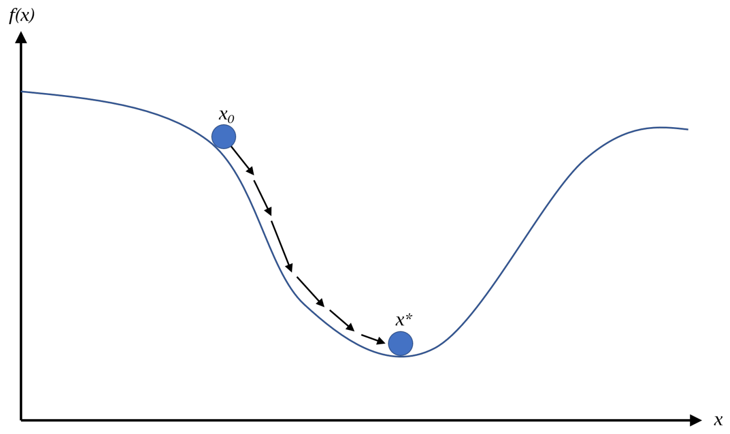


Figure 2.19: The gradient descent method finding a global minimum from an initial pose.

.

2.5.5 Conjugate gradient

Conjugate gradient method is a class withing optimization that can handle large-scale nonlinear problems in an effective way [19]. The conjugate gradient method is originally developed for the quadratic optimization problem, but are further extended to handle general optimization problems [10]. Some important features for the nonlinear conjugate gradient method are that they require no matrix storage and have fast convergence rate [20].

Algorithm 2 *Conjugate gradient - The Fletcher Reeves method*

```
Given $x_0$;
Evaluate $f_0 = f(x_0)$,$\nabla(f_0) = \nabla f(x_0)$;
Set $p_0 \leftarrow -\nabla f(x)$, $k \leftarrow 0$;
**while** $\nabla f_k \neq 0$ **do**
    Compute $\alpha_k$ and set $x_{k+1} = x_k + \alpha_k p_k$;
    Evaluate $\nabla f_{k+1}$;
        $\beta \leftarrow \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}$;    (2.13a)
        $p_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1} p_k$;    (2.13b)
        $k \leftarrow k+1$;    (2.13c)
**end while**
```

In algorithm 2, each iteration requires an evaluation of the objective function and gradient. Only a few vectors need to be stored for the next iteration, and no matrix operation is needed. The line search parameter, α_k, represents the step size of iteration k. From equation 2.13b, α_k needs to satisfy the *strong Wolfe* condition in order to have a descent search direction. The strong Wolfe condition is stated below:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k, \tag{2.14a}$$

$$|\nabla f(x_k + \alpha_k p_k)^T p_k| \leq -c_2 \nabla f_k^T p_k, \tag{2.14b}$$

Where $0 < c_1 < c_2 < \frac{1}{2}$. The inner product between the gradient vector ∇f_k and equation 2.13b are:

$$\nabla f_k^T p_k = -||\nabla f_k||^2 + \beta_k \nabla f_k^T p_{k-1}, \tag{2.15}$$

By applying lemma 5.6 from [20], it can be shown that any line search procedures that computes an α_k satisfying the equation set 2.14a, will ensure descent search direction for the objective function f.

Chapter 3

Method

The objective of this project is to perform optimal motion planning in terrain. FFI has already a complex UGV, which uses sensor-information to plan a path through the terrain. The motion planner for FFI uses a map that represents the environment as a grid, where each cell includes a cost based on the traversability degree. To find a path in the grid, an exploring-tree search is used by FFI to find the goal state. A disadvantage of using a tree search is that the number of nodes explored increases exponentially by the distance searched. This directly affects the motion planner's running-time and limits the planning horizon for each planning sequence. FFI is therefore interested to see if there exist promising alternatives, where one promising option may be to smooth a discrete graph search.

From subsection 2.2.4 and 2.2.5, A* search proved to be the most efficient and best graph search algorithm to use. This thesis will focus on an alternative motion planner for FFI that utilizes the A* algorithm to find an optimal path in a grid, where the grid cells are weighed with respect to terrain characteristics(as in [3] and [12]). Since A* finds a path in a two-dimensional grid, A*'s path is discrete, and the motion planner will apply an optimizing algorithm that smooths the path while keeping the path at low-cost ([5], [37]). The motion planner also needs to account for the vehicle's kinematic constraints, either in the path search ([17], [35]) or post-process the discrete path considering the curvature [5]. A practical challenge for the optimization problem is to design a cost-function that yields the path's desired driving behavior. This is a challenge because we want a cost-efficient path, but at the same time are smooth and keep a comfortable distance to high-cost areas.

The UGV navigates through the terrain with a GPS-set goal, which may be outside the sensors' range. If the goal location is outside the sensors' range, the UGV navigates towards waypoints which work as temporary goal location. As the UGV moves towards the goal, undetected terrain is perceived through the sensors, which means that the motion planner needs to be as efficient as possible and create optimal paths continuously. In this chapter, it is assumed that the environment from the vehicle to the goal is fully observable, thus generating waypoints is not needed.

This chapter will firstly present the available sensor and vehicle information received from FFI, and how this affects the path planning process. Further on, the implementation of A* will be elaborated, including the importance of choosing correct heuristic, calculating cost, dilation, and radial-constrain the A* search. Lastly, local optimization will be performed with gradient descent ([14], [5]).

3.1 Data and map representation

In this section, information about the data- and map-representation in this thesis will be presented. The first paragraph from this section are from the specialization project Motion Planning for Terrain Vehicles (2019)[2], and is included for the completeness of the thesis.

FFI has already a complex setup for the autonomous vehicle OLAV with a lot of sensors and data. A 65-seconds recording of a drive with OLAV was provided from FFI for this project. This recording contains all information the vehicle needs to perform its navigation, which includes poses relative to frames, camera images- and information, lidar sweeps, vehicle measurements, odometry information, and much more. The sensor information from both cameras and lidar are fused to create a cost field (see figure 3.1). This can be used as an input-map to determine the optimal path through the terrain for a search-algorithm.

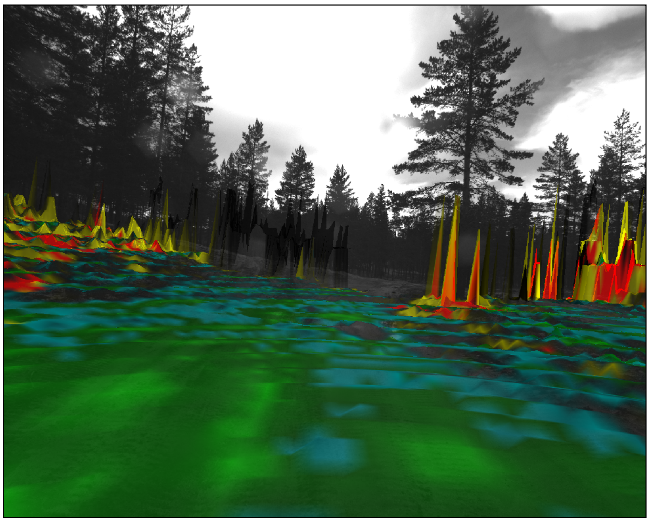


Figure 3.1: How OLAV perceives the vegetation

FFI generated the cost field with estimation from traversability cost, obstacles, attractive-potential, and distances. All this information was combined and are stored as a weighed 2D array with dimension 320x320, where -1 is undetected areas, and the numbers from 0 – 100 define the driveability at the different grid cells. Each cell represents an area of 0.25 meters, hence the 2D array represents an area of 80x80 meters.

To make the process of developing a search algorithm easier, the cost field can be visualized as a grayscale image. The level of brightness in the picture defines the driveability, where the brightest area is the preferred region to navigate in. The white areas are the undetected areas.

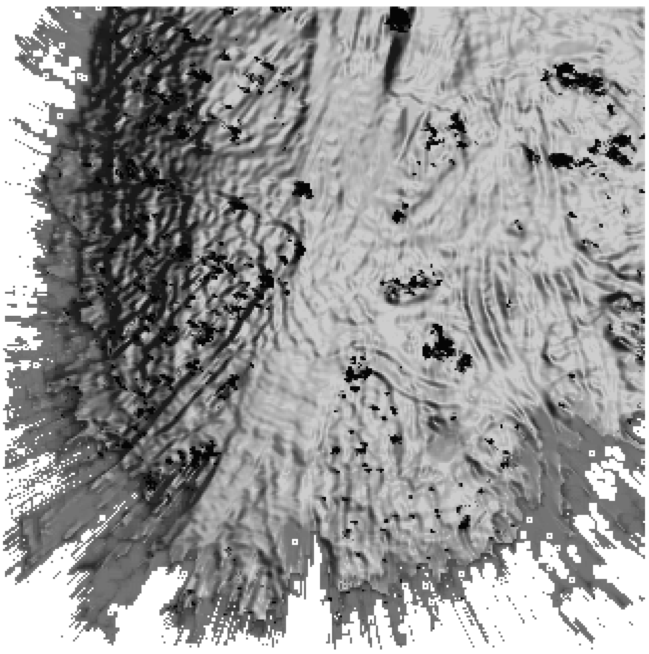


Figure 3.2: Map of the terrain

3.1.1 Polaris Ranger XP900 - Vehicle specifications

FFI uses a vehicle called Polaris Ranger XP900 for autonomous navigation. As a consequence of transforming the vehicle from being a regular ATV to a UGV, the car is restructured. Because of the modifications, the weight of the vehicle is changed, but this is not considered an important part of the path planning process. The essential specifications for the vehicle are the dimensions and the steering angle. These are listed below [23].

Width	152 cm
Height	193 cm
Length	296 cm
Length between axles	206 cm
Steering angle ϕ	± 0.5 rad

Table 3.1: Ranger CP 900 Specifications

The algorithm developed in this thesis is supposed to be generic, and will hopefully work for different vehicles with different specification. To change the algorithm to work for different vehicles, both the dimensions and steering angle needs to be redefined. The dimensions are important because of the spatial occupancy to the vehicle need to be considered in the path planning process, see figure 3.3 for illustration.

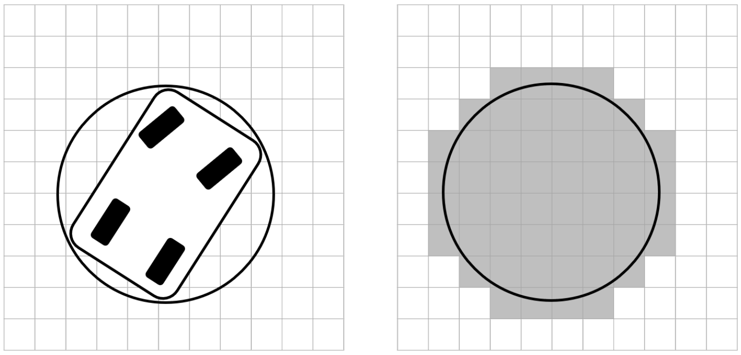


Figure 3.3: Illustrates the spatial occupancy for a vehicle in a grid. The occupancy is circle-shaped to consider all angles the vehicle can have at this location.

The optimal path depends on the area the vehicle is covering. If the vehicle is big, a greater distance needs to be considered for clearance against obstacles and high-cost areas than with a smaller car. The unit used for the dimensions in the algorithm are *cells*, where each cell covers an area of 25 cm. The most important element in the dimension of the vehicle is the diameter because this contributes to illustrate which area it covers when the vehicle rotates. The diameter is simply gathered with Pythagoras sentence:

$$Diameter = \sqrt{Length^2 + Width^2} = 332.74cm \tag{3.1}$$

$$Diameterincells = \frac{332.74cm}{25\frac{cm}{cell}} = 13.31cells \tag{3.2}$$

Since cells are considered whole, the diameter of the vehicle is 14 cell units in the discrete grid. The vehicle will maximum cover an area with a diameter of 14 cell units.

The steering angle is also varying from one vehicle to another. A vehicle with a big steering angle can navigate differently than one with a smaller steering angle. Different paths will then be produced for the different steering angles, where both paths can be feasible for one vehicle, but not the other.

3.2 A* implementation

In section 2.2, it was explained that a graph search algorithm can be applied to a 2D coordinate system. The same can be done to the costmap of terrain traversability, where the nodes are located mid-center of each cell in the grid. The edges which are connecting the nodes are weighed with the cost of a driveability of the terrain between the nodes. The transformation from a costmap to a weighed grid-graph is illustrated in figure 3.4.

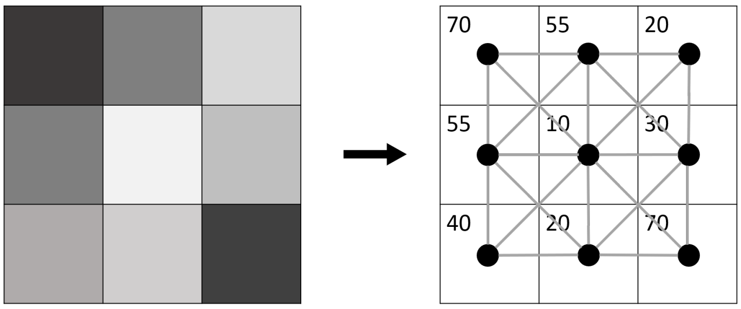


Figure 3.4: From map to graph

As the A* algorithm is a discrete search algorithm, the search for a path and choice of the next cell is limited, either with 4- or 8 potential descendants. To allow the algorithm to create paths that yield closest to a non-holonomic vehicle driving behavior, it is considered for this path planner that a cell has 8 neighbors, thus A* is searching in horizontal, vertical, and diagonal directions.

The A* algorithm is an informed graph search algorithm that uses a heuristic as a guide to finding the shortest path to a goal. This algorithm requires a graph, and a start- and a goal-location located inside this graph to execute. For the UGV navigating in the terrain, the input graph is the costmap of the terrain, the start location is the GPS position of the UGV, and the goal location is the GPS-set goal location(or the temporary waypoints). The algorithm stores its information in two sets, the *Open* set, a queue which prioritizes which traversability cell to expand next, and one *Closed* set where the expanded cells are added to. In the *Closed* set, information about where the cell was expanded from are stored i.e. its predecessor, and how expensive it is to get to this cell location.

Furthermore, the *while* loop is looping until *Open* is empty. This loop is starting at the first cell in the priority queue (which is the start state in the initialization). For each iteration, A* checks if the current cell is at the goal state, if it is, the algorithms have reached its target and the search can terminate. If current ≠ goal state, A* iterates through all adjacent cells of the current cell, horizontal, vertical, and diagonal. The cost (see subsection 3.2.1) for each neighboring cell is calculated and evaluated iff: this cell has not been visited before *or*, the cost to this cell is less than the already-stored cost traveling to this cell. If any of these conditions are met, the adjacent cell is added to the *Open*-set together with the *new* calculated cost. The current cell is also stored as the predecessor of the adjacent cell to be able to backtrack which route led to this cost. The implemented algorithm is stated below:

Algorithm 3 *The A* Algorithm*

```
1: Given $x_s \cap x_g \in \mathcal{M}$
2: Set Open $\longleftarrow \emptyset$, Closed $\longleftarrow \emptyset$, Predecessor($x_s$) $\longleftarrow \emptyset$, Open.put($x_s$, 0)
3: while not Open $\neq \emptyset$ do
4:    current = Open.popMin()
5:    Closed.push(current)
6:    if current = $x_g$ then
7:       break
8:    end if
9:    for next in current.adjacent() $\in \mathcal{M}$ do
10:      $g \longleftarrow g(current) + cost(current, next)$
11:      if next $\not\subseteq$ Closed or $g < g(next)$ then
12:         $g(next) \longleftarrow g$
13:         $h(next) \longleftarrow Heuristic(next, goal)$
14:         Open.put(next, $g(next) + h(next)$)
15:         Predecessor(next) $\longleftarrow$ current
16:      end if
17:   end for
18: end while
```

Figure 3.5 shows a solution path found with the implemented A* from algorithm 3.

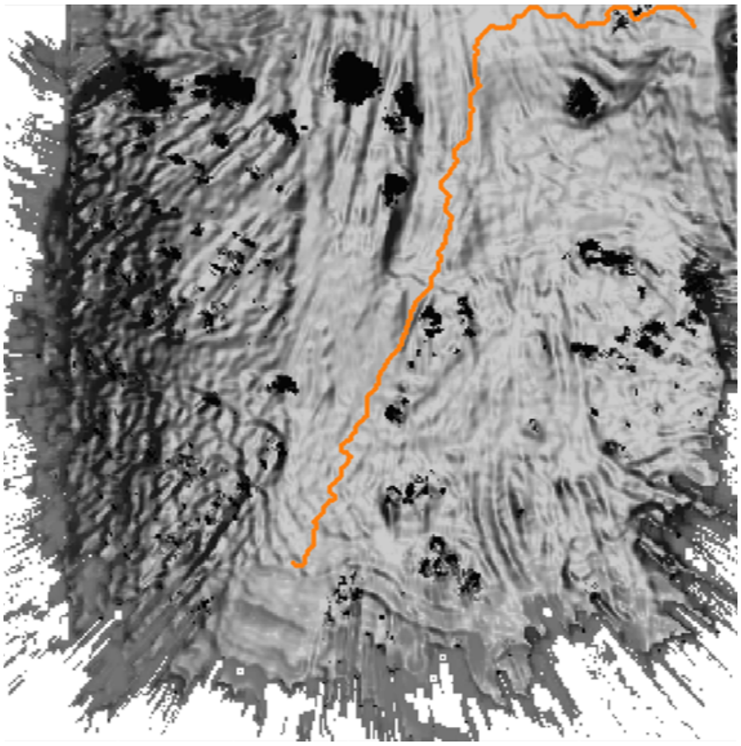


Figure 3.5: Shows a solution path of A* from a start- to a goal-location

3.2.1 Traversability cost

In algorithm 3 the $cost(x_1, x_2)$-method are anticipated to simulate the vehicle moving through the terrain from one cell node to another. Figure 3.6 illustrates a discrete example of a path through a cost grid. The cost from x_1 to x_2 depends on the driveability in this cell and the segment length in both cells. In the illustrated discrete example, there are two types of segments, one horizontal/vertical and one diagonal, where the length(diagonal) = $\sqrt{2} \cdot$ length(horizontal). Setting the pixel length to 1 gives these two costs:

$$cost(x_1, x_2) = \frac{20 + 10}{2} \cdot 1 = 15 \tag{3.3}$$

$$cost(x_2, x_3) = \frac{10 + 20}{2} \cdot \sqrt{2} = 21,2132 \tag{3.4}$$

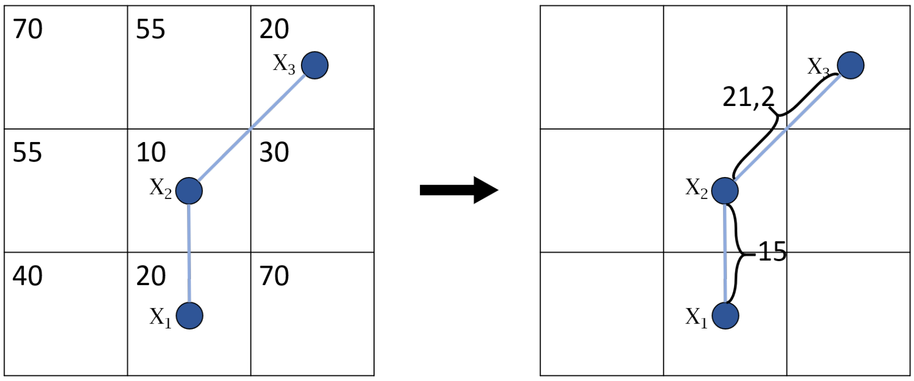


Figure 3.6: Discrete cost illustration

3.2.2 Heuristic

The heuristic function which is used in A* depends on how the algorithm searches. If the algorithm is only allowed to search horizontal or vertical, the best heuristic to use is the Manhattan distance. The Manhattan distance (equation 2.2) estimates the total x- and y-units moved from one location to another, i.e. horizontal and vertical directions. Manhattan distance is the best choice for horizontal and vertical search because this heuristic yields the resulting path behavior for this type of search. If the A* algorithm allows diagonal search, the Euclidian distance is the best heuristic estimate. This heuristic calculates the straight line distance from one state to another, thus yields the shortest path in a continuous environment.

By using a heuristic function between node x_1 to x_3 in figure 3.6 would estimate a distance of 3 with the Manhattan distance, and 2.24 with the Euclidean distance. The real distance in figure 3.6 is $1 + \sqrt{2} = 2.41$. This means that only the Euclidean distance is an admissible heuristic for this problem, and the Manhattan are not, as this overestimate the cost to from x_1 to x_3.

3.2.3 Map dilation

The regular A* algorithm finds the optimal path in the map of the terrain. This solution path is generated solely based on the cost of the different cell the path are directly crossing(, and the heuristic). The solution path does not take account of the surrounding cells, how close the path is high-cost cells, or too-narrow passages. From subsection 3.1.1, it was stated that the vehicle from FFI has a width of approximately 7 cells, which means that the UGV will cover a larger area that the solution from A* is first based on, thus have a larger cost than first anticipated. The generated optimal path may therefore not be optimal for this specific vehicle. When the UGV is tracking the optimal path, it will pass over more cost-expensive cells than the cost of the solution path is stating. Figure 3.7 shows zoomed-in sections from 3.5 where the A* algorithm generated a solution path without accounting the cost of neighboring cells.

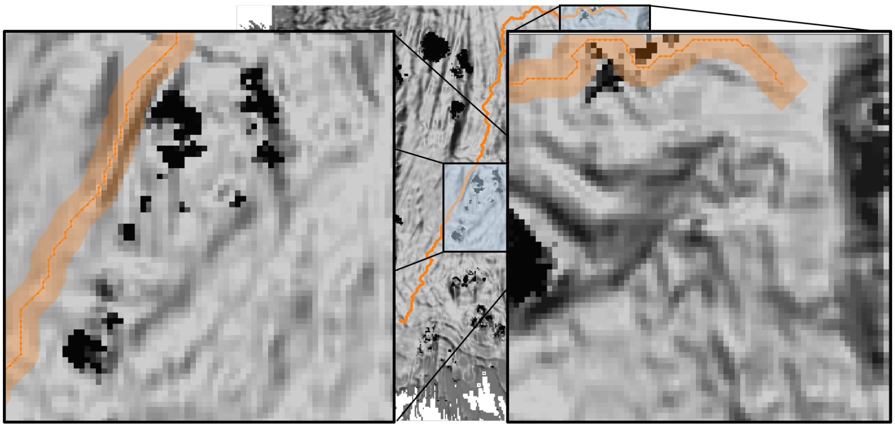


Figure 3.7: Illustrated segments of solution paths from A*. The orange-shaded area illustrates the width of the vehicle if it was to drive this path.

Figure 3.7 shows that a cheap path may be in the neighborhood of high-cost cells, which will make the solution path expensive for a vehicle to track. To address the issue of keeping a clearance of high-cost cells and narrow passages, the traversability map can be pre-processed by dilation.

The size of the kernel in the dilation process will represent the size of the vehicle which is going to follow the map. In the dilation process, the kernel is choosing the largest cell value inside the kernels range and moving this value to the new dilated picture. The dilation are performed with the *opencv* module in Python [9].

The dilated picture inflates the most expensive cell inside an area with the spatial occupancy of the vehicle in focus. By using the dilated map as an input map to A* instead of the full-scale terrain map, the solution path will keep a clearance against the most expensive cells and narrow passages. Figure 3.9 illustrates a solution path from A* in dilated map of figure 3.5. The kernel is chosen based on FFI's vehicle's dimension, where the diameter of the spatial occupancy is approximately 14 cells, as mentioned in subsection 3.1.1. The kernel has a shape of a circle to take precaution that the vehicle may counter high cost cells from all directions, and then maximizing the possibility for A* to generate the safest route for the vehicle. In figure 3.8, is is shown a dilated costmap originated from 3.2.

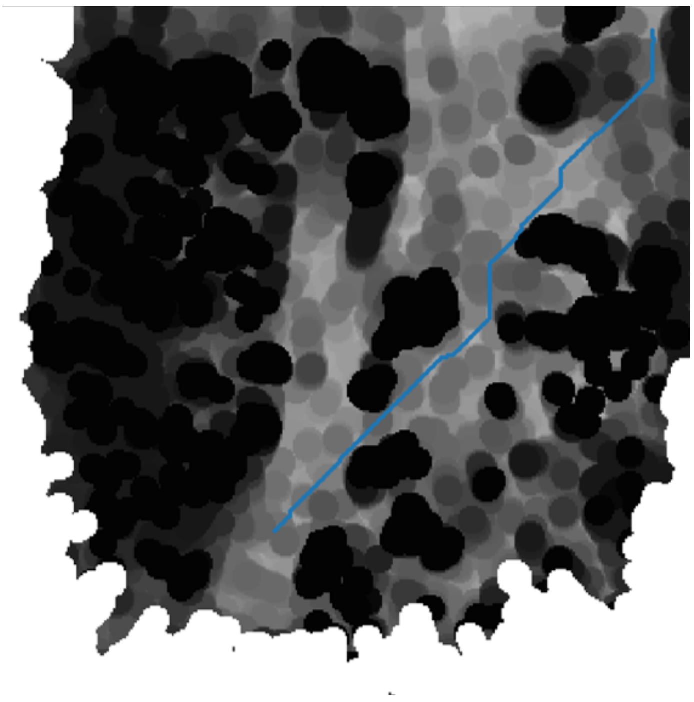


Figure 3.8: Solution path from A* in a dilated map, with a kernel diameter of 14 cells.

Figure 3.8 shows a solution path from A* in a dilated costmap. In the figure below, the path from the original- and dilated-costmap are compared in the original costmap.

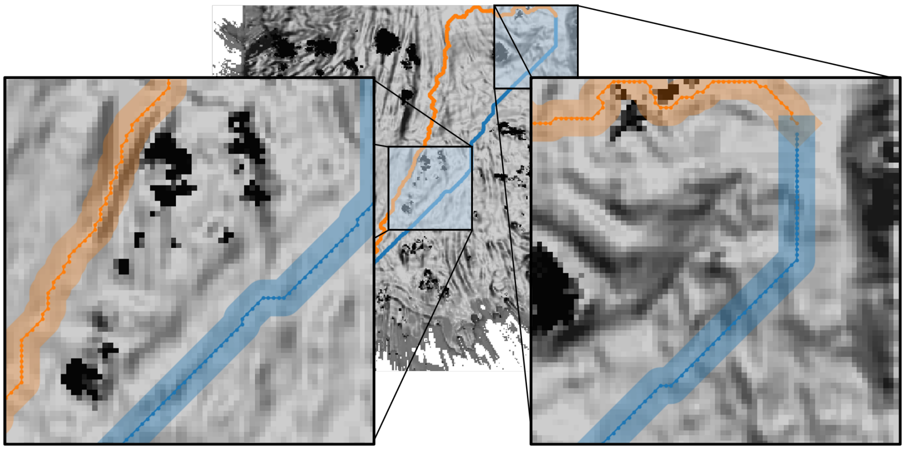


Figure 3.9: Solution paths from A* in a dilated- and original-costmap. The paths are shaded with the vehicles width of 7 cell lengths.

It can be seen that two entirely different paths have been generated from the same start- to the goal-location. The orange path generates a path that is really close to black/dark cells, while the blue path keeps a good clearance against the darkest cells, hence generates a more driveable option for the vehicle in focus.

3.2.4 Node selection based on minimum turning radius

The vehicle which is used by FFI is constrained by the steering angle. The front wheels can not turn further than ± 0.5 rad. From equation 2.4 and with the distance between the axles at 2.06 meters, we get a turning radius of 3.77 meters which are approximately 15 cell-lengths in the grid. Since the vehicle has a constraint in the steering angle, not all paths are feasible for this specific UGV. This limitation can be taken advantage of in the A* search, by pruning nodes which lead to infeasible paths.

An approach to consider the vehicles turning radius in the search is to implement an *if*- statement inside A*'s *while*-loop where it is evaluating the *next*-node, i.e. the neighbour of *current* in algorithm 3. This *if*-statement contains a function which is checking if the location of *next*-node and its predecessors are creating a feasible segment based on the allowable turning radius for the resulting path. If the segment is infeasible, the turning radius is less than the minimum allowed turning radius, and the *next*- node will not be added to the *Open*-set.

The *check_radius*-function calculates the potential turning radius for all adjacent cells to the *current* node. If the calculated turning radius is less than the minimum allowed turning radius, the node is not added as a descendant to the current node. One way to calculate the turning radius in the path is to use the circle equation. If three points are obtained in a grid, there exists a circle that is crossing all three points. The three points have the same distance to the center of this circle. The radius of this circle will also be the same as the turning radius if the three obtained points are in order, hence, forms a path. The equations which is used in the *check_radius*() are elaborated in subsection 2.3.2.

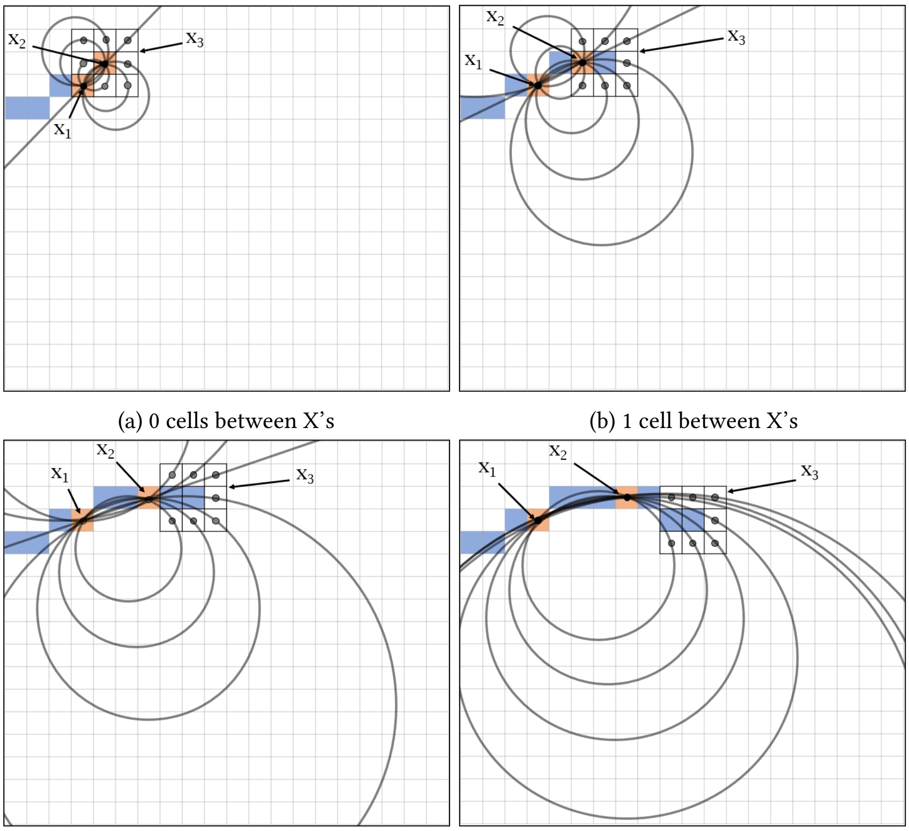
Algorithm 4 *The Check Radius function*

```
Given $X_1, X_2, X_3$, $minimum\_turning\_radius$
Set $(x_1, y_1) = X_1$, $(x_2, y_2) = X_2$, $(x_3, y_3) = X_3$
    $x_c = \frac{(y_3 - y_1)(x_2^2 + y_2^2 - x_1^2 - y_1^2)) + (y_1 - y_2)(x_3^2 + y_3^2 - x_1^2 - y_1^2)}{2((x_1 - x_3)(y_2 - y_1) + (x_1 - x_2)(y_1 - y_3))}$
    $y_c = \frac{x_2^2 + y_2^2 - x_1^2 - y_1^2 + x_c(x_1 - x_2)}{2(y_2 - y_1)}$
    $\rho = \sqrt{(x_1 - x_c)^2 + (y_1 - y_c)^2}$

if $\rho \geq minimum\_turning\_radius$ then
    return True
else
    return False
end if
```

To be able to use the suggested method in a discrete search algorithm, the choice of the variables X_1, X_2 and X_3 are critical. If the radius is checked for three succeeding points, the resulting turning radius is either very small or infinity, see figure 3.10a. If a $check\_radius$ function is implemented for X_1, X_2 and X_3 and the allowable turning radius is bigger than the greatest circle in 3.10a, the algorithm will never find the goal, because it will only add *next* nodes which creates a straight line from the existing path to the $Closed$-set.

The choice of X_i ($i \in [1, 2, 3]$) needs to be considered carefully. Instead of choosing three succeeding point, every other, second or third (or more) may be chosen as in figure 3.10b, 3.10c and 3.10d. By varying the distance between the cells, it allows the algorithm to be more generic, and adapt to fit a greater selection of vehicles (and steering angles). If the algorithm is generating a path for a vehicle with a great steering angle, it is not necessary with a large distance between the cells for the choice of X_i. But if the steering angle is very limited, it has to be a certain distance between the cells to choose the X's for the path to reach the goal. FFI's vehicle requires (3.77 [meters] $\cdot$ 4 $\frac{[cells]}{[meters]} \approx$) 15 cells as turning radius, thus counting from 3.10 require 3 cells or more between the X's to successfully generate a path to the goal.



(a) 0 cells between X's

(b) 1 cell between X's

(c) 2 cells between X's

(d) 3 cells between X's

Figure 3.10: Four scenarios of simulating the turning radius of three-point circles. Two of the points are the orange cells, the third represents the *next* node to be evaluated.

To emphasize, the *check_radius* function is added to prune candidates who lead to an infeasible solution based on the steering angle/turning radius. The heuristic and cost are still calculated as in the regular A* algorithm, and the path which has the lowest cost is chosen to be the solution path. The location where the *check_radius*-function is implemented is shown in algorithm 5.

Algorithm 5 *The A* Algorithm - Turning Radius Limitation*

```
...
if next $\not\subseteq$ Closed or $g < g(next)$ then
    if check_radius(Closed, current, next, $minimum\_turning\_radius$) then
        $g(next) \leftarrow g$
        $h(next) \leftarrow Heuristic(next, goal)$
        Open.put(next, $g(next) + h(next)$)
        Predecessor(next) $\leftarrow$ current
    end if
end if
...
```

3.3 Trajectory optimization

The implemented A* algorithm generates a discrete path for a continuous environment. This path is cost-efficient, keeps a good clearance to obstacles and narrow passages, and respects the maximum steering angle for the vehicle. Since the path is generated in a two-dimensional grid, the coordinates for the path are discrete and not continuous. As a result of the discrete coordinates, the path generated from the A* algorithm is piecewise linear, and not a preferred route for a vehicle. To make the path more driveable, A*'s solution path can be smoothed by optimization.

The smoothing of the path can be treated as an optimization problem. The initial guess, X_0, will be the solution path from the A* algorithm. The nodes that forms the path are the variables that needs to be optimized. If the turning radius in the solution path is too low, the solution is not feasible, thus, the turning radius can be a constraint in the optimization problem. The cost function F, or object function, sets a quantitative measure of what to prioritize in the optimization problem. The cost function F should reflect the vehicles desired driving behavior, which is to have a smooth path, however, by smoothing the path, other characteristics need to be maintained as keeping the total cost of the path low, and enforcing the non-holonomic constraints of the vehicle. The cost function F can be designed by three terms with respect to the path. The cost function are described below([5],[14]):

$$F = F_{smo} + F_{cost} + F_{cur} \tag{3.5}$$

Each of the terms in the cost function 3.5 serves its own purpose, which will be explained in more detail.

The cost function F has to be minimized in order to optimize the trajectory and find a feasible solution. The solution can be minimized iteratively by finding the gradient of the objective function. The gradient shows which direction each variable needs to be perturbed in order to maximize or minimize the cost for each step. The method used in this section is a gradient-based optimization algorithm which is explained in more detail in subsection 2.5.3.

3.3.1 Smoothing term

The F_{smo} is a measure of smoothness in the path. The term penalizes unequal succeeding displacement vectors. The smoothness term will assign a high cost if the transition from one vector to the next is abrupt, thus rewarding the node if the transition is smooth. The term is as following[5]:

$$F_{smo} = w_s \sum_{i=1}^{N-1} (\Delta x_{i+1} - \Delta x_i)^2 \tag{3.6}$$

where w_s are a weight that influences the impact on the path. The sequence of nodes is defined as $x_i = (x_i, y_i)$, $i \in [1, N]$, and the displacement vector as $\Delta x_i = x_i - x_{i-1}$. The gradient of the smoothness term at node x_i gives the direction to perturb node x_i to smooth the angle between the displacement vectors Δx_i and Δx_{i+1}. The gradient of F_{smo} is given below and the impact of the perturbations is decided by the weight w_s.

$$\frac{\partial}{\partial x_i} F_{smo} = \frac{\partial}{\partial x_i} (\Delta x_{i+1} - \Delta x_i)^2 \tag{3.7}$$

$$\frac{\partial}{\partial x_i} F_{smo} = \frac{\partial}{\partial x_i} (x_{i+1}^2 + 4x_i^2 + x_{i-1}^2 - 4x_{i+1}x_i + 2x_{i+1}x_{i-1} - 4x_i x_{i-1}) \tag{3.8}$$

$$\frac{\partial}{\partial x_i} F_{smo} = -4x_{i+1} + 8x_i - 4x_{i-1} \tag{3.9}$$

Figure 3.11 shows smoothing of a path. The initial guess is the discrete black path, and the blue paths illustrate different levels of smoothing. The smoothing gradient are perturbing the nodes to form a straight path from the start- to the goal-location. To prevent the smoothing to form a straight line, a *tolerance* can be used as a stopping criterion. This tolerance can e.g. be evaluated from the object function F_{smo} or the paths curvature.

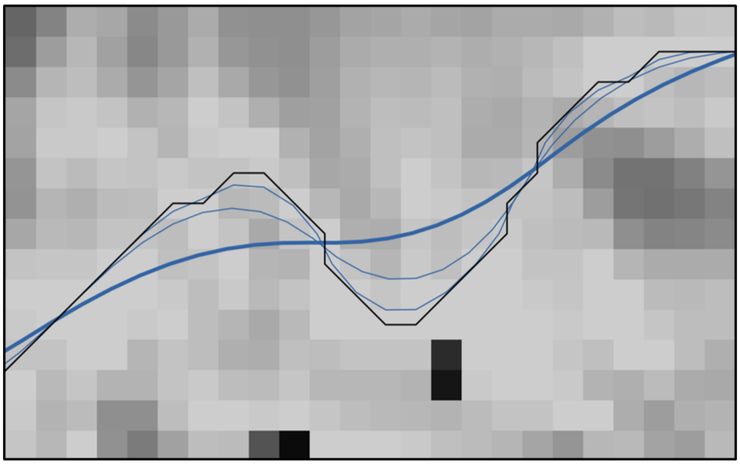


Figure 3.11: Shows different stages of smoothing

3.3.2 Cost term

The cost term is a quantitative measure of how expensive it is to follow the given path. This term penalizes expensive paths and reward cheap ones. By using an averaged map with kernel size reflecting the vehicle's size to calculate F_{cost}, the size of the vehicle is also considered in the calculations.

$$F_{cost} = w_o \sum_{i=1}^{N} cost(x_i, x_{i-1}) \tag{3.10}$$

The gradient of F_{cost} is the directions to perturb the path to reduce the cost. The gradient for each node can be found by using a gradient field, an array of vectors that gives the direction towards the point with the highest ascent at that coordinate. Creating a gradient field of a dilated array will not serve the purpose of the gradient, as big areas of the dilated map are monotone. By using an averaging filter, instead of a maximum filter for the dilation, the contours and differences are more enhanced in the terrain map. The averaging filter will have the same shape as the vehicle's spatial occupancy, simulating what the cost would be if it was located at that location. A gradient field can then be created from the averaged array which will help to guide the path more efficiently towards cheap paths. By using a gradient field in the optimization, it is utilized gradient-information that is already available and is obtained inexpensively as stated in [37]. The weight w_o controls F_{cost} impact on the path.

3.3.3 Curvature term

The curvature term upper bounds the curvature of the trajectory at every node and enforces the non-holonomic nature of the UGV. F_{cur} evaluates the curvature at every transition from one displacement vector to the next and checks if it is smaller than a maximum curvature. The term penalizes nodes that have a curvature $\frac{\Delta\phi_i}{|\Delta x_i|}$ bigger than κ_{max} [5].

$$F_{cur} = w_\kappa \sum_{i=1}^{N-1} \left(\frac{\Delta\phi_i}{|\Delta x_i|} - \kappa_{max} \right) \tag{3.11}$$

κ_{max} is defined from the vehicle's maximum turning radius which is 0.5 rad, hence $\kappa_{max} = \frac{0.5rad}{2m \cdot 4\frac{cells}{m}} = 0.0625$ radians per cell. The curvature weight w_c controls the impact of this term at the solution path.

The change of the angle between the segments at node i can be expressed as:

$$\Delta\phi_i = \cos^{-1}\left(\frac{\Delta x_i^T \Delta x_{i+1}}{|\Delta x_i||\Delta x_{i+1}|}\right) \tag{3.12}$$

To find which direction to perturb node x_i to enforce the curvature, the derivatives of the curvature needs to be found with respect to three points: x_{i-1}, x_i and x_{i+1}. The gradient will then be

$$\frac{\partial}{\partial \kappa_i} F_{cur} = \sigma_\kappa (\frac{\partial \kappa_i}{\partial x_{i-1}} + \frac{\partial \kappa_i}{\partial x_i} + \frac{\partial \kappa_i}{\partial x_{i+1}}) \tag{3.13}$$

and the derivatives of the curvature with respect to the tree points are [5]

$$\frac{\partial \kappa_i}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\frac{\Delta\phi_i}{|\Delta x_i|} - \kappa_{max}\right) = \frac{1}{|\Delta x_i|} \cdot \Delta\phi_i' + \Delta\phi_i \cdot \left(\frac{1}{|\Delta x_i|}\right)' \tag{3.14}$$

$$\frac{\partial \kappa_i}{\partial x_i} = -\frac{1}{|\Delta x_i|} \cdot \left(\frac{\partial \Delta\phi_i}{\partial \cos(\Delta\phi_i)} \frac{\partial \cos(\Delta\phi_i)}{\partial x_i}\right) - \Delta\phi_i \cdot \frac{1}{|\Delta x_i|^2} \tag{3.15a}$$

$$\frac{\partial \kappa_i}{\partial x_{i-1}} = -\frac{1}{|\Delta x_i|} \cdot \left(\frac{\partial \Delta\phi_i}{\partial \cos(\Delta\phi_i)} \frac{\partial \cos(\Delta\phi_i)}{\partial x_{i-1}}\right) + \Delta\phi_i \cdot \frac{1}{|\Delta x_i|^2} \tag{3.15b}$$

$$\frac{\partial \kappa_i}{\partial x_{i+1}} = -\frac{1}{|\Delta x_i|} \cdot \left(\frac{\partial \Delta\phi_i}{\partial \cos(\Delta\phi_i)} \frac{\partial \cos(\Delta\phi_i)}{\partial x_{i+1}}\right) \tag{3.15c}$$

3.3.4 Gradient descent optimization

As the gradient of all three terms from the objective function are obtained, an optimization algorithm can be applied to the discrete path. This algorithm are iterating over the initial path, using the gradients in each node and searching for a local minimum in the neighbourhood based on the objective function F. Normally, in a linear or quadratic problem, an optimization problem will most likely converge in a local/global minimum. But as this problem is highly non-linear with using three independent terms in F, another stopping criterion need to be applied to terminate the gradient descent algorithm. A stopping criterion can e.g. be a constant number of iterations, or when the whole trajectory are within bounds of the non-holonomic constraints. The algorithm for optimization F is stated below and inspired by [14]:

Algorithm 6 *Gradient Descent Optimization*

```
Given $(x_k, k \in [0, N]) \in X_0$
Set $\alpha$,$w_o$,$w_s$,$w_c$, $maxIterations$, $Iterations \leftarrow 0$
*fakePath*$\leftarrow$ copy($X_0$),*newPath*$\leftarrow$ copy($X_0$)
**while** Iterations<maxIterations **do**
    **for** i **in** range(1,N-1) **do**
        *gradient* $\leftarrow$ (0,0)
        *gradient* $\leftarrow$ *gradient* - $w_s \cdot$ SmoGrad(newPath[i-1],newPath[i],newPath[i+1])
        *gradient* $\leftarrow$ *gradient* - $w_c \cdot$ CurvGrad(newPath[i-1],newPath[i],newPath[i+1])
        *gradient* $\leftarrow$ *gradient* + $w_o \cdot$ CostGrad(newPath[i-1],newPath[i],newPath[i+1])
        fakePath[i] $\leftarrow$ newPath[i] + $\frac{\alpha}{w_o+w_s+w_c} \cdot gradient$
    **end for**
    newPath = fakePath
    *Iterations* $\leftarrow$ *Iterations* + 1
**end while**
```

This algorithm is given a discrete path from a path planner as an input. Further, the step-size α, the curvature-, smoothing- and cost-weight are set, as well as maximum iterations. Copies of the existing path are taken to be able to find the correction of node *i+1* without considering the correction of the previous node. The algorithm then iterates the pre-set maximum iterations, node for node calculating the gradient from smoothing term, curvature term, and cost term. All the gradients are added to the corresponding node in a fake copy of the path, such that the gradients are not influenced by each other in the same iteration. The changes are added at the end of each iteration, such that all node changes in the path are done simultaneously.

3.4 Overview of the motion planner

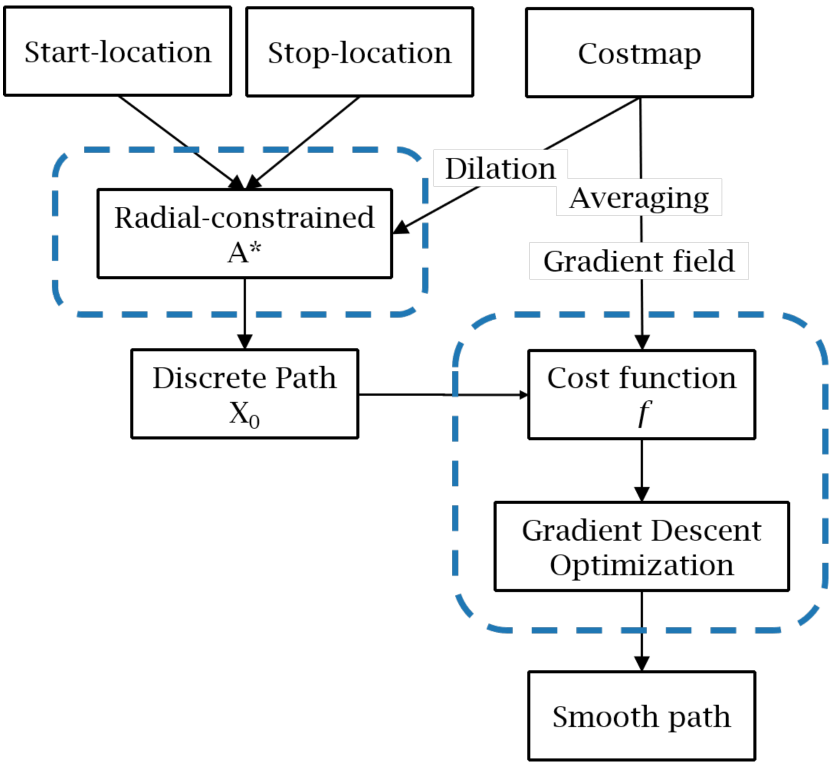


Figure 3.12: Shows a flowchart of the path planning process conducted in this thesis. The blue-dotted box to the left is the path planner, finding a discrete and cheap path that respects non-holonomic steering limitations. The blue-dotted box to the right is the trajectory optimizer, which uses a cost function that yields the vehicle's driving behavior. The gradient of this cost function is calculated and used in gradient descent optimization, outputting a smooth and driveable path.

Chapter 4

Results and discussion

During the development of the motion planner, different experiments were conducted to examine the aspects of the path planner. Some of the experiments show the importance of restricting the turning radius, and some show the effect of optimizing a path based on traversability cost.

The algorithms will be tested in the same environment as in chapter 3, in a traversability map where the brightness of the map defines the driveability. To analyze if the execution is successful or not, the quality of the path will be evaluated in the form of the total cost, smallest turning radius, total length, and cost per length. The radius will be compared to the allowable turning radius for FFI's vehicle, which is 15 cells. The units which are used are cell-lengths from the costmap, where one unit is 0.25 meters and the dimension of the maps is 320x320. For clarification, the word feasible will be used of both discrete and continuous paths if the minimum turning radius criterion is met.

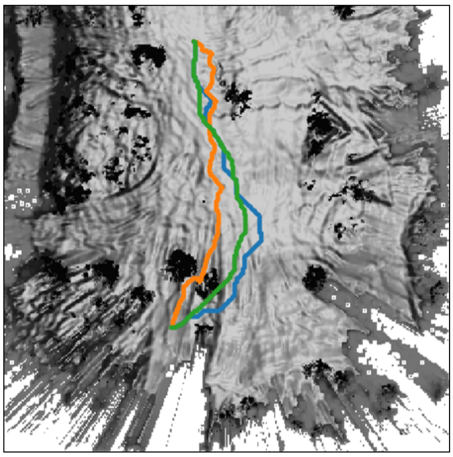
The total cost will be calculated from a traversability map modified by averaging (subsection2.4.2) in all test-cases. This map is used to add consistency to the comparisons, using a costmap that considers the vehicle's width, and have not been applied as an input in any graph-search. The total cost of each path can then be compared relative to each other to analyze the performance.

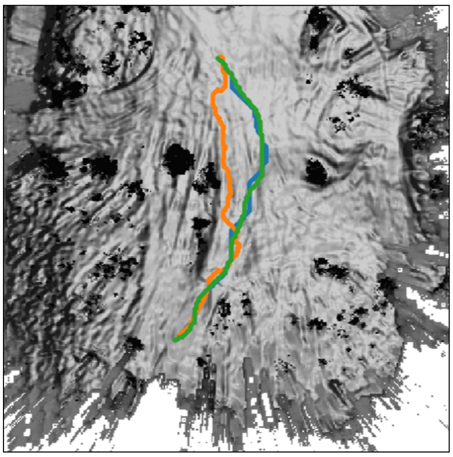
Run-time has not been the main focus while implementing the path planner. Yet, to get a perspective in how fast the paths are generated and optimized relative to each other, the run-time for the different searches will also be stated, but not thoroughly discussed.

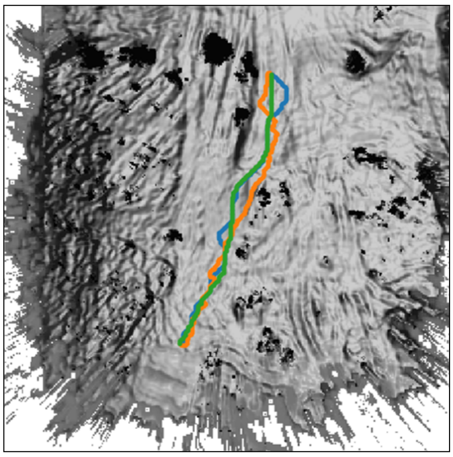
4.1 Discrete radial-constrained A* algorithm

In Section 3.2, it was proposed two eminent changes to the discrete A* search in the costmap. One change was to dilate the costmap, keeping a clearance from high-cost cells and narrow passages. The other change was to modify the A* algorithm, pruning nodes which led to infeasible paths according to a minimum turning radius. This section will analyze the consequence of these changes, by generating three different paths from three types of algorithms. The first path (orange), is generated by the regular A* algorithm in a traversability map. The second path (blue), is also generated by the regular A* algorithm, but in a dilated traversability map. The third path (green), is generated by the radial-constrained A* algorithm in a dilated traversability map.

The three algorithms are tested in four different maps with the same start- and stop-locations. The paths are visualized in figure 4.1. Figure 4.2 shows the paths in a regular costmap compared to a dilated costmap.


(a) Map 1


(b) Map 2


(c) Map 3

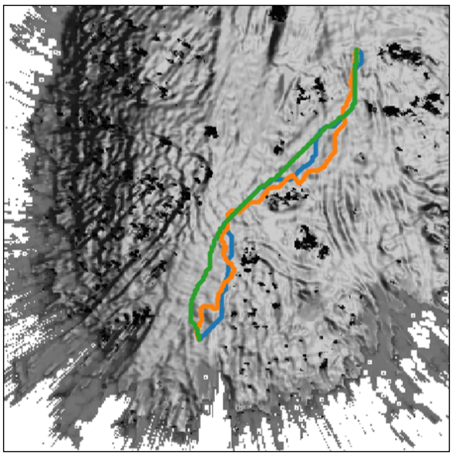
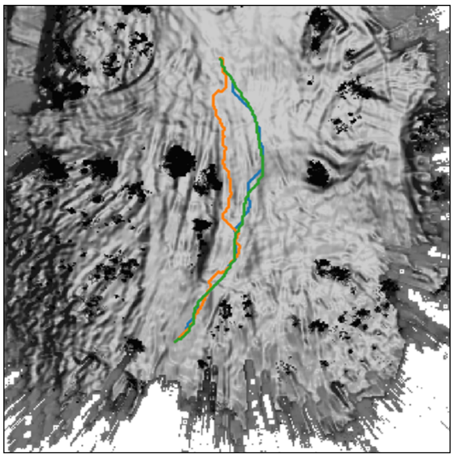

(d) Map 4

Figure 4.1: **Orange**: Regular A* path. **Blue**: Path from A* at dilated map. **Green**: Path with radial constriction in A* at dilated map.


(a) *Map 2.* Three paths in regular costmap.

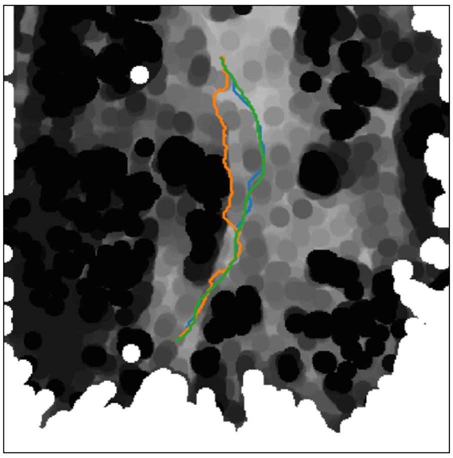

(b) *Map 2.* Three paths in dilated costmap.

Figure 4.2: Shows the environment where the three paths are created. The orange path uses the map from 4.2a as input-graph, while the two other paths use the map from 4.2b as input.

	Total Cost	Smallest Radius	Path Length	Run-time
Regular A*	4506.3	2.9	251.4	1.35s
Regular A* in dilated map	3258.0	5.0	269.0	1.90s
Radial-Constrained A*	4287.0	15.8	249.6	2.14s

Table 4.1: Table for map1

In figure 4.1a, it can be seen that the orange path is choosing narrow passages, resulting in the most expensive choice based on cost (from table 4.1). The orange path has also too sharp turns, with the smallest turning radius at 2.9 units, which will not be feasible for FFI's vehicle to follow. The regular A* search in the dilated map, the blue path, is the cheapest choice, but the turning radius is not legal. The radial-constrained search finds the path with a turning radius at over 15 units, but an expense, it has to pass over an expensive area with very low driveability (dark cells).

	Total Cost	Smallest Radius	Path Length	Run-time
Regular A*	3520.4	4.0	251.4	1.32s
Regular A* in dilated map	2543.9	5.0	241.6	1.65s
Radial-Constrained A*	2699.2	15.8	240.3	1.81s

Table 4.2: Table for map2

In figure 4.1b, the orange path is choosing a path close to high-cost areas, while the green and blue paths are choosing approximately the same route but with good clearance to the high-cost areas. These choices end in a significant difference between the paths generated in the dilated map versus the path generated in the map *not* considering the car's dimension. The green path satisfies the minimum turning radius due to the radial-constriction.

	Total Cost	Smallest Radius	Path Length	Run-time
Regular A*	3347.8	2.9	244.7	1.19s
Regular A* in dilated map	2485.7	5.0	232.6	1.64s
Radial-Constrained A*	2486.2	15.8	223.1	1.71s

Table 4.3: Table for map3

	Total Cost	Smallest Radius	Path Length	Run-time
Regular A*	3959.8	2.8	306.4	1.49s
Regular A* in dilated map	3253.6	4.3	264.8	1.67s
Radial-Constrained A*	3263.0	15.8	259.1	1.80s

Table 4.4: Table for map4

In the two remaining figures, 4.1c and 4.1d, the behavior of the path are approximately the same as explained to figure 4.1b. The orange paths are taking many too-sharp turns to reach the cheap areas in the regular A* search, while the blue and green paths are choosing nearly the same paths. The green paths deviate from the blue paths where the blue trajectories are turning too sharp, this is where the *check_radius()*-function prunes infeasible paths (from algorithm 4).

From the conducted tests, it clearly exists a pattern in the performance of the three algorithms. For the orange paths, the A* algorithm is directly choosing its optimal path (assuming the heuristic is admissible) from the costmap, but this is not the cheapest- or a feasible path for a non-holonomic vehicle. For the blue paths, A* is choosing the optimal path from the dilated costmap, but this path does not consider the vehicle's non-holonomic constraint. For the green paths, A* is choosing a cheap and feasible discrete-path for a non-holonomic vehicle.

Whether, or whether not the radial-constraint modification in A* finds the optimal solution can not be guaranteed. However, based on the facts conducted from figure 4.1 and the associated tables are that the green path finds cheap paths, and all of them are feasible. As the green paths are located in the neighborhood of the blue paths, which are optimal in the dilated maps, we can reason that the green paths lie in the neighborhood of a local minimum, hopefully, the global minimum.

Conclusively, dilating the costmap to keep a clearance to high-cost areas and radial-constricting the A* algorithm generates a discrete path that is cheap and guaranteed feasible. The discrete path will lie in the neighborhood of a minimum, and are a solid fundamental for further optimization.

4.1.1 Run-time

Through the experiment which was done in this section, it can be seen from the run-time in the associated tables, that the fastest algorithm was the regular A*, then the A* in the dilated map, and the slowest was the radial-constrained A*. The time-difference between search in a regular costmap and dilated costmap emerge due to a more monotone costmap from the dilation. In the dilated map, there are generated more promising and cheap paths to explore before reaching the goal, hence more iterations and explored nodes than A* in regular costmap. The radial-constrained A* is the slowest, because of an added constriction that is calculated in the algorithm. In the table below, run-time and memory-consumption from the algorithm executed in map 2(4.1b) can be seen.

	Run-time	Nodes expanded	Iterations
Regular A*	1.32s	31958	47417
Regular A* in dilated map	1.65s	45854	54285
Radial-Constrained A*	1.81s	40998	46830

Table 4.5: Table of run-time and memory-consumption in the algorithms. *Iterations* are the number of operations in the *Closed*-set, which are nodes added and re-evaluations of nodes.

The different searches can be visualized as a search map, where the expanded nodes stores the associated cost to get to this cell from the start location, where a simplified version is shown in figure 2.7. The different searches from figure 4.1b are visualized in figure 4.3.

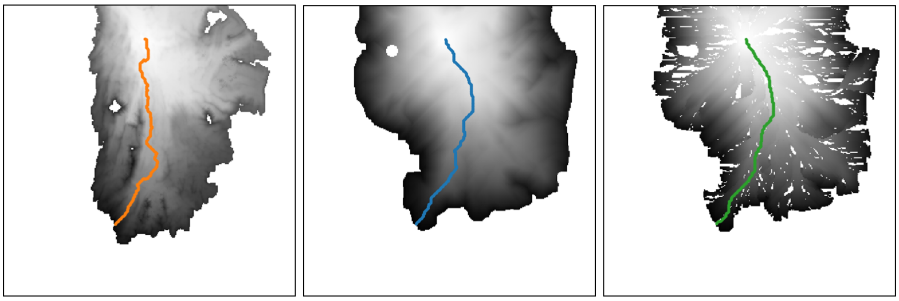

(a) Regular A* (b) A* in dilated map. (c) Radial-constrained A*

Figure 4.3: *Map 2.* Shows the search area for the different path found for map 2 (4.1b), i.e. the expanded nodes with associated cost from the *Closed*-set.

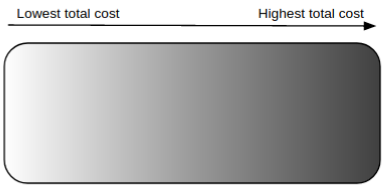


Figure 4.4: Illustrates the color-coding for figure 4.3

It can be seen that the regular A* search from subfigure 4.3a searches a smaller area than A* in dilated map in 4.3b. Subfigure 4.3c covers an equal area to the A* search in the dilated map, but with pruned nodes which are not expanded.

4.1.2 Bug with radial-constraint

With a good eye, one could notice in figure 4.1d that there are some unnecessary and sharp turns in the green path from the zoom-in showed in 4.5. These occur sometimes when the radial-constriction is used to prune nodes that lead to infeasible paths.

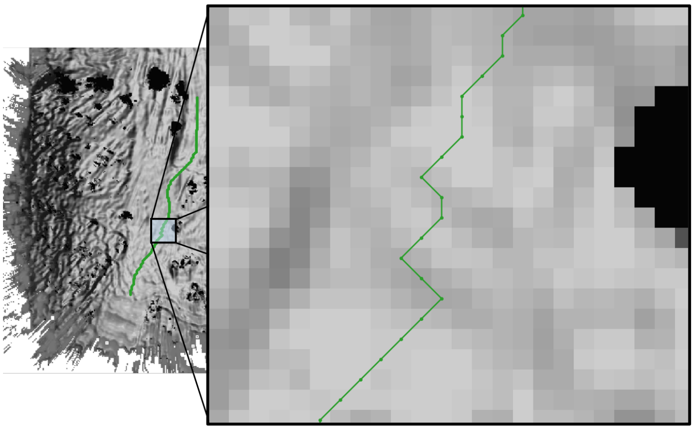


Figure 4.5: Shows an enhanced section of the radial-constrained path from 4.1d

This happens as a consequence when *creating* larger circles in a discrete grid with few points as shown in figure 3.10. The radial-constriction is performing its purpose, by creating circles that are able to generate flexible paths with a greater turning radius than 15 units. In figure 4.6 it is illustrated three-point-circles which are created by every four node. All of the circles which are possible to create do not violate the turning-radius at 15 units, which are the main focus of the radial-constraint modification of A*.

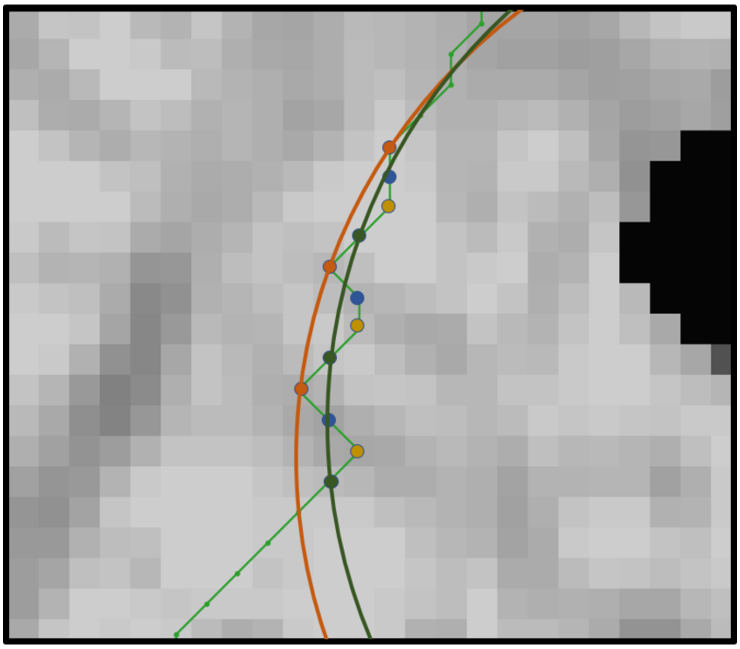


Figure 4.6: Three-point-circles created with every four nodes to illustrate large-scale turning radius

As long as the turning radius is legal all over the path at every 4. node, a simple smoothing algorithm as explained in subsection 3.3.1 will fix the unnecessary turns stated in figure 4.5. In the optimization, the sharp turns will be penalized and smoothed, and fade in few iterations.

4.2 Cost optimization

The purpose of optimization a path is to find the optimal path from an initial guess. The optimal path has to be cost-efficient, smooth, and can not contain any sharp turns. Sometimes when the sharp turns are smoothed, the smoothed path may cut corners resulting in a path that is more expensive than the original path. To prevent the path to be smoothed over too expensive cells, a cost-term is used in the objective function for the optimization.

To demonstrate the effect of cost-optimizing, a simple segment of a path was chosen from figure 4.1c. Figure 4.7 shows A* choosing a cheap path in a regular costmap.

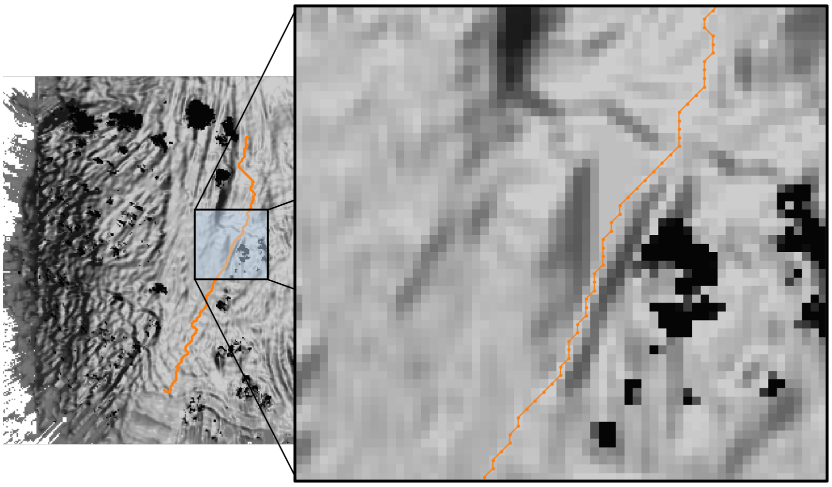


Figure 4.7: Zoom-in of a narrow passage from figure 4.1c

Figure 4.7 shows an example of a seemingly cheap path, but the chosen route is in a narrow passage between high-cost cells. It is not enough to only smooth the discrete orange path in figure 4.7 as the vehicle size needs to be accounted for. To demonstrate the functionality of the cost-term, an optimization was applied on this path in an increased number of iterations, where the iterations represent the length of optimization and how many times the nodes have been perturbed.

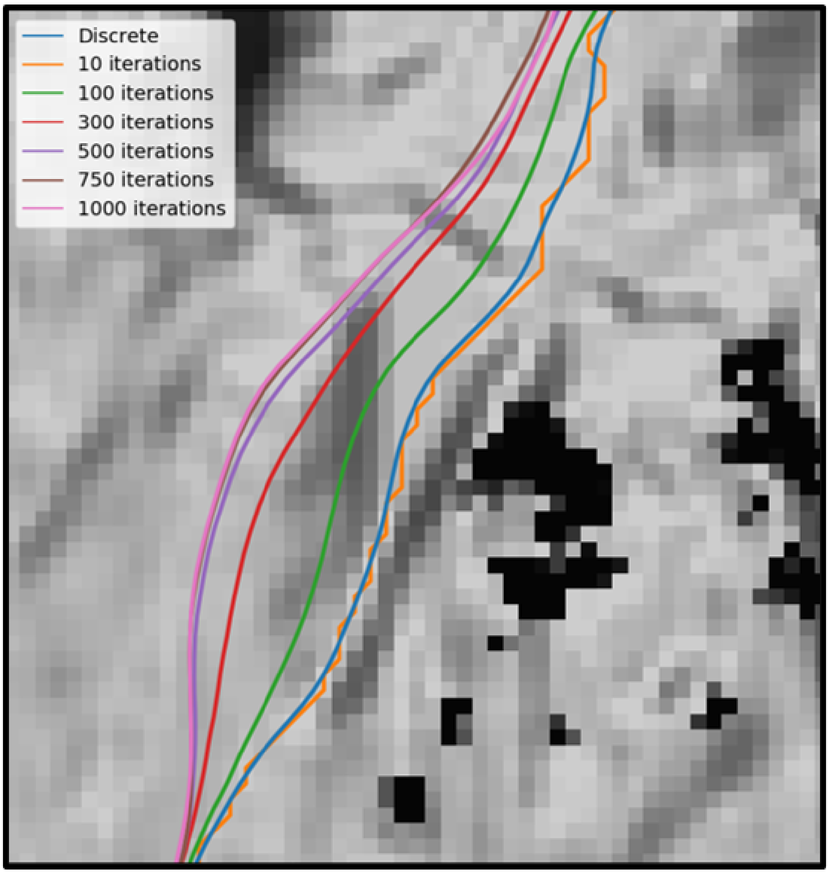


Figure 4.8: Shows the evolution of a cost-optimized path

Figure 4.8 shows different paths that are optimized in an increased amount of times. It can be seen that the blue path is smoothed, but it does not only cut the corners, but it also smooths slightly towards a cheaper area. The same happens for the following

	Total Cost	Length	Cost per length
Discrete	1329.6	71.8	18.5
10 iterations	1238.6	66.5	18.6
100 iterations	1086.7	65.4	16.6
300 iterations	933.4	65.7	14.2
500 iterations	896.6	67.0	13.4
750 iterations	932.4	67.6	13.8
1000 iterations	945.6	69.5	13.6

Table 4.6: Table of the total cost, length and cost per length.

paths with 100 and 300 iterations, they search towards an area that is cheaper for the vehicle to traverse. The search continues until the path converges into a local minimum, but, searching only in the neighborhood of the initial path. The paths with 500, 750 and 1000 iterations are almost identical, which means that the path has most likely converged.

From table 4.6, it can be interpreted that the path with 500, 750, and 1000 iterations have converged, as the cost per length has stabilized. As the path gets further optimized, it finds a cheaper path around the obstacle, causing the path to be stretched and results in a longer path.

The total path-cost can be monitored during the optimization. The total cost of each of the paths from figure 4.8 is presented in figure 4.9.

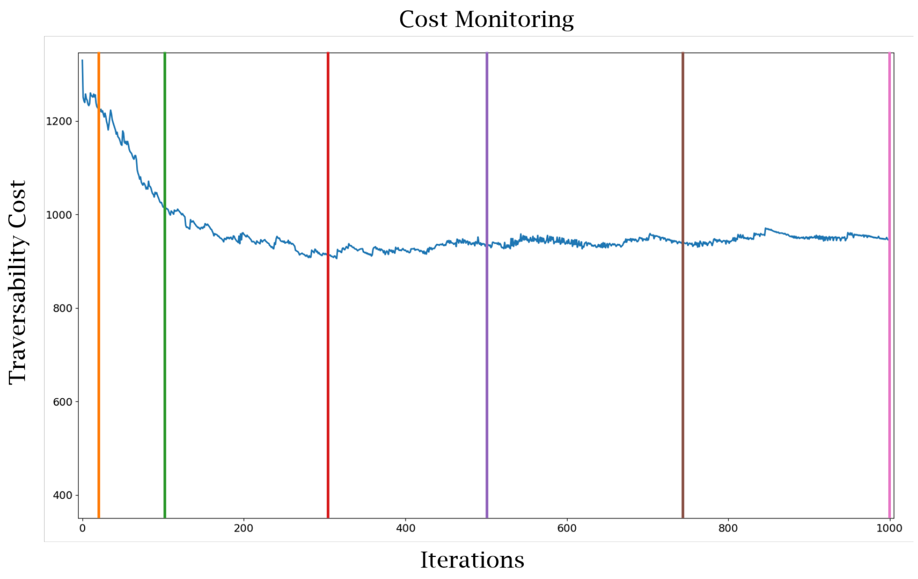


Figure 4.9: Shows monitoring of total cost in the optimization of the section presented in figure 4.8. The color-coding represent the corresponding path from figure 4.8

From figure 4.9 it can be seen that the total cost of the discrete path is most expensive (at 0 iterations), but the cost is decreasing as the iterations increases. The total cost decreases steadily for the first 300 iterations, and are fluctuating around a cost of 900 for the next 700 iterations while the path stretches. Concluding from this observation is that the cost-term are forcing the individual nodes to create a path around the obstacle, causing a longer path and traversal of more cells. The optimized nodes are located in cheaper coordinates, but since the path stretches to get around the obstacle, the total cost does not decrease.

The technique used by the cost-term to find the direction to a cheaper area is firstly to average the map. The kernel's dimension will represent the spatial occupancy of the vehicle in the grid. By using this type of map, the driveability-cost in each cell will simulate how cheap one cell's neighborhood is relative to the adjacent cells' areas.

To find the direction towards the cheapest area from a given cell, a gradient field was produced from the averaged map. This field consists of vectors pointing towards the area with the largest accent, as explained in 2.4.3. The gradient field in figure 4.10 was used to find the convergence path.

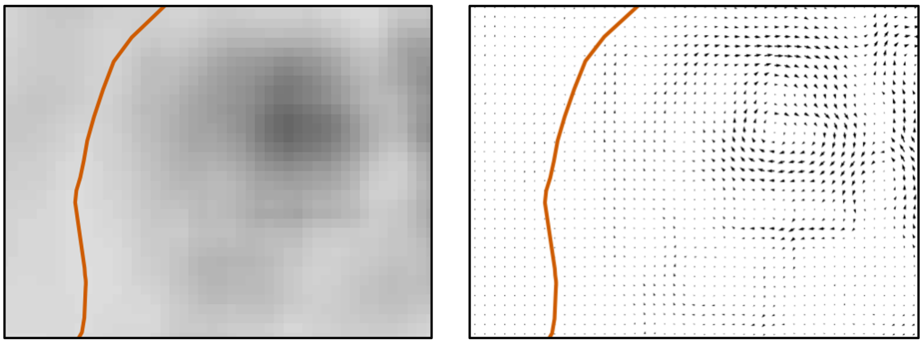


Figure 4.10: Shows an averaged section of an expensive area to the left, and the corresponding gradient field to the right. The section is withdrawn from figure 4.8.

It can be seen from the averaged array to the left in figure 4.10 that the path has converged in a bright area. This area proved to be the cheapest passage for the size of FFI's vehicle.

The first thought of cost-optimizing something is to greedily update towards the cheapest location, however, this can not be directly done when optimizing paths. A path needs do maintain certain characteristics, as keeping the path together and smooth, and as well feasible for a non-holonomic vehicle. From figure 4.9 it could be seen that the path approximately reached the cost-minimum at 300 iterations, but from 4.8 the path has still not converged to its minimum at 300 iterations. The path kept moving because the cost-term is based on the cost of the specified node and not the total cost. The path-nodes are constantly perturbed towards a cheaper area, causing the path to stretch around high-cost areas. The stretching causing the nodes towards cheaper cost-cells to form longer paths, which again results in a higher total cost.

4.3 Optimizing the radial-constrained A*

The initial guess for the optimization, the solution path from the radial-constrained A*, is a solid fundamental for the optimization. The discrete path will have clearance to high-cost cells while satisfying the vehicle's turning radius limitation. If the search of the radial-constrained search is good, a feasible continuous solution will be close to the discrete solution.

The simplest type of optimization would just to smooth the path to be continuous. In most cases, a simple smoothing would be sufficient, but to get our algorithm to work in *all* cases, the cost- and curvature-term need to be included. Fortunately, the impact of cost, curvature, and smoothing-gradients can be adjusted with weights w between the optimizations. By adjusting the weights, one can decide to prefer a cheap path over a smooth path or the other way around. The weights are used in the gradient descent algorithm 6 explained in chapter 3, where each weight w_s (smoothing), w_o(cost) and w_c(curvature) decide the corresponding gradients influence on the path. That is, the smoothing weight w_s decide the smoothing gradient's influence on the path. The weights are adjusted relative to each other, prioritizing its corresponding term separately.

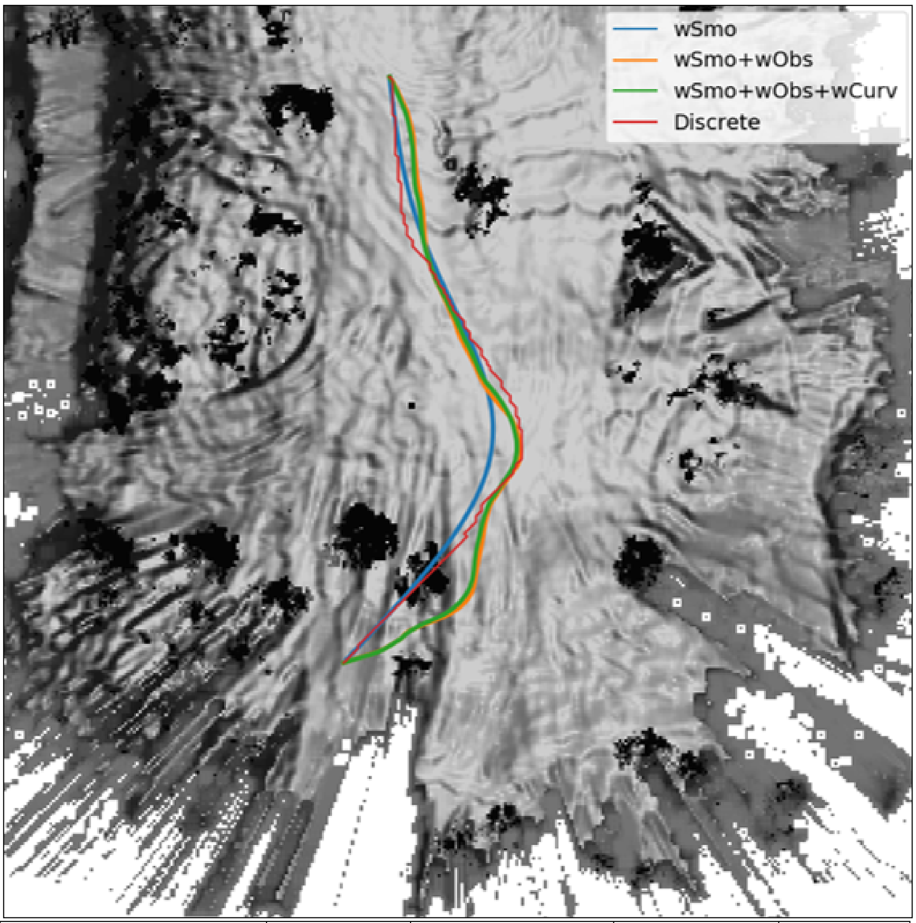
The step-size α is also used in the gradient-descent optimization 6, deciding the proportion of each perturbation. From 2.5.4, α is chosen either by *backtracking* or *exact line search*. Since this optimization problem is non-linear with different types of gradients, the step-size will have a fixed step size [1] for this path planner. A large α will need fewer iterations to reach the minimum, but may never converge because of too-large steps. A small α will may never reach the minimum or require too many iterations to reach the minimum. To be able to converge to a satisfying solution, it is eminent to find a good tradeoff between the different weights and α.

To see the effect of the different terms in the optimization, the radial-constrained paths from figure 4.1 were optimized. The paths were optimized with 500 iterations each with the gradient-descent algorithm proposed in chapter 3. Three different types of optimization were performed with the parameters stated in table 4.7.

	w_s	w_o	w_c	α	Iterations
wSmo	0.9	0	0	0.1	500
wSmo+wObs	0.9	0.5	0	0.1	500
wSmo+wObs+wCurv	0.9	0.5	0.8	0.1	500

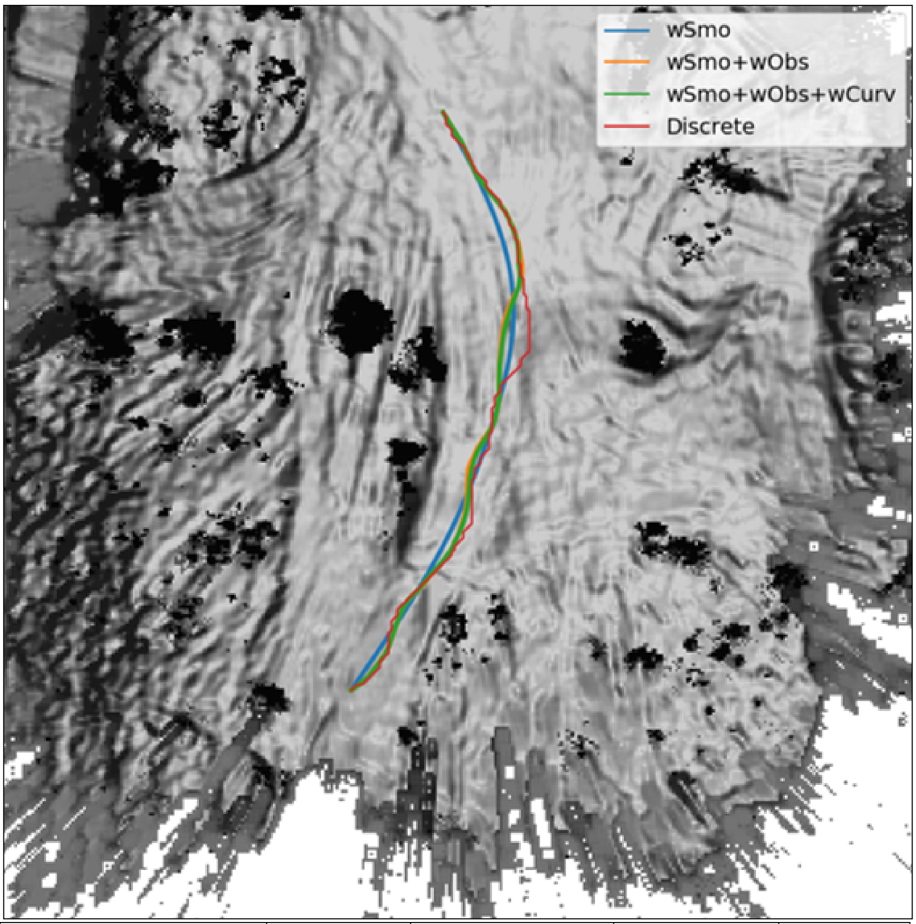
Table 4.7: Parameters for optimization on four different maps.

The parameters are selected in this manner to show the effect in the different terms. By setting a weight to 0, means that the corresponding terms do not contribute to this optimization. The exact choices of why the different parameters are 0.5 for w_o, 0.9 for w_s, 0.8 for w_c and 0.1 for α is a guided *trial-and-error* technique, where you start at some parameters and see the optimizations response. According to the optimizations output-path, the weights can be adjusted depending on the priorities, to have a more/less cheap or smoother path. The optimization of the radial-constrained discrete paths are shown in figure 4.11 to 4.14, where three different optimizations are shown in four maps.



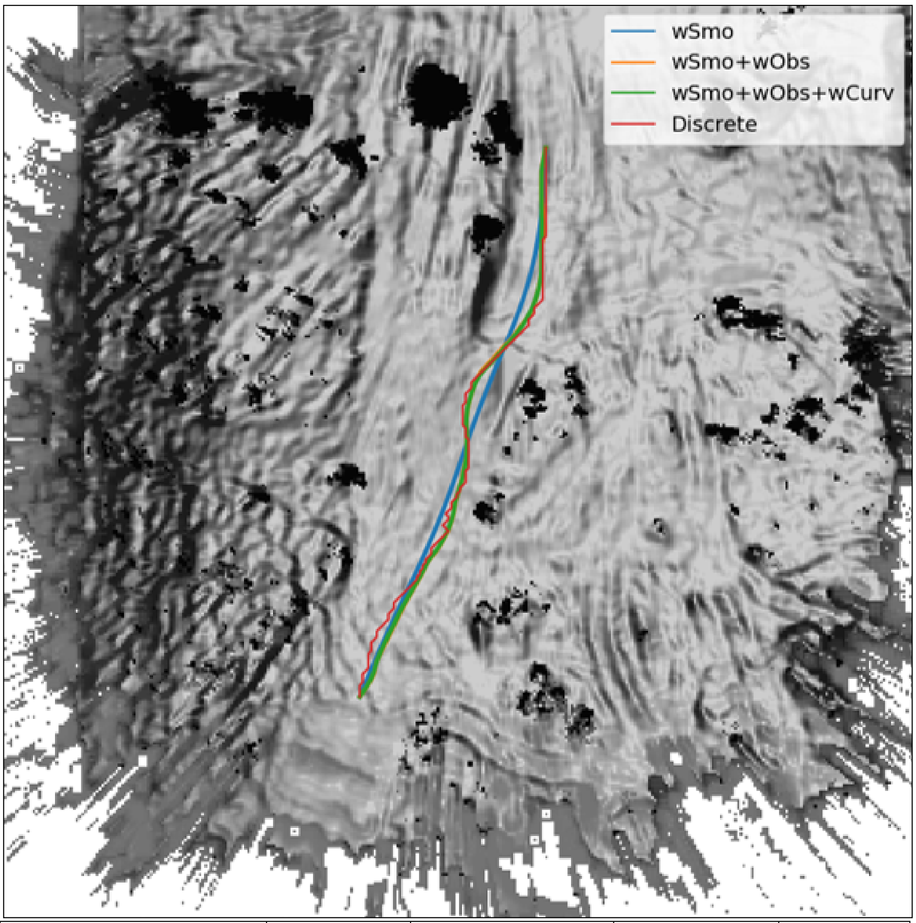
	Total Cost	Smallest Radius	Path Length	Cost/Len
Discrete	4287.0	15.8	249.6	17.2
wSmo	4191.1	47.7	227.5	18.4
wSmo+wObs	3039.6	9.9	245.6	12.4
wSmo+wObs+wCurv	3240.7	16.8	243.1	13.3

Figure 4.11: *Map 1.* The **blue** path is only smoothed and deviates from the discrete path by cutting corners. The **orange** path is also smoothed but deviates from the blue path where the cost gets expensive. The **green** path is smoothed and cost-optimized, but consider the curvature to not exceed the minimum turning radius of the vehicle.



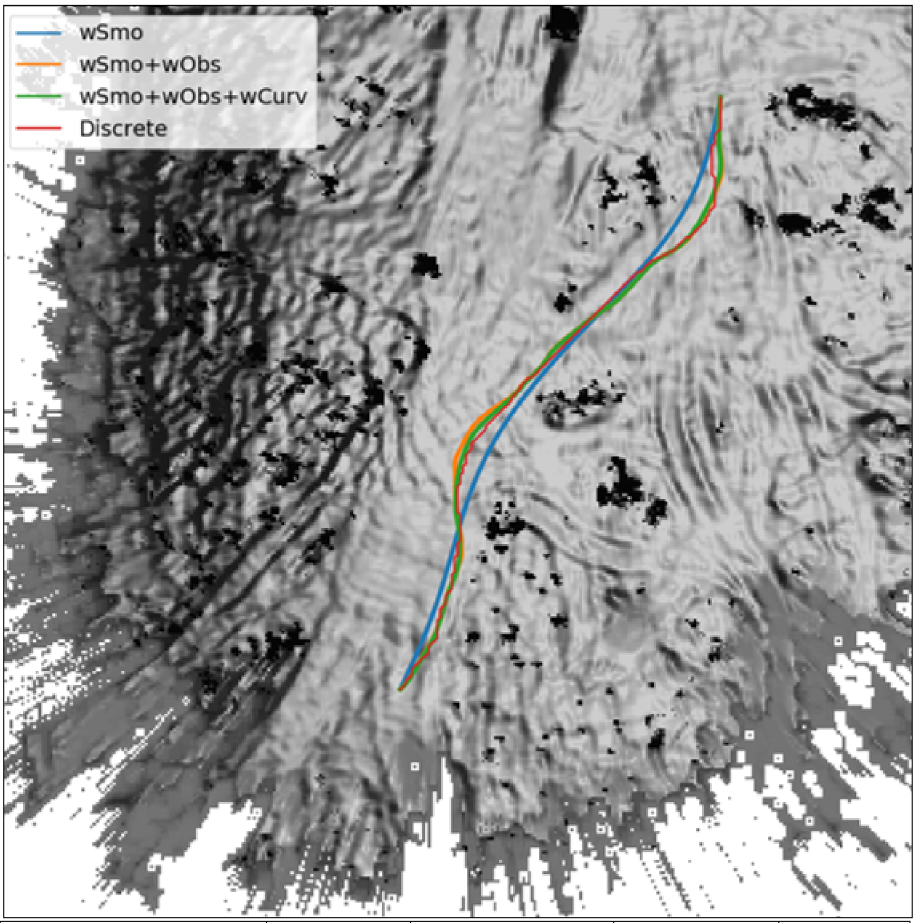
	Total Cost	Smallest Radius	Path Length	Cost/Len
Discrete	2699.2	15.8	240.3	11.2
wSmo	2548.5	80.3	219.7	11.6
wSmo+wObs	2372.6	11.6	225.5	10.5
wSmo+wObs+wCurv	2407.8	14.8	224.7	10.7

Figure 4.12: *Map 2.* Three types of optimization of a discrete path. The three continuous paths are in in the neighborhood of the discrete path



	Total Cost	Smallest Radius	Path Length	Cost/Len
Discrete	2486.2	15.8	223.1	11.1
wSmo	2478.8	129.9	204.1	12.1
wSmo+wObs	2392.1	9.7	208.9	11.4
wSmo+wObs+wCurv	2428.2	20.0	208.6	11.6

Figure 4.13: *Map 3.* It can be seen that the initial guess is good, avoiding high-cost areas like the **orange** and **green** path does. The **blue** path is nearly converging to a straight line, not worrying about the higher-cost areas.



	Total Cost	Smallest Radius	Path Length	Cost/Len
Discrete	3263.0	15.8	259.1	12.6
wSmo	3345.3	84.8	239.5	14.0
wSmo+wObs	3148.9	10.6	252.0	12.5
wSmo+wObs+wCurv	3145.2	19.3	249.8	12.6

Figure 4.14: *Map 4.* The **green** and **orange** following the discrete path tightly. One exception is in one of the most distinct curve, where the **orange** and **green** are taking a longer turn to a cheaper area than the discrete path. The **orange** path is violating the minimum turning radius doing this, but the **green** path has a curvature term that penalizes too-small turning radius.

The four figures 4.11-4.14 shows what role the different terms in the objective function F has. For simplicity, we restate the objective function from eq. 3.5 and the gradient used in algorithm 6:

$$F = F_{smo} + F_{cost} + F_{cur} \tag{4.1}$$

$$\frac{\partial}{\partial x_i} F = \frac{\alpha}{w_s + w_o + w_c} \cdot \left(w_s \cdot \frac{\partial}{\partial x_i} F_{smo} + w_o \cdot \frac{\partial}{\partial x_i} F_{cost} + w_c \cdot \frac{\partial}{\partial \kappa_i} F_{cur} \right) \tag{4.2}$$

The blue paths are solely optimized by the F_{smo}-term. This is the fastest option, optimizing 500 iterations in around 2.5 seconds. These paths focus only to straighten up the sharp corners, while slowly converging to one-straight-line. From the four optimization examples, it can be seen that when the optimization is "blindly" smoothing the path, the cost per length is slightly increased than the discrete path. However, since the initial guesses were good and already located in a safe neighborhood, the smoothing did not increase the cost significantly.

The orange paths are optimized by the F_{smo}- and F_{cost}-term, taking about 3 seconds for 500 iterations. The cost-term is leading the paths towards cheaper areas, while the smoothing-term keeps the path smooth. The balance between the weights w_s and w_o decide rather prioritizing having a very smooth path, finding the cheapest route, or something in between. From the tables in figure 4.11-4.14, it can be seen that the orange paths are the cheapest and lowest cost per length, but at the expense of having a too-small turning radius. This is caused by the gradient of the cost-term, that it greedily guiding the nodes towards a cheaper area without considering the curvature. A greater turning-radius can be achieved by a larger w_s or smaller w_o, but this would again cause a higher total cost for the path.

The green paths are optimized by the terms F_{smo}, F_{cost} and F_{curv}, taking approximately 10 seconds for 500 iterations. The behavior of this optimization is equal to the optimization of the green paths. The exception is that the F_{curv}-term is lower bounding the curvature by a minimum turning radius, which is 15 units. The green path is not always cheaper than the orange path, but the path will hopefully have a greater turning radius than the minimum-allowed turning radius. By using smoothing, combined with cost- and curvature-optimization will also, with almost guarantee create a safer path than just a simple smoothing on a discrete path.

4.3.1 Mass simulation

To test the robustness and consistency of the path planner, it was simulated 819 executions of the radial-constrained A* with the three types of optimizations as in figure 4.14. The simulations were conducted in map number four because this covered the biggest area of traversable cells and would hopefully serve as the best test-arena.

819 start- and goal-locations was randomly generated within the detected area in the costmap. To prevent too short paths and bad data, the minimum distance between the start-and goal-location was set to a Euclidean distance of 150 cell lengths. The radial-constrained A* was executed 819 times, and should be optimized respectively with three sets of parameters, which are the same as stated in table 4.7.

The radial-constrained A* found 751 paths out of 819 start- and goal-locations. This means that the path planner failed to find the goal 68 out of 819 executions. The 751 paths were optimized with the parameters stated from table 4.7 and the results are stated below:

	avg. Cost	avg. Length	avg. C/L	avg. Rad	Rad>15
Disc	3458.4	218.2	15.8	15.8	751/751
S	3665.9	199.5	18.4	112.5	751/751
S+O	3589.6	211.0	17.0	8.7	36/751
S+O+C	3543.5	208.7	17.0	13.2	266/751

Table 4.8: Table that shows the results from the mass simulation of three types of optimization on one path. The average is from all successful simulation and the number of feasible paths is stated.

Table 4.8 shows the results from the 751 optimizations. The cost from the discrete cost are the lowest, then the parameters from $wSmo + wObs$ and $wSmo + wObs + wCur$ are about the same value, and the optimization with $wSmo$ have the highest average. The paths for $wSmo + wObs$ and $wSmo + wObs + wCur$ are only smooth and feasible in 36/751 and 266/751 times, but the smallest radius for $wSmo + wObs + wCur$ have an average of 13.2.

To get some clarity in how the different execution of the optimization was, the *cost per length* for the optimized paths was compared to the *cost per length* of the same discrete path. The plot can be seen in figure 4.15.

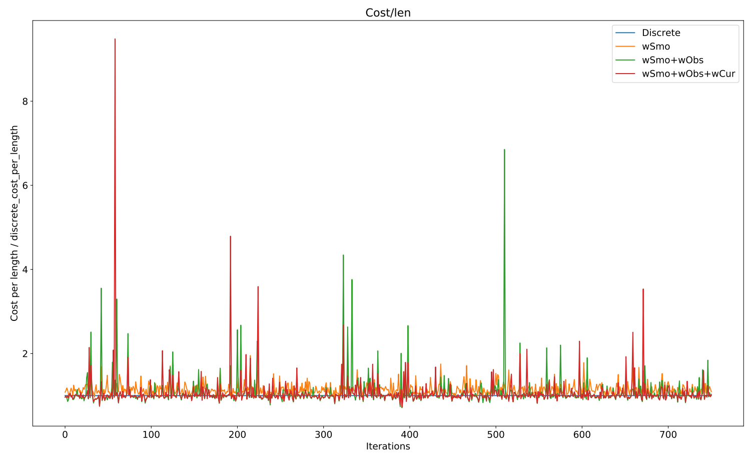


Figure 4.15: The cost per length of the optimized paths relative to the cost per length of the discrete path.

From the graph in figure 4.15 it can be seen that some optimizations have made the cost far worse than the cost of a discrete path. These paths are most likely located at the edge of the detected terrain. As this type of terrain has not any clear gradients to cost-optimize the path, the gradient-term feeds directions on wrong data. This reason may be the cause of the clear spikes in 4.15.

Comparing table 4.6 with the tables from figure 4.11 to 4.14, it can be seen that the four tables represent the typical performance of the optimizer. The radius and path lengths are approximately the same relative to the discrete instances. The relation between the cost are different, where the four figures had the lowest cost in $wSmo + wObs$ and $wSmo + wObs + wCur$. The average cost-per-length is significantly higher than the cost-per-length in all four figures 4.11-4.14. The start- and stop-locations in the four figures are carefully considered in a fully detected area, resulting in four sufficient paths with relatively low cost. This is not the case of the mass simulation, where random points were selected as the start- and goal-locations.

Conclusively, to get a mass simulation with good paths and data, the start- and goal-locations have to be selected more cleverly or hand-picked. Also, the parameters from table 4.7 gave satisfying solutions through open terrain, but maybe another selection of parameters would give better results.

4.3.2 Selection of weights

Balancing the weights between the three types of gradients can be a difficult task. If the objective of the optimization is to get the cheapest path as possible, the cost-weight w_o can not be pumped up without consequences. By setting w_o much higher than w_s and w_c, gives too much freedom to the cost-gradient term, causing the nodes to gather in the low-cost areas. The nodes will then cluster in the cheap areas, having difficulties to maintain a feasible curvature. This problem is illustrated in figure 4.16.

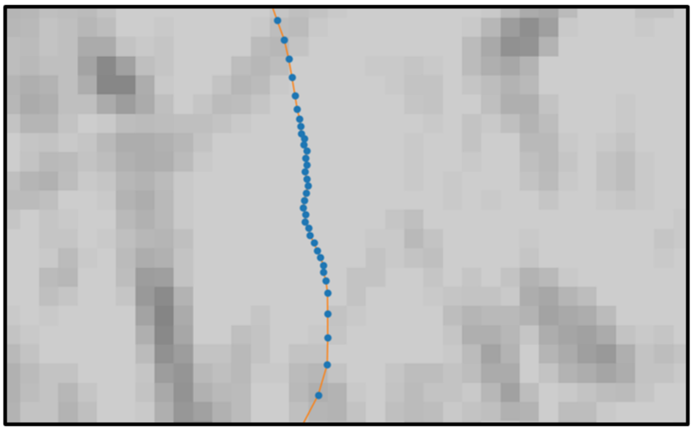


Figure 4.16: Illustrates nodes clustering in bright/low-cost area. w_s and w_c is set to 0.2 while w_o=0.9. α = 0.1 and the duration is 500 iterations.

The smoothing term contradicts the effect illustrated in figure 4.16. In addition to keeping the path smooth at all times, the smoothing term tries to maintain an equal distance between the nodes. Since the w_o had a significantly higher number than w_s, the smoothing-term was under-prioritized in the optimization in figure 4.16, causing the nodes to cluster in a cheap area.

Setting the curvature-weight higher than the two other terms will not cause abrupt changes other than a normal smoothing. The curvature- and smoothing term have similar roles, both can be used to smooth a discrete path, but the curvature-term stops the smoothing when the path has become feasible. The curvature term is needed to maintain the cost term's changes, but have difficulties to contradict the effect when the nodes gather in the cheap areas (as in figure 4.16), and have also problems keeping the curvature feasible when the nodes are clustering.

When selecting weights, it is important to always have a high weight priority on the smoothing-term to keep the path feasible. If the path is supposed to smooth towards cheaper areas, the weight for the cost-term has to increase, as well as the weight for the curvature-term to maintain the path feasible.

4.3.3 Stopping criterion

Recognizing the minimum where the path is converging is not an easy task, especially when there are multiply terms in the objective function with each their purpose. By visualization, it is easy to see when the path has converged because the paths are not moving any more during the optimization, e.g. the paths with most iterations from figure 4.8.

Ideally, the optimization should stop when the path has been perturbed into a local minimum. Since the optimization problem in this thesis is highly non-linear, it is not easy to numerically find a local minimum. Instead, another stopping criterion can be used to stop the optimization. One example of criteria can be a definite number of iterations, which is used in subsection 4.3. This stopping criterion may be more memory-consuming than necessary, that is if the path converges before the maximum number of iterations is reached. When the path got optimized with a definite number of iterations in figure 4.11 to 4.14, each path optimization took about 10 seconds with 500 iterations, which is time-consuming if the motion planner is supposed to be constantly updated on the best route.

One should decide what to prioritize when optimizing the path. If the main goal for the optimization is only to make the path driveable, a stopping criterion can be the path's smallest turning radius. By using the turning radius as a stopping criterion, one can guarantee that the smoothed path is driveable (as in [5]), but one can not guarantee if there exists a cheaper path in the neighborhood.

If the path's traversability cost was to be used as a stopping criterion, the cost can be monitored during the optimization. Figure 4.17 shows an example of cost monitoring of the path per iteration in the gradient-descent optimization.

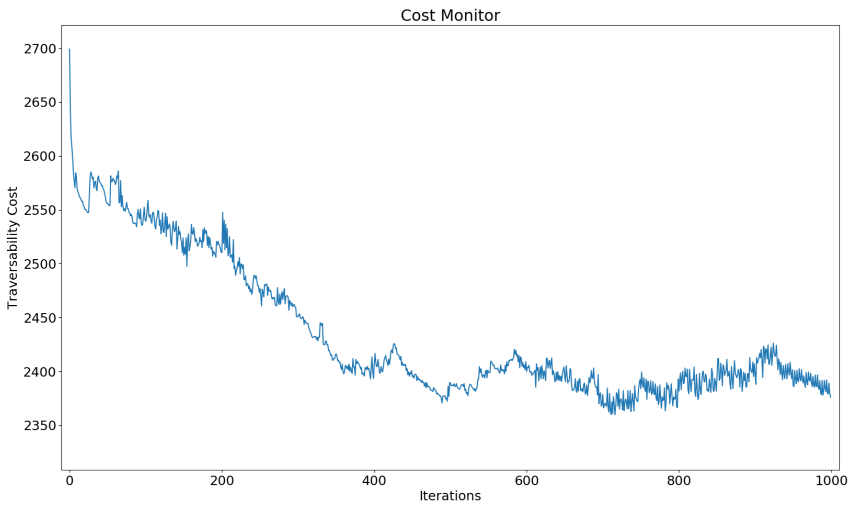


Figure 4.17: Shows a path's cost for each iteration during optimization. Starting at 2700 and converging in about 2400. Max iterations was 1000

From the graph in figure 4.17, it can be seen that the cost is decreasing in an approximately constant manner for the first 500 iterations, and from this point, the cost is pulsating around a 2400-cost for the next 500 iterations. Recognizing the point where the cost-decreasing eases could be used as a cost based stopping criteria.

If the path's feasibility was to be used as a stopping criterion in the optimization from figure 4.17, the optimization would terminate after 18 iterations resulting in a cost of 2554. The two optimized paths are visualized in figure 4.18.

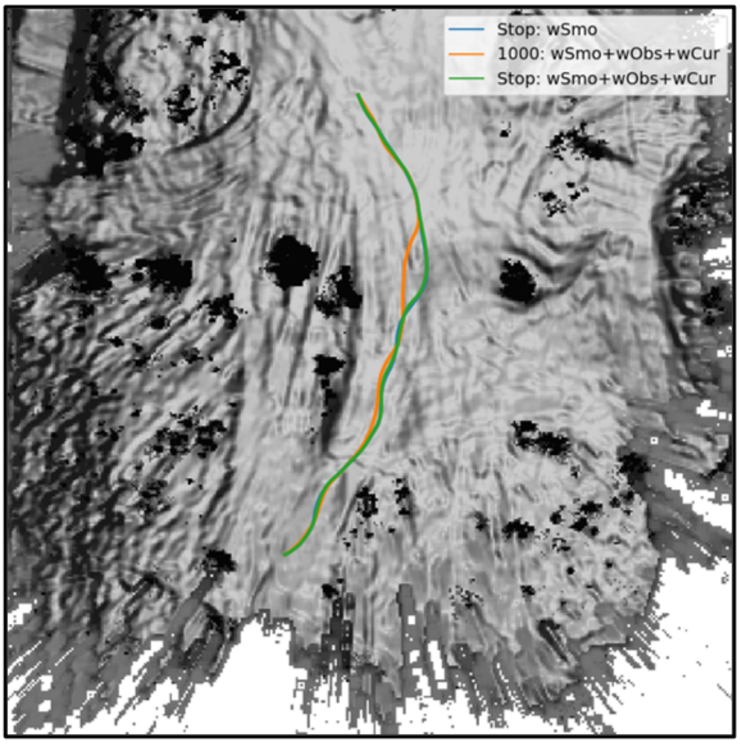


Figure 4.18: Shows three optimized paths.Two paths which are optimized until driveable, and one path which are optimized with a duration of 1000 iterations.

	Total Cost	Stopping criterion	Run-time	Iter.	Length
wSmo	2581.7	Turning-radius	0.046 s	8	227.6
wSmo + wObs + wCurv	2549.0	Turning-radius	0.539 s	21	227.7
wSmo + wObs + wCurv	2392.0	Iterations	21.05 s	1000	224.7

Table 4.9: Table for figure 4.18. Shows run-time for optimization.

It can be seen that all three paths are fairly equal, with the exception that the orange path diverges from some darker areas. All paths are driveable, with a turning radius of over 15 units. The orange path is a cheaper and safer route with a 150 lower cost but at the expense of using 20 seconds more on optimization.

Two of the paths are almost overlapping, the green and blue paths with turning-radius as a stopping criterion. The cost per length for the blue is 11.34 (cost/length) and the green is 11.19 (cost/length), but the blue path uses only 10% of the running-time than the green path. Both paths are feasible, but the green path is only optimized with the smoothing gradient, and the blue path is optimized with the smoothing-, cost- and curvature-gradient.

Conclusively, choosing a stopping criterion for the optimization lies in the user's priorities. If the algorithm is supposed to be fast and safe, one could use feasibility as a stopping criterion. If the user wants run-time consistency, a maximum iteration can be used as a stopping criterion. Lastly, one could use the cost as a stopping criterion to recognize the local minimum. It is also possible to combine multiple criteria in A*'s *while* loop, that is, both radius-check and cost-requirement. There is not implemented a technique to stop the optimization based on the cost in this thesis, but using an *Early stopping*- technique might work [36].

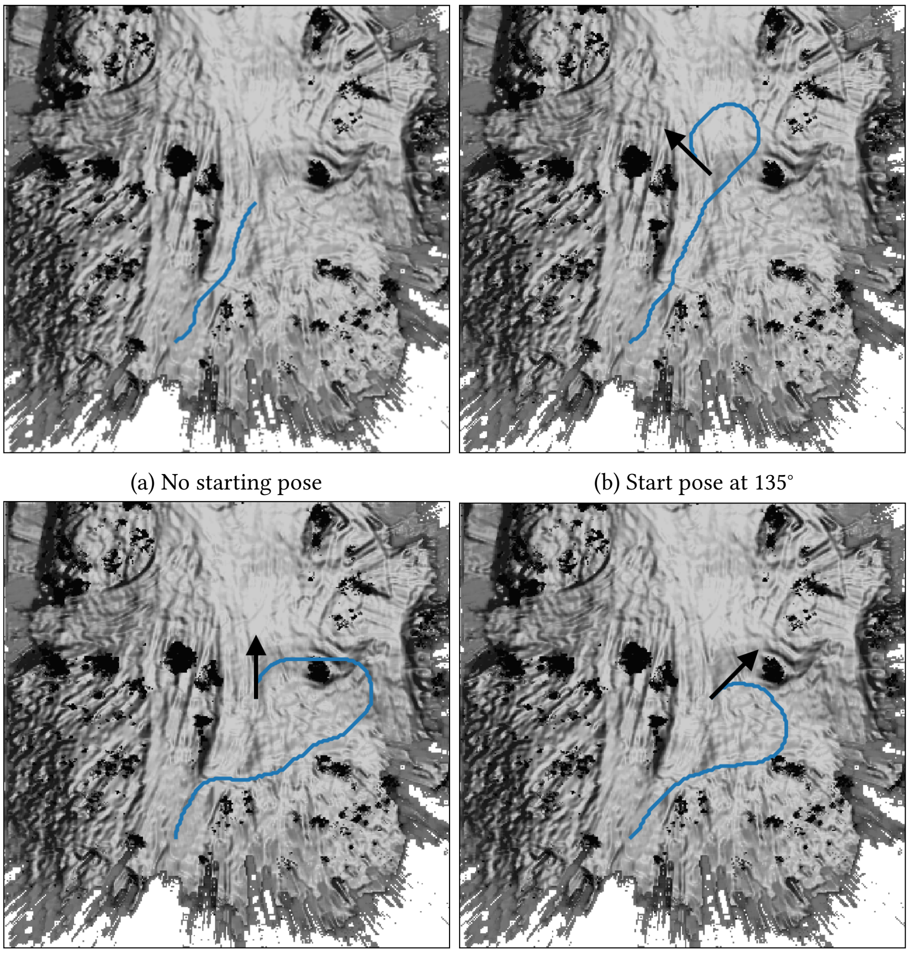
This section have investigated what role the different terms and corresponding constants have in the optimization. If only a smoothing term is used in the optimization, the optimization should include a stopping criterion to avoid deviating too much from the initial cost-efficient discrete path. A stopping criterion should also be used when an objective function consist of a smoothing- and cost-term, to make sure that the solution path will stay feasible. The reason for this is that none of the terms, smoothing or cost, are penalized if the path exceed a turning-radius demand. However, if a curvature-term is included, the gradient of this term will perturb the nodes who violates the minimum turning-radius. A downside of using the curvature term is that the calculations is heavy, resulting in a more slow optimization than with pure smoothing.

4.4 Three DOF start state with radial-constrained A*

Usually, in a path planning problem, the intention is to find the best path from a start- to a stop-location, where the start- and stop-locations have two degrees of freedom (DOF), x- and y-coordinates. However, when a path is generated to fit a route for a non-holonomic vehicle, there are certain limitations for the movements of the vehicle, as the wheels can not slide. To mimic non-holonomic movements through a path, the start state has to include a third DOF, which is an angle describing the vehicles pose relative to the world-frame.

The beginning of the path will be different when using three DOF start state rather than two DOF. Instead of executing the shortest path toward the goal, the search needs to account for the pose of the vehicle at the start position, forcing the path to take a detour from the start to enforce the non-holonomic constraints.

To demonstrate the importance of accounting the pose of the vehicle in the planning process, some paths were generated with the same start- and stop-location. The start position was located in a narrow passage, triggering the algorithm to take a detour if the goal location was located behind the starting pose. Figure 4.19 shows four paths from the same locations, where one path finds the shortest path to the goal, while the three others need to account for a starting pose.


(a) No starting pose

(b) Start pose at 135°

(c) Start pose at 90°

(d) Start pose at 45°

Figure 4.19: Shows the different paths generated at approximately the same start location with different start angles relative to the world frame.

It can be seen that four different paths have been generated from the same start location with different start poses. Subfigure 4.19a chooses the shortest path without regarding any non-holonomic constraints, while the three other paths have distinctively different and longer routes due to the starting pose.

The start angles in subfigure 4.19b-4.19d have been hard-coded, manually shaped the first few nodes to form a start angle. This was to simulate how the radial-constrained A* would perform with a starting pose at the start node. Evaluating from figures, the performance is in general pretty good, but it can be seen that there are also some issues with the radial-constrained A*. From subfigure 4.19c it can be seen that the algorithm chooses a path through a high-cost area rather than perform a u-turn in the low-cost area in front of the start position.

Figure 4.20 visualizes the search for the path in subfigure 4.19c. This is done by creating a map of the *Closed*-set from the search, showing how expensive it is to reach each location.

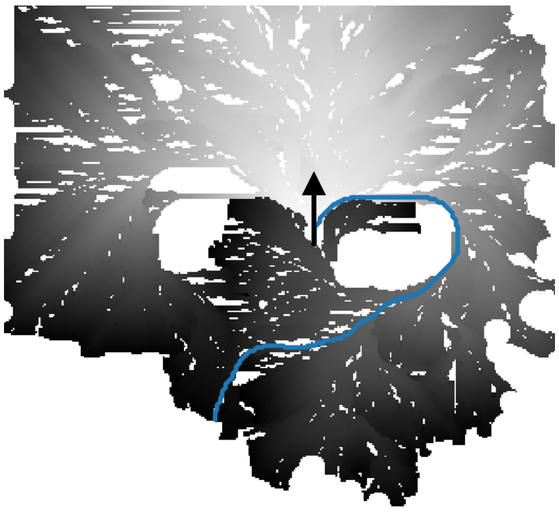


Figure 4.20: Shows a heuristic map of the radial constrained search from subfigure 4.19c. The path reaches the brightest spots cheapest, and require the highest total cost to reach the black areas.

Figure 4.20 shows the path chosen through the search area for the path in subfigure 4.19c. It can be seen that the search never considers performing a u-turn, due to the abrupt transition from light to dark beside the start position. Ideally, the path should go forward, creating a circle and turn back driving right past the starting position. Since the radial-constrained A* search has already searched the area in front of the start position, the search never considers this opportunity, because there exist "a cheaper path" from the start node to this area. This flaw forces the A* to continue its search, resulting in an undesirable path. The same flaw forced the discrete path to choose a high-cost path in figure 4.11.

To overcome the issue that the radial-constrained A* having problems creating u-turns, the A* search can include an *accessed-from* angle when evaluating a node. By implementing this modification, nodes can be reevaluated from different angles and creating different paths to the same node. In other words, including a third degree of freedom which represent the angle between the world frame and the body frame of the UGV.

4.5 Further discussion

The radial-constraint A* in the dilated map is an algorithm that considers the width of the vehicle in the terrain, as well as respecting the non-holonomic constraints. From section 4.1, all three types of algorithms choose their cheapest alternative from the A* algorithm. Based on the fact that the radial-constraint A* algorithm has to generate a different path than the two other discrete-planners, shows it was necessary to constrict the turning-radius. The deviation between the paths in figure 4.1 emphasizes that the customization made in the radial-constrained A* in the dilated map was a reasonable modification to create a feasible solution for non-holonomic vehicles.

On the grounds of the problem elaborated for figure 4.20, there is room for improvements of the discrete radial-constraint modification. By not including the third DOF, the *accessed-from* angle, limits the opportunity to create the best- and cheapest route when the goal is not directly reachable from the start state. This would most likely contribute to get more successful executions of the radial-constrained A* simulation conducted in the mass simulation 4.3.1.

The discrete path from radial-constrained path planner is still a good initial guess for the optimizer. Depending on the priority of the optimization, the path can be made continuous and feasible from about 50 ms to 10-20 seconds (from table 4.18 and section 4.3). Since the radial-constrained A* finds a cost-efficient path with clearance to high-cost areas, optimizing the path for 10-20 iterations is sufficient to make the path continuous and driveable.

The objective function designed for this path planner was to yield the desired driving behavior, to keep the drive smooth, cheap, and feasible. From the conducted tests made in chapter 4 where all terms of F (from 3.5) were included, the optimized paths proved to be the most cost-efficient and feasible option. From these tests, we can conclude that the design of the cost-function for gradient descent optimization was successful.

Choosing a cost-efficient path or the problem of cost-optimization are all dependent on a costmap. The costmap is supposed to represent the traversability in the terrain, where the low-valued cells have the highest driveability. It is assumed that a smooth, cheap, and feasible path found in FFI's costmap is driveable, but there is no guarantee before it is tested on a physical autonomous vehicle. What is certain, is that the algorithm developed in this thesis only need a map represented of numbers, the dimension of the non-holonomic vehicle as well as maximum steering angle to produce a cheap, smooth and feasible path.

Run-time has not been the main focus while developing this path planner. But to give the reader a feeling of how memory-consuming the different implementation was, the run-time was demonstrated together with the execution of the different algorithms. The radial-constrained A*'s run-time can be compared relative to the run-time of the regular A* implementation, which is known to be a time-efficient graph-search algorithm. This gives an impression on how much work remains to get the path planner to operate in real-time.

To get a lower run-time of the optimization, would be to change optimization algorithms from gradient descent method to conjugate gradient method. The conjugate gradient method is better suited to handle large-scale nonlinear problems, and will most likely give a positive response on the path planners run-time.

Chapter 5

Conclusions and future work

Non-holonomic motion planning in terrain is a challenging task, as the world can not simply be divided into traversable and non-traversable areas. Different techniques in the literature have been developed to address the problem of navigating a non-holonomic vehicle in terrain, by discrete- and continuous-graph search algorithms, non-holonomic navigation, and trajectory optimization.

FFI provided a costmap which is used by the unmanned ground vehicle OLAV to navigate in terrain. This map is based on terrain characteristics and is used as an input for the developed path planner. The costmap is dilated to enhance the most critical parts in the terrain and maximize the chance to create a safe path. Further on, the A* algorithm is modified to create discrete paths through the dilated costmap that respects a non-holonomic vehicle's maximum steering angle. When the radial-constrained discrete path is obtained, a gradient descent optimization is performed to smooth the path, while maintaining the curvature and cost-efficiency of the path.

Assuming the start- and stop-coordinates are not randomly generated, it can be concluded that the developed path planner in this thesis is generating sufficient paths for a non-holonomic vehicle in terrain. The radial-constrained A* generates cheap- and discrete-paths that respect the minimum turning radius of a vehicle which gives a great fundamental for the gradient descent optimization. The optimization smooths the path efficiently and can keep it cost-efficient and maintaining the curvature during the optimization.

5.0.1 Further work

There are two elements from the results that appear to need further advancement, the node representation of radial-constrained A* and the implementation of the path planner. The node representation of the A* search are most likely responsible that the search sometimes fails(4.3.1), or create too expensive roads to the goal (4.1a,4.19c). It would be interesting to investigate the consequence of including the angle between the body frame and world frame when a node is expanded to widen the search and allow entering the cell from multiple angles.

Some work still remains on the implementation to get the motion planner to work in real-time. The radial-constrained A* had some higher run-time relative to the regular A*. The run-time of radial-constrained A* will most likely improve by carrying out a better implementation of A*. Lastly, a stopping criteria based on the paths cost- and feasibility should be developed to ensure a cost-efficient and safe path.

References

[1] Boyd, S., Boyd, S. P. and Vandenberghe, L. [2004]. *Convex optimization*, Cambridge university press.

[2] Bø, S. [2019]. Motion planning for terrain vehicles, *Project report in TTK4551*, Department of Engineering Cybernetics, NTNU – Norwegian University of Science and Technology.

[3] Carsten, J., Rankin, A., Ferguson, D. and Stentz, A. [2007]. Global path planning on board the mars exploration rovers, *2007 IEEE Aerospace Conference*, IEEE, pp. 1–11.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. [2009]. *Introduction to algorithms*, MIT press.

[5] Dolgov, D., Thrun, S., Montemerlo, M. and Diebel, J. [2008]. Practical search techniques in path planning for autonomous driving, *Ann Arbor* **1001**(48105): 18–80.

[6] Ferguson, D. and Stentz, A. [2006]. Using interpolation to improve path planning: The field d* algorithm, *Journal of Field Robotics* **23**(2): 79–101.

[7] FFI [2019]. Unmanned ground vehicles for the armed forces. Accessed: 08.12.2019.
URL: *https://www.ffi.no/en/research/unmanned-ground-vehicles-for-the-armed-forces*

[8] Games, R. B. [2016]. Introduction to the a* algorithm. Accessed: 01.12.2019.
URL: *https://www.redblobgames.com/pathfinding/a-star/introduction.html*

[9] Geeks [2020]. Erosion and dilation of images using opencv in python. Accessed: 20.02.2020.
URL: *https://www.geeksforgeeks.org/erosion-dilation-images-using-opencv-python/*

[10] Gill, P. E., Murray, W. and Wright, M. H. [2019]. *Practical optimization*, SIAM.

[11] Goldberg, S. B., Maimone, M. W. and Matthies, L. [2002]. Stereo vision and rover navigation software for planetary exploration, *Proceedings, IEEE aerospace conference*, Vol. 5, IEEE, pp. 5–5.

[12] Guastella, D., Cantelli, L., Melita, C. D. and Muscato, G. [2017]. A global path planning strategy for a ugv from aerial elevation maps for disaster response., *ICAART (1)*, pp. 335–342.

[13] ja72, s. [2020]. Get the equation of a circle when given 3 points. Accessed: 25.01.20.
URL: *https://math.stackexchange.com/questions/213658/get-the-equation-of-a-circle-when-given-3-points*

[14] Kurzer, K. [2016]. *Path planning in unstructured environments : A real-time hybrid a* implementation for fast and deterministic path generation for the kth research concept vehicle*, Master's thesis.

[15] Larson, J., Trivedi, M. and Bruch, M. [2011]. Off-road terrain traversability analysis and hazard avoidance for ugvs, *Technical report*, CALIFORNIA UNIV SAN DIEGO DEPT OF ELECTRICAL ENGINEERING.

[16] LaValle, S. M. [2006]. *Planning algorithms*, Cambridge university press.

[17] Liu, Y. and Bucknall, R. [2016]. The angle guidance path planning algorithms for unmanned surface vehicle formations by using the fast marching method, *Applied Ocean Research* **59**: 327–344.

[18] McREYNOLDS, T. and BLYTHE, D. [2005]. Chapter 19 - illustration and artistic techniques, *in* T. McREYNOLDS and D. BLYTHE (eds), *Advanced Graphics Programming Using OpenGL*, The Morgan Kaufmann Series in Computer Graphics, Morgan Kaufmann, San Francisco, pp. 501 – 530.
URL: *http://www.sciencedirect.com/science/article/pii/B9781558606593500214*

[19] Møller, M. F. [1990]. *A scaled conjugate gradient algorithm for fast supervised learning*, Aarhus University, Computer Science Department.

[20] Nocedal, J. and Wright, S. [2006]. *Numerical optimization*, Springer Science & Business Media.

[21] opencv [2020]. 2d convolution (image filtering). Accessed: 10.05.2020.
URL: *https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html*

[22] Petereit, J., Emter, T., Frey, C. W., Kopfstedt, T. and Beutel, A. [2012]. Application of hybrid a* to an autonomous mobile robot for path planning in unstructured outdoor environments, *ROBOTIK 2012; 7th German Conference on Robotics*, VDE, pp. 1–6.

[23] ranger [2020]. Ranger xp 900 specifications. Accessed: 10.05.2020.
URL: *https://ranger.polaris.com/en-us/2019/ranger-xp-900-sage-green/specs/*

[24] Ruder, S. [2016]. An overview of gradient descent optimization algorithms, *arXiv preprint arXiv:1609.04747* .

[25] Schulman, J., Ho, J., Lee, A. X., Awwal, I., Bradlow, H. and Abbeel, P. [2013]. Finding locally optimal, collision-free trajectories with sequential convex optimization., *Robotics: science and systems*, Vol. 9, Citeseer, pp. 1–10.

[26] scipy [2020a]. Map of norway. Accessed: 18.05.2020.
URL: *https://www.batteriet.no/ressurssider/kart-norge/*

[27] scipy [2020b]. numpy.gradient. Accessed: 10.05.2020.
URL: *https://docs.scipy.org/doc/numpy/reference/generated/numpy.gradient.html*

[28] slant [2020a]. Dijkstra vs a*. Accessed: 30.03.20.
URL: *https://www.slant.co/versus/11584/11585/ dijkstra-s-algorithm_vs_a-algorithm*

[29] slant [2020b]. Dijkstra vs bfs. Accessed: 27.03.20.
URL: *https://www.slant.co/versus/11584/11586/ dijkstra-s-algorithm_vs_breadth-first-search*

[30] Soltani, A. R., Tawfik, H., Goulermas, J. Y. and Fernando, T. [2002]. Path planning in construction sites: performance evaluation of the dijkstra, a-star, and ga search algorithms, *Advanced engineering informatics* **16**(4): 291–303.

[31] Stout, B. [1997]. Smart moves: Intelligent pathfinding. Accessed: 01.12.2019.
URL: *http://www.cis.umassd.edu/ ivalova/Spring08/cis412/Old/SMARTMOV.PDF*

[32] Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G. et al. [2006]. Stanley: The robot that won the darpa grand challenge, *Journal of field Robotics* **23**(9): 661–692.

[33] Webster, M. [2020]. Optimization. Accessed: 18.04.2020.
URL: *https://www.merriam-webster.com/dictionary/optimization*

[34] Wilson, G. N. and Ramirez-Serrano, A. [2014]. Terrain roughness identification for high-speed ugvs, *Journal of Automation and Control Research* **1**: 11–21.

[35] Yakovlev, K., Baskin, E. and Hramoin, I. [2015]. Grid-based angle-constrained path planning, *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, Springer, pp. 208–221.

[36] Yao, Y., Rosasco, L. and Caponnetto, A. [2007]. On early stopping in gradient descent learning, *Constructive Approximation* **26**(2): 289–315.

[37] Zucker, M., Ratliff, N., Dragan, A. D., Pivtoraiko, M., Klingensmith, M., Dellin, C. M., Bagnell, J. A. and Srinivasa, S. S. [2013]. Chomp: Covariant hamiltonian optimization for motion planning, *The International Journal of Robotics Research*

32(9-10): 1164–1193.
URL: *https://doi.org/10.1177/0278364913488805*

Sondre Bø

Motion planning for terrain vehicles: Path generation with radial-constrained A* and trajectory optimization

NTNU
Norwegian University of
Science and Technology

FFI Forsvarets forskningsinstitutt
Norwegian Defence Research Establishment