Bachelor thesis
Computing Science

Radboud University

Routing Algorithms for Autonomous Agricultural Vehicles

Author:
Laura Philipse
s4528751

First supervisor/assessor:
dr. Nils Jansen
N.Jansen@cs.ru.nl

Second supervisor:
Johan Kleuskens
johank@phact.nl

Second assessor:
prof. dr. Frits Vaandrager
f.vaandrager@cs.ru.nl

January 22, 2020

Abstract

We investigate a routing problem for autonomous agricultural vehicles. Such vehicles operate on farmer fields where work needs to be done on tracks and these tracks need to connected efficiently. We represent the field as a graph and describe this problem as the Rural Postman Problem (RPP), where a certain subset of edges is marked as required and the goal is to make a route with all the required edges. We implement a Monte Carlo based algorithm for the RPP and compare it to a naive solution, where every track is just connected to the track next to it. This algorithm turns out to be an improvement, although we find different values for the parameters than the original paper.

Contents

1 Introduction **2**

2 Research **4**
2.1 The problem 4
2.2 The algorithm 6
2.3 Results 9
2.4 Discussion 13

3 Related Work **14**

4 Conclusions **18**

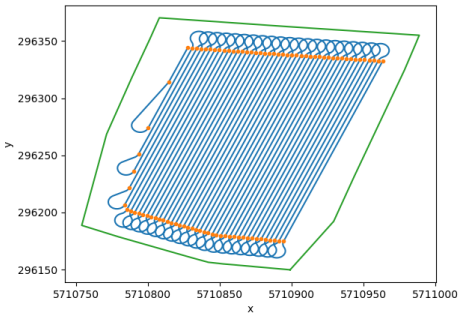
A Appendix **21**

Chapter 1

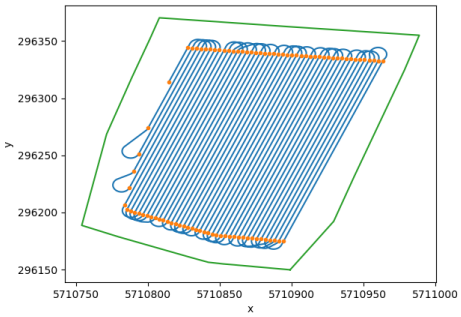
Introduction

A lot of research has been done into coverage path planning algorithms for autonomous robots with tasks such as lawn mowing or vacuum cleaning [3, 9]. For agricultural vehicles, most research has been about optimally positioning the tracks [15]. Our case however, is different. The company Phact [2] is currently working on developing software for an autonomous agricultural vehicle and they are looking for a way to efficiently connect the tracks that the vehicle needs to do work on. This is more complicated than it seems, as these vehicles typically cannot make narrow turns, so simply connecting the nearest tracks might not be efficient. As the fields are often very big, every improvement in the route can save the vehicle a lot of time and fuel.

We decided to represent the field as a graph and use a solution for the so-called Rural Postman Problem, a graph routing problem where a certain set of edges is marked as required and the goal is to find a route that contains all the required edges. In this case, the tracks are the required edges, while all the possible turns between the tracks are edges that are not required.



(a) The naive route for field 1, with length 8648.295



(b) The found route for field 1, with length 7911.374

Figure 1.1: Example routes for field 1

We implemented a Monte Carlo based algorithm by Fernández de Córdoba et al. [6] for the Rural Postman Problem. We compared this algorithm to a naive solution, where every track is just connected to the one next to it. As it turns out, the length of the route found by the algorithm in Figure 1.1b is indeed shorter than the length of the naive route in Figure 1.1a.

Chapter 2

Research

2.1 The problem

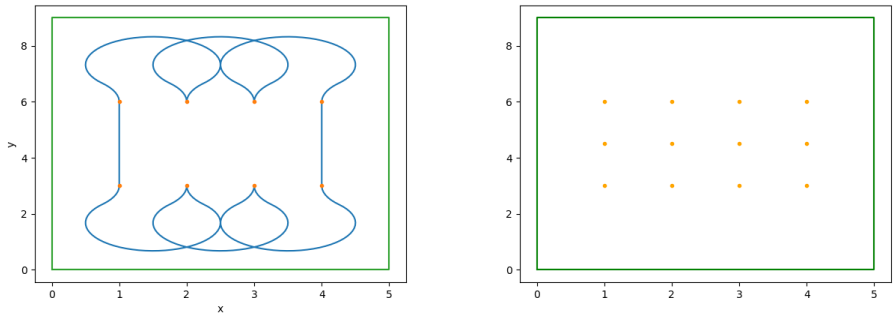
We want to solve a routing problem on farmer fields for autonomous agricultural vehicles. The farming fields have lines where the crops need to be. We call these lines tracks. We already have these tracks, but they need to connected into one route for the autonomous agricultural vehicle to follow. This vehicle has a big turning radius, so just connecting every track to the track next to it is probably not that efficient.

Our problem can be defined as a graph by attaching a node to the end of each track and then connect the nodes at the same side of the field to each other with turns. The weight of each edge is the distance between the nodes. For the turns, the distance is calculated using Dubins, which is an algorithm that calculates a path between two points while taking into account the orientation and turning radius of the vehicle. It can be debated how useful it is to connect all the nodes at the same side with each other. It could be the case that only the nearest ones need to be connected.

A field can be defined as a undirected graph $G = (V, E)$ where all the vertices $v \in V$ are the ends of the each line on the field. The edges $e \in E$ consist of the given lines and the turns between the vertices on the same side of the field. The weight of all $e \in E$ is defined as the distance.

Over the years, researchers have defined lots of different routing problems, such as the Traveling Salesman Problem [7] and the Chinese Postman Problem [7]. As we defined our problem as a graph, we could use solutions to existing routing problems to solve it. We have found two problems that could be useful in this case.

First, the Traveling Salesman Problem (TSP). The TSP aims to find a path



(a) A not valid solution that does contain all nodes

(b) The field with extra poins in the middle

Figure 2.1: Some examples

where each node is visited exactly once and then returns to the start node. This could be applicable to our problem, but we need to make sure that each track is visited instead of just using the turns to visit every point, because just visiting every point can result in invalid solutions. An example of a route that contains all nodes, but not all tracks can be seen in Figure 2.1a. This can be done by giving the tracks a weight of zero or even negative, which makes it more likely the algorithm will use that edge. Another option is to add an extra node in the middle of each track, as seen in Figure 2.1b, as that makes sure that a route can only visit all points if it uses all tracks. Seyyedhasani et al. [17] used this strategy to solve the vehicle routing problem with multiple vehicles.

Secondly, the Rural Postman Problem (RPP). The RPP is a variant of the Chinese Postman Problem (CPP). The CPP aims to find a route where the start and end point is the same and each edge is traveled at least once. The RPP aims to visit a certain subset of edges at least once. We could use this to make the tracks required while the turns are not, but the problem is that we want to make sure that the tracks are traveled exactly once, while with the RPP, required edges are visited at least once. To solve this, we could give tracks a relatively high weight to make sure they are used only once. Another option is to make sure that all nodes are in the found solution exactly once.

An overview of some papers that solve these problems can be found in the Related Works section.

2.2 The algorithm

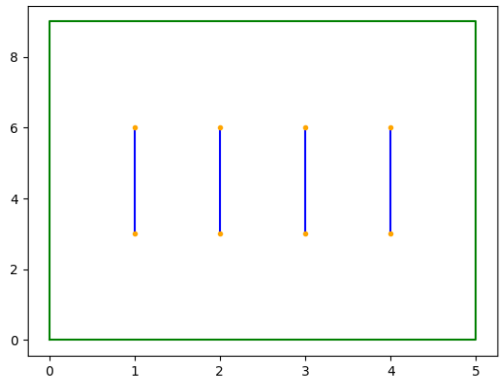


Figure 2.2: The example field

We decided to implement this problem and the Monte Carlo based algorithm described in Fernández de Córdoba et al. [6], which is a randomized algorithm based on probabilistic choices, in Python. A field consists of edge points and lines. The field edge points consist of x and y coordinates in a list. For the example field in Figure 2.2, the field edge points look like this: `[[0,0], [0,9], [5,9], [5,0], [0,0]]`. The lines consist of two points, with both x and y coordinates. The lines of the example field are:
`[[[1,6], [1,3]], [[2,6], [2,3]], [[3,6], [3,3]], [[4,6], [4,3]]]`

The field coordinates are just used for drawing the outline of the field and are not considered in planning the route. For both the field and lines, we converted these structures in a list with the x coordinates and a list with the y coordinates to make them easier to plot. Then we calculated a distance matrix, where the distances between each of the points were stored. If a point is not reachable, the distance is noted as inf. For the turns, we used a python package called Dubins [1] to calculate the distance. For this, we need to know the turning radius of the vehicle. We also save the turns in a separate matrix to be able to plot them later.
The distance matrix for our example field would look like this:

```
[[0, 3.0, 6.0325, inf, 3.1416, inf, 4.1416, inf],
[3.0, 0, inf, 6.0325, inf, 3.1416, inf, 4.1416],
[6.0326, inf, 0, 3.0, 6.0325, inf, 3.1416, inf],
[inf, 6.0325, 3.0, 0, inf, 6.0325, inf, 3.1416],
[3.1416, inf, 6.0325, inf, 0, 3.0, 6.0325, inf],
[inf, 3.1416, inf, 6.0325, 3.0, 0, inf, 6.0325],
```

```
[4.1416, inf, 3.1416, inf, 6.0325, inf, 0, 3.0],
[inf, 4.1416, inf, 3.1416, inf, 6.0325, 3.0, 0]]
```

A possible route is represented as a list of integers, where each integer refers to a point from the field. We also define a function that can calculate the length of a solution and a function that checks if the required edges are in a solution. Then we defined a function to plot the outline of a field and the points and a function to plot the outline of a field and a solution. We also defined a naive route to use as a baseline. The naive route for the example graph in Figure 2.2 is in Figure 2.3.

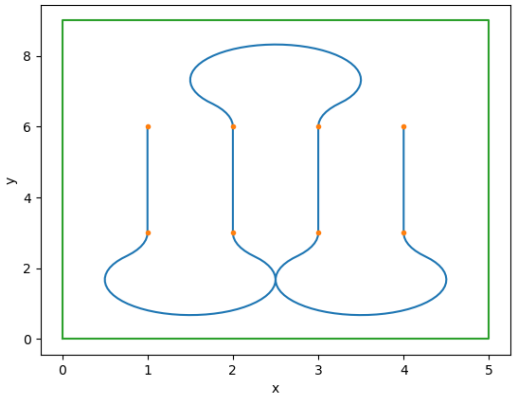


Figure 2.3: The naive route for the example field

The algorithm has a few parameters: it has the number of nodes, the distance matrix, the value of α, the number of iterations and the value of x. The α is part of the formula used to calculate the probability as presented in the paper. The x, however, only exists because of the way we chose to implement this. Every iteration, the algorithm generates a route by adding nodes to a list until the list contains all nodes. It starts by adding a random node, then if the required edge attached to the last node of the route isn't in the route yet, it will select that as the edge to take and adds the node at the other end of the edge to the list. If that edge is already in the route, it selects all nodes that are reachable from the last node in the route and aren't in the route yet, and randomly chooses one of them based on the probability. We chose to implement this by making a list with all the options, where the more often the item is in that list, the more likely it is that that item will be chosen. This process repeats until the route contains as many nodes as the number of nodes. Then it checks if the route is a valid route and if the route is shorter than the one found in previous iterations. If so, it replaces the shortest route with the one found. After went through the number of

iterations, it returns the shortest route.

```
def paper41(nrofnodes, distance, α, iterations, x):
    shortestRoute = []
    shortestDist = ∞
    for every iteration:
        route = []
        add a random node to route
        for every node i:
           if the required edge is not in the route yet:
                    add the other node attached to that edge to
                        route
           else:
                options = []
                for every node j:
                    if the distance from node i to node j is not
                         ∞ and node j is not in route:
                        add node j to options

                total = 0
                for every option k:
                    calculate the probability of this edge with
                        p = 1/(distance[route[i]][k])^α
                    add p to the total
                for every option k:
                    calculate how many times k should be in the
                        list with p/total * x
                    add k that many times to options

                choose a random option based on the probability
                   from the options list and add it to the route
        if the route contains all nodes:
            if the route is a valid route:
                if the length of the route < the length of the
                    shortest route:
                    set this route as the shortest route
    return the shortest route
```

For our example field, the length of the naive route is 30.0976 while the length of the route from the Monte Carlo based algorithm described in Fernández de Córdoba et al. [6] is 22.4248. This is an improvement of 25.4931%.

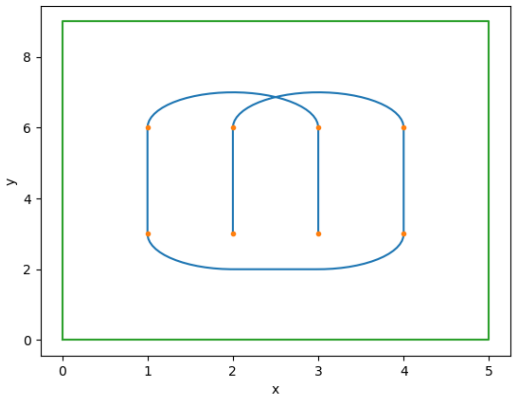
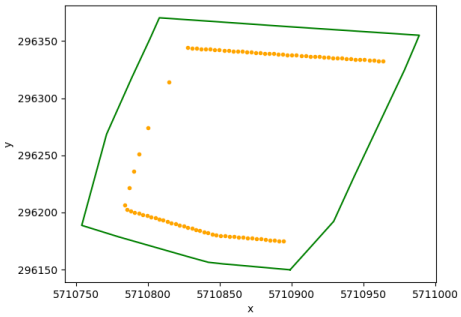


Figure 2.4: The algorithm applied to the example field

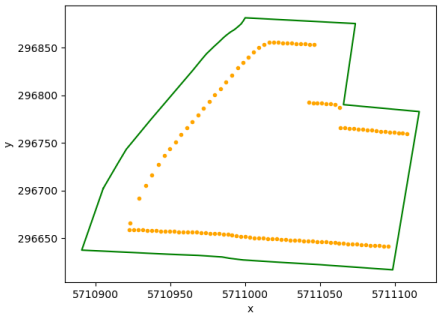
2.3 Results

We test this algorithm on three fields as seen in Figure 2.5. These fields are actual fields that are used by Phact to test their software. Field one is a relatively square field, just as field two. Field three is a more complex field. The naive route for field three, as seen in Figure 2.5d, is very long, so for field three we will often find bigger improvements than for the other fields. Fields are special cases of graphs, as every node is connected to exactly one required edge. Because of this, the algorithm may behave differently than as described in the paper. For example, the paper assumes that it needs to be possible to visit nodes multiple times, but in our case the algorithm can always visit all the required edges without doing so. The paper also gives some simplification routines to optimize routes that contain some edges more than once, but this also does not happen in our case, so there is no need to use the simplification routines.

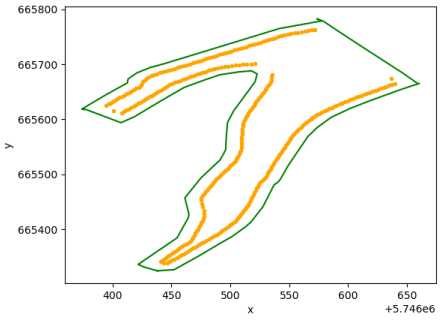
The paper tried several values between 1 and 20 for α and found that around 4 the results were the best, and they used 1000 for the number of iterations. Because we already saw that we have a special case, it might be better to use different values. We also need to find out the influence of our newly introduced parameter x. We ran all tests on a 64-bit laptop with 4 GB RAM and an Intel Core i5-2520M CPU at 2.50 GHz running Windows 10 Pro. The values in the tables are an average of running the algorithm 100 times, as this algorithm is Monte Carlo based, so the outcomes may vary in every run. We report the computing time in seconds and the length of the found route. We also report the length of the naive route as a comparison. We rounded all values to three decimals.



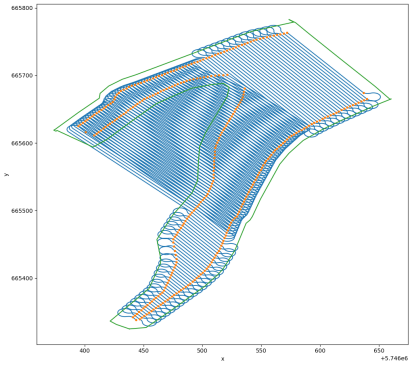
(a) Field 1



(b) Field 2



(c) Field 3



(d) Naive route for field 3

Figure 2.5: The three fields and the naive route for field 3

We start by comparing different values for α with 100 as the number of iterations and 1000 for x. We only test up to $\alpha = 115$, as using 120 with field 3 results in an overflow error.

Field	Naive solution	$\alpha = 4$		$\alpha = 10$		$\alpha = 50$	
		Time	Solution	Time	Solution	Time	Solution
1	8648.295	0.847	8164.692	0.830	8008.982	0.860	7941.379
2	10640.645	1.273	10024.837	1.301	9778.680	1.342	9713.675
3	25714.613	10.302	14357.313	10.394	13459.836	10.436	13187.970

Field	Naive solution	$\alpha = 65$		$\alpha = 80$		$\alpha = 95$	
		Time	Solution	Time	Solution	Time	Solution
1	8648.295	0.832	7940.997	0.850	7939.252	0.821	7941.619
2	10640.645	1.318	9717.129	1.360	9716.638	1.352	9716.091
3	25714.613	10.422	13232.852	10.459	13229.079	10.521	13203.307

Field	Naive solution	$\alpha = 100$		$\alpha = 110$		$\alpha = 115$	
		Time	Solution	Time	Solution	Time	Solution
1	8648.295	0.840	7936.453	0.847	7938.828	0.860	7935.594
2	10640.645	1.333	9712.975	1.371	9719.603	1.347	9716.975
3	25714.613	10.472	13193.147	10.552	13215.742	10.420	13171.820

A higher α generally results in better solutions, while the time does not increase. So while the paper found 4 the best value for α, we opt for 115, even though for field 2, the shortest route was found using $\alpha = 100$, because that might have been due to the randomized nature of this algorithm.

We will now test some values for x with $\alpha = 115$ and 100 as the number of iterations.

Field	Naive solution	$x = 10$		$x = 100$		$x = 1000$	
		Time	Solution	Time	Solution	Time	Solution
1	8648.295	0.332	8647.953	0.378	8055.724	0.575	7946.604
2	10640.645	0.645	11503.506	0.703	10080.842	0.981	9747.370
3	25714.613	8.567	30326.396	8.702	20176.747	9.500	14328.179

Field	Naive solution	$x = 1000$		$x = 1500$		$x = 2500$	
		Time	Solution	Time	Solution	Time	Solution
1	8648.295	0.836	7937.230	1.092	7936.323	1.620	7931.064
2	10640.645	1.336	9717.701	1.686	9706.977	2.324	9697.799
3	25714.613	10.429	13241.632	11.421	12859.505	13.472	12618.310

Field	Naive solution	$x = 3500$		$x = 4500$		$x = 10000$	
		Time	Solution	Time	Solution	Time	Solution
1	8648.295	2.059	7931.357	2.563	7931.904	5.188	7930.303
2	10640.645	3.036	9695.403	3.739	9696.532	7.316	9692.496
3	25714.613	15.238	12515.713	17.817	12478.326	27.794	12403.260

When x increases, so does the computing time, while the length of the solution route gets shorter. However, at some point the time might increase too much compared to the improvement in route. What exactly the optimal value for x is, is up to debate. We will go with 1000 for now.

Now we test some values for the number of iterations with $\alpha = 115$ and $x = 1000$.

Field	Naive solution	iterations = 10		iterations = 100		iterations = 200	
		Time	Solution	Time	Solution	Time	Solution
1	8648.295	0.085	7978.090	0.877	7938.113	1.683	7928.959
2	10640.645	0.139	9796.326	1.340	9717.450	2.807	9702.650
3	25714.613	1.075	13663.946	10.666	13212.244	21.179	13134.703

Field	Naive solution	iterations = 300		iterations = 400		iterations = 1000	
		Time	Solution	Time	Solution	Time	Solution
1	8648.295	2.578	7926.705	3.391	7925.173	8.658	7914.822
2	10640.645	3.963	9693.681	5.373	9688.895	13.8112	9674.544
3	25714.613	32.011	13051.831	42.161	13001.729	105.880	12894.209

Just as with x, as the number of iterations increases, the length of the solution routes gets shorter while the calculation time gets longer.

2.4 Discussion

It is not possible to say for sure what the optimal values are for x and the number of iterations. This is up to debate depending on how long the calculations is allowed to take and how much time one is willing to sacrifice for how much improvement. There are also other factors that need to be taken into account. When the algorithm is run on a better machine, the calculation time will decrease. It also might be the case that there is a more efficient way of implementing this algorithm that we have not considered. It might also be worth it to implement this algorithm in another programming language to decrease the computing time. We also mentioned before that it might not be needed to allow turns from every track to all the other tracks, but that just the first few might do. This can also decrease the computing time, as the graph will have less edges, so the algorithm has less options to choose from.

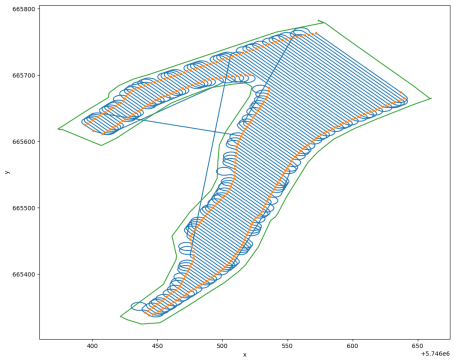


Figure 2.6: A route for field 3 that crosses some tracks

A limitation is that this implementation of the algorithm does not take the field edge into account, so turns might go outside the edge. It also does not take into account that the route cannot cross any of the tracks. As seen in Figure 2.6, it is possible that the algorithm produces a route that crosses some of the tracks. Future research needs to be done to avoid this. Our representation of the field currently does not take into account that the vehicle drives at a slower speed when making a turn compared to a straight line. The main problem here is that the 'turns' returned by the dubins algorithm may contain straight lines, for example when connecting two tracks that are far apart, and we currently have no way of separating those parts from the turning parts. If a way to take the speeds into account is found, it might be possible that other solutions than the ones found now are more optimal.

Chapter 3

Related Work

Seyyedhasani and Dvorak (2017) [17] transformed two fields, a basic rectangular field and a non-convex one, into graphs by adding nodes at the end and in the middle of every path, and then giving invalid paths a value at least ten times higher than the cost of any valid path, to make sure these paths are never used. They then used a modified Clarke-Wright algorithm and Tabu Search to solve the problem for varying number of vehicles, ranging from 1 to 10. It costs Tabu Search at least two hours on an Intel i7 processor to calculate one scenario, while the modified Clarke-Wright algorithm takes less than a second. While with multiple vehicles Tabu Search performed better, the differences in performance with just one vehicle were minimal.

Yaghini et at. (2011) [18] compare a Simulated Annealing (SA) algorithm and an Ant Colony Optimization (ACO) algorithm for finding the shortest Hamiltonian path. They implemented the algorithms using Java and ran it on a computer with core2 CPU at 2.66 GHz, 4 Gigabytes of Ram that had Windows Vista as its operating system. They then tested the algorithms on the standard problems ulysses16, a280, berlin52, gr666, kroA100, pr1002, rd100 and u1060. The average best solution relative gap is 0.0736 for SA and 0.0602 for ACO. The average of the average of the relative gap is 0.104 for SA and 0.071 for ACO. The average CPU time is 22.46 s for SA and 50.23 s for ACO. SA is faster, while ACO generally gets a better solution. They then ran both algorithms on 1071 Iranian cities. For 1071 nodes, SA took on average 45.76 CPU seconds, while ACO took 205.89 CPU seconds. They suggest that it may be possible to get better results by using other ways to tune the parameters.

Pearn et al. (1995) [16] compared the heuristic Christofides et al. algorithm with a modified and a reverse version for the RPP. The Christofides et al. algorithm has a complexity of $O(|V|^3)$. They have ten sets of 20 problems that they solved using the algorithms. Then they did an addi-

tional test with another 50 problems. They ran the algorithms on the PC 486 DX-33 and none of the problems, not even those with 50 nodes, 199 edges and 61 required edges, took more than 0.3 CPU seconds to be solved. Both the modified and the reverse algorithm produce significantly better solutions than the original algorithm, while the reverse performs better than the modified algorithm if there are a relatively high number of nodes with an odd degree.

Corberán et al. (2001) [4] give a cutting plane algorithm for solving the General Routing Problem, where one wants to find the shortest route where a set of required edges and a set of required nodes is passed at least once. If the set of required nodes is empty, this is a Rural Postman Problem. They coded the algorithm in C and ran it on a machine with a speed comparable to a 66 MHz Pentium. They had 118 RPP instances in total, 116 of those were solved to optimality by cutting planes alone and the other 2 were solved after invoking branch and bound. The longest time seemed to be 109.11 s for 3 instances.

Ghiani et al. (2000) [11] present a new formulation, polyhedral properties and a branch-and-cut algorithm for the RPP. They implemented the algorithm in C using the Watcom Integrated Development Environment and the CPLEX library and they ran it on a PC with a Pentium processor at 90 Mhz with 16 MB RAM. They then tested their algorithm on the Albaida1 graph as a real-world example and they also randomly generated three different types of graphs. Their algorithm can solve instances up to 350 vertices to optimality, where their main limitation was the memory. The longest time they reported was 1368.0 CPU seconds for five instances.

Corberán et al. (2007) [5] presented a Branch and Cut algorithm for the Windy General Routing problem that also can be used for the RPP. They coded the algorithm in C/C++ using Cplex 9.0 MIP Solver with Concert Technology 2.0. They ran the tests on a Pentium IV at 2.8GHz machine with 1GB RAM with a time limit of 10 h. The algorithm solved all RPP instances to optimality. For the instances that on average have 886.2 nodes and 2289.9 edges the computing time is on average 3033.7 seconds, while the instances with on average 665.7 nodes and 1698.41 edges have an average computing time of 2873.3 seconds. The instances with on average 446.0 nodes and 1128.9 edges have an average computing time of 26.6 seconds.

Fernández et al. (2003) [8] examine three existing mathematical programming formulations of the RPP and then define a new variant that substantially reduces both the number of flow variables and the number of "link" constraints. They then present a new algorithm to get lower and upper bounds for the RPP and test it on various problems. They coded the algo-

rithm in C using the CPLEX 6.5 library and ran it on a Sun Ultra2 Model 1200 with two 200 MHz processors but using only one processor (SPECint95 6.90). The longest average CPU time was reported for a group of problems called Node250 with 232.6 vertices on average, namely 1229.849 seconds, while Node300 with an average of 277.6 vertices only took 1077.504 seconds on average. Groups with an average of 181.8 vertices or less took significantly less long, with a few exceptions. 160 of the 173 problems were solved to optimality. For those 13 for which optimality was not proved, the percentage gaps between the obtained upper and lower bounds are very small and always below 1%. There is room for improvement by future work.

Ghiana et al. (2006) [10] present a insertion heuristic for the RPP. They coded it in C and ran it on a Pentium-1GHz personal computer. They then compared their heuristic with the classical Frederickson procedure and they also applied the 2-opt procedure developed by Hertz et al.[13] to the solutions provided by the two constructive heuristics. Without using 2-opt, the new heuristic was better in 25 cases, worse in 28 and got the same result in 1 case. With 2-opt, it was better in 16 cases, worse in 25 and got the same result in 13 case. Without 2-opt, the insertion heuristic was slower every time, while with 2-opt, it was faster in 14 cases. They used instances with 350 vertices or less and the longest time for the insertion heuristic without 2-opt was 20.544 seconds. With 2-opt, the longest time was 9170.910 seconds.

Fernàndez de Córdoba et al. (1998) [6] present a heuristic algorithm based on Monte Carlo methods for the RPP. They ran it on a Personal Computer with a Pentium 90 processor. The source code has less than 700 FORTRAN 77 lines (including comment lines) and the executable file has no more than 150 kbytes. They tested their algorithm on the 24 instances of Christofides et al. and the 2 Albaida instances and compared the results to the Christofides et al. heuristic. They tried a few different variations and a few different values for their parameters, but it is hard to say for sure which is best as the algorithm is based on randomness. In what they consider the best version of the algorithm, they got the optimal result in 19 instances. In the other cases, the solution was not that far from optimality. The longest time they reported for this version was 8.49 seconds. The algorithm is easily adaptable to other variations of the problem. The solution may also be used for algorithms that need an initial solution.

Hertz et al. (1999) [13] present several improvement procedures for the RPP, including 2-opt. These can be used to construct an initial solution from scratch, to obtain a feasible solution starting from an infeasible solution, or to postoptimize a feasible solution. They coded the heuristics in Pascal and ran them on a Silicon Graphics Indy 194 MHz IP25 Processor.

They tested the algorithms on randomly generated instances and on the 24 Instances of Christofides et al., the Albaida Instances and the 10 Corberán Instances. They compared the results from the Frederickson heuristic with Frederickson + 2-opt and Construct + 2-opt. When Frederickson wouldn't find an optimal solution, the other two versions would find a better one, although their computation time would be longer. Especially with the Corberán instances the increase in time is significant. Where Fredrickson takes less than a second to complete any of the instances, the shortest time with 2-opt is 1056 seconds.

Groves et al. (2005) [12] present several heuristics, including a Two-Opt and a Three-Opt, for the RPP. The heuristics are based on a local search framework. They use the heuristic of Frederickson (1979), which has complexity $O(|V|^3)$, to generate an initial solution and then compare this solution with the improved solutions found by Two-opt and Three-opt. They ran their tests on an Intel Pentium IV processor (2.8 GHz) with 512 Mb memory. They used the 24 instances by Christofides et al. (1986) and the two Albaida instances. They later omitted instance 18 from the results as they got a valid solution that is shorter than the reported optimal cost. For Two-Opt, all times were 0 seconds, while for Three-Opt there were two instances where the time was 2 seconds. They also randomly generated some new instances that are bigger, ten of those were between 3000 and 5000 vertices and between the 20000 and 30000 edges. The longest time reported on those instances for Two-Opt is 5901 seconds. They also mention that the Two-Opt heuristic from Hertz et al. [13] has complexity $O(|E|5)$ per iteration.

Muyldermans et al. (2005) [14] present several heuristics, including 2-opt reverse, which is based on the 2-opt for the TSP and 2-opt dir-opt $\circ$ flip. They coded the algorithms in C/C++ and run on a personal computer (Pentium II, 500 MHz, 128 MB RAM) under Windows 98. They tested the algorithms on the 24 instances of Christofides et al., the 2 Albaida instances and several more instances. They then compared the results from Frederickson & 2-opt and Construct & 2-opt from Hertz et al. [13] with several of their methods.

Chapter 4

Conclusions

We solved the routing problem for autonomous agricultural vehicles on farmer fields by representing the field and the tracks where work needs to be done as a graph where we need to solve the Rural Postman Problem. For the Rural Postman Problem, a route needs to be made that contains a required subset of the edges in the graph. We then implemented a Monte Carlo based algorithm presented in Fernández de Córdoba et al. [6] and compared the results to a naive solution, where each track is connected to the one next to it. The algorithm showed to be an improvement over the naive solution. We did find that different values than the ones proposed in the paper for the parameters work better in our case. Future research would be to make sure the algorithm does take into account that the field edge and the tracks should not be crossed. Another possible improvement would be to take into account the driving speeds of the vehicle.

Bibliography

[1] dubins. `https://pypi.org/project/dubins/`.

[2] Phact. `https://phact.nl/`.

[3] Howie Choset. Coverage for robotics – a survey of recent results. *Annals of mathematics and artificial intelligence*, 31(1-4):113–126, 2001.

[4] Angel Corberáan, Adam N. Letchford, and José María Sanchis. A cutting plane algorithm for the general routing problem. *Mathematical Programming*, 90(2):291–316, Apr 2001.

[5] Angel Corberán, Isaac Plana, and José M. Sanchis. A branch & cut algorithm for the windy general routing problem and special cases. *Networks*, 49(4):245–257, 2007.

[6] P. Fernández de Córdoba, L.M. Garcría Raffi, and J.M. Sanchis. A heuristic algorithm based on monte carlo methods for the rural postman problem. *Computers & Operations Research*, 25(12):1097 – 1106, 1998.

[7] Moshe Dror and E. Benavent. *Arc routing: theory, solutions and applications*. Kluwer Academic Publishers, 2000.

[8] Elena Fernández, Oscar Meza, Robert Garfinkel, and Maruja Ortega. On the undirected rural postman problem: Tight bounds based on a new formulation. *Operations Research*, 51(2):281–291, 2003.

[9] Enric Galceran and Marc Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, 61(12):1258 – 1276, 2013.

[10] Gianpaolo Ghiani, Demetrio Laganà, and Roberto Musmanno. A constructive heuristic for the undirected rural postman problem. *Computers & Operations Research*, 33(12):3450 – 3457, 2006. Part Special Issue: Recent Algorithmic Advances for Arc Routing Problems.

[11] Gianpaolo Ghiani and Gilbert Laporte. A branch-and-cut algorithm for the undirected rural postman problem. *Mathematical Programming*, 87(3):467–481, May 2000.

[12] GW Groves and JH van Vuuren. Efficient heuristics for the rural postman problem. *ORiON*, 21(1), 2005.

[13] Alain Hertz, Gilbert Laporte, and Pierrette Nanchen Hugo. Improvement procedures for the undirected rural postman problem. *INFORMS Journal on Computing*, 11(1):53, 1999.

[14] Luc Muyldermans, Patrick Beullens, Dirk Cattrysse, and Dirk Van Oudheusden. Exploring variants of 2-opt and 3-opt for the general routing problem. *Operations Research*, 53(6):982–995, 2005.

[15] T Oksanen and A Visala. Path planning algorithms for agricultural machines. *Agricultural Engineering International: CIGR Journal*, 2007.

[16] W.L. Pearn and T.C. Wu. Algorithms for the rural postman problem. *Computers & Operations Research*, 22(8):819 – 828, 1995.

[17] Hasan Seyyedhasani and Joseph S. Dvorak. Using the vehicle routing problem to reduce field completion times with multiple machines. *Computers and Electronics in Agriculture*, 134:142–150, 2007.

[18] M. Yaghini, M. Momeni, and M. Sarmadi. Finding the shortest hamiltonian path for iranian cities using hybrid simulated annealing and ant colony optimization algorithms. *International Journal of Industrial Engineering and Production Research*, Mar 2011.

Appendix A

Appendix

```
import matplotlib.pyplot as plt
import math
import numpy as np
import itertools
import random
import timeit
import dubins

def readArrays(lines, field):
    x = []
    y = []

    a = []
    b = []
    for i in lines:
        x.append(i[0][0])
        x.append(i[1][0])
        y.append(i[0][1])
        y.append(i[1][1])
    for i in field:
        a.append(i[0])
        b.append(i[1])
    return x, y, a, b

def plotPoints(xlines, ylines, xfield, yfield):
    fig, ax = plt.subplots( )
    line1, = ax.plot(xlines, ylines, '.')
    line2, = ax.plot(xfield, yfield)

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    plt.show()

def calculateTurns(xlines, ylines, turnRad):
    turns = []
    for i in range(0,len(xlines)):
        m = []
```

```
    for j in range(0,len(xlines)):
        if (i%2==0 and j%2==0) or (i%2==1 and j%2==1):
            if(i%2==0 and j%2==0):
                if((xlines[i]-xlines[i+1]) == 0 and (xlines[
                    j]-xlines[j+1]) == 0):
                    t1 = 1/2*math.pi
                    t2 = -1/2*math.pi
                elif((xlines[i]-xlines[i+1]) == 0 ):
                    t1 = 1/2*math.pi
                    t2 = math.atan(abs(ylines[j]-ylines[j
                        +1])/abs(xlines[j]-xlines[j+1]))-math
                        .pi
                elif((xlines[j]-xlines[j+1]) == 0):
                    t1 = math.atan(abs(ylines[i]-ylines[i
                        +1])/abs(xlines[i]-xlines[i+1]))
                    t2 = -1/2*math.pi
                else:
                    t1 = math.atan(abs(ylines[i]-ylines[i
                        +1])/abs(xlines[i]-xlines[i+1]))
                    t2 = math.atan(abs(ylines[j]-ylines[j
                        +1])/abs(xlines[j]-xlines[j+1]))-math
                        .pi
            elif(i%2==1 and j%2==1) :
                if((xlines[i]-xlines[i-1]) == 0 and (xlines[
                    j]-xlines[j-1]) == 0):
                    t1 = -1/2*math.pi
                    t2 = 1/2*math.pi
                elif((xlines[i]-xlines[i-1]) == 0 ):
                    t1 = -1/2*math.pi
                    t2 = math.atan(abs(ylines[j]-ylines[j
                        -1])/abs(xlines[j]-xlines[j-1]))
                elif((xlines[j]-xlines[j-1]) == 0):
                    t1 = math.atan(abs(ylines[i]-ylines[i
                        -1])/abs(xlines[i]-xlines[i-1]))-math
                        .pi
                    t2 = 1/2*math.pi
                else:
                    t1 = math.atan(abs(ylines[i]-ylines[i
                        -1])/abs(xlines[i]-xlines[i-1])) -
                        math.pi
                    t2 = math.atan(abs(ylines[j]-ylines[j
                        -1])/abs(xlines[j]-xlines[j-1]))

            q1 = (xlines[i], ylines[i], t1) #A configuration
                 is (x, y, theta), where theta is in radians,
                 with zero. along the line x = 0, and counter
                -clockwise is positive
            q2 = (xlines[j], ylines[j], t2)
            dub = dubins.shortest_path(q1,q2,turnRad)
        else:
            dub = None
        m.append(dub)
    turns.append(m)
return turns
```

```
def calculateDistanceMatrix (xlines, ylines, turns):
    distance = []

    for i in range (0,len(xlines)):
        m = []
        for j in range (0,len(xlines)):
            if i==j:
                m.append(0)
            elif ((i%2==0 and j%2==0) or (i%2==1 and j%2==1)): #
                turn
                val = turns[i][j].path_length()
                m.append(val)
            elif ((i%2==0 and j-i==1) or (j%2==0 and i-j==1)): #
                required edge
                val = math.sqrt((xlines[i]-xlines[j])**2+(ylines
                    [i]-ylines[j])**2)
                m.append(val)
            else:
                m.append(math.inf)
        distance.append(m)
    return distance

def makeBasicRoute(length):
    z = list(range(0, length))
    for i in range(0, length):
        for j in range(0, length):
            if (i%4==2 and j%4==3 and j-i == 1):
                swap = z[i]
                z[i] = z[j]
                z[j] = swap
    return z

def plotRoute(route, xlines, ylines, xfield, yfield, turns):
    fig, ax = plt.subplots( )
    xx = []
    yy = []
    for i in range (0, len(route)):
        xx.append(xlines[route[i]])
        yy.append(ylines[route[i]])
        if (i+1<len(route) and ((route[i]%2==0 and route[i
            +1]%2==0) or (route[i]%2==1 and route[i+1]%2==1))):
            con = turns[route[i]][route[i+1]].sample_many(0.1) #
                sample rate doesn't change distance
            a = [q[0] for q in con[0]]
            b = [q[1] for q in con[0]]
            for i in a:
                xx.append(i)
            for i in b:
                yy.append(i)

    line, = ax.plot(xx, yy)
    line2, = ax.plot(xlines, ylines, '.')
```

```
    line3, = ax.plot(xfield, yfield)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    plt.show()

def checkRoute(route):
    numPairs = len(route)/2
    for i in range(0, len(route), 2):
        if (abs(route[i+1] - route[i])==1):
            numPairs -= 1
    if (numPairs == 0):
        return True
    return False

def solutionLength (solution, distance, stop):
    length = 0
    for i in range(1, len(solution)):
        if (length>=stop):
            break
        length = length + distance[solution[i-1]][solution[i]]
    return length

def paper41(length, distance, alpha, iterations, x):
    shortestRoute = []
    shortestDist = math.inf
    for i in range(0,iterations):
        route = []
        route.append(random.randint(0,length-1)) #start at
            random node
        for j in range (1,length):
           if (route[j-1]%2==0 and route[j-1]+1 not in route):
                    route.append(route[j-1]+1)
           elif (route[j-1]%2==1 and route[j-1]-1 not in route):
                    route.append(route[j-1]-1)
           else:
                options = []
                for q in range (0, len(distance[j-1])):
                    if (distance[route[j-1]][q]!=math.inf) and (
                        q not in route):
                        options.append(q)
                if (options):
                    opt = options
                    total = 0
                    p = []
                    for k in opt:
                        a =1/(distance[route[j-1]][k])**alpha
                        p.append(a)
                        total = total + a
                    for k in range(0, len(opt)):
                        add = p[k]*(1/total)*x
                        for g in range(0,int(add)-1):
                            options.append(opt[k])

                    ran = random.randint(0,len(options)-1)
```

```
                route.append(options[ran])
            else: #this case should never happen
                print("no options")
                break
    if (len(route)==length):
        if checkRoute(route):
            dist = solutionLength(route, distance,
                shortestDist)
            if (dist < shortestDist):
                shortestDist = dist
                shortestRoute = route
return shortestRoute
```