Determining the Optimal Route for a Tethered Manure Applying Robot

Robin Oosterbaan

TU Delft

Determining the Optimal Route for a Tethered Manure Applying Robot

by

Robin Oosterbaan
4601556

This thesis was made at:
Computer Science
Algorithms group
Delft University of Technology

With support from the Cognitive Robotics department

In collaboration with Lely Technologies

Preface

This thesis is the result of months of hard work. Perhaps a bit late due to taking a leap year because of a lack of motivation due to a certain pandemic. Since my background is Computer Science and this project has a lot of overlap with Robotics, I had to explore topics beyond the scope of Computer Science and gain knowledge along the way. Next to learning the ins and outs of Robotics, I learned that I enjoy this area of work. In the future I hope to continue working with the combination of Computer Science and Robotics.

First, I would like to thank my supervisor Neil Yorke-Smith, for giving me the freedom to pursue this topic and for the insightful discussions we had along the way.

I also want to thank Yke Bauke Eisma for both his guidance, feedback and encouragement to start this endeavour. These supervisors are the reason I was able to pursue this project and have it come to a sound conclusion.

During my time at Lely I quickly learned a lot and was able to discuss my problems with them. For this I would like to thank Jasper Wijkhuizen for: supervising, discussions and his expertise in route planning. There is also the Jojo team consisting of: Pieter van Driel, Howard Sie and Karel van den Berg. Who I also want to thank for helping me with the vehicle, the insightful discussions and ideas.

Lastly I would like to thank my partner for the discussions, proofreading and patience.

Abstract

Lely is developing a tethered manure applicator robot, designed to work on the fields of farms. This vehicle called the Jojo, is constrained in its movement due to the attached tether. It can not make tight turns and can only reverse by backtracking the driven path. This thesis explores the path planning issue and provides a novel solution. The problem is best described as the shortest path from a starting point to a set of strokes that cover the field, with a constrained turning radius. We first explore existing solutions and discover that none exist that are suitable. Further research finds that the behaviour of this vehicle is best compared with steerable needles, but at a different scale.
First, we construct a simulation environment for the vehicle. This simulation is used to develop the plan execution engine, path follower and safety checks. This is necessary for testing the solutions in the simulation and in the real world.
Secondly, with some inspiration from the solutions provided for the steerable needles, we arrive at a method for finding a Jojo path. The method makes use of a large graph that represents translations associated with a given location. We also provide methods for attaching the starting point and destinations in the form of strokes, either via the start or end of the strokes or via the middle. A Jojo plan is then obtained by performing a directed Steiner tree approximation on this graph, where the starting point is the root and the terminals are the strokes. In order to make the approach fast enough, we spend a substantial amount of time profiling and analysing the program and design a number of performance improvements. These allow for the approach to solve instances that are larger than necessary.
Thirdly, we test, compare and improve the approach. We look at both small instances to perform numerical analysis in order to optimise a set of parameters. We then inspect larger instances that are based on real world scenarios.
Lastly, we perform a set of real world tests using the vehicle. One of these tests highlights a shortcoming of the algorithm, which we discuss and provide a solution for.

Glossary

DSTP	The Directed Steiner Tree Problem is the same concepts as STP, but in a directed graph. Rather than only specifying a set of required vertices, the vertices should all be reachable from a given starting point.
GNSS	Global Navigation Satellite System (GNSS), the Global variant of satellite geopositioning.
Jojo	The tethered manure application robot.
PID controller	Control loop mechanism commonly used for continuous systems, the mechanism incorporates historical and live feedback to determine the next set point. This mechanism is often used with cruise control to determine engine power output.
Pit	The docking point for the vehicle.
ROS2	Robot Operating System 2, a software suite consisting of libraries and tools for robot applications.
RPP	Regulated Pure Pursuit. An improved version of the carrot following algorithm.
RVIZ2	Software program for visualizing robot states in a 3D environment.
STP	The Steiner Tree Problem in graph theory, the set of edges that connect all required vertices of a graph with the least total cost.

Contents

1 Introduction

This chapter is meant to familiarise the reader with the Jojo problem and explain the goal of this thesis. This research is performed in collaboration with the company Lely [18]. Lely is a Dutch concern known for its autonomous cow milking robots, automatic cow feeding robots and manure collecting robots. The world of dairy faces many challenges regarding the environment and as such, Lely is currently developing a robot that aims to improve the circularity in the sector. The concept is an 'on-demand' manure distributing robot that tries to limit the time between manure collection and application. The intended usage of the vehicle is for it to drive over each part of the field and cover it in manure. The robot carries a ~500 metres long hose that can be attached to docking points in the fields, it is used to provide the manure by pumping it through, but the hose is also a contributor as to why route planning for the vehicle is challenging. Unlike the often used drag hose manure applicators, this vehicle does not drag the hose, nor does it move the hose, it unrolls it when driving forward and rolls it up when driving backwards, hence the name Jojo. This means that reversing is always done by backtracking the already driven path. Field coverage planning for agricultural vehicles is a well studied topic and so is the topic of tethered robots. The intersection of these two however, makes for a novel problem, which this thesis aims to provide a solution for. We first research the vehicle and the control mechanisms, the path planning problem that arises, investigate multiple potential solutions and lastly perform real world tests. This chapter starts with an introduction to the Jojo problem in 1.1, we then introduce the main research question and the accompanied sub questions in 1.2 and lastly we summarize the contributions this thesis aims to make in 1.3.

1.1. Motivation

Lely already has a software suite designed for calculating mowing patterns for their robotic mower. Lely now requires an algorithm that can be added to this suite, but rather than constructing routes for the mower, it should construct routes for the Jojo. This suite is designed to be operated by trained professionals and thus the algorithm has different requirements as opposed to a system without a human in the loop. This for instance means that the calculation time should be in the scale of a few minutes in order to allow for iterative improvements and still be user friendly, but we get the benefit of manual interventions, insights and modifications. The current version of the software suite allows for creating fields and generating strokes within these fields. These strokes are straight lines over which the vehicle should drive and distribute manure. The missing piece is planning a route towards these strokes. As we discover during this research, the problem at hand is challenging to solve due to the constraints of the vehicle as well as the size of the problem instances. The goal of this thesis is to provide a software solution that can aid a trained professional with constructing a route.

1.2. Research questions

In order to reach the goal as presented in 1.1, we introduce the main research question to be formulated as follows:

What is a suitable solution for calculating the optimal route a tethered solar powered manure applying vehicle must follow, in order to reach a set of strokes located within a field?

We also formulate the set of sub-questions that we will use in order to be able to answer the main question to be the following:

1. What would be 'suitable' solution for the Jojo problem?
2. What is a supposed good method to structure and create the solution?
3. Does the solution scale sufficiently to allow for real world scenarios to be solved within a reasonable amount of time?
4. How does the solution compare in different scenarios?
5. Are the generated solutions executable in the real world?

1.3. Contributions

The contributions of this thesis are the following:

- Develop a path following algorithm based on a trivial pure pursuit algorithm.
- Develop a Jojo plan execution engine.
- Integrate the path follower and execution engine to work in a vehicle simulation.
- Develop a novel and performant method for finding a curvature constrained shortest path from a starting point to a set of end points.
- Develop a fast and memory saving graph representation.
  - Investigate methods to best represent the problem instances to satisfy the performance constraints.
  - Develop a method to translate the Jojo problem into such a representation to then be solved with the aforementioned path planner.
  - Develop a method to translate the solution from the path planner into a Jojo plan which the vehicle can execute.
  - Develop and benchmarked multiple performance improvements made to allow the algorithm to be used on large instances.
- Benchmark the proposed method on smaller scale instances to optimize the parameters and show the difference between the used solvers.
- Benchmark the proposed method on large scale instances to show calculation times and discuss the quality of the solutions.
- Perform real world tests on small instances that are chosen to contain the difficult to execute scenarios.

1.4. Outline

This thesis aims to provide an algorithm for aiding a professional when constructing Jojo plans, that can be integrated with the existing Lely software suite.

Chapter 2 discusses the related works, that highlight the novelty of the problem at hand. Chapter 3 introduces the reader to the Jojo system and explains the difficulty with controlling this vehicle. It also provides an explanation of how the path follower is designed and how the vehicle is simulated. Chapter 4 is designed to explain the thought process behind the reason for ignoring some constraints and why we opted to use a graph-based path planning approach. Chapter 5 is a large chapter that explains how we represent and solve the problem to create a Jojo plan. This chapter also benchmarks subsections of the algorithm to show why certain design decisions were made. Chapter 6 describes a set of conducted

experiments with small instances to benchmark the method and optimize variables. We also use real world scenarios from the farm on which the vehicle is currently located to generate multiple Jojo plans. The latter instances are simulated to show execution times. Chapter 7 describes conducted experiments using the real vehicle using small problem instances, but chosen to represent the difficult scenarios. The last chapter 8 summarises and reflects on the proposed algorithm.

2

Relevant Literature

This chapter is meant to give an overview of the state of the art research that is regarded necessary for solving the Jojo planning problem. It explores the basic concepts of tethered vehicles and the combination of coverage planning for tethered vehicles. Especially helpful for the reader is the topic of agricultural path planning, as this will highlight common difficulties of field segmentation and efficient vehicle usage. While a strong attempt is made to provide reliable and accepted sources, the research will sometimes be from a less accepted sources, since tethered vehicles are a niche topic and therefore not much research has been conducted. This chapter is responsible for answering the first sub-question: "how are similar coverage planning problems described and solved in literature?"

2.1. Agricultural path planning

Route planning in the agricultural world is often the coverage planning problem. One has a field that needs to be treated using a machine with specific constraints, think of driving width, overlap between passes or allowed steering behaviour. The most common approach one finds in literature is to draw parallel lines over the field in a suitable direction and come up with a method to best connect each line with each other. Oksanen et al. [26] provide a more sophisticated method to design these lines whereas the lines are often connected using a specific pattern. These path planning techniques are designed to work with regular farm vehicles, not one that is attached to a tether. Although the generated strokes seem usable, the generated headlands are not. Constructing optimized coverage patterns is difficult for regular fields, but becomes increasingly difficult for irregular shapes.

2.2. Path planning

In general [9], but also specifically for robots, this topic is widely researched [14]. For this research we focus on the planners that are compatible with the Jojo. This requires the planners to take maximum steering angle and obstacles into account. The most used and proven doctrines are: Hybrid-A* and State lattice. The implementation of these algorithms are also included in the Navigation 2 software stack designed by Macenski et al [21]. The first algorithm runs the A* algorithm on a three dimensional grid, where the last dimension is used to express rotation. The algorithm expands to locations on the grid that are reachable for the vehicle, the most commonly used version for robots is designed by Dolgov et al. [6]. The latter makes use of the same grid, but uses a pre-calculated set of transitions to turn the grid into a graph. These transitions are chosen to be suitable for the vehicle to execute, which makes traversing any direction in the graph a feasible route for the vehicle. The commonly used implementation is based on the design of Pivtoraiko et al. [28]. The topic of multi-goal steering constraint path planning is also heavily researched. Think of the Dubins Travelling Salesman Problem [25]. The topic of being constrained by a tether is quite unexplored, but will be discussed in the next section.

2.3. Tethered vehicles

The idea behind a vehicle or robot that is attached to a cable, rope or commonly referred to as tether is to use it as an advantage. One may use the tether as a way to power the vehicle, say for a robot vacuum [15] or underwater robot. Others use the tether as a way to return back to base, often used for exploration and rescue robots [2]. Another possibility is to use the tether as a force vector, this is often done by hanging from the tether in order to reach previously inaccessible places, say craters on the moon [23]. These vehicles all make use of strong and flexible tethers and the robots are often able to rotate around their axle using differential drive [22]. One can conclude that although they look similar, none of these robots are similar to the Jojo, which is designed around a fragile and immovable tether. Another interesting and perhaps more similar field of research is steerable needles. They are attached to a flexible and strong tether and are constrained in their ability to make turns. These needles are curved and as a consequence moving forward causes a steering behaviour, moving straight requires frequent rotation. The turning radius of these needles is restricted to the curve they are given. Due to surrounding tissue, the needle can only return the way they came by means of pulling the tether. Reaching multiple places is often done by retracting the needle and steering forward using a different angle. These needles are used in scenarios where the target is hard to reach and the tissue is vulnerable, this thus requires computer assisted path planning. Since the search spaces are relatively small and the solution is of high value, researchers often make use of brute force methods to determine the best path such a needle should take. Liu et al. [19] have developed a three dimensional greedy guided fractal tree that expands until it reached the target. These trees are fast due to the GPU acceleration, but are constrained to small working areas. Fu et al. [8] criticise these fractal trees by stating they are not always correct due to the greedy nature, where the tree is steered based on a coarse resolution. One may argue that increasing the compute ability would alleviate said problems, but the researchers introduce a resolution complete motion planner for these needles, which is proven to run in finite time. This area of research is mostly focused on finding the optimal path for a needle from A to B, but Lobaton et al. [20] introduce a path planner for the multi-goal version. They construct a search space by randomly placing circles with the circumference of the turning radius and connecting nearby circles. They then solve the Steiner Tree Problem [10] on this search space in order to find the multi-goal shortest path. They show that the cost of multi-goal optimisation is lower compared to individual path planning, but perhaps more important for our use case is the creation of a search space that is kinematically feasible for the needle to traverse to later be solved using a Steiner Tree solver.

2.4. Steiner Tree Problem

Also known as the STP is best described as a variation of the minimum spanning tree problem, but rather than including all nodes, only a given selection of vertices is required. This problem also applies to a directed graph and is named the DTSP. Both problems are regarded as NP-Hard, but the directed version is seen as the most difficult [12]. Multiple approximation algorithms exist for the DTSP, but recently Watel et al. [39] developed a novel approach that, although capable of creating bad approximations gives good results for most instances.

2.5. Conclusion

In this chapter, we summarise and highlight relevant literature to help solve the Jojo problem. We find that stroke generation on farm land for regular vehicles is trivial, but perhaps not entirely suitable for the Jojo. Existing path planning algorithms are also found to be both sufficient and insufficient for our use case. Planning a path from A to B is trivial, the same holds for multi-goal path planning, but not for the tethered version. We continued with exploring existing tethered vehicles to find potential solutions for the Jojo problem. We learned that although they appear similar, all tethered vehicles use the tether in a different way compared to each other and compared to the Jojo. We do find a similar field of research if we treat the Jojo as a steerable needle. These needles experience the same kinematic constraints, are attached to a tether and have existing planners for multi-goal problems. Although these planners are perhaps undesirable due to compute requirements and ability to scale due to the intended use case, the topic is the most similar to the Jojo problem.

3

System & Control of the Jojo

In this chapter we first explore how the Jojo system is designed to work and explain the constraints that arise from this. We then look at how the vehicle behaves and how it is controlled. For this, we design a custom vehicle controller that is able to follow paths. Since the vehicle is large and heavy, we put special care in preventing sudden movements by designing a custom acceleration control scheme and we provide additional safety mechanisms.

3.1. System

This section aims to give the reader an explanation of how the Jojo system is intended to work. This section also looks at the current status of the prototype and its capabilities and limitations.

3.1.1. Vehicle

Figure 3.1: The Jojo

The vehicle, also known as Jojo, is designed to have two rigid rear wheels and a steering front axle, also using two wheels. The weight of ~2000kg and even more when filled with fluid, would normally be regarded as a large, but in the agricultural setting this is 'lightweight'. A large reel is placed in the centre of the vehicle and below the reel is a fluid applicator attached to the hose. This reel is where the hose is stored. Driving forwards unrolls this hose, driving backwards rolls it back up. Important to note is that unlike most tethered vehicles [35], the unrolled hose stays stationary on the floor when the vehicle is moving; it is not pulled or pushed. As a consequence, the vehicle must always drive backwards the same way it went forward. Failing to do so will break the hose as it is quite brittle. The current prototype uses a hose of 250 metres, but in the future this can become more than 500 metres depending on the farm size.

Steering
The car like vehicle has a distance of 2.55 metres between the two axles. The theoretical steering angle is 55°, which gives a radius of 1.7 meters, however, this angle is not used due to the large slip forces acting on the rear wheels and due to the hose not being able to make these tight turns. Note that the hose is being guided by a cart. This cart moves left to right and back again when rolling and unrolling the hose. As one can imagine, the position of the cart determines the maximum steering angle. If we make a left hand turn, with the cart on the right, the hose will be more relaxed as it experiences a larger turning radius than that of the vehicle, but if the cart is also on the left, the turning radius will be less than that of the vehicle.

Field tests have shown that it is possible to steer with 45°angle (3.6 meter radius) when driving with a hose and the cart in a favourable location, but, due to the forces still acting on the hose, in this scenario it appears more comfortable to constrain the vehicle to 35°steering angles. 45°is even more difficult with the cart in the unfavourable location and even though Lely is currently working on improving this, no guarantees are given, therefore we conclude that the 35°limit is appropriate for our scenario. In practice this means that an acceptable turning radius is about 3.6 meters. If the hose in in the unfavourable position it experiences a turning radius of around 2.5 meters.

Localization
The vehicle uses a combination of sensors in order to determine its location. A dual-antenna GNSS system with an accuracy of sub centimetre and sub degree yaw precision is primarily used, but the encoders on the steering and drive wheels are also used to augment this precision. The latter is important for distinguishing whether the vehicle is standing still or slowly moving, since GNSS jitter can also be interpreted as minor movements while drive motors having an RPM of 0.0 can only mean the vehicle is stationary.

3.1.2. Attachment points

In the grass fields on the farm, multiple attachment points are placed where the reel of the vehicle can be attached to. These points are placed in such a way (in the middle of a field) that the vehicle is able to reach all corners of the field. It is not possible to drive over these points and they should be avoided by the vehicle. Figure 3.2

Figure 3.2: An attachment point with the hose attached

3.1.3. Vehicle representation

Due to the flat nature of grass fields in the Netherlands, the spatial representation of the vehicle is done using a 2-dimensional grid. The centre of the rear axle is chosen as the centre point of the vehicle. This is often chosen in literature when working with car-like models [34], but is not mandatory. We, however, have a good reason to use this centre point since the manure applicator is placed near the rear axle and the unrolling mechanism of the hose is also placed nearby. When we use the center of the rear axle as center point of the vehicle, we can use said point to track where the vehicle has already applied manure and we can say that the tightness of a turn at this point is identical to the tightness the hose experiences.

The GNSS coordinates are translated to a X, Y and yaw with respect to a chosen 0 location. This location is often chosen to be the corner of the farm, this is done to make all coordinates a positive value. The yaw is set to 0 when facing North and increased anti clockwise. Due to the high static and dynamic precision of the GNSS system used, we use this input as the default for determining velocity and acceleration. However, when the vehicle is not in motion, the GNSS system has very small fluctuations in the location. This is due to tiny inaccuracies, atmospheric disturbances and the vehicle shaking due to winds. In order to properly determine the difference between slowly moving and being completely stationary, the GNSS input is merged with feedback from the rear wheel encoders.

3.2. Control

In this section we look at the kinematic model of the vehicle and the control system used for navigating. Both are necessary for simulating and controlling the vehicle. We look at the vehicle dynamics, steering behaviour and path following algorithms in order to provide the basis for simulating and testing the generated paths.

3.2.1. Vehicle kinematics

Since the environment is represented in 2D, the vehicle state can be represented using a 8-dimensional state: $(X, Y, yaw, \dot{X}, \dot{Y}, \dot{yaw}, \ddot{X}, \ddot{Y})$. Although the vehicle has four wheels, it does not use the Ackermann steering geometry as found in cars. The two front wheels rotate around one point. This makes it so that the vehicle behaves more like a tricycle, where the front axle can be reinterpreted as a single virtual wheel as shown in figure 3.3. Since the rear wheels do not steer this effectively makes the vehicle model similar to that of a bicycle.

The robot frame kinematics are:

$$
\begin{aligned}
v_x(t) &= v_s(t) cos\alpha(t) \\
v_y(t) &= 0 \\
\Delta\theta(t) &= \frac{v_s(t)}{d = 2.55} * sin\alpha(t)
\end{aligned}
$$

Where v_s is the linear velocity, α is the steering angle and d is the distance between the rear axle and the steering wheel, in our case 2.55 metres.

As such the robot kinematics in the world frame become:

$$
\begin{aligned}
\dot{x}(t) &= v_x(t) * cosyaw(t) \\
\dot{y}(t) &= v_x(t) * sinyaw(t) \\
\dot{yaw}(t) &= \Delta\theta
\end{aligned}
$$

In practice the simplified representation is not accurate. The environment is not always 2D, the wheels slip and are not inflated identically, the control of each motor is not perfect and neither is the control of the steering angle. Tests show that when performing manual control the delay in setting a steering angle and the vehicle reaching the angle is also between 0.5 and 2.0 seconds, depending on travel distance. Accelerating and decelerating also gives noticeable but predictable delay. Since Lely is currently also using a similar approach for controlling their vehicles and they work well, we say that the simplified representation is deemed 'good enough' for controlling and simulating the vehicle for now.

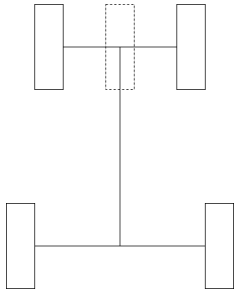


Figure 3.3: The virtual third wheel is placed in the centre of the front steering axle

3.2.2. Attached hose

When the Jojo is attached to a docking point, the vehicle must manage the hose it is attached to. Driving forward places down the hose, while driving backward retrieves it. This means that there are some restrictions to driving when attached to the hose. For one, the velocity of the vehicle should be controlled, but more importantly, the turning radius of the vehicle is reduced. Driving backwards must always be done by following the path that was followed when going forward. Failing to comply to these restrictions may result in damage to the hose or vehicle. Another limitation is the length of the hose, which makes it so that the vehicle can not drive forward indefinitely. These restrictions are the reason the route planning can be seen as unique. Literature often references tethered vehicles, but never vehicles that can only backtrack along driven routes.

3.2.3. Simulation

In order to first test the control algorithms for the vehicle, a simulation had to be built. For this, the bridge between software and hardware is disabled and replaced with a virtual bridge. The virtual bridge receives the commands the vehicle would normally receive, simulating the behaviour and giving feedback accordingly. The bridge uses a maximum linear acceleration and deceleration of 0.5 m/s^2, exceeding this value gives an error. The steering angle can be simulated with or without a delay in reaching the position. A future improvement could be to better simulate the steering wheel behaviour, by also simulating the PID controller that is normally running to control the angle. Visualisation of the behaviour is achieved by using RVIZ2 as shown in figure 3.5.

3.2.4. Path Following

Controlling vehicles can be made as difficult as one may desire. Model predictive controllers can be designed and tuned to both safe and reliably follow a path [4], but are difficult to implement. A carrot follower is one of the simplest methods, where a 'carrot' is placed before the vehicle and the steering wheel is aimed directly at said carrot. If the vehicle moves forward, the carrot is also moved along this path. The algorithm used for path following on the Jojo is a modified version of the Regulated Pure Pursuit algorithm [5], which is an improvement to the carrot following algorithm. This is necessary because the textbook implementation and the currently used one from the ROS2 navigation stack [21] lacks safeguards and behaves poorly at the last metres of a driven path, where the remaining path is shorter than the look ahead distance. In practice this leads to strong and abrupt steering corrections in the last metres of a path. The improvements made to the controller are summarised in this section:

- **Variable look ahead** is used to improve accuracy at lower velocities. The Pure Pursuit algorithm is normally given a fixed value for the look ahead distance. A shorter distance may cause the vehicle to overshoot, a longer distance will cause the vehicle to converge undesireable slow towards the target path. Since the Jojo has an identical reaction time on the steering wheel for both slow and fast driving, it can be argued that when driving slow, the look ahead distance can be shorter and while driving fast, the distance should be greater. This concept is not new and appears to work quite well [1]. As such, the controller scales the distance within a range of 1.0 to 4.0 metres depending on the actual velocity of the vehicle.
- **Interpolated look ahead points** is a small modification that makes it so that the look ahead point is not tied to the resolution of the generated path points. As mentioned in the introduction of

this section, when approaching the end of a path, the look ahead point comes to lie closer to the vehicle than the intended look ahead distance. In practice the vehicle is often a few centimetres removed from the desired path, when looking far ahead, this error results in a small steering correction. When approaching and thus looking a small (<20cm) distance forward, this error becomes large, resulting in strong steering corrections. The observed effect is that the vehicle stops at the correct point, but has a wrong heading. In our case, it is more desired to have the xy-error over the heading error. The simple fix is to stop steering when near the end of a path and just drive straight. However, we choose to interpolate the path and select a look ahead point from that interpolation. As a result, the vehicle no longer performs strong steering corrections at the end of the path which has the trade off that the vehicle may reach the destination target with a greater xy-offset than before. The benefit is that we increased the yaw precision.

- **Variable velocity** is used to address the issue of braking before cornering, which is preferred over limiting the velocity according to the steering angle. The latter causes the steering wheel to move before having reached the desired velocity that is tied to such a steering action, resulting in an error in follower accuracy but also causing dangerous situations due to fast cornering. We resolve this by calculating the velocity the vehicle needs when driving through a corner and expand these values to extend to before and after the corner as well. This way we have careful control over enter and exit velocity.
- **Acceleration control** is the last shortcoming that we solve. When designing vehicle controllers, one design decision to make is regarding acceleration and deceleration: does the low level hardware regulate this or the vehicle controller? The tables 3.1 3.2 explain the pros and cons of both approaches. The approach for the Jojo is to use a hybrid approach. The low level hardware regulates the acceleration and deceleration with high, but mechanically acceptable values. The controller is designed to regulate the acceleration and deceleration within said values. Figure 3.4 shows an abstraction of the velocity curve the controller uses. The controller is given a maximum acceleration value, a 'distance from target' at which to start decelerating and a distance at which the vehicle moves at minimal allowed velocity. The latter is used to slow the vehicle down to the minimum speed of 0.05 m/s at 0.1 metres away from the target, then roll forward until the target is reached and hard brake at that location. This approach assures that the vehicle gracefully accelerates and decelerates while also being able to stop exactly at the target location (Field tests on concrete have shown that the vehicle is able to consistently go from 1.0 m/s to standstill within 0.02cm of the desired location.)

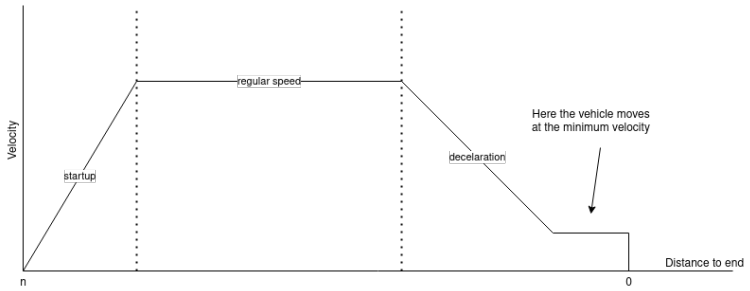


Figure 3.4: Abstract diagram highlighting the difference acceleration and deceleration phases the controller uses.

Pros	Cons
Acceleration variables can be dynamically modified depending on the driving scenario.	Higher risk of sudden acceleration and deceleration.
Sending a 0 velocity means the vehicle will stop directly.	More complex vehicle controllers are required.

Table 3.1: Pros and cons of high level acceleration & deceleration control

Pros	Cons
The vehicle will never accelerate or decelerate faster than the set value.	The vehicle will always be late with achieving the desired velocities.
Less prone to failure.	Stopping implies slowly coming to a standstill, which may cause the vehicle to overshoot the target.

Table 3.2: Pros and cons of low level acceleration & deceleration control

Lastly, the textbook version of the Pure Pursuit algorithm is often used in scenarios with small vehicles in dynamic environments that are using localisation techniques that are inherently more unreliable in nature. The control algorithms are designed to recover from faulty vehicle behaviour, GNSS uncertainty which can result in jumps and change in environment. For the Jojo there is no desire to recover from such situations, aborting is the suitable action. Scenarios that are detected include:

- failing to respond to movement instructions;
- deviation from the path above a set threshold;
- vehicle orientation compared to the path not being within the threshold;
- shake in the GNSS signal lowers the maximum velocity;
- loss of connectivity (GNSS requires a steady network);
- the pressurised path follower stops with working when the connection with the pump is lost.

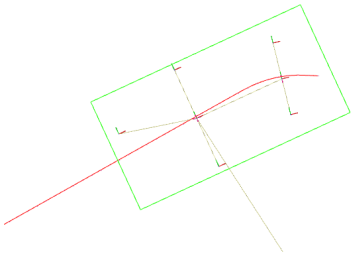


Figure 3.5: RVIZ2 visualization of the vehicle following a path running in a simulation

3.3. Conclusion

This chapter first explores the system, where we look into the design and limitations of the vehicle. We then look at how the vehicle is controlled. Since there was no existing path follower, we designed a custom one more suited for the large, heavy and tethered Jojo. We also incorporated a set of extra safety features which are important when are performing the real world tests with the vehicle. We also make it possible to simulate the vehicle and use that simulation to confirm the workings of the custom controller. In the end this chapter serves as the foundation for answering the sub questions of both "what would be a 'suitable' solution?" and "how would one go about validating such a generated solution?", the first requiring the knowledge of the limitations of the system and both the path follower and simulation being the foundation of answering the latter question.

4

Exploration

Due to the unconventionality of the "Jojo problem" we dedicate this chapter to searching for promising methods for solving the problem. This chapter first defines a simplification of the problem and then looks at different approaches, some self made and some from literature. We touch upon the workings of each approach and give a short analysis. The most promising approaches are implemented, tested, reviewed and viewed in the scope of the non simplified "Jojo problem". This chapter concludes that a novel approach performs much better than all others. As a result, the next chapter will give a more thorough explanation of the approach when applied to the full problem.

4.1. Simplified problem

The most important aspect of the Jojo problem, but also the reason we are not able to use conventional methods is the turning radius constraint. Therefore in order to initially develop and compare algorithms we use an informal simplified representation of the Jojo problem, that combines the turning radius constraint and shortest path to multiple points minimization into one problem. This approach is used due to the novelty of the problem at hand, which warrants the exploration of multiple approaches for solving the problem. As such, we define the simplified problem, explore multiple approaches for solving said problem and give a short evaluation of the performance and lastly make a comparison between the promising approaches. The most promising approach will then be used to solve the regular Jojo problem.

The simplified Jojo problem is presented as an optimization problem using a set of variables:

- The orientated starting point given as $P_s = (X_s.Y_s.\theta_s)$.
- Outer boundary, described by a non complex polygon named B.
- A set of orientated destination points given as $D = \{P_0, P_1, P_2...\}$.
- The turning radius of the vehicle given as R.
- The bounding box of the vehicle.

The constraint for the path are the following:

1. **Turns** in the path may not be sharper than the turning radius of the vehicle.
2. **Jumps** between two locations on the path are allowed.
3. **Collisions** between the bounding box placed anywhere on the path and the outer boundary are not allowed.
4. **Forks** in the path are allowed.

The problem description would be: Find the shortest constraint satisfying path, starting from P_s to set D. Figure 4.1 shows a visual representation of the problem.

One can see that this representation will not suffice for solving the actual Jojo problem. Specifically the lack of deciding at which end of a stroke the vehicle should enter and the hose having a maximum length, makes this representation less than optimal, however, being able to solve problems like this is fundamental to solving the latter.

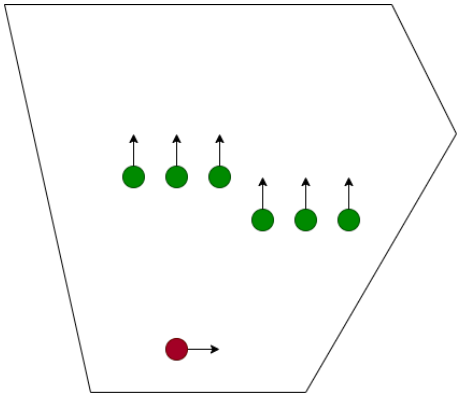


Figure 4.1: Visual representation of a simplified instance. Here the outer boundary is the polygon, the starting point is the red circle, the destination points are in green

4.2. Planner based algorithms

In this section we list and discuss potential algorithms that are fundamentally designed to use path planners that provide kinematically feasible routes between two given points. We effectively explore a promising set of path planning algorithms from the PythonRobotics project [30] that seem usable for our use case. We use this library since most path planners are implemented and are easy to use, which allows for quick exploration. These approaches benefit from the fact that the generated paths are not fixed to a grid and provide infinite precision. These planners are capable of finding a kinematically feasible route from A to B. The drawback of relying on such path planners is the inability to create paths that are less optimal for reaching a destination, but provide benefits in the future when navigating to the next point. There is a fundamental difference between finding a shortest path from A to B and then from B to C, and finding a shortest path from A to B and C. In short, these approaches do not think ahead.

- **Naive** is the simplest method. It simply calculates all routes from the root to the set of terminals using the Hybrid-A* planner [6] and uses these paths as the solution. One can see that this approach gives undesired results, but we include this approach in order to allow for a numerical comparison between the other algorithms. The algorithm is also quite fast due to only needing to calculate $|D|$ paths.
- **Greedy** is a simple improvement made to the naive implementation. The algorithm finds all shortest kinematically feasible paths from the starting position to the all destinations using the same planner as before. In this case the best path is the shortest path, the other paths are discarded. The algorithm then calculates the shortest paths to all non-reached destinations, starting from N locations sampled over the already created path. The shortest path is added to the existing path. The algorithm repeats the previous step until all destinations are reached. In order to improve the performance of such an approach, we can use a type of gradient descent. The list of points that need evaluation is sorted by distance from the closest destination point to the furthest. The algorithm keeps working through these points, but if improvements in shortest path are not made for a certain amount of samples, the algorithm stops further searches. This algorithm gives a better solution than the naive approach due to the larger search space. Initial tests showed the need for performance improvements due to the exponential nature of this approach. However, even with the gradient descent improvements, 50x50 metres instances with about 10 terminals resulted in computation times above 30 minutes.
- **RRT** or Rapidly exploring Random Tree was another interesting type of planner that could be used. Not only is this a well-studied way of path planning, the solutions that can be found appear quite similar to the kind of solution that we may desire. These algorithms are normally used for path planning between two points. Kinematically constrained versions also exist [38]. For our use case, we would use such an algorithm to construct a tree without a destination, starting from the starting point and have the boundaries be the outer boundary. Then we need to attach each of the destination points to the tree. For this step, we can use multiple motion planners, in our case Dubins paths are used. With all the destinations attached to the tree we then keep removing

all leaves that are not attached to a starting point or a destination. The resulting tree would be the solution. Performance wise the RRTs seemed promising. Nevertheless, the solutions that RRTs provide suffer due to the randomness introduced. The paths tend to swirl in suboptimal directions and as a consequence makes the solution appear unstructured. During testing, we also noticed that increasing the search area quickly introduces long computation times. The most difficult part, however, is attaching the terminals to the tree. The question of where to best attach is answered with "it depends". It depends on where the other terminals are attached. We are effectively left with a variation of the problem we originally had.

In this section we explored promising looking path planners from the PythonRobotics project, but next to receiving suboptimal solutions, we also quickly learned that the size of our problem will be the biggest challenge. All small instances that we handmade are about 1% of the size of the regular Jojo instances, but apart from the naive approach, the approaches already required multiple minutes to run in order to find a solution. Trying the planners provided by the library to find paths from A to B on the instances that we may encounter (500x500m) fields gave computational difficulty. As such, no more effort is put in these approaches and we continue with a drastically different approach in the next section.

4.3. Graph-based algorithm

Due to the undesired conclusion of the previous section, we continued exploring. This section explores the usage of grid based graphs. The philosophy behind translating the problem into a graph is that this allows for graph based algorithms to be used. In particular the Steiner tree problem [10] or short STP is of interest here. It is defined as: "Given a directed graph $G = (V, E)$ with weights on the edges, a set of terminals $S \subseteq V$, and a root vertex r, find a minimal weight out branching T rooted at r, such that all vertices in S are included in T." [41]. In our case, it means that the starting position is the root r and the desired points are the terminals S. The solution T would then also be the shortest path. Note that we cannot directly solve the Steiner tree problem due to the NP-Hard nature of the problem [12]. As such in this section we explore the alternatives for solving the instances.

First, we start with constructing a graph instance that is similar to the problem instance for the solvers to solve. We use the following procedure:

1. Create a grid that is aligned with the outer boundary;
2. For each point in the grid and a given orientation, test if it lies within the outer boundary. If so, include it in the graph (this implies that on the same location, multiple nodes are present in order to represent the amount of desired orientations (buckets) that are used);
3. For each node in the graph, search in a box around it for nodes that can be reached from the location and orientation the node represents. If so, add the length of the path as an arc in the graph.

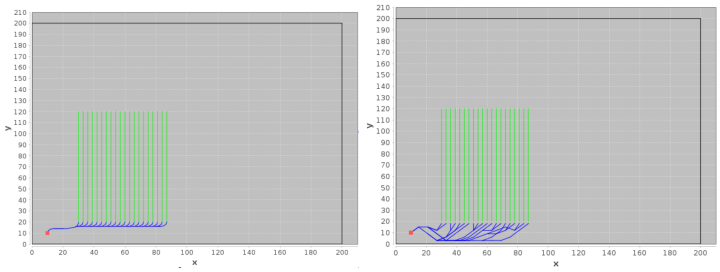
We now have a graph where if the vehicle is placed on a point and it keeps following connected nodes, it will never violate kinematic constraints.

Next, we look at a couple of solvers that we tested.

- **Approximation** algorithms were the first choice due to the bad scaleability observed for exact solvers on the steinlib instances [16]. Multiple approximation algorithms exist for solving the Steiner T problem. One naive approach is to calculate the shortest path to each terminal and return the intersection of these paths, similar to the Naive approach attempted before. A more practical approach is to find the shortest path between each terminal and the root using Dijkstra's algorithm, then set the weights for that path to 0 and repeat until all terminals are reached. This approach creates suboptimal paths due to the lack of thinking ahead. The paths are always the shortest from A to B, not the shortest from A to B and C. This approach is similar to the Greedy approach attempted before. However, quite recently (2016) the approximation algorithm GreedyFLAC [39] was developed. Since the release it has been used for scheduling traffic in data centers [24] [31] and can be seen as a breakthrough in approximation algorithms for the Steiner tree problem. The approximation ratio is not as good due to unlikely edge cases, but in general the average ratio is better than the Dijkstra approach. As such, this became the algorithm of choice. The results

of a simple test are shown in figure 4.2a and highlight the quality of the achieved solution. The solution shown as the blue path is identical to the best possible solution.

- **Ant colony optimisation** was also used to attempted to solve the DSTP. We use ACO to find the set of arcs in the graph that have the lowest total cost, but create a path starting at the start node that branches out towards all desired endpoints. The algorithm used is a variation of the one given by Prossegger, Markus, & Abdelhamid Bouchachia [29]. However, instead of using an undirected graph, we implement it using the directed graph. An ant is placed on each terminal and bootstrapped by first solving the path to the root using Dijkstra's algorithm and applying pheromones accordingly. Then for each iteration, each ant traverses the graph until either termination or reaching the root. The original algorithm dictates the use of the minimum spanning tree in order to combine the generated paths for each iteration. However, the minimum spanning tree is too hard to solve for the directed graph instance, thus we skip this step and combine all paths directly. The cost of the created graph is calculated and pheromones are applied to each arc accordingly. The idea behind using ACO was to allow simple incorporation of currently unused constraints. The individual ants can be blocked or punished for violating for instance the total length constraint. As such the algorithm was given a more critical evaluation. Initially, the ant colony algorithm achieved decent results on small instances, increasing the instance size, however, caused a large increase in computation time, which is why the bootstrapping of ants was introduced. Nevertheless, even after allowing the algorithm to go on for ~5 minutes, the result as shown in figure 4.2b highlight the shortcomings of this approach. The ants manage to find a route, but they fail at finding an optimal one. Giving the ants more time to run did not yield better solutions. Larger instances appeared out of reach for the ant colony optimisation.



(a) GreedyFLAC **(b)** Ant Colony Optimization

Figure 4.2: Visual representation of the best solutions achieved on a simple instance.

4.4. Conclusion

The disappointing results from the previous section 4.2 were the reason an attempt was made to use a graph-based approach. The idea of using graphs for route planning is not new, nor are these lattice motion structures. The combination of such a structure with solving a Steiner tree problem in order to find a valid path, is a new concept. The ant colony optimisation had high hopes due to the future benefits, but testing showed that even with aid and tuning, the ants still had difficulty traversing large instances. The other approach, using GreedyFLAC, originally thought to be less scalable, appeared to be much more reliable and better performing than any other attempted approach thus far. The conclusion of this chapter can also be translated into the answer to the sub question of: "what is a supposed good method to structure and create the solution?", since it was shown that the graph-based approach worked on the simplified problem. However, much work is still required to actually use this approach. Next chapter will further focus on the design of such an approach, but on the non-simplified version of the problem.

5

Algorithm design

Last chapter we arrived at the conclusion that graph-based approaches, which rely on approximation algorithms that solve the Steiner tree problem, are most promising for solving our problem. This chapter aims to apply said approach to the "Jojo problem". We first do a deeper dive in lattice graph construction and performance improvements that were made. We look at ways to allow the vehicle to enter the strokes at both ends and even provide a method to allow for entering in the middle and have the algorithm decide what approach is best. This chapter also goes over some aspects of the problem that cannot be solved formally, but we do provide a "fix afterwards" solution. Another important factor is the performance on larger instances. For this, we provide programmatic and algorithmic solutions that when combined hopefully provide sufficient performance in order to have reasonable calculation times. Lastly, we look at how the generated path is represented and stored in order for the vehicle to execute.

5.1. Design concept

As mentioned in the previous chapter, the idea behind the algorithm is to create a graph that is structured like a two-dimensional grid. We use a transition lattice to link nearby nodes in the graph to each other. If we carefully select the placement of the nodes and arcs, we can effectively create a graph that represents the search space of the Jojo problem. If all the nodes and arcs in the graph are checked for collisions with the boundary and the arcs represent feasible transitions, we can be sure that the vehicle can safely transition over this graph. This graph can then be used to solve the Jojo problem. We mark the root to be the starting location and the terminals to be the strokes, we can run a solver for the Steiner tree problem and use the solution as a valid Jojo path. Figure 5.1 shows the concept as a flow diagram. The first step is retrieving a Jojo instance, this is to be provided by the user interface. The second step is the generation of strokes, as discussed before we classify this step as trivial due to the knowledge Lely has and as such use a simple approach. The blocks highlighted in blue is what this chapter is about: The algorithm receives a Jojo instance with the strokes already calculated, creates a directed graph and approximates a directed Steiner tree solution, which is then used to construct a JSON formatted Jojo plan. The last step being the execution of the plan, will be left for the testing chapters.

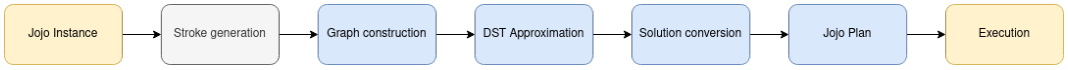
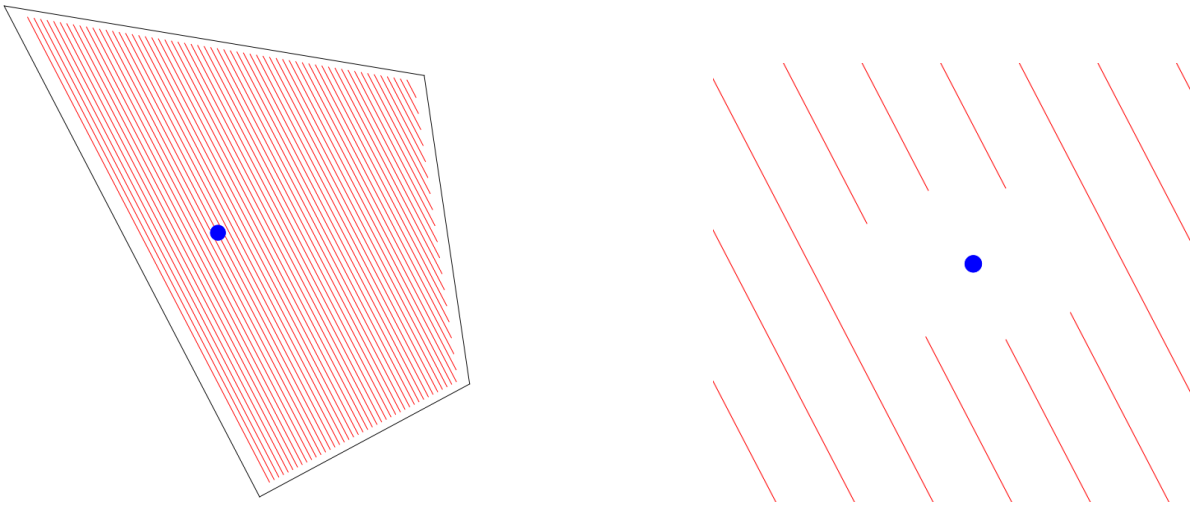


Figure 5.1: Flow diagram of the algorithm concept.

5.2. Stroke generation

The generation of strokes in agricultural settings is not new. [26, 27, 11] As such, we do not investigate this topic much. As also explained in the introduction, Lely already has in house stroke generation algorithms and desires only an algorithm that connects these strokes rather than one that also optimises the strokes. As such, we use a simple implementation where we place strokes 3 metres apart (the width of the vehicle) along the longest angle in the field polygon. This algorithm yields the results as shown in

figure 5.2a. The algorithm starts of with an outer and inner polygon that represent the field that needs manure and the boundary the vehicle may move inside of. We also use the location of the attachment point as an extra boundary since it is both not allowed to apply manure in its vicinity and the vehicle should not collide with it. This solution is also shown in figure 5.2b



(a) Simple field division.

(b) Strokes near the attachment point.

Figure 5.2: Visual representation of the simple field division algorithm on the left, and zoomed in at the attachment point on the right.

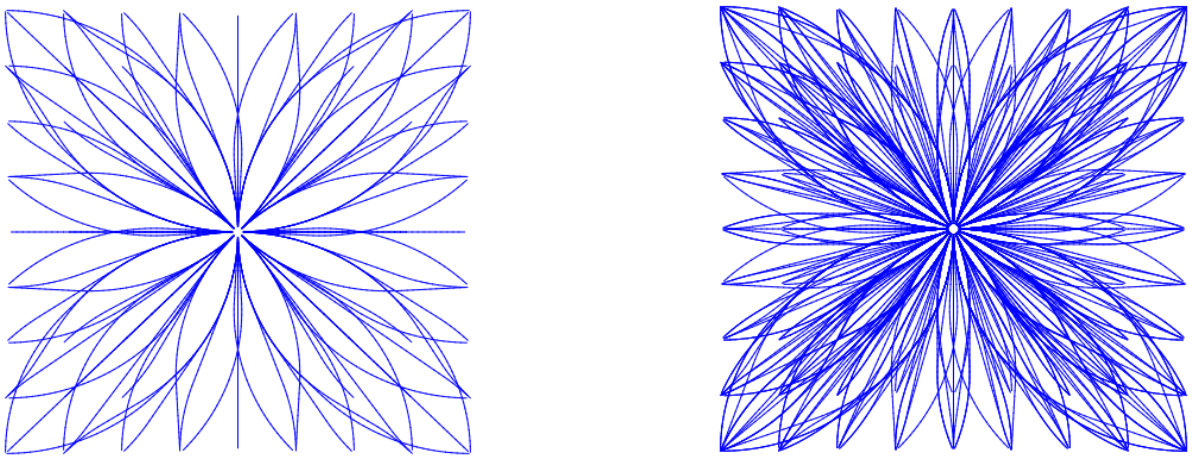
5.3. Lattice generation

Before we can start on graph construction, which is the next step in the program as shown in Figure 5.1, we must first calculate the translations that this graph will use. In the previous chapter the construction of the graph relied on manually-calculated transitions the vehicle could make between two nodes in the graph. This section focuses on creating a larger and more complete and representative transition table in order to better represent the capabilities of the vehicles. Paths constructed from chains of lattice arcs are in general better if there are more types of arcs to choose from. A simple example would be a 3x3 lattice versus a 5x5 one: Reaching a point 2 forward and 1 to the right (2, 1) translation, would cost the first lattice 2 steps of (1, 0) and (1, 1). The second lattice can perform a direct (2, 1) translation, which yields a shorter path. We look at path generation using different parameter and define when the path should be added to the transitions. As shown in figure, 5.3a the lookup table is designed to represent transitions in a NxN (N is odd) sized square, where the starting point is the center of the square. The figured lattices also show that when the number of lattices and dimensions is increased, the number of arcs also increases strongly.

These lattices can be configured in many ways, creating lattices with a large number of paths will result in a large graph. This is clearly visible in figures 5.3a 5.3b, where doubling the number of buckets results in an exponentially larger amount of arcs. The following list of parameters can be tuned:

- Maximum turning radius.
- Distance between nodes.
- Amount of angles (buckets) per location.
- Size of the lattice.
- Width of the cone.
- Turning penalty.

The dimensions of the lattice can be tuned accordingly. The bigger the lattice, the better representation of reality the graph becomes, at the cost of an increase in graph size. Creating a lattice of insufficient size can cause the graph to be too sparsely connected, making it so that certain nodes can be hard to reach. Therefore, selecting a lattice is a balancing act between performance and quality. For now, we



(a) 8 buckets.

(b) 16 buckets.

Figure 5.3: Visual representation of a potential lattice with a 1.0 metre resolution and a 9x9 grid with 8 and 16 buckets.

define the default lattice to be used with a resolution of 1.0 metres and 36 buckets and be $7x7$ in size. It is difficult to characterise the trade off between these variables. In general we can state that larger lattices will yield larger grids, but the quality of the solution is not directly tied to larger lattices. In the next chapter we will dedicate the section 6.2.3 to better understand the performance and quality trade off between the lattice parameters.

5.3.1. Path generation

The calculation of a transition from the center of the square to any other point and orientation could be done by drawing a circle between these two points, but we are better of using Dubins paths as they provide the shortest feasible path between two points. However, instead of using the shortest obtained path from the possible types (LSL, LSR, RSL, RSR, RLR and LRL), we test all six possible types for validity. If multiple types satisfy, the shortest one is used. The test criteria consist of the following:

- The entire path should be contained in a cone of a given width. We normally use a 60°cone as this seems to fit well with the turning radius, but this value can be modified at any time.
- No point in the path may have a distance from the start greater than the distance between the start and end node.

These conditions effectively limit the path to be both contained within a cone and a circle around the starting point. The appendix contains examples that show the arcs created for a set of configurations: Figure 9.1 shows the paths created when using low resolution (3 metres between nodes) grid size and figure 9.1 shows the same directional bucket but for a higher resolution grid size. Better methods exist for selecting the optimal motion primitives [3], but for our use case it is deemed sufficient to use the aforementioned approach.

5.4. Graph construction

This next step refers to the Graph construction step as shown in Figure 5.1. Construction of the grid based graph is done by first creating an evenly spaced grid and finding all nodes on the grid that are properly contained within the boundaries of the field and not near the docking point. This means that per grid location all possible rotations (buckets) are added as a node. All points on the grid that are at least a certain distance from any border are added straightaway. The bordering points are tested for collisions between the bounding box and the field boundary for each possible rotation and only the combinations that fit within the boundary are added to the graph. Next, for each node we use the look up table created in the previous section and determine for each transition if the graph contains the destination node also. If so, the transition is added as an arc in the graph, with the cost being the precalculated path distance that is associated with the transition. Since all impossible states are

not present due to the exclusion of impossible nodes, the next stop is to verify if no path contains an impossible state. As such: if any of the two nodes involved in the transition is associated with a location near the border of the boundary, the entire path must also be checked for collisions in order to prevent jumps between small barriers. Figure 5.4 shows how a segment of these graphs could look like when all the transitions are drawn.

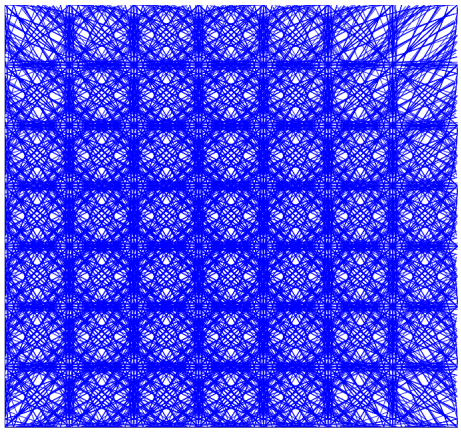


Figure 5.4: Visual representation of the lattice structures applied to a grid.

5.4.1. Representation

Constructing a graph for a 500x500 metres field, using a grid with a one metre resolution gives ~250.000 nodes. Using a lattice of size $7x7$ and 10°(36 buckets) rotation gives ~5 million arcs. Originally, both Guava's graph and JGraphT were used, but it was quickly discovered that these graph libraries are not designed for the intended usage. Therefore, a custom graph representation is used. In order to be both memory efficient and have high performance, we use a convenient 32 bit representation for each node. The first 6 bits represent the orientation, the remaining $2x13$ bits represent both X and Y locations in the grid respectively. The arcs are represented using 64 bits, where the first 32 bits represent the "from" and the last 32 represent the "to" in the arc. The graph is represented using two maps, that maps nodes to lists of nodes. These lists are sorted in descending order depending on the associated arc cost. This is done to facilitate the GreedyFLAC algorithm in the future. In addition to this, two maps are also used to keep track of the cost per arc. One small map that is used to store the costs for each transition, effectively replicating the costs associated with the transitions from the lattice. The latter map is used to store "special" costs. Later sections will explain why and how this map is used. This means that when finding a cost of an arc, we first check if the arc has a special cost associated with it. If not, we return the value in the transition table.

5.4.2. Strokes

In the simplified problem from section 4.1, we searched for a path towards a set of locations. This simplification lacks the real representation of a stroke, which, in reality, is able to be accessed from both sides.

In order to be able to have the algorithm decide which side of entry is better, we use a virtual node. This node has two incoming arcs, each coming from the nodes at each end of a stroke. This virtual node is configured to be the desired node for the Steiner tree solver.

The figures 5.5 and 5.6 show how this is implemented. The stroke visualized in the first figure is represented as shown in the second figure. The blue point corresponds to the "virtual" point and is not effectively attached to a location.

5.4.3. Grid decoupling

A consequence of using the stroke representation as described in the previous subsection is the difficulty of reprenting strokes that are not aligned with the grid. The grid, using a resolution of one metre, is sufficient when all the strokes are spaced either vertically or horizontally, because the vehicle is exactly

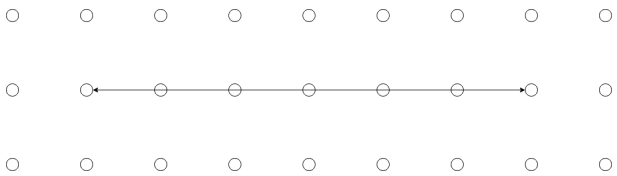


Figure 5.5

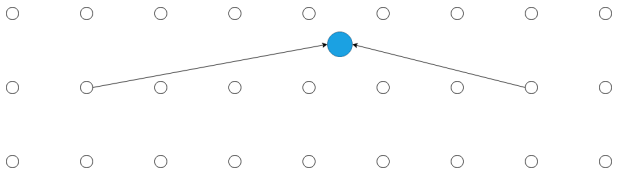


Figure 5.6

3 metres wide: being a multitude of one. The problems arise when the strokes are orientated in any other direction. The strokes begin to either overlap or leave gaps. This is the inevitable shortcoming of using a grid, which is why in order to address this issue, the strokes have to be decoupled from the grid. This is done by calculating paths from nearby nodes to both of the side of the stroke, using the same Dubins paths as earlier. This creates a large amount of arcs that all point to the earlier created virtual node. These custom arcs are also the reason the graph representation has a second map for keeping track of the custom arcs and associated cost as mentioned earlier. The figure 5.7 shows how this looks like in practice. Per stroke, the search area is a half circle at both ends. All nodes that can be reached from here are added as an arc to the virtual node as used in the previous section. The size of these areas is determined by the resolution of the grid. These areas are not intended to create many arcs in the graph, rather values of around 20 to 30 arcs is deemed reasonable. Smaller areas will lack substantial options for reaching the node, where large areas quickly expand to produce a high number of arcs. This is unnecessary and hits hard on performance.

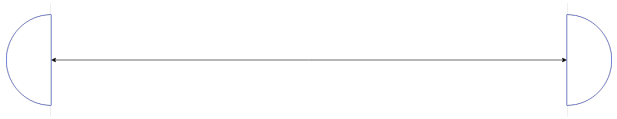


Figure 5.7: Visual representation of the 2 search areas that are used per stroke.

5.4.4. Mid stroke insertion

Lastly, another improvement to the strokes is the ability for the solver to decide on splitting the strokes and applying manure starting from the middle, rather than from one of the both ends. A mid intersection is intended to avoid the scenario where the vehicle is placed near a stroke, but first has to drive to one of the two ends to start applying manure. These intersections may produce small gaps relative to the size of the strokes, but allow for great reductions in total path length. In order to facilitate this possibility, we make further use of the grid decoupling approach. Not only the outer edges of a stroke are mapped to the virtual node, but also the nodes in the middle segment of a stroke can be added if a 'fork' can be constructed. Figure 5.8 shows all three regions that are now used when coupling the stroke to the grid. The middle segment is now added to allow mid stroke intersections.

Constructing a good fork can be quite difficult due to the many possibilities that are available. The most important being the width of the stroke that is cut in order to allow for steering room. Small widths create self intersecting forks, while increasing this size causes larger segments of stroke to be ignored. Figure 5.9 shows how the vehicle can intersect the stroke in the middle when starting from a node. In order to generate the forks we use the following approach:

1. Given a Point P and a Stroke S, first check if P lies in the boundary as shown in the figure 5.8.
2. Construct 2 circles with a size equal to the turning radius, one left of P and one to the right, check if these circles do not intersect the stroke.

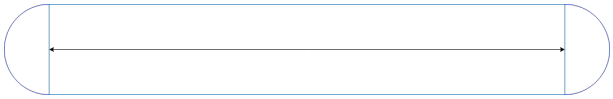


Figure 5.8: Visual representation of the 3 search areas that are used. The middle segment is used for mid attachments, the outer segments are used for regular attachments to the grid.

3. If the point succeeds at these two tests, now interpolate along the vector associated to the point, to find the intersection with the stroke S.
4. Find the shortest LSL and a RSR Dubins path respectively to the left and right of the intersecting point as shown in figure 5.9.
5. If both paths can be constructed and are not too long, they can be added as virtual node to the graph.

We also make sure that the remaining segments of the stroke are of sufficient length and allow for adding a 'punishment' in the form of an increased arc cost when choosing to use the mid stroke intersection. Since this is more a preference than an optimisation, the usage of this variable is left for the end user. We leave the punishment at 0, but the minimum stroke length is set to 15.0 metres.

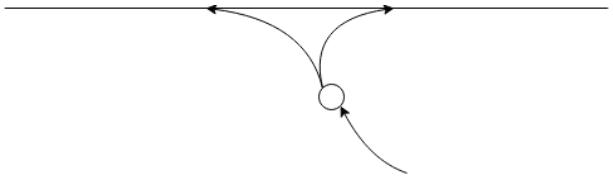


Figure 5.9: Visual example of a possible mid stroke intersection.

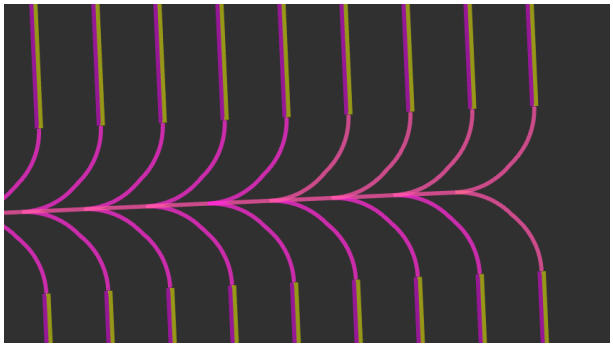


Figure 5.10: Mid stroke insertions as shown in the simulation.

5.4.5. Docking point

As explained in the introduction chapter, the starting point for the vehicle is located a certain distance away from the docking point. The vehicle first docks, drives forward for about ten metres and is only then ready. Since the docking orientation is chosen by the end user, the algorithm only needs to make sure that there are no strokes colliding with the docking point and enough safety margin is created. The use of driving forward for about ten metres after docking is to provide enough distance between the docking point and vehicle to allow the hose to touch the floor. Driving perfectly straight is no hard requirement. The first ten metres of driving can therefore also be used to our advantage, in order to align the vehicle with the grid. This means that unlike the strokes, the starting point requires no decoupling. The starting point of the graph is therefore a node in the graph that is closest to the location ten metres away from the docking point.

5.4.6. Conclusion

In this section we explained how we translate any Jojo problem instance into a directed graph. The idea being, that solving the directed Steiner tree problem on this graph will yield a suitable solution to the Jojo problem. The graph building process is done by constructing a representative graph of the environment and making sure all transitions are feasible for the vehicle. We then use abstractions for the strokes in order to be able to represent them without coupling them to the grid. These abstractions can be constructed to support end and mid insertions of the stroke. We also provide a method to represent the docking points in such a way that the solvers can find the most optimal orientation for the vehicle. Lastly, we make sure that this graph representation is represented in such a way to be both performant and memory efficient. The assumption is that this representation, which is also an abstraction, is sufficient for solving most of the instances. There are of course limitations which are to be discussed at the end of this chapter. With the representation in order, we can now focus on solving the directed Steiner tree problem, which is to be addressed in the next section.

5.5. Approximation algorithms

We compare three algorithms for approximating the Steiner tree problem.

1. **Naive** is a sanity check where we calculate the shortest route from the root to each other terminal and use the set of all used arcs as solution. The performance of this method is heavily dependant on the problem instance.
2. **Guided Dijkstra** is a smarter version than the naive approach. We first find the shortest path to any terminal, then set the cost of this path to 0 in the graph and repeating until all terminals are reached. This approach will always yield better performance than the naive implementation. It is even possible to steer this algorithm more by either setting the cost to a value below 0 or exaggerating the cost of all arcs. The latter is preferred due to the limitations of Dijkstra's algorithm.
3. **GreedyFLAC** [39] is a flow simulation based approach for approximating the Steiner tree algorithm. The algorithm expands from each terminal and combines flow strength when two or more terminals collide. Since the approach includes each terminal in the calculation and is therefore able to balance paths between multiple terminals and thus deviates from the shortest path from the root to the terminals, it can find paths the other two approaches can not.

The difference in behaviour of these algorithms is best visually explained. Figure 5.11 shows a clear difference in tree size between the naive algorithm and the other two. The difference between GreedyFLAC and guided Dijkstra is not directly visible, but is present. GreedyFLAC achieves a ~10 metre shorter path due to the 'thinking ahead'. The Dijkstra algorithm finds the shortest path to any stroke, in this case the second from the left, it then keeps on finding the closest path. However, this initial shortest path is already less optimal due to having to reverse further along this path in order to reach the strokes on the right side. The GreedyFLAC algorithm is able to balance this more and will balance the shortest path between all strokes, rather than the shortest path to just one. The authors of GreedyFLAC [39] provide an evaluation suite that has the state of the art approximation algorithms implemented. Using this suite, we created a few large graph instances and arrived at the same conclusion of the original authors. GreedyFLAC performed either similar or better both in speed and solution quality. This conclusion combined with the frequent decisions to use GreedyFLAC over other solvers in state of the art research led to the decision to stay with these three algorithms.

5.6. Performance improvements

Constructing the graph is no simple task due to the scale of the problem. Large fields could be sizes of up to 1000x1000 metres, using a grid size of one metre and ten°buckets would give 36 million nodes and ~800 million arcs. Therefore, as mentioned before: the graph, nodes and arc representations are designed with such scale in mind.

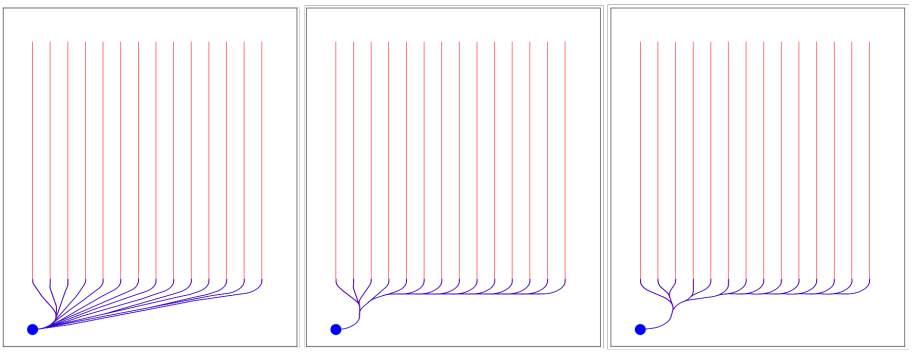


Figure 5.11: Visual representation of the generated routes, form left to right: Naive, Guided Dijkstra and lastly GreedyFLAC.

5.6.1. Primitive types & libraries

Since the software suite from Lely is developed in Java, we also use this language. One of the drawbacks is that the default data structures in this language make use of reference types. Since we are using a 32 and 64 bit representations, of which primitive types exist, it is more performant to use data structures that are compatible with the primitive types. For this, we compare the usage of three performance enhancing libraries for these primitive types. GNU Trove, fastutil and Eclipse Collection (Formerly Goldman Sachs collection).

We test the difference in performance by constructing a naive grid and running GreedyFLAC to find a shortest path to a set of random points. These grid graphs are not representative of instances that we normally use, but they are useful when comparing performance. Testing revealed that in all cases the primitive type libraries perform better than the regular types. However, there were large differences between the libraries as well. Figure 5.12 shows the computation times for these libraries when solving small instances. The GNU Trove quickly became to slow to solve the larger instances. This both due to library limitations as well as slower performance for frequently used functionalities. The differences between fastutil and Eclipse Collection were less apparent due to more similar functionalities. Nevertheless, fastutil appeared both more performant and less memory intensive in all scenarios.

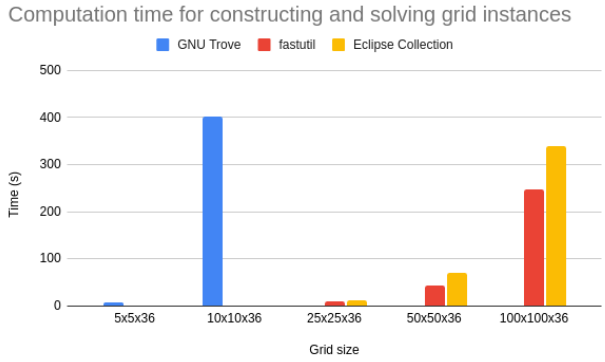


Figure 5.12: Simple comparison between the three performance enhancing libraries.

In order to better understand what parts of the computation that are problematic for performance or were implemented wrongly, we use VisualVM [37], a realtime profiling utility for inspecting both function times and total memory usage. Figure 5.13 highlights the extensive usage of primitive types. The int array is responsible for ~85% of the used memory. This way we also discovered that there are some operations that may look fast, but are substantially slower than initially assumed. One example of this is when using a Map that maps an Integer to a List. Inserting an item first requires the initialisation of the List, which is done with the operation "putIfAbsent(idx, new List)". However, it is better to use "computeIfAbsent(idx, () -> {new List}), due to only executing the closure after finding idx is absent in the map and thus initializing List only when necessary.

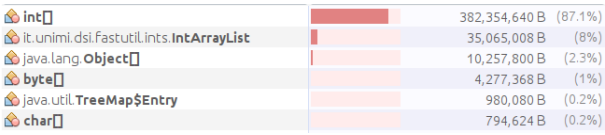


Figure 5.13: Screenshot from VisualVM when inspecting memory usage after a large graph is constructed.

5.6.2. Guided Dijkstra improvements

One of the approximation algorithms we use is a reiterative shortest path planning algorithm. The naive implementation was quite slow since we first find all shortest paths and start from the shortest one. The costs of the arcs for this shortest path are then set to 0. The algorithm then keeps finding the shortest paths until are terminals are reached. If we increase the number of terminals, the computation time drastically increases. The simple improvement made here is to rather than running Dijkstra's algorithm for each terminal, we add extra arcs in the graph from each terminal to another "virtual" node that is set as the destination. Dijkstra will always yield the shortest path and thus the path taken will have passed one of the terminals, which consequently has its arc to the virtual node removed. This effectively means that this approach has the same runtime complexity as the naive approach.

5.6.3. GreedyFLAC improvements

For implementing GreedyFLAC, we start with the original source code from the paper [39], but heavily modified. Where possible, we use primitive types and also rely on the fastutil data structures. VisualVM also revealed that the NonEmptyIntersection operation and consolidating the Fibonacci heap were responsible for most of the computation time. The complexity of GreedyFLAC is defined as $O(mlog(n)k + min(m, nk)nk^2)$ and an approximation ratio of k where k is the number of terminals, n is the number of arcs and m is the number of nodes. The improvements that we make in this section are all based on programming improvements. The complexity stays identical, the relative performance is increased.

Intersections

GreedyFLAC performs a sort of flow simulation, starting from each of the terminal nodes. When two of these expansions collide, the expansion that originates from the collision is accelerated, depending on how many terminal nodes have reached the node. The intersection is used to test if the flow simulation performed collides. The original algorithm keeps a set per node to represent each terminal. Profiling with VisualVM showed that this intersection method was responsible for half of the processing time during GreedyFLAC.

The performance improvement comes from the fact that there is always a defined list of terminal nodes. Therefore, rather than keeping a hash set of terminals per node, we keep a binary array of a set size, where each bit represents one terminal. The collision is then simply tested by performing an AND operation and verifying if the result is 0. Complexity wise, this change would yield no improvement, but the constant time operation is accelerated in such a significant manner that the NonEmptyIntersection is no longer a large contributor to the computation time.

Heap implementation

GreedyFLAC normally uses a Fibonacci heap due to the $O(1)$ complexity of the decrease-key operation. In literature, this heap is often used to prove that the complexity of an algorithm has a certain bound. In practice, the Fibonacci heap has such a large constant cost for said operation that the actual performance is worse off [7]. An attempt was made to use the Pairing heap, but testing showed that this yielded no measurable increase in performance and sometimes even a decrease. Further investigation revealed that the delete-min instruction is responsible for a large portion of the compute times of GreedyFLAC and that both heap implementations required near identical times for the operation. Anecdotally speaking: if the algorithm requires 25 seconds to run, this instruction is responsible for 13 seconds of computation. Since the complexity of the Fibonacci heap is more favourable ($O(1)$ vs $O(log(n))$ we choose to keep the Fibonacci heap over the Pairing heap.

5.6.4. Other improvements

This section is to address design decisions that result in marginal performance gains, but do not warrant a specific section.

- **Java Topology Suite** or JTS is used to perform the polygon based calculations. Testing collisions and finding the points near the grid points are a good example. This library uses well programmed polygon representations that make use of spatial indices in order to accelerate computations.
- **JVM tuning** is also used to improve performance. The Java Virtual Machine is able to be tuned extensively [33] [13]. We made sure the heap allocations were of sufficient size and also used more performance accelerating flags [36].
- **JIT warmup** is also used during benchmarking. We first run a couple of instances before performing the actual tests.

5.6.5. Performance results

In order to show the performance differences achieved by programming improvements, we compare the original code used for the introduction of GreedyFLAC [39] with the improved version. We create simple graphs by creating a NxN grid with B buckets. The arcs are constructed in a 3x3 grid around each node to each reachable node. Note that this approach creates a approximately ~322 arcs per node, this is a larger amount of arcs than we would normally use. We compare runtime and memory usage. Both algorithms are executed on the same graph structure to create a fair comparison. Lastly, we also made sure the received solutions were identical to make sure the implementations are correct.

Memory usage

We first look at the memory used before we run the approximation algorithm. One issue is that the implementations use Java as language. This has the drawback that measuring memory usage is difficult due to the unpredictable garbage collector. We therefore measure the size of the entire heap after graph construction and force a garbage collection. The memory usage during execution is also difficult to measure. Java will use more memory to boost speed if memory is available, which makes direct comparisons less trustworthy. Nevertheless, since the differences are of such an order of magnitude, it is important to highlight the difference to show how these improvements move the feasibility of execution on large graphs from impossible to very reasonable. Figure 5.14 shows the log scale of memory used. Note that the default version quickly reached high values and could not be tested further.

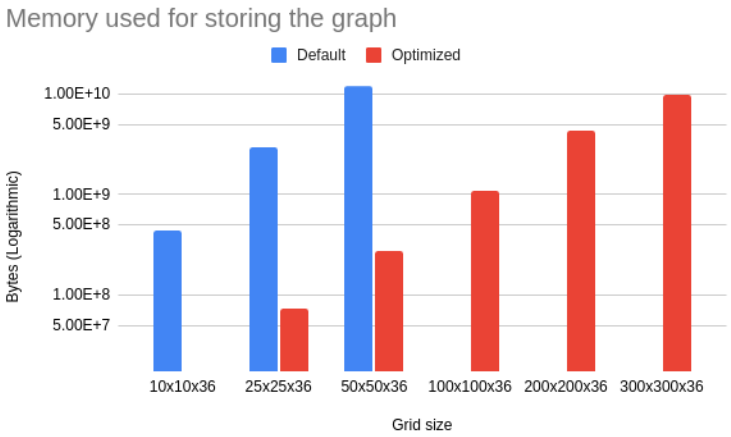


Figure 5.14: Chart that shows the difference in memory usage between the default graph implementation and the custom made version when storing a grid based graph.

Execution time

Next, we inspect the difference in execution times. For this, we first compare the time it takes to construct the graph with the time it takes to execute the GreedyFLAC approximation. For this, we create identical problem instances with a set carefully placed terminals. The starting point is placed in the left lower

corner and the terminals are placed in the middle and three other corners (this is a difficult scenario). We also compare performance effects of increasing the amount of terminals. In order to create good results, we use JMH to eliminate performance differences due to JIT warmup and we make sure the computer has sufficient memory available.

We again stopped the default implementation after the 50x50x36 instance due to the memory constraints. The results in figure 5.15 show that the execution time also improved with an average factor of 15. We also see that the improved version shows signs of significant slowdown when at the 200x200x36 instance. The 300x300x36 instance is solvable, but runs into memory constraints.

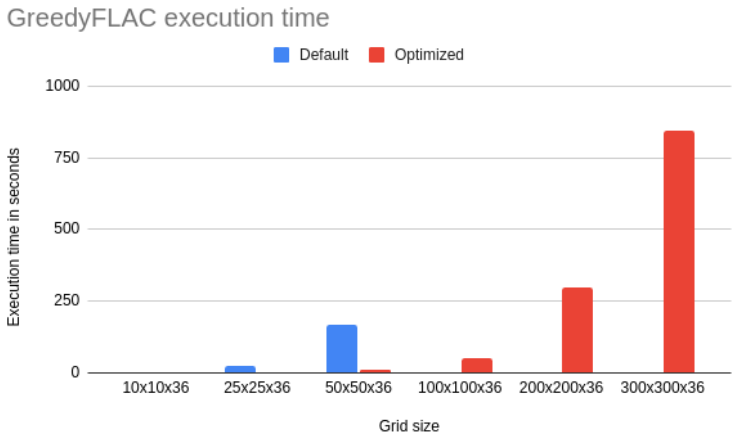


Figure 5.15: Chart that shows the difference in execution times of GreedyFLAC between the default implementation and the optimized version.

5.7. Jojo Plan

In the previous sections we managed to represent the problem in a graph and provide sufficient performance improvements in order to solve large instances and create a tree that represents the path. We must now create an instruction manual for the Jojo from this tree. In order to transform from a tree to an instruction we move through the tree using a left first iterator. It behaves similar to a depth first iterator, but has a preference for the most left path when approaching a split. Using this ordering makes it so the vehicle moves along the tree with the least amount of movement.

The instructions that are generated using the left first iterator are stored using the following five instruction types:

- **DriveToStart** is used as placeholder to represent the vehicle docking and driving to the starting location of the created plan. This instruction uses the regular path follower.
- **DriveForward** is accompanied with a path that instructs the vehicle to move along this path. This instruction uses the regular path follower with a reduced speed.
- **DriveBackward** is the same as DriveForward but the vehicle now moves backwards. This instruction uses the regular path follower with a reduced speed.
- **DistributeForward** is accompanied either with a path or start and end location to represent the stroke that manure needs to be applied on. This instruction uses the pressurized path follower.
- **DistributeBackward** is the same as DistributeForward, but now for the reversed motion. This instruction uses the pressurized path follower.

Each of the instructions and accompanied information is then stored in a large JSON file. In order to execute the program, the JSON is loaded into a queue and each action is executed after the other. As a safeguard, a check is made whether the vehicle is at the right location for the action to be executed. For the DriveToStart, DriveForward and DriveBackward instructions this means that the start position has to be almost equal to the actual position, whereas for DistributeForward and DistributeBackward actions allow for the vehicle to be positioned on anywhere on the path. The latter allows for the vehicle to interrupt the distribute actions, which may be necessary.

5.8. Limitations

The solution presented using a graph based approach also comes with limitations, this section aims to highlight and discuss the shortcomings. We highlight the shortcomings and provide potential solutions when possible.

- **Approximation** algorithms make mistakes. Figure 5.16 shows both the Dijkstra approach and GreedyFLAC making an obvious mistake for this problem instance. In both cases the path starting from the blue marker should have steered more to the left in order to better reach the first stroke. The GreedyFLAC approach appears to make a small correction to the left, causing only one stroke to attach in a weird way. The Dijkstra approach lacks this manoeuvre and fails to properly connect the two left most strokes. We can always argue that the optimal solution does not differ that much in cost to the given solutions, but it is still a shortcoming that we must address. GreedyFLAC appears susceptible to making mistakes when a cluster of terminals is near each other. These clusters will prefer to reach the terminal directly over adding a small path deviation to add a few more terminals.
- **Graph misalignment** which is a result of translating polygons of infinite precision to a grid. Field shapes can exist that are difficult to align with the grid, which can lead to an exclusion of possible paths. Say for instance a narrow corridor where the vehicle can just fit through, if the grid is not perfectly aligned with the corridor it will not be represented as a valid path. A possible solution for this would be to detect these scenarios and then create special nodes in the graph, that move the vehicle along this precise path.
- **Maximum hose length** is the most obvious limitation. The approach does not keep track of the maximum hose length and whether the solution violates the constraint. As mentioned before, the Steiner tree problem is already difficult to solve and adding this constraint makes it even more difficult. There are limited solutions found in literature for the undirected instance [17]. Most of them focus on the topic of wiring on microchips [32]. These solvers all have in common that they are designed to work for small instances, not the large graphs that are used in this approach. As for now, an alternative solver is deemed infeasible. Another argument one could make, is that the likelihood of violating this constraint is not as high when the docking point is placed in a logical location and the fields are of reasonable sizes. The reality is that the placement of the attachment points is also decided beforehand and may be best determined by using this algorithm to find the best placements. The vehicle is also tailored to each farm it is placed on, this means that the hose length is also decided upon during installation.

  Nevertheless, an attempt is made to provide multiple solutions for these edge cases. We start with a set of recommendations for the user to keep the maximum length of each stroke within a reasonable size. This may require the strokes to not be orientated to provide the longest strokes, but rather in such a manner that they are not too long. Next, to keeping the strokes a reasonable size, the fields and attachment point locations should also be designed in such a way that long paths are avoided.

  Another approach one can use is to iteratively run the naive Dijkstra algorithm, where if a generated path exceeds the maximum hose constraint, the intersection between the new path and the partial solution can be viewed as too large. One can then modify the costs in the search graph in such a way that the costs of the arcs in the partial solution are not all set to 0. An approach could be to set the first half of this intersection to 0 and the rest of the partial solution to a very high value. Rerunning the shortest path algorithm will then yield a result with a smaller intersection between the new path and the partial solution, in effect reducing the distance to the root, if the distance is still too much, the same steps can be repeated to find an even shorter path until the distance constraint is satisfied.

  The same approach can also be used to 'repair' solutions created by GreedyFLAC, but as opposed to fixing a violation during execution, this approach must do a retroactive repair. If a solution violates the maximum hose constraint the path has to be analysed to find the reason the path is deviating from the shortest path to a terminal which has the violated the constraint. The solution is effectively to decrease the depth of the tree by better balancing the tree. This can be done either automatically or by relying on manual intervention.

- **Branching from strokes** is not possible. In this scenario, the vehicle first moves forward over a stroke, then drives to the next stroke and completely drives over it, after which it drives backwards and finishes the first stroke. Due to the design of the graph, the strokes act as a sink and thus do not provide a method to extend the path after reaching a stroke. There may be a possibility to implement this with more arcs and nodes, but the maximum hose length limitation mentioned earlier, that has difficulties with long paths, led to the design decision not to solve this issue.
- **Maximum size**, even with the performance improvements is still a limitation. Using grid resolutions in the range of centimetres or having extremely large instances could be desired, but are not feasible. We are effectively on the edge of feasibility. Should the dimensions be doubled in size again, we would not be able to calculate a solution within a reasonable amount of time using a conventional computer.

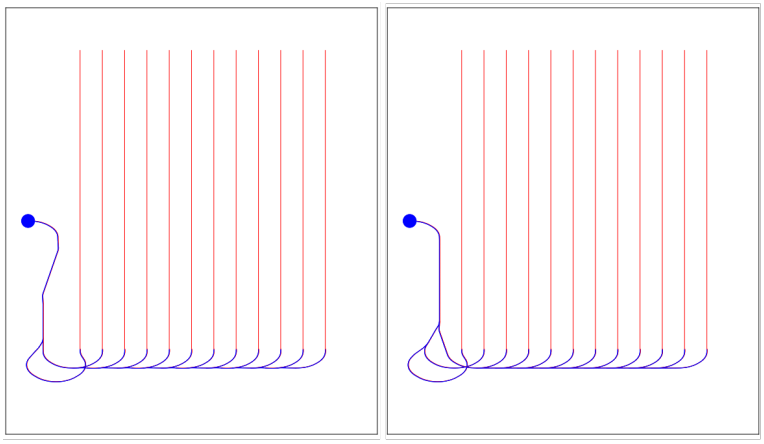


Figure 5.16: Visual representation of the solutions created by GreedyFLAC on the left and Guided Dijkstra on the right.

5.9. Conclusion

In this chapter, we presented an extensive explanation of how the simplified graph approach is transformed to be applicable for the general Jojo problem. We first look into how the lattices are generated and how they are used to construct the directed graph. We then look at how each aspect of the problem is incorporated into the graph. Strokes can be approached from both ends and be split from the middle, while disconnecting the strokes from the grid. We introduce the different approximation algorithms that can be used and we provide an analysis of their workings.
In this chapter, we also learn that in order to achieve usable performance, extra effort has to be put into constructing efficient software. Therefore, we also introduce multiple solutions for speeding up the algorithm. The node representation is designed to be fast and the memory usage is also optimised. Multiple attempts were made to improve the speed of GreedyFLAC, the switch to primitive types proved to be most effective. Switching the underlying data structures for others yielded no substantial difference. These performance improvements are also highlighted and discussed. In the end we managed to accelerate the computations in such a manner that we moved this limitation beyond the field dimensions that we aim to use.
In the next chapter we will look at how well the algorithm behaves, after which we can discuss whether the proposed solution will fully answer the sub question of: "what is a supposed good method to structure and create a solution?". For now the we can state that with the performed benchmarks on each stage of the algorithm, that the full solution will probably be sufficient.

6

Algorithm verification & testing

This chapter aims to evaluate and compare the quality of the generated Jojo solutions created using the algorithm as explained in the previous chapter. This chapter is divided in two parts, where in the first we investigate the three approximation algorithms and perform parameter optimisation. For this we use a large number of small test cases, which enables us to perform numerical analysis. While in the second part of this chapter, with the information we gathered in the first part, we use larger, more representative instances. These are based on the fields available to us for future real world testing. For these instances, we provide a per instance comparison and discussion, where we look at path length, simulation performance and integrate some user opinions.

6.1. Parameters

The graph construction algorithm and consequently the lattice generation have important variables that can be tuned. The parameters and their respective default values are given in Table 6.1. The default values are initially chosen based on an educated guess, but are to be optimised with the experiments in this section. For this, we specify a range of suitable variables in order to compare. Note that we do not disallow grid decoupling as the generated solutions are regarded as unsuitable as explained in chapter 5.

6.2. Generated instance experiments

In this section we will first define a test set of smaller auto generated instance, these are designed to be solvable within a reasonable amount of time. We then provide a closer inspection of the different results that the algorithms provide and try to explain the difference in the results. This section continues with a set of experiments that run the algorithms on a large collection of instances while we vary the values of the parameters. We conclude with an analysis of the results.

Parameter	**Data Type**	**Default**	**Values**
Resolution (m)	Float	1.0	{0.5, 1.0, 1.5, 2.0}
Turning radius (m)	Float	3.6	[2.55-5.0]
Lattice dimensions	Integer	7x7	{3, 5, 7, 9, 11, 13, 15}
Angle buckets	Integer	36	{8, 12, 16, 20, 24, 28, 32, 36}
Cone width (degrees)	Float	60.0	[45.0-90.0]
Grid decoupling	Boolean	True	[True]
Mid insertion	Boolean	False	[False, True]
Cost function	Function	Distance	[Distance, Travel time]

Table 6.1: Table with the default parameters

6.2.1. Instance creation

In order to both test and verify the algorithm we use a collection of hand made instances that encounter edge cases and we make use of randomly generated instances. The initial was mostly used during development and is now only used to verify the results of the algorithms in order to make sure no implementation faults exist. The random instances are created by generating a shape and placing random strokes within this shape and making sure they are all reachable. We also have the ability to create random rectangles within the outer boundary, which are used to create blocks of strokes as opposed to the random variant. Figure 6.1 shows a small collection of example instances with an accompanied solution generated by one of the algorithms.

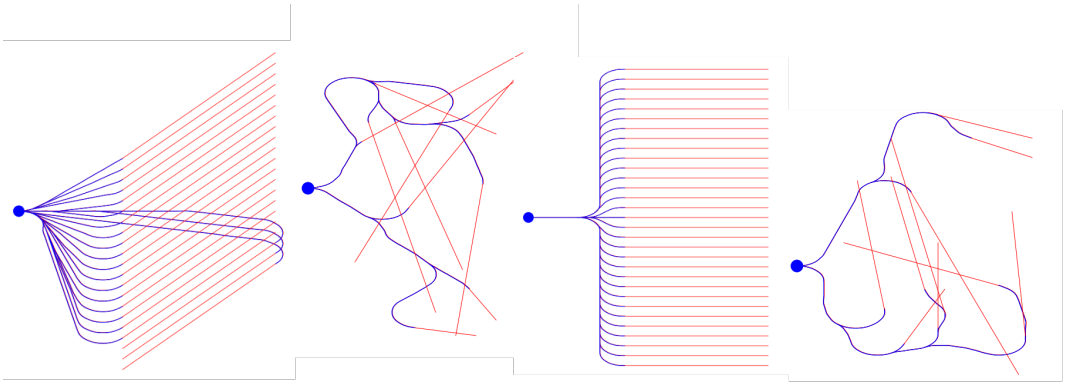


Figure 6.1: A collection of small instances with a potential solution.

6.2.2. Experiment 1: Approximation algorithms

In this first experiment we aim to understand the performance characteristics of the three approximation algorithms. The assumption is that GreedyFLAC yields the best quality and performance. In order to find out, we run all three algorithms on 100 generated instances using the default parameters as specified in Table 6.1 and using four fields of the sizes 50x50, 100x100, 150x150 and 200x200 with 10 to 20 strokes. Each combination is represented equally for this test. Since each algorithm is deterministic, we only have to run the solvers once per instance.

Results & Discussion

We randomly selected ten results to better show the difference in computation times. Figure 6.2 shows the difference in path length between the naive and other approximation algorithms. The Guided Dijkstra and GreedyFLAC are both comparable, but GreedyFLAC always created an as good as, or better solution over the Guided Dijkstra.

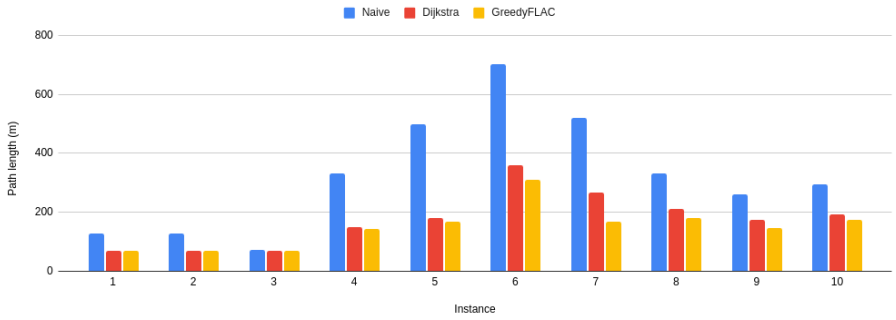


Figure 6.2

The aggregated results are presented as an average score over all 100 and shown in Figure 6.3. The story for the first ten instances can be retold here. For now the Dijkstra and GreedyFLAC algorithms are almost identical in performance.

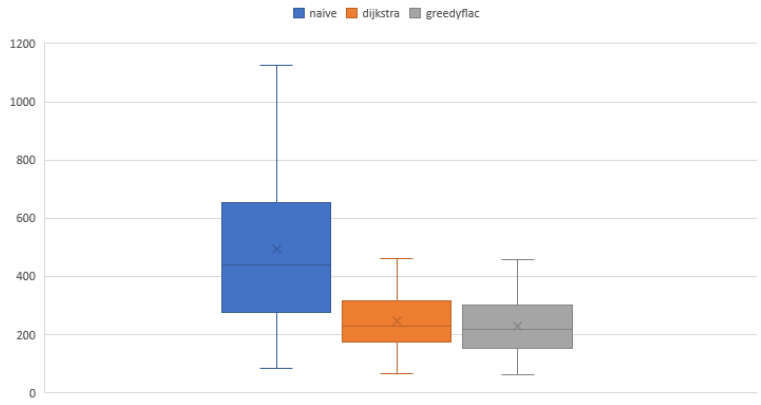


Figure 6.3

6.2.3. Experiment 2: Lattice optimization

This second experiment is aimed at optimising the three variables: Resolution, Lattice dimension and Angle buckets. These three variables are tied together in multiple ways. For instance, increasing the resolution of the lattice may give better results due to having more points represented in the grid, but it may also lower results due to covering less surface with each lattice. Increasing the resolution may even require more dimensions in order to make sure all points are reachable. The same analogy can be made with the number of buckets, a reduction nets less arcs in total, but may also limit the graph in such a way that not all possible vehicle movements are represented sufficiently. All three variables essentially influence the size and quality of the search graph, which in turn has consequences for performance. As such we try to optimize these variables by performing an exhaustive search. We use the default parameters as specified in Table6.1 and the value ranges of the three variables to measure the impact. Since this exhaustive search is time consuming due to the 224 possible combinations we only use ten random instances on a 50x50 grid for the comparison. Due to the results of the previous experiment, we only use the GreedyFLAC approximation algorithms. We also measure the time it takes to both construct the graph and find the solution. This is due to the uncertainty the graph size has on approximation performance.

Results & Discussion

The number of variables that we are representing is relatively large. Therefore, in order to show the results, we plot the three variables on the respective XYZ axis and use color and size to represent computation time and size to represent the score. Figure 6.4 shows the results when using GreedyFLAC using a 3d plot. The raw data is given in appendix 9 in tables 9.1 and 9.2. As expected, the general trend is that increasing the values results in both higher compute times and a better score. However, careful inspection shows that the values with a high lattice dimensionality and resolution, but a lower amount of bucket, still performs reasonable.

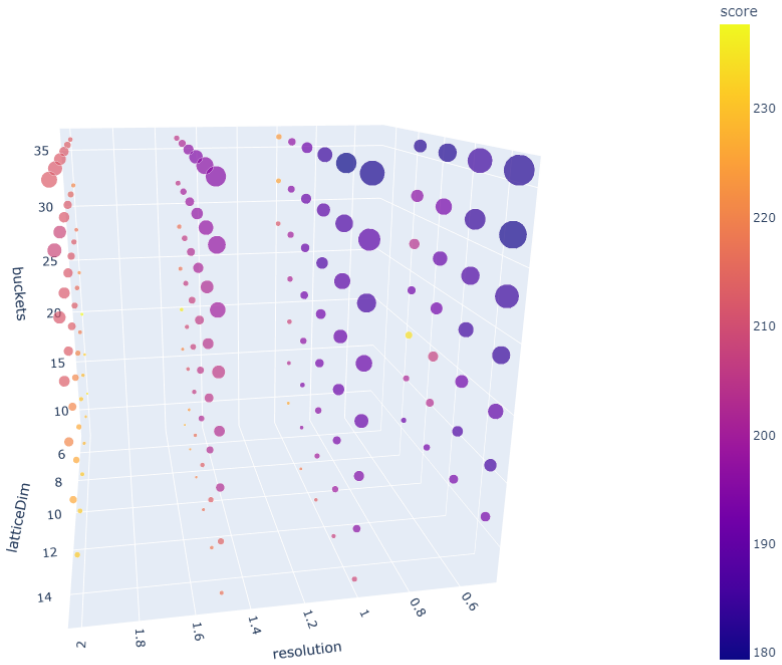


Figure 6.4: Visualization of the results in a 3D Scatterplot when approximating with GreedyFLAC. The score is visualized using colors and the computation time is represented with the size of the dots.

Say we specifically look at the best performing and slowest configuration of: (R=0.5, D=15, B=36) and a neighbouring configuration of: (R=0.5, D=15, B=16). The average score lowers from ~182 to ~185 meters, but the total computation time is also lowered from 376 to 64 seconds. Also interesting to note is when we compare the outer scenarios of (R=2.0, D=5, B=8) with (R=0.25, D=19, B=36), we get a performance increase of ~160x (410 seconds to 2.5), but only gain a distance improvement of ~20%. A potential conclusion one can make is that an increase in values is tied to an increase in computation times, but not necessarily tied to a proportionate increase in quality. Figure 6.5 shows the connection between lattice size and computation times. The colour filter based on the resolution exposes the linear scaling trend in computation times compared to the size. If we look at the lattice size versus score results using a fixed resolution shown in figure 6.6, we see that larger lattice sizes do trend towards a better solution, but with diminishing returns. Since performance gains are relatively low and time penalties high it is wise to pick lattice parameters that keep the size to a minimum.

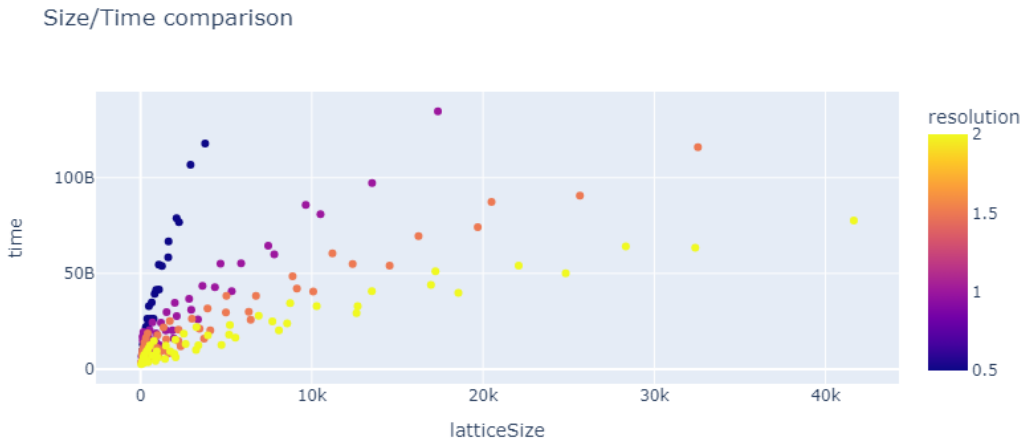


Figure 6.5: Visualization of lattice size compared to calculation times.

Future implementation may need to look at better methods of generating lattices as such as proposed by Bergman Et al. [3], where the set of best combinations are programmatically determined. For now, a reasonable set of parameters appear to be (R=1.0, D=9, B=16), the same as shown in figure 5.3b, the score is ~194 meters and can be calculated in ~10 seconds.

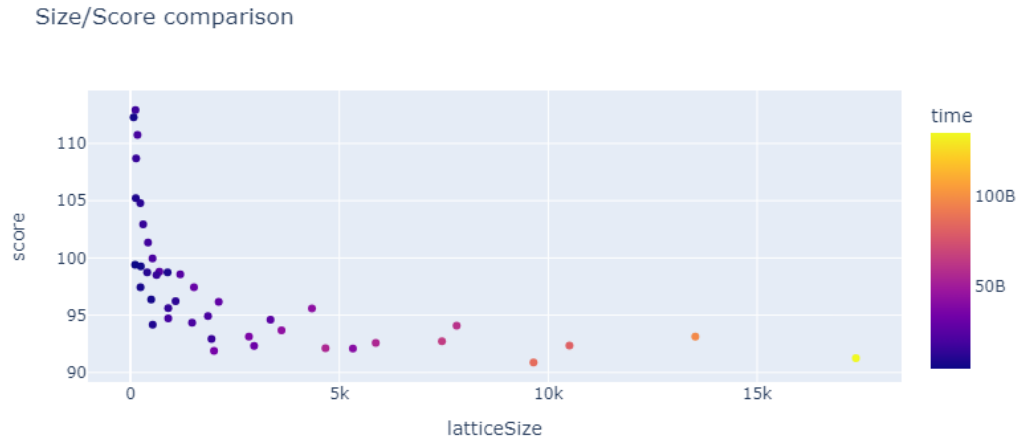


Figure 6.6: Visualization of lattice size compared to score for the resolution of 1.0.

6.2.4. Experiment 3: Stroke count

This third experiment is aimed at understanding the performance impact of having more available strokes. For this, we generate an increasing amount of stokes on the same 50x100 field and measure the performance. The assumption is that more stroke means a larger computing time for all three algorithms, but GreedyFLAC will experience the least relative impact as opposed to the other two algorithms who will probably scale linearly with the number of strokes. This experiment only focuses on the three approximation algorithms and therefore only measures the time it takes to run those and not the time it takes to construct the graph as performance impact of adding strokes is negligible.

Results & Discussion

The results for this exceed the initial guess of performance scaling. Initially, we did not expect such a large difference as we expected GreedyFLAC to also take a performance penalty, but the results in figure

6.7 show how much the number of terminals changes the performance of each approximation. The naive and Dijkstra approach perform an iteration per node where GreedyFLAC performs the calculation in one go and thus gains performance in that respect. The performance differences are of such an order of magnitude that it can be argued that only GreedyFLAC is suitable for these larger instances. However, it must be noted that the Dijkstra approach can be made much faster by using only one iteration to calculate the paths to each terminal.

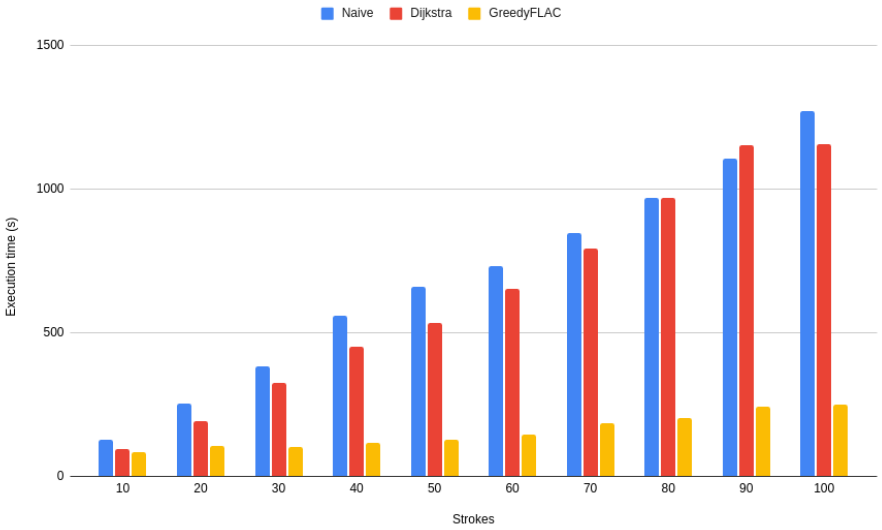


Figure 6.7: Execution times per approximation algorithm of increasingly more strokes added to the same 50x50 grid.

6.2.5. Conclusion

In this section regarding the tests on generated instances where we compare the three approximation algorithms, we arrive at the conclusion that GreedyFLAC gives better performance both time and quality wise, especially on instances with a large number of strokes. During parameter optimisation, we also notice that it is difficult to optimise the lattices due to the many variables that influence the quality and size of said lattices. In general though the combination of chosen parameters should result in a lattice that is both of sufficient size in order for all points in the grid to be reachable and not too large to prevent high compute times. The quality of the final result is influenced less by the parameters than initially thought, while performance is heavily dependant. Perhaps the two suitable conclusions to this chapter are: lattice generation should be made smarter and GreedyFLAC as approximation algorithm is deemed the most suitable due to slightly better results over Dijkstra, but especially due to the performance difference when confronted with many terminals (strokes). The small instances gave a good insight in how results are influenced. In order to make a final assessment of the quality we continue in the next section with more representative instances.

6.3. Real world instance experiments

In this section of the chapter, we will be conducting experiments and discussing representative scenarios. We will also be using what we learned in the previous part of this chapter. Namely the parameters that we optimised and the conclusion that GreedyFLAC delivers the best results. Also, rather than only comparing the path lengths, we will be comparing the simulation performance of these instances as well. For these experiments we will be using the same field the vehicle is normally operating on. Figure 6.8 shows the three fields with their solutions using GreedyFLAC. The other two approximations did not yield a solution within a reasonable amount of time. Note that in the real world, the left most field is the only one currently equipped with an attachment point. We also decided to use different stroke angles for each field in order to test the algorithm behaviour under different angles. Figure 6.10 shows the behaviour near the pits.

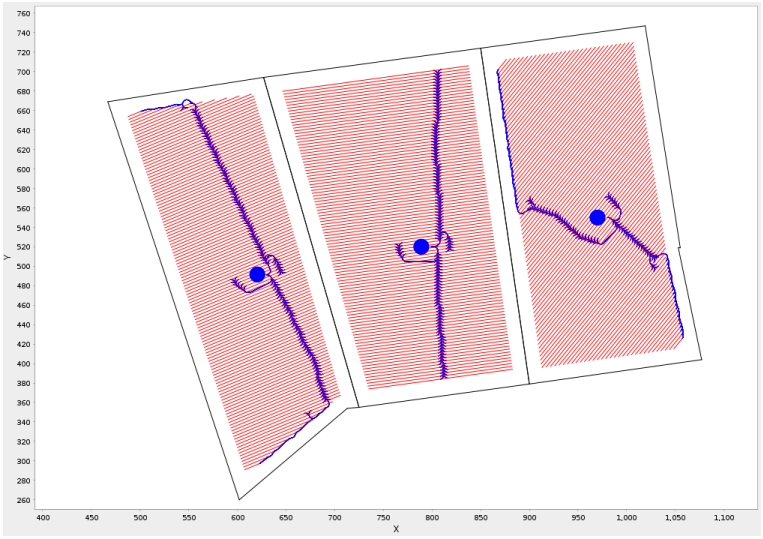


Figure 6.8: Here the solution of the mid instance is pictured when zoomed in to be near the docking pit.

6.3.1. Simulations

The simulation makes use of different settings per part of the Jojo plan. The vehicle velocity when driving over the strokes is set to $0.8m/s$, while in any other case the velocity is set to $0.5m/s$. For the simulations, we also assume the vehicle never runs out of power or manure. As shown in the table 6.2, the lengths of both the drive and distribute segments are known, from which we can deduce that the simulations require a substantial amount of time to complete since we know vehicle velocity. Nevertheless, this guess would be an upper bound, since the vehicle controller as described in chapter 3 slows down for corners and uses slow accelerations. Also note that during each step in the simulation, checks are run to verify if the vehicle is actually doing what it needs to do.

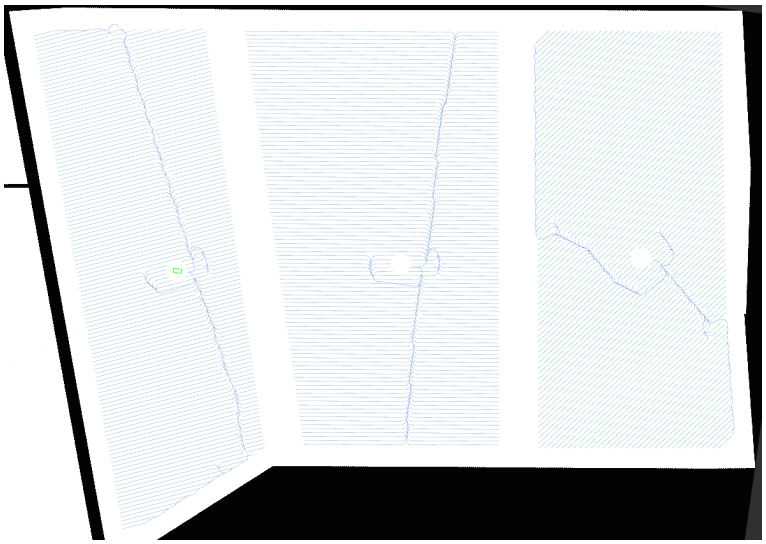


Figure 6.9: RVIZ2 visualization of the plans in the simulation.

6.3.2. Results & Discussion

Since the simulation takes a large amount of time to complete, we only run each instance once. The results are given in table 6.2 and show how the length of the paths influence the simulation times. Inspecting the generated paths as shown in figures 6.8 and 6.10 we see that the solutions make heavy use of the mid stroke insertions. Perhaps the pit being positioned in the middle of the field is the result of this. We also see in the pit area both how well the algorithm performs in finding a solution in such a

Field	Size (m2)	Compute time (s)	Path length (m)	Drive length (m)	Distribute length (m)	Simulation time[1](h)
Left	59,402	43.5	31,458	4,257	27,201	17
Middle	70,056	24.3	38,003	3,723	34,280	22
Right	60,255	60.8	32,202	2,911	29,291	19

[1] Simulation may have introduced a slowdown during execution

Table 6.2: Table containing both heuristic and simulated results

situation, but also the shortcomings. The two branches on the left and right, which cover the strokes that are cut in order to avoid the pit, branch of from the start of a mid stroke insertion. A better solution would have branched off from the ends of such a mid stroke insertion.

During the simulation, we also noticed that the vehicle almost perfectly followed the path, the simulation can perhaps be augmented to sometimes make localization errors. The simulation also did not appear to run in real time due to computation power required, making it difficult to estimate execution time. Since there is a strong connection between the path lengths and the execution times, it can be argued that the times can be approximated in a faster way, rather than simulating each instance. This approximation may even use the acceleration and path curvature as input to further increase its accuracy. Nevertheless, the simulation was invaluable during both development and gaining confidence in real world testing.

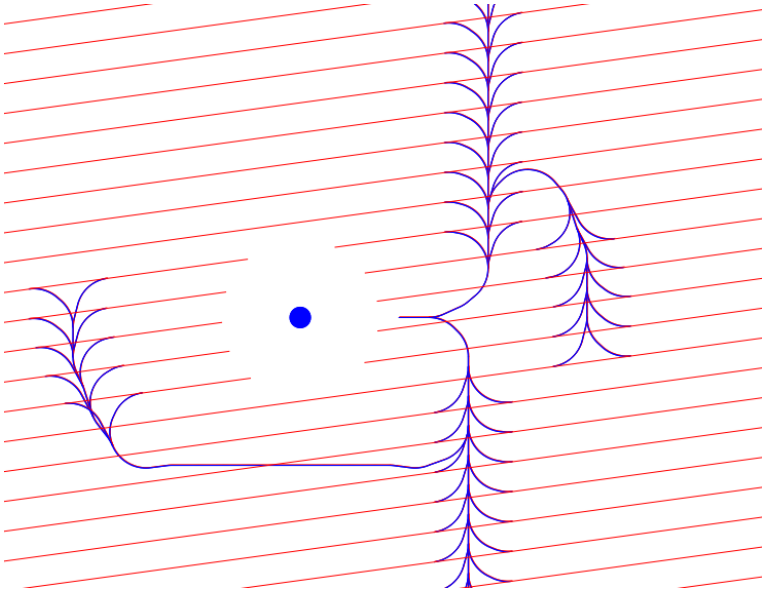


Figure 6.10: Visual representation of the fields and strokes for a solution on a real world instance at the middle field.

Lastly, we must also address an observed shortcoming of the planned path. We already discussed the limitation of the vehicle not being able to continue the path either from the end or middle of a stroke. Figure 6.10 shows how both the left and right branches that are forked from the vertical segments require the vehicle to drive back to the start of a mid intersection, rather than branch of from such an intersection. Figure 6.11 highlights this even better. The path could be improved by merging the branch to the right with the end of a mid intersection. For the first two test fields we count a total of five of these occurrences, for the last field we also count five. We can argue that in this case the extra driven distance is not substantial, but we must assume that there are instances where this behaviour is more frequent.

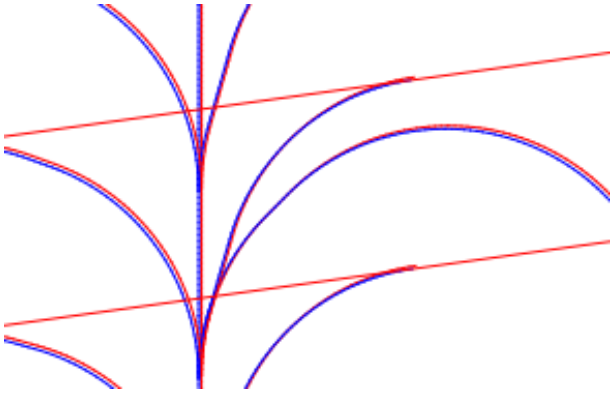


Figure 6.11: Visual representation of the fields and strokes for a solution on a real world instance at the middle field, zoomed in at a suboptimal segment.

6.4. Conclusion

In this chapter we started with testing and optimising the algorithm as specified in the previous chapter. We learn that the lattice parameters originally assumed to be chosen well, were actually not as performant as thought. The better variables were found by an exhaustive search, which showed the diminishing returns of solution quality at the cost of large computation times. We also investigate the difference between the three approximation algorithms and learn that the modified Dijkstra performed quite well on the simple instances, but lost competitiveness to GreedyFLAC when increasing the number of strokes resulted in a drastic increase in compute time.

In the second part of this chapter we continue testing, but on the aforementioned real world scenarios. For this, we use the only field currently available to us and show that the algorithm indeed manages to find a sufficient solution to the three instances within a very reasonable amount of time. We then simulated the solutions to these three fields, although they take a long time to complete in the current form, they show that the solutions are feasible for the vehicle to execute.

Considering that during the initial phase of development in this thesis the assumption was made that even generating a suboptimal solution is difficult, all the solutions that we managed to get for both random and real world scenarios appear both visually and numerically sound with minimal suboptimalities, especially when the path lengths are dwarfed by the total driving distance. As such, it is possible to argue that the answer to the sub question of: "how does the solution compare when using different farm layouts?" can be partially answered with: performing above initial expectation, but not completely due to the lack of comparisons as a result of both the novelty of the problem and the proposed solution. Whereas the sub question of: "does the solution scale sufficiently to allow for real world scenarios to be solved within a reasonable amount of time?" can be answered with a yes as we have shown in section 6.3

Another interesting fact can be observed in table 6.2. The size of the calculated path is dwarfed by the size of the distribute segments. Perhaps this opens the possibility for allowing worse path approximations, but allow for the maximum hose constraint to be adhered.

7 Experiments

In this chapter, we will be conducting real world experiments using the Jojo plans that we are able to generate using the aforementioned methods. The experiments and simulations that we conducted in the previous chapter are sufficient to show that the generated solutions perform as expected. This chapter takes it a step further by performing a couple of real world tests intended to verify whether the solutions are also feasible in the real world. In order to save time, we conduct a few experiments that perform the difficult segments of the solutions, rather than the entire solution, since executing a route that would take at minimum 16 hours to complete is infeasible.

7.1. Parameters

For these experiments we use the variables listed in table 7.1 for generating the Jojo plans. These are the variables we obtained during the variable optimisation performed in the previous chapter.

Parameter	**Data Type**	**Value**
Resolution (m)	Float	1.0
Turning radius (m)	Float	3.6
Lattice dimensions	Integer	9x9
Angle buckets	Integer	16
Cone width (degrees)	Float	60.0
Grid decoupling	Boolean	True
Mid insertion	Boolean	True
Cost function	Function	Distance

Table 7.1: Table with the parameters used for the real world experiments

7.2. Experiment 1: Mid field intersections

This first test will investigate whether the vehicle is able to perform execute the mid field intersections. This scenario appears to happen quite often and can also be deemed difficult for the vehicle to execute, due to the sharp turns. Therefore these intersections warrant an experiment to test whether the intersections in the current form are suitable. Figure 7.1 shows the constructed instance with the associated solution. The route is executed by always preferring the left most branch before backtracking, as explained in the previous chapter. This solution has a drive length of ~196 meters, a distribute length of ~480 meters and the tightest corners have a radius of 4.0 meters. We will be testing the vehicle by driving a velocity of 0.5 m/s and look ahead distance of 2.0 meters on the drive segments and a velocity of 0.8 m/s and a look ahead of 4.0 meters on the distribute segments. Since we are interested in whether the vehicle is able to perform these intersections, we need to record the path the vehicle follows. We do this by logging the GNSS data on 1 second intervals. The second point of interest is whether the attached hose experiences no damage during testing. An operator is continuously checking if the hose

experiences no straining. If so the operator will halt the vehicle. This experiment will be repeated 3 times.

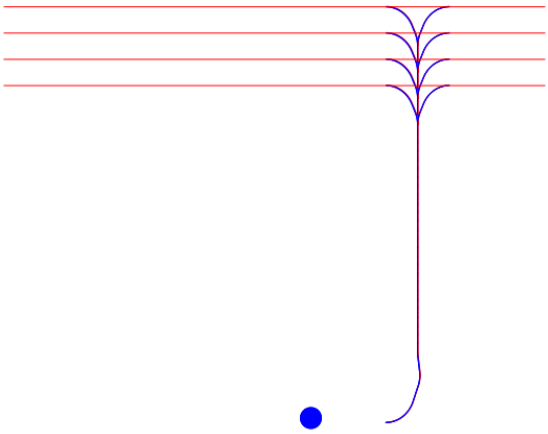


Figure 7.1: Visualisation of the generated solution containing multiple mid intersections in a row.

7.2.1. Results & Discussion

Figure 7.2: Picture of compressed grass after the vehicle drove over multiple times.

We specifically choose a patch of grass where the grass was quite high in order to visually show the results. Figure 7.2 shows the compressed grass after the mid intersections. The grass is highly compressed in the areas where the vehicle drove over 12 times due to the multiple iterations during the experiment. The image appears to show good results, but for further investigation we will look at the GNSS points. Figure 7.3 shows these traces. The results are quite promising: the vehicle can actually execute the route and the vehicle follows the route quite well. The figure also shows the 3 traces laid over each other: the repeated paths are almost identical. This is the result of both the route being valid to drive and the path following algorithm working well. The largest area of improvement appears to be related to the path follower experiencing understeer during corners, especially at higher velocities. This can probably be reduced by decreasing the look ahead distance, but as mentioned in chapter 3 this may introduce undesirably strong steering behaviour on straight segments.

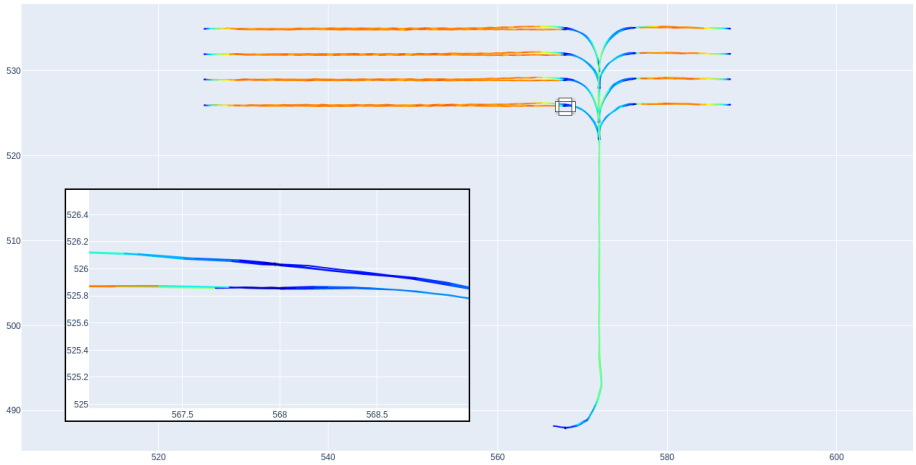


Figure 7.3: Visualisation of the 3 recorded GNSS series. The colors indicate the velocity.

We also executed a run with a 0.4 m/s increase in velocity, figure 7.4 shows this behaviour. The turning segments show signs of oscillation, a common issue with RPP due to a too small lookahead distance. The distribute lines behave well due to the further lookahead distance of 4.0 meters as opposed to the 2 meters during the drive segments. If the end user would want the vehicle to move faster, the best approach would be to keep the driving velocities as is, but allow for the vehicle to go faster on the distribute segments. However, perhaps the best solution would be to better tune the variable look ahead functionalities as explained in a previous chapter in section 3.2.4.

Nevertheless, the results show that the vehicle was able to repeatedly execute the solution, showing that a commonly used movement pattern in the solution can be executed well.

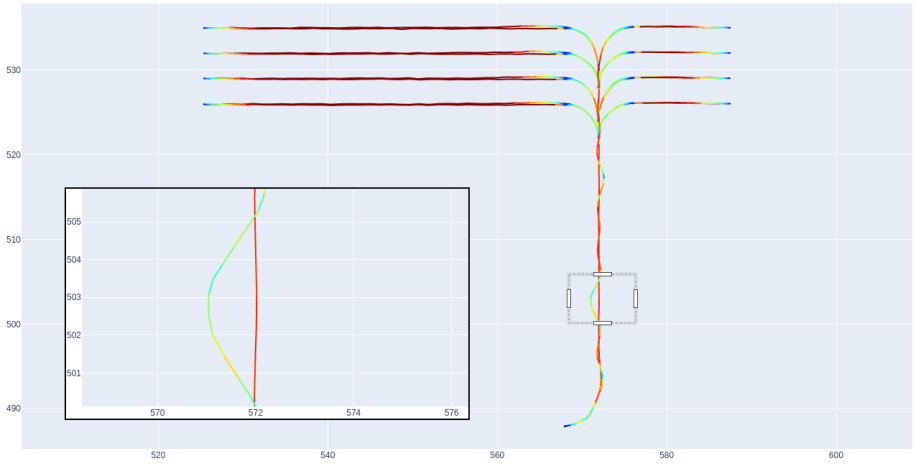


Figure 7.4: Visualisation of the GNSS series of a run with 0.9 m/s velocity on the drive segments and 1.2m/s on the distribute segments.

If we now inspect the velocity graph shown in figure 7.5, we notice a few things:

- The vehicle indeed drove the configured speeds of 0.5 m/s in the steering segments and 0.8 m/s in the distribute segments.
- The repeatability of the execution is also confirmed by the almost identical velocities.
- The total execution time was about 1600 seconds or 27 minutes.
- The maximum velocity is not achieved during the steering behaviour, as configured.
- The vehicle spends a considerate amount of time at the velocity of 0 between each step of the Jojo plan. On average this means that there is about ~300 seconds lost to standing still between actions. Out of scope for this research, but perhaps a good focus for future improvement.

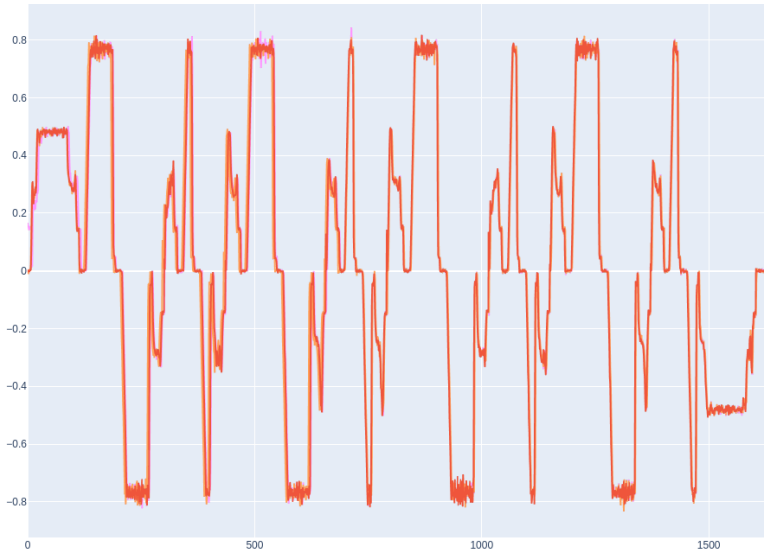


Figure 7.5: Visualisation of the 3 recorded velocities over time. The vertical axis is specified in M/S and the horizontal axis is in seconds.

Lastly, we also investigated whether the hose managed to stay intact, which appears to be the case. During execution the vehicle did not have to be stopped. Afterwards the hose was unrolled once more to visually inspect the hose for any signs of damage. Even though the hose was not damages in the testing, we do recommend that the current minimum turning radius should be increased. Partly because we did observe minor straining in the corners, but more as a preventative measure for dealing with uneven fields. During manual testing, we noticed that when the vehicle autonomously drives over uneven ground, the vehicle may steer too much due to the GNSS antennas on the roof tilting and causing a misrepresentation of the location. If such an unlucky bump is placed on the same location as where the vehicle needs to turn, it may cause a too tight corner to be made and as a consequence break the hose.

7.3. Experiment 2: Docking point

This second experiment is designed to test if the behaviour near a docking point is desirable. For this experiment we construct a solution without mid intersection, but with strokes placed over the docking point. Figure 7.6 shows the generated solution. Note that next to testing the docking point, the tight circle with a radius of 3.775 meters is also an interesting scenario. This experiment will again log the GNSS data.

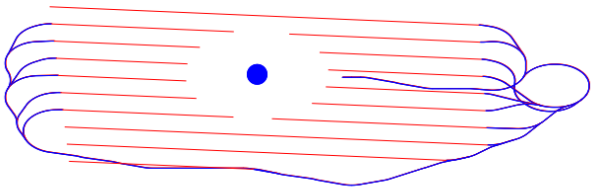


Figure 7.6: Visualisation of the generated solution with strokes placed over the docking point.

7.3.1. Results & Discussion

This experiment had to be aborted as the tight circle at the start was straining the hose beyond an acceptable level. This behaviour repeated on the second try, as was expected with the repeatability observed in the first experiment. The same is true for the conclusion of the previous experiment that stated that the used minimum turning radius was too small, but not for the same reason.

The straining of the hose was originally not observed when determining turning radius based on the corners made in the lattices. Problems arise when these motion primitives are repeatedly chosen to be the same tight corner. Put simply, a tight 90°turn exerts less force on the hose than a 180°turn of similar tightness. The suspected reason for this behaviour is that after the first 90°there is a build-up of tension, which results in an even greater build-up when performing another 90°turn in the same direction. Perhaps the build-up also interferes with the unrolling mechanism, causing the increased tension in the corners. The solution would be to increase the minimum turning radius to prevent such straining. However, one can argue that the Jojo plan used in the first experiment consisted of corners having a similar tightness and that increasing the turning radius would in turn reduce the quality of the generated solution. The basis of this flaw stems from an assumption made in the first section in chapter 5: *"If all the nodes and arcs in the graph are checked for collisions with the boundary and the arcs represent feasible transitions, we can be sure that the vehicle can safely transition over this graph."*, which as we can see is not always the case.

We must conclude that this behaviour and the proposed solution can be seen as a shortcoming of the plan generator since it does not make decisions based on existing tension in the hose.

Another problem that should be discussed can be seen in figure 7.6. The strokes surrounding the pit should be cut out better. This is due to them being cut of in a circular shape. This approach, however, does not take into account the size of the vehicle. Specifically the cuts in the second and 7th stroke from the top are a consequence of this approach. In reality, these cuts are not required. At the end of the stroke the vehicles bounding box is already touching the other side of the cut stroke, making the cuts unnecessary. However, this problem can be solved using better grid division algorithms and as discussed, are not the focus of this research.

8

Conclusion

Solving the Jojo problem and testing the generated solutions required this thesis to discuss a multitude of different topics and fields of study. We first discover that there exists a research gap concerning the curvature constrained path to n points that covers the least area. Many topics of research exist regarding the flexible tethers that are focused on preventing tangling. The most similar area of research appeared to be the topic of steerable needles, where only a small portion of work is focused on non exhaustive search.

The Jojo however, is on many fronts not as similar to steerable needles as we discuss thereafter. In order to get the vehicle to function we first had to implement a few core components as described in chapter 3. The most important being the observed lack of sufficient path following capabilities and as such, we explore the dynamics of the vehicle and design a custom path follower that suits the needs, which can both be tested in a simulated environment and is regarded safer due to the constraints.

We then continue to focus on solving the Jojo problem in chapter 4, for which we introduced a simplified version of the problem. Simple tests and observations were used to make the decision to use a graph based approach with approximation algorithms for the DST-problem. At this point we are aware of the potential shortcomings of this approach, but since we only observed reasonably scalable behaviour from this approach, we deem this a fair trade off.

With understanding of the problem, vehicle and a promising approach, we design and implement the graph based approach, but now on the full Jojo problem. This is done in chapter 5. We start with lattice construction methods, for which we use a trivial approach and decide upon a memory efficient method to represent the underlying graph. Next we introduce solutions to both the grid alignment issue by introducing decoupling and also allow for the path to enter a stroke midway. We also introduce the three approximation algorithms that we use for solving the DST-problem on the graph and highlight performance improvements we made. We then continue with general performance improvements, mostly due to on better programming decisions and extensive runtime inspections. Lastly, we also present a method for the solution to be represented as and implement an execution engine that can execute such a plan both in the simulation and real world.

In chapter 6 we test the algorithm on both small and large instances. The small ones are used to benchmark performance and compare the different approximation algorithms. After an exhaustive search we learn that the initially assumed to be reasonable variables for the lattices are actually not as good. We also learn that the quality of the solution is not influenced as much by reducing the lattice size, but compute times are lowered greatly. Nevertheless, we are able to confirm the workings of the algorithm and reliably construct Jojo plans from a given problem instance.

The large instances are based on the farm the vehicle is currently at. We split a large field in three parts and perform both numerical and visual analysis. Lastly, we also simulate the generated solutions to determine the real world execution times and only learn that these are quite long and that the execution engine works well. We observe that the calculated path only represents a small part of the total driving distance since the strokes account for most of the distance and if we are critical does highlight that perhaps the generated paths may be less optimal but apply more ignored constraints.

Lastly in chapter 7 we perform real world tests by generating small instances that are representative of difficult to execute segments of a Jojo plan. The first experiment, designed to test the frequently used

mid stroke intersections, performs well each try and mostly remarks the inaccuracies of the vehicle controller, especially when driving at a higher velocity.

The second experiment is designed to test behaviour near the attachment points and also makes use of long and tight corners. This experiment had to be interrupted due nearly damaging the hose due to straining. Previously we verified only the driveability of the most difficult paths in the lattices and made the assumption that any combination of the paths attached are also driveable. We observe that tight corners introduce a strain into the hose which after performing another tight corner in the same direction builds up even more strain. This eventually leads to critical straining and termination of the experiment. The proposed solution requires the solver to use a greater minimum turning radius.

Nevertheless, we can conclude that we have developed a novel method to solve the Jojo problem on large instances within a reasonable time. The shortcomings will be discussed in the next section, but based on the initial difficulty assumptions and lack of related work, the obtained results are a promising start.

8.1. Limitations

The proposed solution also has a list of shortcomings and limitations that we will summarise en discuss as well.

- **Maximum hose length** also known as maximum depth in the tree is not a constraint this solution is able to take into account. When using the graph based approach there exists no (fast) algorithm that is able to solve the DST-problem while taking maximum delay into account. Even if such an algorithm were to exist, we would still not be able to take this constraint into account since the length of the strokes is not accounted for with the current method. We therefore made the decision to ignore this constraint when calculating a route and present potential methods to fix solutions that violate it. Nevertheless, we also argue that it is better to prevent scenarios where the constraint is violated. Since this algorithm is intended to aid a human operator, we have this luxury.
- **Minimum turning angle is not the actual minimum** but has to be artificially increased to make sure the vehicle is able to execute a series of strain increasing transitions. Where one such transition may be possible, a multitude is not. In order to solve this a higher minimum turning radius must be used. One can see that since we a re limiting the vehicles movement set, the search space is in turn decreased to accommodate said scenario, such a decrease may lead to less favourable solutions.
- **Branching from strokes** and from the paths that connect the strokes to the grid, is not supported. This also introduces a decrease of the search space, again potentially decreasing the quality of a solution. However, there may exist a fix afterwards solution for this problem as well. We can perhaps detect when a segment of the path can be replaced with branching off a stroke.
- **Future performance** could also be a concern. Currently the implementation is fast enough to be user friendly, but larger problem instances for even longer hoses (think more than 1000 meters) could pose a problem. We do believe that implementing the algorithm in a less constrained programming language and using a more memory efficient graph representation can increase the maximum size of problem instance, but the exponential nature of the problem will not be overcome with a better implementations.

8.2. Future work & Recommendations

This thesis presents a novel method for solving the Jojo problem, which can in no doubt be improved and perhaps also be used in different areas of research. In this section, we summarise the most important areas of research to build upon using this research:

- **3D environments** are perhaps the next frontier for this approach. As mentioned before, we used the research topic of steerable needles as inspiration for this solution, the similarity of the problem may allow for this approach to be used when it is modified to work in a 3D environment.

- **Cost functions** currently used were limited to the length of the path since we assume this is representative. Perhaps it is also interesting to see if the vehicle dynamics are used as a cost function.
- **Post processing** methods should also be researched. We solution quality is limited by the grid resolution. Perhaps if a post processing step that first generates all the 'splits' and then uses a high resolution planner to calculate the routes between these splits is used, the quality of the solution would automatically improve. In the Jojo use case such an approach is probably necessary to facilitate the end user who may wants change the routes slightly.
- **Alternative solutions** using completely different methods would also be beneficial to solving the Jojo problem. Anecdotally: as a human, after observing the solutions presented by the algorithm, it becomes intuitive to predict the outcomes. This may also be the case of an AI based approach. It could also be possible to incorporate the field division with the path planner in order to be able to optimise both.
- **Dynamic scaling** is perhaps another avenue to explore. The grid is currently constrained to a set resolution. Perhaps it can be better to dynamically scale the grid to be of either higher or lower resolution depending on the situation. Perhaps a low resolution grid can be used, but with a higher resolution at constrained locations.
- **Guided search space** as seen with neural A* path planning [40], is perhaps also an inspiring direction. Neural A* guides the regular A* algorithm which is not something the DST-approximation can make use of, but perhaps the idea of using a variation of this approach to decrease the total search space could be a possibility. One can image a heat map an AI can generate based on a problem instance, which can either determine the boundaries of the grid or be combined with the aforementioned dynamic scaling.

References

[1] Joonwoo Ahn et al. "Accurate path tracking by adjusting look-ahead point in pure pursuit method". In: *International journal of automotive technology* 22 (2021), pp. 119–129.

[2] Syed Ali Ajwad and Jamshed Iqbal. "RECENT ADVANCES AND APPLICATIONS OF TETHERED ROBOTIC SYSTEMS." In: *Science International* 26.5 (2014).

[3] Kristoffer Bergman, Oskar Ljungqvist, and Daniel Axehill. "Improved optimization of motion primitives for motion planning in state lattices". In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2019, pp. 2307–2314.

[4] Jingwei Cao et al. "Trajectory tracking control algorithm for autonomous vehicle considering cornering characteristics". In: *IEEE Access* 8 (2020), pp. 59470–59484.

[5] Edoardo Cocconi. "Enhanced Pure Pursuit Algorithm & Autonomous Driving". In: (2019).

[6] Dmitri Dolgov et al. "Practical search techniques in path planning for autonomous driving". In: *Ann Arbor* 1001.48105 (2008), pp. 18–80.

[7] Michael L Fredman et al. "The pairing heap: A new form of self-adjusting heap". In: *Algorithmica* 1.1-4 (1986), pp. 111–129.

[8] Mengyu Fu, Oren Salzman, and Ron Alterovitz. "Toward certifiable motion planning for medical steerable needles". In: *Robotics science and systems: online proceedings* 2021 (2021).

[9] Alessandro Gasparetto et al. "Path planning and trajectory planning algorithms: A general overview". In: *Motion and Operation Planning of Robotic Systems: Background and Practical Approaches* (2015), pp. 3–27.

[10] Michel X Goemans and Young-Soo Myung. "A catalog of Steiner tree formulations". In: *Networks* 23.1 (1993), pp. 19–28.

[11] Ibrahim A Hameed, DD Bochtis, and CG Sorensen. "Driving angle and track sequence optimization for operational path planning using genetic algorithms". In: *Applied Engineering in Agriculture* 27.6 (2011), pp. 1077–1086.

[12] Mathias Hauptmann and Marek Karpinski. *A compendium on Steiner tree problems*. Inst. für Informatik, 2013.

[13] Vojtěch Horkỳ et al. "Dos and don'ts of conducting performance measurements in java". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. 2015, pp. 337–340.

[14] Anantha Sai Hari Haran V Injarapu and Suresh Kumar Gawre. "A survey of autonomous mobile robot path planning approaches". In: *2017 International conference on recent innovations in signal processing and embedded systems (RISE)*. IEEE. 2017, pp. 624–628.

[15] Emin Faruk Kececi and Fatih Kendir. "Wired Autonomous Vacuum Cleaner". In: *Mechatronics and Robotics Engineering for Advanced and Intelligent Manufacturing*. Springer. 2017, pp. 167–175.

[16] Thorsten Koch, Alexander Martin, and Stefan Voß. *SteinLib: An updated library on Steiner tree problems in graphs*. Springer, 2001.

[17] Valeria Leggieri, Mohamed Haouari, and Chefi Triki. "The Steiner Tree Problem with Delays: A compact formulation and reduction procedures". In: *Discrete Applied Mathematics* 164 (2014), pp. 178–190.

[18] *Lely company website*. `https://www.lely.com/nl/`.

[19] Fangde Liu et al. "Fast and adaptive fractal tree-based path planning for programmable bevel tip steerable needles". In: *IEEE Robotics and Automation Letters* 1.2 (2016), pp. 601–608.

[20] Edgar Lobaton et al. “Planning curvature-constrained paths to multiple goals using circle sampling”. In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 1463–1469.

[21] Steven Macenski et al. “The Marathon 2: A Navigation System”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020.

[22] Sandeep Kumar Malu, Jharna Majumdar, et al. “Kinematics, localization and control of differential drive mobile robot”. In: *Global Journal of Research In Engineering* 14.1 (2014), pp. 1–9.

[23] Jaret B Matthews and Issa A Nesnas. “On the design of the Axel and DuAxel rovers for extreme terrain exploration”. In: *2012 IEEE Aerospace Conference*. IEEE. 2012, pp. 1–10.

[24] Mohammad Noormohammadpour et al. “Dccast: Efficient point to multipoint transfers across datacenters”. In: *arXiv preprint arXiv:1707.02096* (2017).

[25] Jerome Ny, Eric Feron, and Emilio Frazzoli. “On the Dubins traveling salesman problem”. In: *IEEE Transactions on Automatic Control* 57.1 (2011), pp. 265–270.

[26] Timo Oksanen and A Visala. “Path planning algorithms for agricultural machines”. In: *Agricultural Engineering International: CIGR Journal* (2007).

[27] Timo Oksanen and Arto Visala. “Coverage path planning algorithms for agricultural field machines”. In: *Journal of field robotics* 26.8 (2009), pp. 651–668.

[28] Mihail Pivtoraiko, Ross Alan Knepper, and Alonzo Kelly. “Optimal, smooth, nonholonomic mobile robot motion planning in state lattices”. In: *Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-15* (2007).

[29] Markus Prossegger and Abdelhamid Bouchachia. “Ant colony optimization for Steiner tree problems”. In: *Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*. 2008, pp. 331–336.

[30] Atsushi Sakai et al. “Pythonrobotics: a python code collection of robotics algorithms”. In: *arXiv preprint arXiv:1808.10703* (2018).

[31] Aretor Samuel, Yudong Zhang, and Ruijie Zhu. “Deadline-aware multicast resource allocation in SDM-EONs with fluctuating delay-sensitive traffic”. In: *Journal of Lightwave Technology* 40.16 (2022), pp. 5355–5368.

[32] Rudolf Scheifele. “Steiner trees with bounded RC-delay”. In: *Algorithmica* 78 (2017), pp. 86–109.

[33] Jack Shirazi. *Java performance tuning*. " O’Reilly Media, Inc.", 2003.

[34] Jarrod M Snider et al. “Automatic steering methods for autonomous automobile path tracking”. In: *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08* (2009).

[35] Yun Seong Song and Metin Sitti. “STRIDE: A highly maneuverable and non-tethered water strider robot”. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE. 2007, pp. 980–984.

[36] *Tuning Java Virtual Machines (JVMs)*. `https://docs.oracle.com/cd/E13222_01/wls/docs81/perform/JVMTuning.html`. Accessed: 2023-04-28.

[37] *VisualVM*. `https://visualvm.github.io/index.html`. Accessed: 2023-04-28.

[38] Jiankun Wang, Baopu Li, and Max Q-H Meng. “Kinematic Constrained Bi-directional RRT with Efficient Branch Pruning for robot path planning”. In: *Expert Systems with Applications* 170 (2021), p. 114541.

[39] Dimitri Watel and Marc-Antoine Weisser. “A practical greedy approximation for the directed steiner tree problem”. In: *Journal of Combinatorial Optimization* 32 (2016), pp. 1327–1370.

[40] Ryo Yonetani et al. “Path planning using neural a* search”. In: *International conference on machine learning*. PMLR. 2021, pp. 12029–12039.

[41] Leonid Zosin and Samir Khuller. “On directed Steiner trees”. In: *SODA*. Vol. 2. 2002, pp. 59–63.

9

A

A

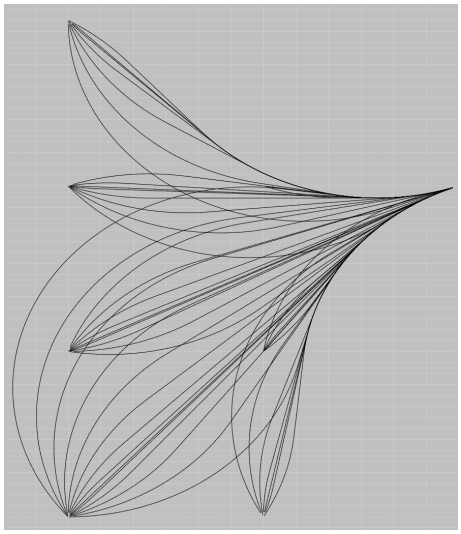


Figure 9.1: Visual representation of the paths created for a 5x5 lattice with 3m between the nodes and 10°buckets, specifically for the 200°bucket

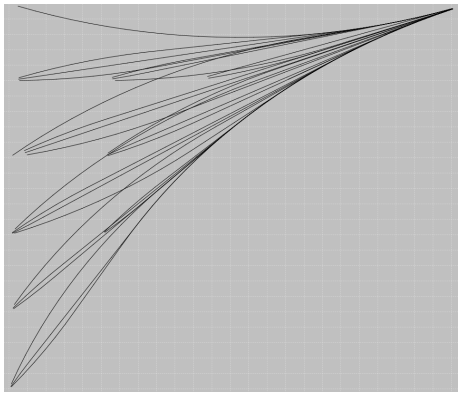


Figure 9.2: Visual representation of the paths created for a 11x11 lattice with 0.5m between the nodes and 10°buckets, specifically for the 200°bucket

resolution	latticeDim	buckets	score	time
0.5	9	12	253.69009850323772	29709340899
0.5	9	16	263.85624980349996	42452846101
0.5	9	20	313.381744821441	50255597000
0.5	9	24	253.0519334230242	74567712500
0.5	9	28	262.78864262637273	99529769200
0.5	9	32	258.7675390640471	138240370800
0.5	9	36	246.68194190896966	178542735100
0.5	11	12	254.50210451176295	44819722101
0.5	11	16	265.51044806044536	68122163200
0.5	11	20	263.5701557722499	81384004300
0.5	11	24	245.4577362547638	140408575000
0.5	11	28	252.4132904172208	183606338700
0.5	11	32	260.28022788479564	237991967100
0.5	11	36	245.18794214856035	303266840900
0.5	13	12	247.6558591119805	68078205700
0.5	13	16	242.20976063048118	100995865300
0.5	13	20	253.12867810508232	143816008499
0.5	13	24	237.8987836619836	233951475099
0.5	13	28	250.06598882543048	306204608399
0.5	13	32	240.16721958558347	394634742800
0.5	13	36	231.09251498833237	508904234300
0.5	15	12	253.5183780674525	92142517800
0.5	15	16	238.4895422037905	165767989300
0.5	15	20	249.0110825059438	202123202799
0.5	15	24	240.89786117512887	344829895900
0.5	15	28	242.88647984366807	473671967400
0.5	15	32	239.13420227134165	634432098501
0.5	15	36	235.15150088619126	811293579099
1	5	28	271.5083418590094	18144528001
1	5	32	290.51043381951854	18505597600
1	5	36	273.6095332618903	22539939800
1	7	12	297.3756247870839	6930124599
1	7	16	255.08584341519781	10884946801
1	7	20	265.86416898466496	17520460300
1	7	24	253.22234961603172	22061251100
1	7	28	251.06220906693002	29455642800
1	7	32	254.26854015675696	37194800000
1	7	36	249.93182490701537	46695707300
1	9	8	266.04366031971625	6873146100
1	9	12	251.89882543287504	12289673201
1	9	16	253.30676458596608	20699595999
1	9	20	251.61080359944896	30706437000
1	9	24	253.47429594976938	41996767300
1	9	28	251.3313717793604	60934499701
1	9	32	247.76861825292613	78328680500
1	9	36	248.33890680247995	94395368800
1	11	8	268.1457239501591	10391548901
1	11	12	260.0722684605633	21564482399
1	11	16	256.7603327151217	35549327200
1	11	20	247.89902327000846	59313497900
1	11	24	249.44402902381825	86022485500
1	11	28	245.95027537992837	114230166599

1	11	32	243.7563589079785	148666031600
1	11	36	231.77333764329495	200492192200
1	13	8	267.55163188677886	18156675300
1	13	12	257.93506416747744	35617356400
1	13	16	254.1478847704438	63837107299
1	13	20	245.56722132736041	102492567399
1	13	24	242.372535467472	146787127700
1	13	28	240.0142140988234	219251188400
1	13	32	243.54185782484228	287609704600
1	13	36	238.38494521848673	408059409100
1	15	8	267.19435472344605	28186017900
1	15	12	257.62657877860823	56229746899
1	15	16	256.28567674113503	106375704599
1	15	20	250.16430344851682	186355656001
1	15	24	250.22222679076003	281527932700
1	15	28	242.80385693100501	402115198999
1	15	32	257.19776556095275	504517054500
1	15	36	236.68360381534984	695658904100
1.5	5	8	273.45732564285083	2449825500
1.5	5	16	276.150345002595	5431696699
1.5	5	20	294.56465635499666	6979173700
1.5	5	24	278.70233495965397	9195911100
1.5	5	28	268.8527458674578	11037397400
1.5	5	32	265.032123222924	14152723900
1.5	5	36	260.18818325296184	17031881900
1.5	7	8	275.0103301621732	3378043500
1.5	7	12	277.2947863019267	5833723200
1.5	7	16	260.7589925040867	8948714699
1.5	7	20	255.4092152169017	12123394700
1.5	7	24	261.9835817905725	19031420400
1.5	7	28	256.2680732651862	25006860301
1.5	7	32	255.3482436431944	31771184600
1.5	7	36	257.9102715382329	38773216700
1.5	9	8	275.4034063345214	5504193901
1.5	9	12	276.93899714387544	10605949800
1.5	9	16	265.89420270899075	15972168300
1.5	9	20	259.8563175003356	25582596000
1.5	9	24	254.84207604501086	37321473200
1.5	9	28	252.0031064130025	54504474600
1.5	9	32	253.4089086678478	65642488100
1.5	9	36	243.87616873180664	87447077400
1.5	11	8	268.7669017123877	8733338100
1.5	11	12	282.36570853546317	18899038300
1.5	11	16	265.5573480160797	32941587001
1.5	11	20	263.6312655708451	47066121400
1.5	11	24	258.8895758008287	73813036000
1.5	11	28	251.75623302935682	106429021900
1.5	11	32	260.85210840626104	137659995700
1.5	11	36	259.2466946513611	189143868600
1.5	13	8	271.0249835335232	13534029400
1.5	13	12	278.0435508457769	32237836400
1.5	13	16	268.02188274329717	55195384299
1.5	13	20	263.425703707739	82990182701

1.5	13	24	270.97716255031867	134208929601
1.5	13	28	259.8229204566751	190494923600
1.5	13	32	264.3717345069539	251214497801
1.5	13	36	261.5370419136915	347778628200
1.5	15	8	275.39330832633357	19251660600
1.5	15	12	285.3068210754473	46167814101
1.5	15	16	269.7206790890024	79768359800
1.5	15	20	264.5550042734385	135461008700
1.5	15	24	271.3250408685527	210465854100
1.5	15	28	265.5208863965412	328689790501
1.5	15	32	264.65661795467867	408227695700
1.5	15	36	266.03188562717014	540245919300
2	5	12	286.59974974256136	2919819299
2	5	16	282.5676643455156	5034418300
2	5	20	293.65382519538605	6421253000
2	5	24	278.4709178711728	8413319000
2	5	28	283.89504785838716	10151064100
2	5	32	285.4582142693089	11758198600
2	5	36	266.3276151564615	14320538801
2	7	12	288.78438554307854	5727174401
2	7	16	288.384157348612	7985901500
2	7	20	285.7989842560129	11114060400
2	7	24	275.90988199161154	16943046600
2	7	28	278.64562682111193	21569670300
2	7	32	274.82473588084383	27099981701
2	7	36	268.4863250073605	34201426100
2	9	12	295.41674056348205	8923945500
2	9	16	295.6614750076086	14193070900
2	9	20	279.99002827765685	22252978100
2	9	24	281.13026228222503	34743807299
2	9	28	278.90940687631536	48288368001
2	9	32	276.4168359209266	61840431700
2	9	36	269.89331910702913	81719227199
2	11	12	294.5199326696751	15781933901
2	11	16	291.0477200166323	26058033700
2	11	20	280.19549525283105	42017335000
2	11	24	284.87372824211036	64497857000
2	11	28	275.81984658476557	87426398000
2	11	32	268.73471257929305	108883423300
2	11	36	277.61614028434803	152828447700
2	13	12	298.5662601631266	25070054200
2	13	16	301.4057032573338	41831654299
2	13	20	290.31491844812297	64562361100
2	13	24	288.40310287656746	101507193499
2	13	28	278.58508616792227	133578429299
2	13	32	272.7598340039498	177398931999
2	13	36	278.87898275963914	229526390701
2	15	12	298.50853442658376	35221280700
2	15	16	301.67717024154945	53151704900
2	15	20	296.79541161422196	86751456600
2	15	24	291.9738868922633	138155053400
2	15	28	283.82945363697934	170677232900
2	15	32	280.2709334968054	210848359201

2	15	36	282.10344740288883	296631804300

Table 9.1: Lattice tuning results for modified Dijkstra

resolution	latticeDim	buckets	score	time
0.5	9	12	191.19003660844908	12095312999
0.5	9	16	201.8635590382503	16392859600
0.5	9	20	234.21656108497808	21562120999
0.5	9	24	191.94140143347158	27859743000
0.5	9	28	203.9790323461996	44654782500
0.5	9	32	196.23065328963497	63063784699
0.5	9	36	182.1185192749186	67651284799
0.5	11	12	192.85121215018918	18496724399
0.5	11	16	203.49610955554425	27174655600
0.5	11	20	207.14792322891057	42113529600
0.5	11	24	192.5266032136379	59105328000
0.5	11	28	190.63885174689204	85632257399
0.5	11	32	192.83971798627994	109041102099
0.5	11	36	181.43302730944114	136672408801
0.5	13	12	193.1779481568407	33620999400
0.5	13	16	187.14170932936068	48070317900
0.5	13	20	191.93660294286315	68688663800
0.5	13	24	185.19507393153557	93878296700
0.5	13	28	186.94501714558137	136108767699
0.5	13	32	184.01179019335387	175114016400
0.5	13	36	184.9325919289602	243504015000
0.5	15	12	192.97704077778036	38729865001
0.5	15	16	185.70737203494656	63664884399
0.5	15	20	189.78977506215284	95962416101
0.5	15	24	184.24677132019045	132556738100
0.5	15	28	184.55082838725326	227904658400
0.5	15	32	180.5409033604289	305127461999
0.5	15	36	182.04951357787496	376523192200
1	5	28	212.06471957664593	10111363300
1	5	32	225.43951656075424	10759390999
1	5	36	223.72311794523551	13592227600
1	7	12	226.29281695478312	4323925700
1	7	16	202.82717433435167	6980525499
1	7	20	208.1375399869483	9778359500
1	7	24	201.77849424612685	13342828399
1	7	28	200.08000084394726	17429815101
1	7	32	195.40253890947878	21184624800
1	7	36	197.03055431981312	23627937700
1	9	8	218.587070726908	4037854000
1	9	12	197.68454005580043	6332748199
1	9	16	194.40631390590377	10307965900
1	9	20	196.60477979177745	18232502100
1	9	24	192.33908907758268	25718670300
1	9	28	190.8809888694831	31168361600
1	9	32	190.831524449033	42216225801
1	9	36	191.4611603422099	48767681200
1	11	8	213.14650027541106	5533400800
1	11	12	200.1468618884397	11856886699

1	11	16	196.18689088261976	18287383200
1	11	20	193.15442718013148	30437284501
1	11	24	192.36932692447584	40642216801
1	11	28	187.34356167981053	55653860600
1	11	32	188.95266585130625	72169614399
1	11	36	184.55159491158145	90276516800
1	13	8	207.4238355613715	8032689000
1	13	12	199.15749878760715	17114295299
1	13	16	193.594197694957	27599795599
1	13	20	191.37660207921004	53827032199
1	13	24	190.90603719365544	78154220300
1	13	28	188.94726162595123	105465702199
1	13	32	186.98512923434083	126428146300
1	13	36	179.37865227163908	167572532199
1	15	8	208.82586127416874	12049741701
1	15	12	196.08683750386703	24756039900
1	15	16	194.72209521647872	42998203500
1	15	20	191.07825377941708	79196743599
1	15	24	190.97763622332883	111468483500
1	15	28	186.06794071561313	147867465200
1	15	32	189.1205978292923	195316112499
1	15	36	181.00309395722857	246012960400
1.5	5	8	227.38054596744286	1990935900
1.5	5	16	220.34699446212093	4753656101
1.5	5	20	235.94321670284435	6120999200
1.5	5	24	217.286161285094	7446759100
1.5	5	28	217.20449652917597	9099320499
1.5	5	32	205.08761548501326	11090255501
1.5	5	36	205.0911440994258	12925894200
1.5	7	8	223.89271971403937	2577117499
1.5	7	12	223.60647001903007	3967753301
1.5	7	16	211.77007683130265	6199586201
1.5	7	20	209.51674858084502	8207729101
1.5	7	24	206.82521463379527	11523925799
1.5	7	28	206.38316382755994	14138209700
1.5	7	32	200.4690220477484	17816487199
1.5	7	36	202.17398100631348	22421644500
1.5	9	8	219.97430131097025	3052716999
1.5	9	12	216.78507218248555	5884106999
1.5	9	16	206.31720631996168	9094313800
1.5	9	20	207.7838908337234	14671967600
1.5	9	24	207.37941366210538	20271946199
1.5	9	28	203.02328647385872	27459700200
1.5	9	32	199.1254916789639	32140231200
1.5	9	36	196.65425333431062	43370914301
1.5	11	8	216.70748225667876	4641502100
1.5	11	12	213.55440490240207	8818801200
1.5	11	16	202.38654839197304	14790270501
1.5	11	20	205.25365347818328	21318429400
1.5	11	24	206.05910071707095	33369667301
1.5	11	28	201.3528943263897	43418336800
1.5	11	32	197.27992255241278	55642699700
1.5	11	36	194.8711159448485	75327219400

1.5	13	8	218.05972496014306	6152323001
1.5	13	12	213.24540368546036	12609534101
1.5	13	16	202.86066571256308	21403285000
1.5	13	20	205.17299555234163	33699149600
1.5	13	24	204.62344951761804	50075925699
1.5	13	28	202.5010283881923	67593836300
1.5	13	32	197.68776789255145	87805661300
1.5	13	36	196.24449532349902	118147532800
1.5	15	8	217.20815338626562	7653734800
1.5	15	12	213.92451417034016	16903491300
1.5	15	16	202.84878525491106	29957988300
1.5	15	20	203.26430313846495	48533161601
1.5	15	24	205.7722824902084	68693952101
1.5	15	28	199.819331807073	99094882200
1.5	15	32	197.6801317993806	126700022100
1.5	15	36	195.5997261635499	165930693100
2	5	12	234.7589167956593	2388926100
2	5	16	231.3938713582038	3399001499
2	5	20	237.76225177744385	4416617299
2	5	24	221.32479754683064	5492408500
2	5	28	217.33817430829535	6648238000
2	5	32	220.64354064526128	8091289001
2	5	36	213.36806334116017	9909613000
2	7	12	228.51498407010735	3282500901
2	7	16	228.52035874056386	5369987200
2	7	20	218.8436435246001	7082982100
2	7	24	216.05058002509514	9772562300
2	7	28	212.67923204316367	12389397701
2	7	32	214.70997854198262	15518668400
2	7	36	213.24548416233242	19635319900
2	9	12	228.32040087026266	4922369501
2	9	16	230.27918231964185	8138240700
2	9	20	220.50751633785003	12172113400
2	9	24	212.9858953873864	16491814899
2	9	28	212.11808740290576	22757788400
2	9	32	212.30841827919153	28521380500
2	9	36	212.441448343736	36216882301
2	11	12	230.68553885309365	7442739500
2	11	16	225.9503654499967	12129130100
2	11	20	220.4648470515188	19283689301
2	11	24	212.84507331997384	25825381701
2	11	28	212.0207082869212	36071768700
2	11	32	212.0613426853604	45257724399
2	11	36	212.30795312687778	57916596301
2	13	12	231.33321636114752	10206384399
2	13	16	225.89930911803276	17316646000
2	13	20	220.41203275582748	26643285200
2	13	24	212.81485549812004	35791111200
2	13	28	211.9316204654013	50968514401
2	13	32	208.32012662226845	66510373299
2	13	36	212.1563106493767	80976439100
2	15	12	231.31527003142452	13211255500
2	15	16	225.89930911803276	22346986399

2	15	20	220.76467127545067	34713244800
2	15	24	212.8129431213065	48137755399
2	15	28	211.93142007992734	62166900199
2	15	32	208.32012662226845	80555768200
2	15	36	212.56518328981747	102932220699

Table 9.2: Lattice tuning results for GreedyFLAC