UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Efficient path planning for multiple agents in agriculture fields

Lior Sinai
M.Sc. Systems and Control
February 2020

Supervisors:
prof.dr.ir G.J.M Krijnen
dr. B. Sirmaçek
dr. M. Poel
dr. A. Kamilaris
N. Botteghi

Robotics and Mechatronics Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Acknowledgements

This report marks the end of my masters in systems and control. Two and a half years ago, I quit my job to study here in the Netherlands, so far away from my home in South Africa. I had not been to the Netherlands before and I was unsure how it would go. I am glad I took this risk, because this masters was a very valuable and enriching experience. It was challenging, but I feel I have grown through it, both through the knowledge I have gained and the experiences I have had.

I am extremely grateful for all those who made this possible. I would like to thank the friends, lecturers and supervisors who have helped me along this journey. I would especially like to thank the following people: my friends from engineering in South Africa; José Esteban Torra; the Jewish community of the Netherlands; Sebastiaan, Toms, Egor and Yuki of Space Society Twente; Mathieu, Sjoerd and the interns at Airbus, and lastly, my family for supporting me throughout this long masters. A special thanks to my brothers Jonty and Rael for their suggestions with this report, and Nicholas for his help with the presentation.

Summary

An active area in research and industry is managing swarms for precision agriculture, as part of a wider move towards increased automation. This report focuses on the particular task of path planning. This problem is NP-hard and is closely related to the vehicle routing problem (VRP) in operations research. Given a known map of an agriculture field, the goal is for the robots to allocate and visit all the targeted crops in an efficient way as a team. Here 'efficient' means minimum distance and time, but it is shown that often both cannot be minimised simultaneously. In particular, using more agents results in a longer total distance but a shorter time overall. Instead a cost function was developed to balance these two metrics. Two environments were developed in Python. A continuous environment with a basic physics model without collisions, which is used to simulate aerial robots, and a grid environment with obstacles, which is used to simulate ground robots moving between crops. The obstacles were static, so Dijkstra's algorithm was used to find the shortest paths around them. This meant the same algorithms could be used in both environments. Exact solutions were found with the Bellman-Held-Karp algorithm, but it runs in exponential time. The focus instead was on more practical approximation algorithms which run in polynomial time. These were: nearest neighbour (NN) ranked, dividing the field into clusters and then using the NN algorithm or Christofides' algorithm, Ant Colony Optimisation (ACO) and Q-learning. Of these, ACO performed the best on the distance and time metrics. However the solution is partly random, which resulted in path crossing - which increases the chances of collisions between agents - and a large spread between the distances agents' travel - which leads to uneven wear and tear. Clustering with NN or Christofides' gave more controlled solutions with comparable metrics.

Contents

List of Figures

List of Tables

List of acronyms

ACO	Ant Colony Optimisation
BHK	Bellman-Held-Karp dynamic programming
CPP	coverage path planning
MDP	Markov decision process
MST	minimum spanning tree
NDVI	normalised difference vegetation index
NN	nearest neighbour
RAM	Robotics and Mechatronics group
RL	reinforcement learning
RNN	recurrent neural network
SLAM	simultaneous localisation and mapping
TSP	travelling salesman problem
UAV	unmanned aerial vehicles
UGV	unmanned ground vehicles
VRP	vehicle routing problem

Chapter 1

Introduction

1.1 Motivation

Precision agriculture is the name given to technology for assessing and responding to local variations in crop health on farms [1]. It involves collecting, interpreting and acting on data. For example, satellites or swarms of robots can take images of crops in a field. These can be used to calculate a metric like the normalised difference vegetation index (NDVI), which indicates whether a region contains healthy vegetation or not.

A current research area is managing robot swarms for precision agriculture [2]. This includes collective exploration, automated collaboration on tasks, communication, and fleet formations. The research in this report focuses on an aspect of collective exploration: path planning through an agriculture field. That is, allocating crops and assigning paths for each robot to follow, in order to complete the tasks in an efficient way as a team.

Two main scenarios are considered. In the first, the goal is to plan paths through the entire field. In literature, this is known as the coverage path planning (CPP) problem [3]. The second scenario is to plan paths through only select points or areas. These points could be preassigned by the farmer or ground workers, or from an image from a satellite or a single high altitude unmanned aerial vehicle (UAV). This scenario is closely related to extensive work on the vehicle routing problem (VRP) in operations research [4]–[6].

These path planning problems are NP-hard. This means that the solution space grows very rapidly with the problem size, and there is no efficient algorithm for finding an exact optimal solution. Therefore, in general, it is best to search for an approximate best solution. For agriculture applications, this should be good enough.

Because this is a new research direction for the Robotics and Mechatronics (RAM) group at the University of Twente, the decision was made to work on high level simulations with a basic physics model. The focus then was on investigating algorithms for path planning. Two models were implemented: a continuous world model, and a grid world with obstacles. Later work could include testing with real robots and more advanced simulations which incorporate robot and sensor data, and decentralised control of the robots.

1.2 Research questions

The aim of this research was to implement various methods for autonomous multiple agent path planning, and to select one which can be used practically in farms or be improved for more complex scenarios. The following research questions were asked:

1. What are the most efficient algorithms for generating paths for a fixed number of particle agents in a 2D environment with known static interest points?
2. How effectively can paths be planned through obstacles representing typical agriculture landscapes?
3. What are the optimal parameters for minimising time and distance travelled for these agents?
4. How does the distribution of the targeted points influence the paths? Specifically, if the task is to visit all points, randomly scattered points, or points scattered in clusters?

The main metric was minimising the distance covered, because this is directly correlated with cost and energy. Other performance metrics included the time to complete the task, overlaps between trajectories and agent utilisation. Overlaps should be avoided because this increases the chance of collisions, and agent utilisation should be evenly distributed for even maintenance. Algorithms were also judged on their processing times.

1.3 Report organization

The remainder of this report is organised as follows: Chapter 2 is a literature review of related research. An analysis of the problem and various methods that are used to solve it are presented in Chapter 3. The design of the 2D test environments in Python and the test cases are presented in Chapter 4. The results of simulations are presented in Chapter 5. Finally, conclusions are reached and recommendations are given in Chapter 6.

Chapter 2

Background

This literature review is divided into three parts. The first part introduces robots used in precision agriculture. The second investigates the coverage path planning (CPP) problem and several applications in agriculture. A common problem with these applications is that the solution is very constrained. The final section focuses on algorithms for solving the general vehicle routing problem (VRP). The VRP is directly related to path planning, and the algorithms are less dependent on the farm geometry. Most of these algorithms are used in chapter 3.

2.1 Robots in precision agriculture

Unmanned aerial vehicles (UAVs) are commonly used for precision agriculture. This has been driven by their increasing durability and decreasing costs. See for example Aerobotics [7] and senseFly [8], two companies that specialise in this field. They are used for imaging tasks such as plant stress detection, nutrient estimation, weed inspection, and for crop management tasks such as chemical spraying [9]. The advantage of UAVs over satellites and aircraft is their higher spacial resolution and lower costs for areas less than 50 ha [10].

Unmanned ground vehicles (UGVs) are also gaining popularity. A commercial example is the Huskey, which was developed by Clearpath Robotics especially for precision agriculture [11]. UGVs have more complex navigation challenges, but they can directly assess and interact with crops. Perhaps because of this more complex navigation, there are many papers on CPP for ground vehicles. This is detailed in section 2.2. Other research has focused on online navigation using GPS and computer vision. See for example Bonadies et al. [12]. However, working with sensor data was not within the scope of this research and is not explored further.

While a single UAV or UGV can be useful by itself, there is also research into using multiple robots. Initially it was thought that solutions for exploration would mostly come from research into swarm robotics. Swarm robotics is the study of controlling multiple autonomous robots separately but achieving a desired collective behaviour [13]. These behaviours include flocking together while travelling, working together to complete a task, or collective exploration. The particular task considered here, path planning, is part of collective exploration. However, this research is not well defined for this task. There does not appear to be a consensus on methods and algorithms. Therefore, the next section investigates the more direct problem of CPP instead.

2.2 Coverage path planning in agriculture

The aim of CPP is to explore an entire area as efficiently as possible [3]. It usually has the additional constraints of simple trajectories to simplify control for autonomous robots. It is an NP-hard problem. There are additional challenges when it is applied to ground vehicles in agriculture. A common constraint is that large vehicles can only turn in the *headland* areas [3], [14], [15]; these are the areas on the edges of the field outside of the main crop regions. For covering an entire farm, there is the extra difficulty posed by the large variety and irregular shapes of farms [16].

Of course, farmers have developed and iterated on their own informal solutions, but there has also been research on determining more optimal paths. For example, Palmer et al. were able to reduce the distance travelled by a sprayer in field by 16% [17]. Other CPP research is detailed in the remainder of this section.

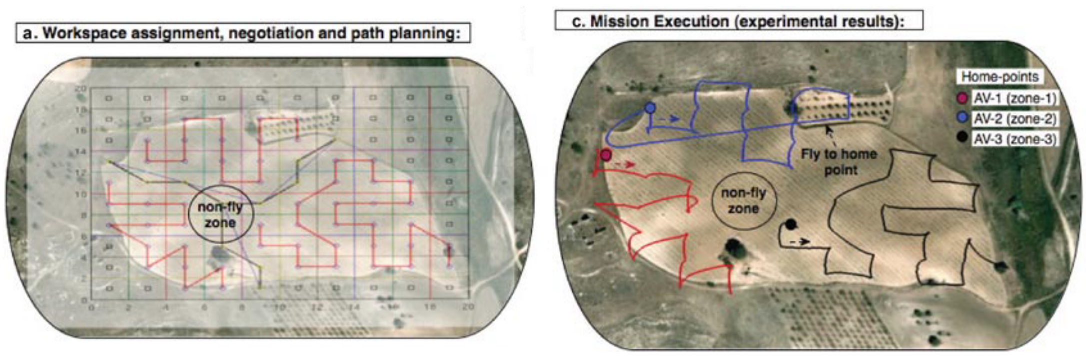

Figure 2.1: Model and execution for CPP with UAVs by Barrientos et al. [18].

There is limited research on path planning for UAVs in agriculture. An exception is the work of Barrientos et al. [18]. They implemented a path planning for a small fleet of three UAVs in a 6.4 ha farm. Their experiment is shown in figure 2.1. They used commercial UAVs but used their own control software to keep to trajectories based on pressure sensors and GPS data. Their CPP algorithm transformed the farm to a low resolution grid, divided the area equally in between the agents, and then used a search algorithm with backtracking to find paths that minimised the number of turns.

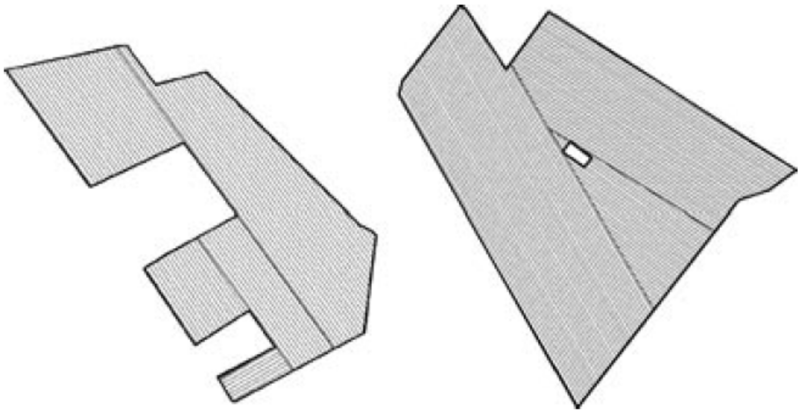

Figure 2.2: CPP for straight edged fields by Oksanen and Visala [3].

There is more research on CPP for ground vehicles; perhaps because this is a more challenging problem. The following papers focus on a single vehicle: Oksanen and Visala [3] developed two algorithms for arbitrarily shaped fields with straight edges. Example solutions are shown in figure 2.2. One algorithm works top-down by first dividing the field into trapezoids, then creates parallel straight line paths and optimises the driving direction. The other works bottom-up by using prediction and brute force search to extend paths throughout the field. They verified their methods on 1500 farms in Finland with an average size of 3.87 Ha. Hameed et al [14] developed an algorithm which also works for fields with curved edges. This algorithm divides the field into convex polygons using a recursive approach, and then generates curved tracks which are parallel to the longest edges. They verified it on 15 farms in Denmark, ranging in size from 0.2 Ha to 44.9 Ha. Bochtis et al [19] developed an algorithm for path planning in rectangular fields with parallel tracks. They transformed the problem into a VRP and used an unspecified VRP solver. They had a detailed model for their cost matrix based on intra- and inter-row operations and turning requirements.

Seyyedhasani and Dvorak [15] developed an algorithm for multiple vehicles. An example solution is shown in figure 2.3. Their algorithm converts the CPP to a VRP. They use a modified Clarke-Wright algorithm to find an initial solution (see section 2.3.2), and 600-700 iterations of a tabu search algorithm to optimise this solution. They have two separate costs, a time cost $cost_{last}$ for the last returning vehicle, calculated as the maximum distance, and a distance cost $cost_{all}$ for all vehicles, calculated as the sum of the distances travelled. They combined these with:

$$cost = z\,cost_{last} + (1-z)\frac{cost_{all}}{m},\ 0 \leq z \leq 1 \tag{2.1}$$

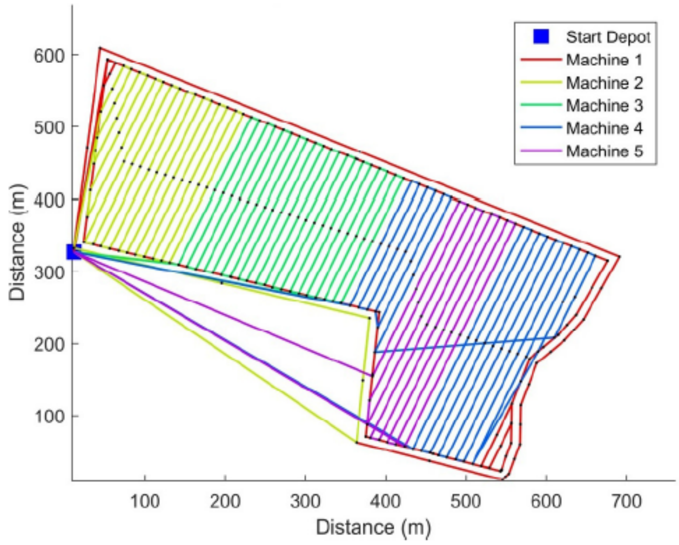

Figure 2.3: CPP for multiple machines by Seyyedhasani and Dvorak [15].

where m is the number of vehicles and is used to create approximate equal magnitudes between the two costs, and z is a weighting parameter. They used $z = 0.8$ for all their tests, because this value ”was found to provide sufficient optimization for all vehicles while still selecting solutions that minimised time to field completion.”

A problem with all these different algorithms is that they are highly constrained solutions. The generated paths cannot navigate around complex geometries. They require the field to be rectangular or made out of simple polygons. The single agent solutions are not easily generalised to the multiple agent case. The decision was therefore made to investigate the more general VRP, because this can offer more flexible solutions.

2.3 The vehicle routing problem

In the travelling salesman problem (TSP), a hypothetical salesman must visit each city once and return to the starting city. The aim is to minimise the distance he travels along this loop. This is an old problem that has been studied at least since the 1930's [4], [20].

The vehicle routing problem (VRP) is a practical extension of the TSP and often includes multiple vehicles. The aim is to minimise a cost which consists of the distance as well as many more variables such as capacity and time restrictions [5], [6]. It can also include dynamic constraints such as fluctuating customer demand [6]. This problem is important for logistics companies, delivery companies and factory management. The path planning problem considered here was interpreted as a type of VRP.

Many algorithms have been developed for solving the TSP and VRP. Several algorithms are explored in the following sections. They are classified as follows [6], [23]:

1. Exact algorithms. These will do a full search for the global minimum cost. For example, for the TSP they will find *the* tour with the shortest length. These algorithms run in exponential time, which is very slow.
2. Approximation algorithms. These are much faster but generally produce a solution with a higher cost than the true optimum.
   (a) Heuristic algorithms. Procedures and rules for finding a good solution.
   (b) Meta-heuristic algorithms. A search process which uses a model to reduce the search space, with some randomness to avoid settling in local minimums and absorb noise.
   (c) Reinforcement learning. A search process where an agent indirectly learns the model and its solution through actions and rewards. The search is less constrained and less biased than meta-heuristic algorithms, but it is more unstable.

The TSP is the simpler problem, but it can be transformed into a VRP by adding additional constraints [21], [22]. For example, a simple transformation from the multi-agent to the single agent case, is to add $m-1$ copies of the origin point for m agents and to remove all but one agent [21]. The distances between these copied origins are set to ∞ or 0 to either prohibit or be indifferent to travelling between them. However, a problem with this transformation is that it introduces a large number of duplicate and degenerate solutions.

2.3.1 Exact algorithms

For n points, a brute force search has the extremely high running complexity of $\mathcal{O}(n!) \approx \mathcal{O}(n^n)$. In comparison other exact algorithms are much faster, even if they are still computationally prohibitive.

The Bellman-Held-Karp (BHK) algorithm is a dynamic programming algorithm with a running complexity of $\mathcal{O}(n^2 2^n)$ and a space complexity of $\mathcal{O}(\sqrt{n}2^n)$[1] [24], [25]. It iteratively solves the following simpler problem: what is the shortest (open) path through a subset of points? If this is solved, the solution can be saved and used to solve the same problem with one more point. The shortest path through the slightly bigger problem will be one of the old solutions plus an extra edge. Figure 2.4 visualises this calculation for a single step.

[1]For a computer with 8GB RAM, this requires $n < 31$, because 8GB= $2^{33} \approx \sqrt{31}(2^{31})$ bits.

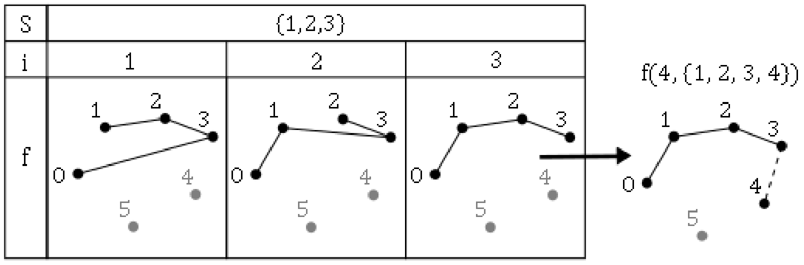

Figure 2.4: Visualisation of a step in BHK [author's image].

Plane cutting is an iterative linear programming technique with a similar run time but a lower space complexity. This was first introduced in 1954 when Dantzig et al. solved a 49 city tour with it [26]. Improvements in this algorithm, along with massive improvements in computing power, have enabled searches for the optimal solution for tours with thousands of cities [20]. The current record is for 85,900 cities, calculated for planning the path of a laser cutter on a circuit board in 2006 [20]. This calculation utilised multiple servers and took over 14 months to calculate the optimal solution (but it only took a few weeks to come within 0.02%).

Both these algorithms are only for the single agent case. They can be extended to the multi-agent case by transforming the problem into a single agent problem as explained above. Even so, these algorithms are unreasonable. Adding an extra point doubles the run time, independent of where it is added. This is why this research focuses on the approximation algorithms in the next section instead. Only BHK is implemented later, because it is the simpler and less abstract of the two.

2.3.2 Heuristic algorithms

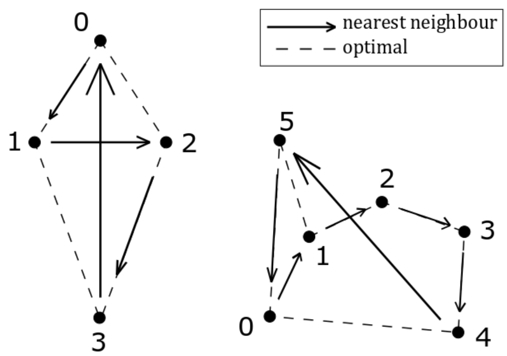

Figure 2.5: Example solutions of the Nearest Neighbour algorithm [author's image].

The simplest heuristic algorithm is the nearest neighbour (NN) algorithm: always go to the nearest point - the nearest neighbour - that has not been visited. It has a complexity of $\mathcal{O}(n^2)$. The solution differs based on the starting point, so it can also be run n times to find the best starting point, giving a complexity of $\mathcal{O}(n^3)$ [4]. On a local level the solution may resemble the optimal solution, but this algorithm is short-sighted and will miss points and backtrack [4], [27]. See figure 2.5. It has a worst case ratio of $0.5\lceil log_2(n)\rceil + 0.5$ to the optimal length [27].

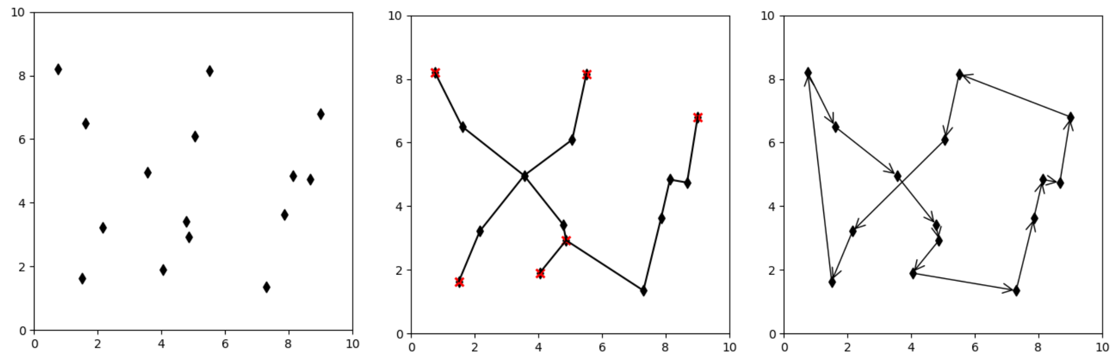

Figure 2.6: Random points (left), their minimum spanning tree (middle) and the Christofides tour (right) [author's image].

The heuristic algorithm with the best guaranteed solution is Christofides algorithm. It always has a worst case ratio of 1.5 or less [28]. It is based on the minimum spanning tree (MST). A spanning tree is a collection of edges that connect all points without creating loops and the MST is the spanning tree with the shortest total length. An example MST is shown in figure 2.6. Unlike the optimal TSP path, this tree can be found efficiently in $\mathcal{O}(n\ log(n))$ time. Also, the optimal TSP path will always be longer than the total MST length. So the MST gives an absolute lowest bound for the shortest TSP path.

A popular algorithm for the VRP is the Clarke-Wright algorithm [29], because it can construct multiple routes and can work with additional constraints. This algorithm calculates *savings* from an origin point, also known as a depot. These savings are the distance reductions for extending the route of a single vehicle against sending separate vehicles. If $d_{i,j}$ is the distance from point i to j, then the savings $s_{i,j}$ relative to the origin 0 is defined as:

$$s_{i,j} = d_{0,i} + d_{0,j} - d_{i,j} \tag{2.2}$$

Paths are then constructed by always adding the segment with the largest savings which has not been used, and which does not violate the other constraints.

These algorithms are developed for the single agent case. This research later investigates extensions for the multi-agent case. For example, a simple idea is to divide the field into segments at the start, so each segment is a single agent problem. A disadvantage of these heuristics though is that the solution is difficult to modify.

For example, it is sometimes necessary to break the NN rule in order to avoid backtracking. While Christofides gives a strong bound for the solution (we know a better solution will never be more than 1.5 times shorter), it is also difficult to modify the solution itself. These algorithms were implemented because they give a good baseline and perform significantly better than random selection. The next section introduces algorithms which are modified and updated through search procedures. This can ultimately lead to better solutions.

2.3.3 Meta-heuristic algorithms

There are many algorithms for the TSP/VRP which use heuristics to reduce the massive search space. These include k-Opt, tabu search, simulated annealing and genetic algorithms [4], [5], [20], [22]. There was limited time to investigate all of these, so the only meta-heuristic algorithm investigated was Ant Colony Optimisation (ACO). This was introduced in 1991 by Dorigo [30].

ACO was inspired by ants and the pheromones (chemicals) that some species lay on a trail. As more ants use the trail, the pheromone will grow stronger and attract more ants. Then, since more ants will tend to travel along the shortest route, the shortest route will tend to grow the strongest pheromone trail.

Algorithm 1 Ant Colony Optimisation

```
procedure ACO
    while termination condition not met do
        ConstructAntSolutions
        DaemonActions                          ▷ optional
        GlobalPheromoneUpdate
```

The high level ACO algorithm is described by algorithm 1. The ants first construct solutions by choosing the next target with a probability rule. Dorigo suggests the following rule for the TSP [30]:

$$p(j|i, J) = \begin{cases} \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in J(t)}[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}} & \text{if } j \in J(t) \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where $J(t)$ is the remaining targets at time t, τ_{ij} is the pheromone value along each edge (i, j) and η_{ij} is the desirability of the target along each edge (i, j). Dorigo uses $\eta_{ij} = \frac{1}{d_{ij}}$ where d_{ij} is the edge length, so that shorter edges are more desirable like with the NN heuristic. The parameters α and β are for tuning. If $\beta >> \alpha$, only η_{ij} is important and this reduces to the NN algorithm. On the other hand, if $\alpha >> \beta$, then only τ_{ij} is important, but this is only useful if a "good" pheromone map has been learnt. This is certainly not the case at the start, and the targets will be random and convergence will be slow. So α and β should be of similar magnitudes to balance these two heuristics. The denominator is a normalising value so that the sum of probabilities is 1.

Next "unnatural" daemon actions can be taken. For example, local search is recommended to improve convergence to the true global minimum.

The final step is to update the pheromone for the next iteration of the algorithm. Pheromone should be deposited on the trail, but also evaporation is needed to prevent convergence to local minimums. Both of these are accomplished with:

$$\tau = (1-\rho)\tau + \Delta\tau \ , \ 0 \leq \rho < 1$$
$$\Delta\tau_{ij} = \begin{cases} \frac{1}{L} & \text{if edge}(i,j) \in P \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

where ρ is the evaporation parameter, $\Delta\tau$ the deposit parameter, P is a relevant path and L is its total length. P could be the current iteration path, the best path so far or the best path in a set of iterations. A variation of ACO called Min-Max ACO also sets minimum and maximum values of pheromone. This helps avoid local minimums and encourage exploration [31].

There have been several applications of ACO to the VRP. They all adapt the original methods of Dorigo for the TSP [30]. Junjie and Dingwei [32] use a transform to the single TSP. They deposit pheromone along all iteration routes. Yu et al. [33] use local search and 2-Opt for the daemon actions. Liu et al. [34] use the Min-Max ACO. They also use the following other modifications: local search; only the global best route has pheromone deposited on it; they randomly sample from the probability distribution with a probability q_0, and take the action with the highest probability with a probability $1 - q_0$.

This algorithm is in many ways similar to reinforcement learning (RL), which is discussed in the next section. In the language of reinforcement learning, the action-value function is given by $Q(S,A) = Q(i,j) = \tau^\alpha\eta^\beta$, the policy is $\underset{j \in J}{arg\,max}(Q(i,j))$ and the learning rate is ρ. RL does not use any other heuristic, whereas ACO uses the NN rule in η. This makes the problems of exploration and exploitation, as discussed with balancing equation 2.3, even more pronounced. However, RL is therefore also less dependent on each specific scenario, whereas a new τ needs to be learnt for every point map. This gives it the potential to handle more challenging scenarios, such as partial visibility (an agent only sees part of the map).

2.3.4 Reinforcement learning

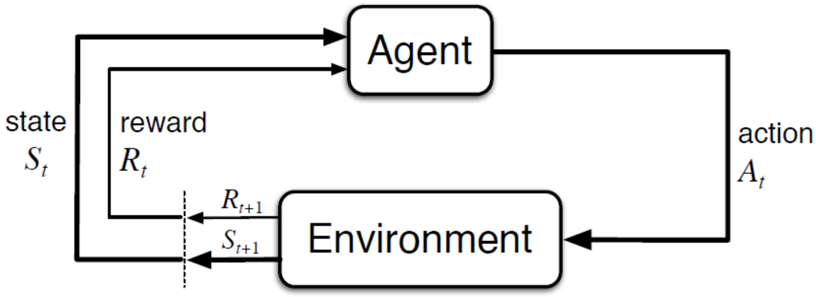

Figure 2.7: The agent-environment interaction in reinforcement learning [23].

Reinforcement learning is a paradigm where an agent learns about an external environment by taking actions and receiving rewards [23], [35]. More formally, the agent takes observations of the current state S_t, uses this to decide on an *action* A_t and then receives a *reward* R_{t+1}. The policy π of the agent for taking actions is modified through simulations to maximise the reward. In contrast to all the above algorithms, the state-transitions are not modelled directly into the policy.

The focus of reinforcement learning is on Markov decision process (MDP). In an MDP, each state S_t of the environment depends only on the previous state S_{t-1}. MDPs are useful for reinforcement learning because there is a direct relationship between action and reward, which make learning policies meaningful. However, MDPs can still be used as an approximation to a non-Markov process, and reinforcement learning has been successfully applied to these problems [23], [35].

Algorithm 2 Q learning

Initialise $Q(S, A)$
for each episode **do**
  RESET(environment)
  Initialise $S_t \in S$
  while S_t not terminal **do**
    $A_t \leftarrow \epsilon$-GREEDY(Q, A_t)
    STEP$((S_t, A_t))$, observe R_{t+1}, S_{t+1}
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t)]$
    $S_{t+1} \leftarrow S_t$; $A_{t+1} \leftarrow A_t$
stop

A popular reinforcement learning algorithm is Q-learning described in algorithm 2 [23], [35]. The main goal is to learn the action-value function $Q(S, A)$. Depending on the state, Q can be encoded as a table/array (like τ in ACO) or as some weighted function for very large states.

Line 7 is the main learning step. The parameter α is used to control the learning rate. There is always an inherent trade-off between *exploration* - trying a new action - and *exploitation* - doing an action that has worked well before. Too much exploration will slow convergence, but too much exploitation will result in a local minimum. Best practice is to start with a large α to encourage exploration, and then reduce it to encourage exploitation and convergence.

The algorithm calls three other functions: RESET, ϵ-GREEDY and STEP. RESET is used to reset the environment to an initial state. Given the current state S_t, ϵ-GREEDY takes a random action with probability ϵ, or takes the policy action $\underset{a \in A}{arg\,max}(Q(S,a))$ with probability $1-\epsilon$. Unlike equation 2.3, this probability does not include any prior information about the model. The STEP function runs another time step of the simulation.

There is very limited research on reinforcement learning for the TSP/VRP. This may be because this process is difficult to model in the above reinforcement learning framework. That said, the following papers are relevant: Gambardella et al. explored the similarities between ACO and Q-learning [36]. They developed an algorithm which combines both and which they claim performs better than ACO. Bello et al. [37] introduced a framework to solve combinatorial problems with neural networks. They focused on the TSP and also a problem called the knapsack problem. They trained a version of the more well known recurrent neural network (RNN) which they call a pointer network, using the actor-critic reinforcement learning algorithm. They achieved tours that were close to optimal for 50 and 100 cities, and outperformed other algorithms such as Christofides. Nazari et al. [38] extended this work to the VRP with capacity restrictions. Their solutions mostly had a 10% or lower deviation to the total optimal length.

RL is the most flexible and general of the algorithms investigated. However, this also made it the most challenging. This is evidenced by the sparse literature on this topic. There is some investigation of RL later in this research, but it was not fully completed. The results section does not include RL at all. It is hoped that this will be investigated at a later stage.

Chapter 3

Analysis and methods

3.1 Analysis

3.1.1 Problem description

The goal is to plan paths for multiple agents through interest points spread over a large area. These agents can either be aerial or ground vehicles. Two environments were created: a continuous world for the aerial vehicles, and a grid world for the ground vehicles. The following assumptions were made to further confine the scope of the problem:

1. Number of agents: there are few agents compared to points/crops.

2. Agent types: there is only one type of agent.

3. Origin: the agents start at a common location.

4. Destination: all agents must return to their start locations.

5. Information: static. The field layout and points of interest do not change.

6. Observability: global. All the field layout and all the points of interest are known at the start.

7. Space: agents move in a 2D space.

8. Interactions: times for tasks at each interest point (imaging/cutting/turning etc) are ignored.

9. Interactions (continuous only): collisions between objects can be ignored.

Most of these assumptions can be relaxed in future work. The first assumption is motivated by the agriculture application and the high costs of robots. It can be removed when modelling many cheap robots. Assumption 5 could be removed to create a dynamic VRP, which is widely researched in literature. Assumption 6 reduces this problem to only localisation and path planning. Dropping this assumption would also make this an exploration problem, which would require simultaneous localisation and mapping (SLAM). This would make most of the algorithms investigated in chapter 2 invalid, because they assume global observability of the environment[1].

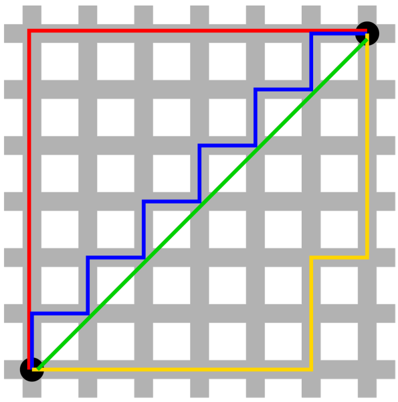

Figure 3.1: Equivalent shortest paths in taxicab geometry (yellow, blue and red) and a unique shortest path in Euclidean geometry (green) [39].

Collisions were easy to implement in a grid world because there is an obvious boundary between the obstacles and free space. In this grid world the agent could only move in 4 directions: left, right, up and down. This is known as Manhattan or taxicab geometry [39]. A consequence of this is that, unlike in Euclidean geometry, the shortest path between two points is generally not unique. In figure 3.1, three equivalent shortest paths are shown, and there are still more. The results for the grid world therefore need to be adapted for a continuous world. However this was not done in this research.

3.1.2 Solution space

It useful to estimate the size of the solution space for the search algorithms. This can be done using the discrete formulations of the TSP with multiple agents. This consists of two separate counting problems: counting the number of ways to allocate points, and then the number of paths through these points. While exact formulas exist for both problems (see appendix A.1), it is simpler to work with approximations to get an estimate of the magnitudes.

[1]The NN algorithm and RL are exceptions.

The number of ways to allocate n objects to m sets is known as a Stirling Number of the second kind $S(n, m)$ [40]. The edge cases are $m = 1$ and $n = m$, where there is only one allocation of points. In between these bounds, $S(n, m)$ grows rapidly with n and m and asymptomatically approaches:

$$S(n, m) \approx \frac{m^n}{m!} \tag{3.1}$$

As an example, $S(15, 5) \approx 250$ million.

For a single agent which starts and ends at the same location, the number of possible paths N through n other points is exactly:

$$N = \frac{n!}{2} \ . \tag{3.2}$$

The factor 2 removes duplicate reverse paths.

For multiple agents finding an approximation is difficult. This number includes one agent doing all the work, so it is necessarily larger than equation 3.2. However, it will still be of a similar order of magnitude.

Equation 3.2 can be modified to get a lower bound for N for m agents. First a transformation is done to the single TSP, by adding $m-1$ origin points and removing all but one agent. Then duplicates are removed as follows:

$$N \geq \frac{(n + m - 1)!}{2^m (m - 1)! m!} \tag{3.3}$$

The 2^m removes reverse routes, the $(m - 1)!$ removes permutations of the copy origin points, and the $m!$ removes permutations of point allocations. N is always larger because there are less duplicates if any agent visits one point (no duplicate reverse path) or no points (less permutations of allocations). If n is much larger than m these effects are less common and the estimate is more accurate.

3.1.3 Cost function

The goals of minimising both distance and time cannot always be accomplished simultaneously. For a single agent these are the same, but it can be faster to have more agents which combined travel further than one agent. This is difficult to prove for a large number of points, but is done easily in the simple problem in figure 3.2.

In figure 3.2, there are two agents and two interest points. Each agent is identical, has a constant velocity of v and starts and ends at the same origin. There are only two solutions: send one agent to visit both points or send each agent to visit a point. In the case of one agent, the costs for the distance L_1 and the time T_1 are:

$$L_1 = d_{0,1} + d_{1,2} + d_{2,0}$$
$$T_1 = (d_{0,1} + d_{1,2} + d_{2,0})/v$$

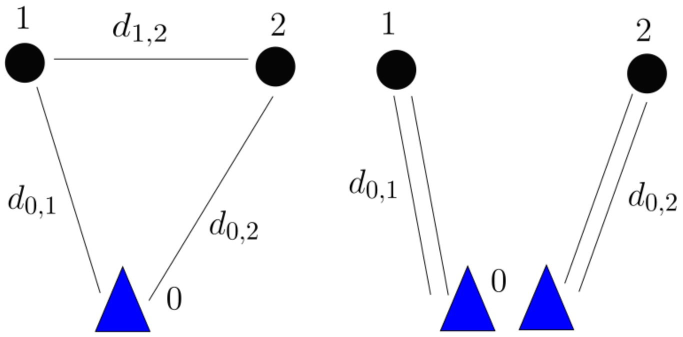

Figure 3.2: A 2 point, 2 agent VRP.

In the case of two agents, the costs are:

$$L_2 = 2(d_{0,1} + d_{2,0})$$
$$T_2 = max\left(\frac{2d_{0,1}}{v}, \frac{2d_{0,2}}{v}\right)$$

Assuming $d_{2,0} \geq d_{1,0}$, the difference between the costs are:

$$L_2 - L_1 = d_{2,0} + d_{1,0} - d_{1,2} \geq 0$$
$$T_2 - T_1 = \frac{d_{0,2} - d_{1,2} - d_{1,0}}{v} \leq 0$$

The inequalities are from the triangle inequality in Euclidean space. Therefore one agent will always cover a shorter distance for two points, but it will always be slower. Therefore these two requirements cannot be minimised simultaneously.

This research will look into balancing these two requirements. For example, compared to UGVs, UAVs may require a more significant weighting on time, because of their more limited flight time. The cost function suggested by Seyyedhasani et al [15] was used (equation 2.1):

$$\begin{aligned} cost &= z\,T + (1-z)\frac{D}{m}, \quad 0 \leq z \leq 1 \\ D &= \sum_{k=1}^{m} d_k \\ T &= \max_{k\in[1,m]} \frac{d_k}{v}, \quad v = 1\ [d/t] \end{aligned} \tag{3.4}$$

Where d_k is the distance travelled by agent k and m is the number of agents. As with equation 2.1, the velocity v is set to 1, so that the maximum distance can be used as a proxy for the time. The agent which travels the longest distance will be the slowest agent, assuming the time for other tasks is ignored.

3.2 Grid world obstacle planning

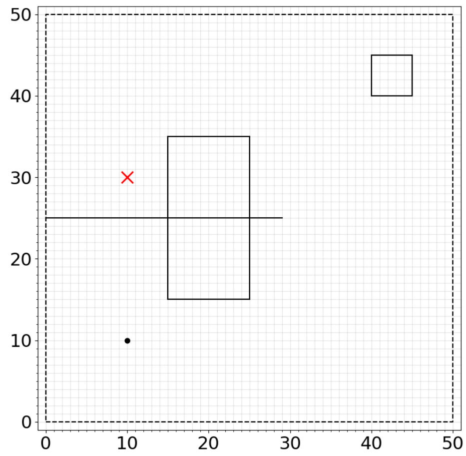

Figure 3.3: Grid world with obstacles. The agent is at the black dot, and the target is the red cross.

Obstacles were implemented in a grid world, but this then introduced the problem of finding the shortest paths around them. For example in figure 3.3, if an agent starts at the black dot, how does it find the shortest path to the red cross around the wall and the box?

Algorithm 3 Dijkstra's algorithm for a 2D grid

procedure DIJKSTRA($start, target, grid$)
    $frontier$.ADD($(start, 0)$)
    for all vertex $v = [i, j]$ **in** grid **do**
        $dist[v] \leftarrow \infty$
    while $frontier$ not empty **do**
        $u \leftarrow$ EXTRACT_MIN($frontier$) ▷ $u = (i, j)$ and is removed from $frontier$
        if $u = target$ **then**
            return solution
        for each neighbour v of u **do** ▷ neighbours are: $(i \pm 1, j)$ and $(i, j \pm 1)$.
            if neighbour **is not** (obstacle **or** boundary) **then**
                $alt \leftarrow dist[u] + length(u, v)$ ▷ $length(u, v) = 1$ for a grid
                if $alt < dist[v]$ **then**
                    $dist[v] \leftarrow alt$
                    $frontier$.ADD((v, alt)) ▷ `ADD` with priority for `EXTRACT_MIN`

Because of the assumptions that the field is static and its layout is known, the agent can plan a path and so Dijkstra's shortest path algorithm was used [41]. This finds the shortest path in a (mathematical) graph. This algorithm exists in many variants. The main idea for all of them is to incrementally explore the graph and repeatedly update estimates for the shortest distances to each node.

Algorithm 3 describes the variation used in this research. It can be visualised as growing a circle around the start node until it reaches the target. The worst case running time is $\mathcal{O}(4rc)$ for a grid with size $r \times c$, for checking all four neighbours for all rc nodes.

A small improvement to this algorithm is the A^* algorithm [42]. This uses a heuristic function to guide the search. In algorithm 3, line 14 is replaced with:

14: $frontier.\texttt{ADD}((v, alt + h(v, target)))$

where $h(v, target)$ is the heuristic function. For a 2D grid, a good choice is the distance to the target if there were no obstacles. This adds a penalty to nodes that are assumed to be far away from the target, so that they moved down the queue of nodes to visit. The distance can either be the Manhattan distance:

$$h(v, target) = |target_x - v_x| + |target_y - v_y| \tag{3.5}$$

Or the Euclidean distance:

$$h(v, target) = (target_x - v_x)^2 + (target_y - v_y)^2 \tag{3.6}$$

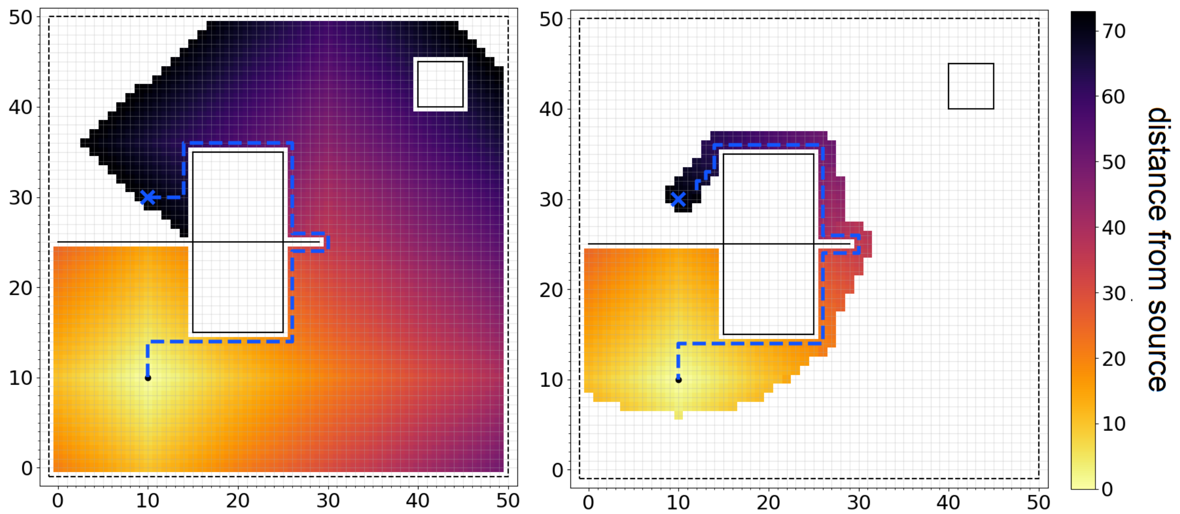

Figure 3.4: Nodes visited for Dijkstra (left) and A^* with Euclidean heuristic (right). White nodes were not visited. Shortest paths are shown with blue dashes.

Figure 3.4 shows that this can greatly reduce the number of nodes visited. For this particular case, Dijkstra visited 1939 nodes, A^* with the Manhattan distance visited 913 and A^* with the Euclidean distance visited 507. The A^* algorithms resulted in a reduction of nodes visited of 53% and 74% respectively. The Manhattan distance performs worse because it is less sensitive to the direction[2].

Dijkstra can also be used to find multiple targets from the same source or even a complete distance map to all points. For this case, the A^* equations have no meaning and are not used. With the distance map, the distance and path from any other point can be read directly. This requires an initial computation cost and a storage cost of $\mathcal{O}(nrc)$ for n points with $r \times c$ sized grids, but it will be much faster for comparing multiple points. This also assumes that the obstacles are constant; therefore agent collisions should be ignored if this approach is used.

A consequence of these assumptions and this approach means that the abstract VRP is the same for the discrete case as for the continuous case. The VRP is essentially to find a minimum combination in a cost matrix, where the cost is a function of the distance. For the continuous case, that distance is implicitly assumed to be the Euclidean straight line distance. For the discrete case, this distance is the Dijkstra path distance, and it has most of the same properties. Therefore the same VRP algorithms can be used to solve both problems.

3.3 VRP algorithms

The following is a summary of the VRP algorithms implemented:

1. Exact algorithm.
   (a) Bellman-Held-Karp (BHK), section 3.3.1 and appendix A.2.
2. Approximation algorithms
   (a) Heuristic
      i. Nearest neighbour (NN), section 3.3.2.
      ii. Christofides' algorithm, section 3.3.2 and appendix A.3.
   (b) Meta-heuristic
      i. Ant Colony Optimisation (ACO), section 3.3.3.
   (c) Reinforcement learning (RL)
      i. Q-learning, section 3.3.4.

[2]With equation 3.5, $|\Delta x| + |\Delta y| = |\Delta x + \Delta y| + |0|$ so diagonal paths are equal to vertical or horizontal paths which pass through the source. But with equation 3.6, $\Delta x^2 + \Delta y^2 \leq (\Delta x + \Delta y)^2 + 0^2$ so diagonal paths are always shorter.

The focus of this research was on the approximation algorithms. The exact algorithm could only be used on small problem sets because of computational limitations.

The heuristic algorithms are the fastest algorithms, because they have a very limited number of iterations. However the solutions are rigid and are not compatible with the cost function, equation 3.4. The meta-heurestic algorithm, ACO, is by design more flexible but also more unstable. It therefore required careful tuning to produce good results. It is much slower and required up to 300 iterations. The solutions are also more chaotic because the exploration is not fully controlled. That is, there are more overlaps between paths and there is a greater spread of the distances' the agents travel. Q-learning is similar to ACO, but the exploration is less biased because it does not use prior information about the model. But this also made it harder to implement.

The rest of this section describes each algorithm in more detail, except for Christofides' algorithm which is in the appendix. This is because it requires additional theory that is not needed for the rest of the work. There is also more detail on BHK in the appendix.

3.3.1 Bellman-Held-Karp dynamic programming

Of the exact methods reviewed in section 2.3.1, only Bellman-Held-Karp was implemented [24], [25]. This is because it is easy to understand, and code was readily available online. The implementation was limited by its very high running complexity of $\mathcal{O}(n^2 2^n)$ and space complexity of $\mathcal{O}(\sqrt{n}2^n)$. For this research, a computer with 8GB of memory was used, and only a maximum of $n = 23$ was possible. In hindsight, the space complexity was too great a limiting factor, and it is recommended that later research uses plane cutting instead.

For the m agents, two methods were investigated:

- Method 1: Transform to a single agent problem, as discussed in section 2.3. This increases the space and time complexity by a multiple of $\mathcal{O}(2^{m-1})$.
- Method 2: Solve the single agent problem for all subset of points (this can be done by "closing" the loops in the single agent iterations). Then search through allocations of these subsets to find the lowest combined cost. The space complexity increases slightly but the running complexity can be much lower than method 1.

For method 2, equation 3.1 shows that the number of allocations is very large, but not all allocations have to be checked. For example, if the first agent makes a higher cost than the current lowest total cost, all allocations for the other agents can be skipped. A recursive back-tracking algorithm was implemented to do this.

For more detailed explanations, see appendix A.2.

3.3.2 Heuristic algorithms

Nearest Neighbour

The Nearest Neighbour (NN) algorithm is to always target the nearest point - also called the nearest neighbour - that has not been visited. As explained in chapter 2.3.2, this strategy is short-sighted and is not optimal. None the less, it provides a basic benchmark that performs significantly better than randomly selecting targets.

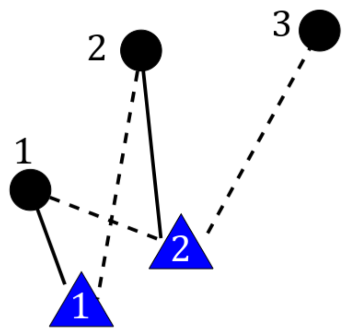

Figure 3.5: Visualisation of the NN ranked algorithm. Agent 1 is sent to point 1 because of the NN rule; agent 2 is sent to point 2, the next nearest point after point 1.

This algorithm does not work well with multiple agents because if the agents start at the same point or meet at a point, they will effectively have the same trajectories afterwards. This is a very inefficient use of resources. A better heuristic is to rank the agents by their distances to each point, and send each to a different target based on this ranking. An example is shown in figure 3.5.

Algorithm 4 Nearest neighbour ranked algorithm

procedure NEARESTNEIGHBOURRANKED(n, m)
  while unvisited points **do**
    for $k = 1, ..., m$ **do**
      Find point $j \in$ not visited with the nearest agent a_k
      Set target for agent a_k as point j
      Remove agent a_k and point j from the selection pool

Algorithm 4 is a slightly more efficient implementation of this idea; it finds the m points with the closest agents and sends the agents to each. It was designed to run in real time with a simulation. It re-allocates targets every time a point is visited.

This is inefficient because an agent will have wasted time if its target is changed. A better version could plan ahead from the start.

Clustering

Another heuristic is to divide the map into m clusters at the very start. Each agent is then given a cluster and can only target points in this cluster. Then any single agent algorithm can be used. If the NN algorithm is used, any potential backtracking is reduced, so performance should be improved. If Christofides' algorithm is used, then within each cluster each agent will travel a path that is never more than 1.5 times the optimal path length (see appendix A.3).

The choice of clustering algorithm is highly dependant on the distribution of the points [43]. For the use cases that will be introduced in chapter 4, the following clustering algorithms were found to be useful:

- Evenly dividing the exploration space into m sectors or rectangles.
- K-means clustering using 2D Cartesian co-ordinates.
- K-means clustering using polar co-ordinates.

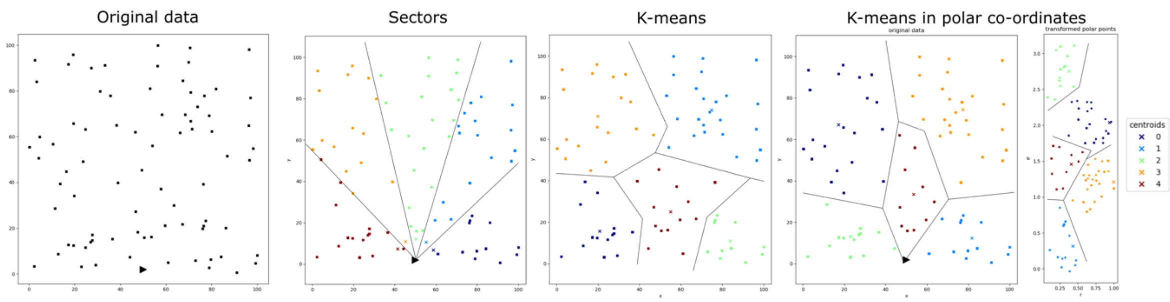

Figure 3.6: Clustering of 80 points into 5 clusters, with an origin at the bottom.

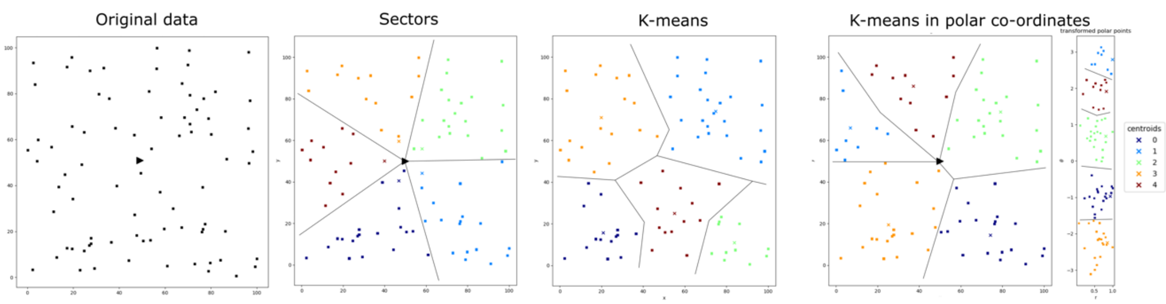

Figure 3.7: Clustering of 80 points into 5 clusters, with an origin in the centre.

Examples of these are shown in figures 3.6 and 3.7. The origin point is in the bottom in the first figure and in the centre in the second figure. The K-means clusters are the same for both figures, because it is independent of the origin.

Evenly dividing the area is a simple but effective strategy. This mostly removes overlaps between paths. For uniform geometries and points, it will also evenly divide the work between the agents. This strategy is not suited for complex geometries or an uneven distribution of points. There are many more suitable clustering algorithms for points in 2D space [43]

K-means is one such algorithm. It is is an iterative clustering algorithm that finds K centroids μ_k that minimises the following error term [43]:

$$\sum_{i=0}^{n} \underset{k \in \{1,\ldots,K\}}{min} (|p_i - \mu_k|^2) \tag{3.7}$$

where p_i is the co-ordinate of each ith point. The centroid μ_k is recursively defined as the mean (average) point of all the points where it used, hence the need for iterations. This clustering algorithm assumes that the data is spherically distributed and fails when it is not [43]. This assumption should result in clusters of similar size, and therefore paths of similar lengths between the agents.

K-means by itself does not take into account the origin point (x_0, y_0), so it was also used with polar co-ordinates. The aim with this strategy is to reduce the start and return journeys. This is done by setting $p = (r, \theta)$ in equation 3.7, where r and θ are defined as:

$$\begin{gathered} r = \sqrt{(x - x_0)^2 + (y - y_0)^2} / r_{max} \\ \theta = arctan\left(\frac{y - y_0}{x - x_0}\right) \end{gathered} \tag{3.8}$$

To follow the spherical criterion, the radii were normalised with r_{max} so that all values were between 0 and 1. The angles varied from $-\pi$ to π; these were not normalised to create a bias on the angle (more spread out angles have a bigger effect than more spread out radii).

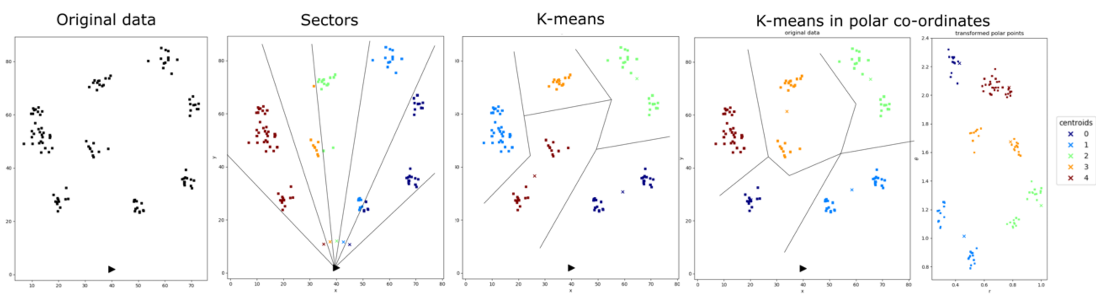

Figure 3.8: Grouping 9 clusters into 5 clusters, with an origin at the bottom.

Figure 3.8 shows that these algorithms can find and group smaller existing clusters. The sectors method performs poorly. It splits three existing clusters. But the results from both K-means clustering methods are good, and differ only with the middle cluster. For path planning, the polar co-ordinates version is probably better, because the agent going to the middle cluster should also go to the cluster at the far side.

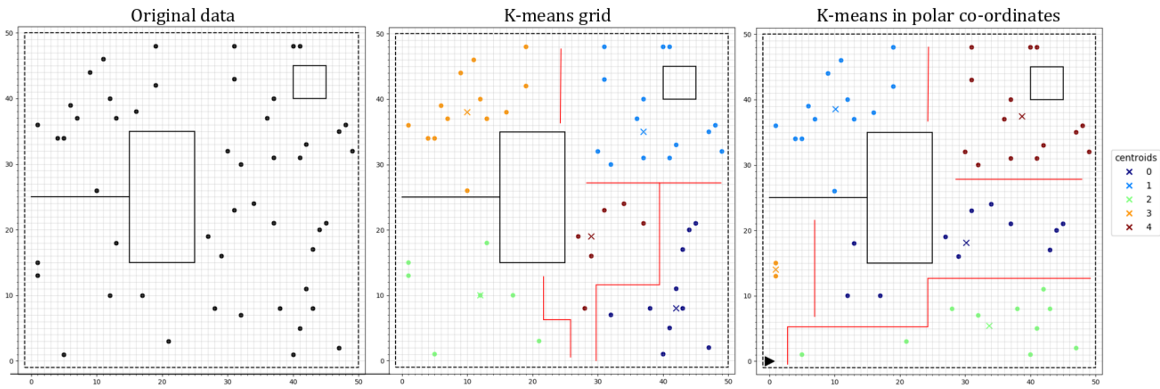

Figure 3.9: Clustering of 50 points into 5 clusters on a grid.

K-means still works on a 2D grid. See the examples in figure 3.9. The minimum distance can be found using Dijkstra's algorithm - see section 3.2. A problem is that the mean point can lie on an obstacle or not be a discrete point. In both of these cases, the distance for equation 3.7 is undefined. The centroid can however be defined as a (non-unique) grid point that minimises the sum of distances to all the points in its cluster. It can be found directly using the distance maps described in section 3.2. It is a (non-unique) minimum value in the sum of these matrices.

The grid points can also be transformed to polar co-ordinates. The only difference is that r should be defined as the shortest path to the origin, which can be found using algorithm 3. Then, because this transformation already takes the distance around obstacles in to account, the continuous space K-means algorithm can be used. An example is also shown in figure 3.9.

3.3.3 Ant Colony Optimisation

Algorithm 5 describes an adaption of ACO from section 2.3.3 for multiple agents. The steps from the high level ACO algorithm 1 are in comments. As with the NN Ranked algorithm above, it was designed to run in real time. Therefore one agent can visit multiple points in the same time another travels to only one point. This is a different approach to the reference papers in section 2.3.3. Agents are allowed to return to the origin early, in case the best solution requires some agents doing more work than others.

Two versions were implemented. In the first, all the agents share and update the same τ vector. In the second, each agent updates its own τ^k vector. An example of

Algorithm 5 Ant Colony Optimisation algorithm for multiple agents

1: **procedure** ACO($points, m$)
2:     $(P^*, c^*) \leftarrow (\emptyset, \infty)$ ▷ Initialise best solution
3:     **while** not converged **do**
4:         RESET(environment)
5:         **while** unvisited points **do** ▷ ConstructAntSolutions
6:             **for** $k = 1, ..., m$ **do**
7:                 select ant a_k with a target point i
8:                 **if** point i visited *and* ant is active **then**
9:                     add point i to path P_k
10:                     select next target j with probability $p(j|i, J)$ ▷ See eq. 2.3
11:                     **if** j=0 *and* number active ants > 1 **then** ▷ DaemonActions
12:                         deactivate this ant
13:                 send ant a_k to target
14:         **if** $cost(P) < c^*$ **then**
15:             $(P^*, c^*) \leftarrow (P, cost(P))$ ▷ Update best solution found so far
16:         **for** $k = 1, ..., m$ **do** ▷ GlobalPheromoneUpdate
17:             **for** edge(i, j) **in** P_k **do**
18:                 $\tau^k_{ij} \leftarrow (1 - \rho)\tau^k_{ij} + \Delta\tau_{ij}$ ▷ See eq. 2.4

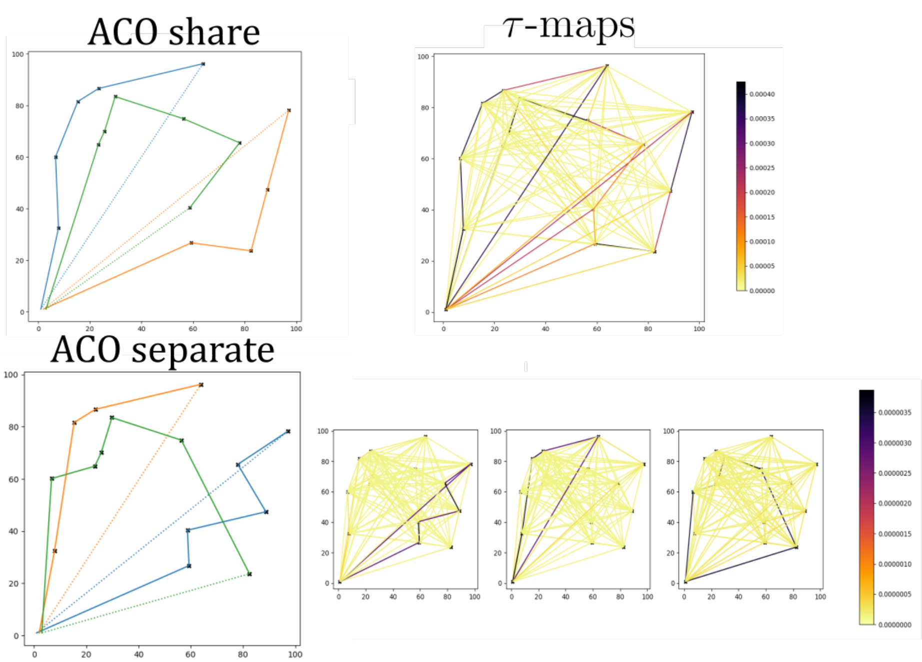

Figure 3.10: ACO with shared or separate pheromone maps, $z = 1$.

each is shown in figure 3.10. The results for both tended to be very similar.

For the pheromone update (line 18 of algorithm 5), only the lowest cost c^* from the best paths P^* received a positive pheromone update. This choice speeds up convergence but also increases the risk of converging on a local minimum. To limit this, maximum and minimum values for τ were set. These are:

$$\tau \in \left[\frac{1}{\rho c^*}, \frac{2}{\rho c^* n}\right] \tag{3.9}$$

These limits are recommended by [31] and should encourage exploration.

3.3.4 Q-learning

Q-learning was implemented using algorithm 2. Only the single agent case was investigated. The simplest action space was chosen: the agent must choose which point to target next. Hence the action space has a size n. Valid actions are targets which have not been visited and the agent must choose from these. The reward is then the negative distance between the two points. This is simpler than for example making the agent choose which direction to move in. In that the case, the agent would have to learn how to move and where to move.

Table 3.1: Different states for RL

	State	Size	Description
1	$(p_1, ..., p_k)$	$\sum_{k=1}^{n} k! \approx \mathcal{O}(n^n)$	k points visited with an order.
2	$(i, \{p_1, ..., p_k\})$	$\sum_{k=1}^{n} k\binom{n}{k} = \mathcal{O}(2^n)$	At point i and k points visited with no order.
3	(i, k)	n^2	At point i and k visits so far.
4	(i)	n	At point i.

The next choice was to choose the state space. Different options are shown in table 3.1. The full state is state 1, which is the points that have been visited and the order they were visited in. This state is huge and is difficult to explore and store. Therefore smaller states were investigated.

State 2 also stores the points that been visited but discards the order. This is still an MDP, because the next action depends only on the last visited point and the other points that have been visited. This state is used in BHK in section 3.3.1. The advantage of Q-learning is that it could potentially be represented by a much simpler and less costly Q function - e.g. a recurrent neural network (RNN) - and it could be learnt faster. A disadvantage is that finding the optimal solution, or indeed even a good solution with known error bounds, is no longer guaranteed. Regrettably, there was not enough time to investigate this further.

States 3 and 4 are very small, but are bad approximations of the MDP. This is because these states only store the current point, but as mentioned above, the action is limited by other points that have been visited too. This is implicit in the `STEP` function, which discards invalid actions. These states are therefore not good representations and they cannot learn long term strategies. The results were not competitive and are not included in this report. State 4 only stores the current point, so the best strategy that can be learnt is the NN algorithm. State 3, which additionally tracks the number of points visited, does not perform much better.

Chapter 4

Design and implementation

4.1 Model design

4.1.1 Software selection

After reviewing various agent modelling software, the decision was made to write custom code. The motivation was:

- The model is simple. It is confined to 2D and the agents are modelled as point particles. No sensor data with measurement noise is taken into account. No terrain or crop interactions are taken into account. The only object interactions are collisions.
- This allowed flexibility with the algorithms. Different paradigms such as reinforcement learning and meta-heuristics could be used in the same code.
- A clear separation was made between rendering visuals and the rest of the code. Visuals slow the code significantly and therefore were suppressed when running multiple iterations.

The code is written in Python 3.7. Python was used because it is an popular, dynamic language with many scientific libraries and packages. A disadvantage is that it is not as memory or time efficient as other languages.

Appendix A.4 details packages and code sources that were used.

4.1.2 Continuous model

The agents were modelled as point particles. Their state was given by their 2D position co-ordinates (x, y) and an orientation θ. Two dynamic models were implemented: zero-order (direct) and first-order (velocity). The former is simpler and means that the focus is only on the path finding problem. The latter is useful for more realistic tests. All agents were assigned a maximum linear velocity v_{max} and a maximum angular velocity ω_{max}. Each model was incremented by a time step of size Δt.

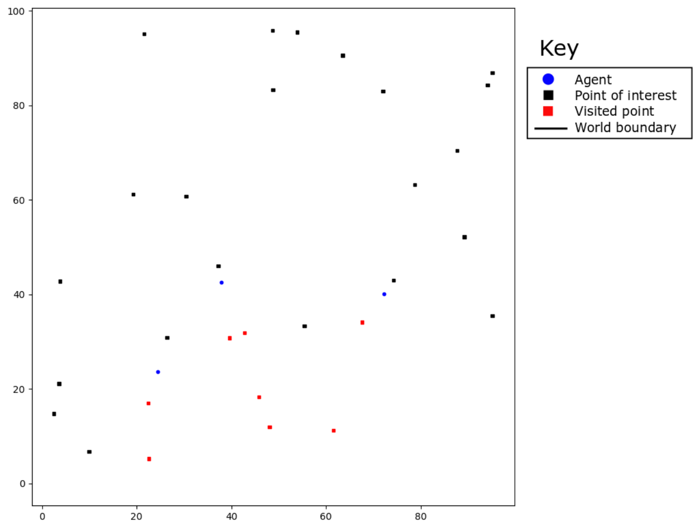

Figure 4.1: Screenshot of the continuous model. Note: agent orientations are not rendered.

For the zero-order dynamics, the control input is $u = (u_1, u_2)$ for each co-ordinate direction. These are bounded between -1 and 1. The next agent state was calculated as:

$$\begin{aligned} x_{t+1} &= x_t + v_{max} u_1 \Delta t \\ y_{t+1} &= y_t + v_{max} u_2 \Delta t \\ \theta_{t+1} &= arctan\left(\frac{u_2}{u_1}\right) \end{aligned} \tag{4.1}$$

This control tended to make the agent follow straight lines. The effective angular velocity is ∞.

For the first order dynamics, the two control inputs were for the linear and angular velocities respectively. They were also bounded between -1 and 1. The next agent states were calculated as:

$$\begin{aligned} x_{t+1} &= x_t + v_{max} u_1 cos(\theta_t) \Delta t \\ y_{t+1} &= y_t + v_{max} u_1 sin(\theta_t) \Delta t \\ \theta_{t+1} &= \theta_t + \omega_{max} u_2 \Delta t \end{aligned} \tag{4.2}$$

This control tended to make the agents follow curved paths.

A detect radius r_{detect} was defined for each agent. Each point was marked as visited if the distance between any agent and that point was less than this:

$$|(x, y)_{agent} - (x, y)_{point}| \leq r_{detect}$$

The simulations were run until all points were visited. Return journeys distances and times were estimated afterwards.

The agents were restricted to a boundary rectangle. All commands to move past this rectangle were ignored.

4.1.3 Grid world model

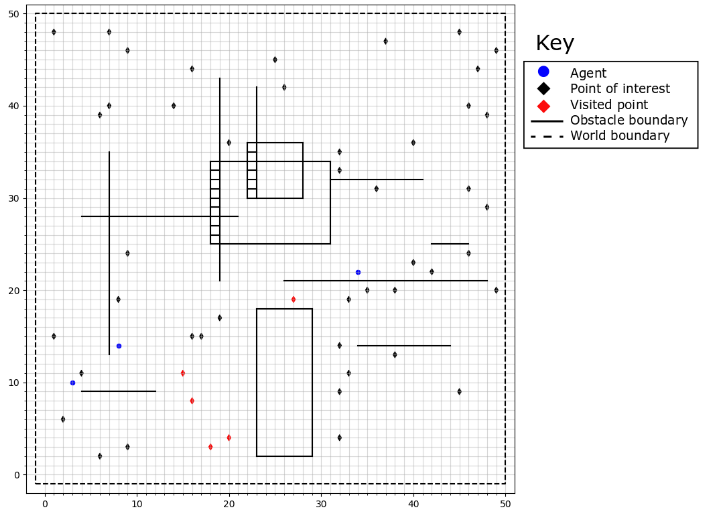

Figure 4.2: Screenshot of the grid model.

The grid was encoded in a 2D array. All agents and points of interests could occupy 1 cell, and obstacles could occupy 1 or more cells. The value of each cell was set based on the object type. This is shown in table 4.1. The agents and points of interest were each given a unique index, and this was set to the value of their cell. The obstacles were given an arbitrarily large index to separate them from agents.

Object	Cell size	Value
Free space	any	0
Point of interest	1	$-$(index+1)
Agent	1	$+$(index+1)
Obstacle	any	$x >> m$

Table 4.1: Keys for the grid.

A collision with an obstacle was defined as an agent attempting to move on to a grid cell with a value greater than some threshold value. This creates flexibility in the code: if the threshold is greater than the number of agents m, agent collisions were ignored. If it was 0, they were not.

A visit with a point was defined as an agent moving onto a cell with a point index. That is:

$$|(x, y)_{agent} - (x, y)_{point}| = 0$$

When an agent moved into a cell, that cell was set to the agent index, regardless of what was in it. Therefore redundant lists were also kept for the agents, points of interest and obstacles. This allowed values to be reset e.g. after an agent moved from a point into free space. This also made look-ups faster - an agent could be found in the agent list rather than looking for it in the grid.

Only a zero-order dynamic model was used. The agents could only move one grid step in one direction per time step. This restriction was to enable diagonal paths. The control input is therefore $u = (\pm 1, 0)$ or $u = (0, \pm 1)$. The next agent states were calculated as:

$$\begin{aligned} x_{t+1} &= x_t + u_1 \\ y_{t+1} &= y_t + u_2 \end{aligned} \tag{4.3}$$

Simulations were run until all points were visited, and return journeys and times were estimated afterwards.

4.2 Experimental design

4.2.1 Algorithm tests

The algorithms were tested on random environments. This was to test their robustness in different scenarios.

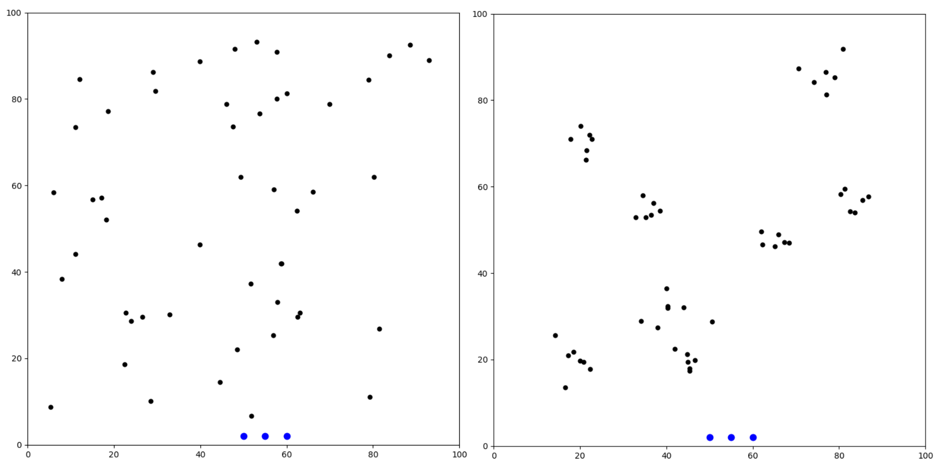

Figure 4.3: Uniformly random points (left) and clustered random points (right).

The tests for the continuous environment were points randomly scattered throughout the area, and points randomly scattered in clusters. The agents always start at the bottom in the middle. Example layouts are shown in 4.3. The clusters were made with the `make_blobs` function from Scikit-learn [43]. This function was initiated with a small set of uniform randomly distributed centroids. Then for each centroid, n_{blob} points were placed based on a normal distribution with the centroid as the mean and the standard deviation as 3 units. The detect radius was set to 1 unit, and the agent velocity to 15 units/unit time.

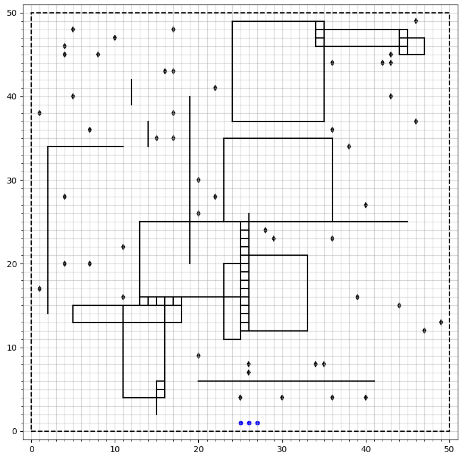

Figure 4.4: Uniform random obstacles and points.

For the grid test, obstacles were randomly placed around the grid. They were combinations of lines and boxes. These restricted access from 15% to 35% of the grid. An example grid is shown in figure 4.4. Points were uniformly distributed around the grid. Points that landed inside obstacles, on obstacle edges, or that were completely surrounded by obstacles, were randomly moved elsewhere. The agents also always started at the bottom in the middle.

All the ”random” objects and choices were made with a `numpy` pseudo-random number generator. The generator is initiated with an integer called a *seed*. This seed can be saved and used to recreate all subsequent ”random” number sequences after it. In this way, the same set of randomised scenarios were made for each algorithm. The OpenAI implementation was used, which hashes seeds to avoid linear correlations between pseudo-random seeds from successive iterations. See [44] for a full explanation.

4.2.2 UAV tests

The UAV tests used the continuous particle environment with no obstacles. These tests addressed the scenario of planning paths through selected regions. These regions would be identified by farmers with another technology. For example, regions with a low NDVI. Points were randomly distributed throughout a shape, or randomly distributed in clusters. It was assumed there was no need for placing the crops in regular patterns because the selected regions will be irregular and the UAVs are not constrained by paths on the ground. Regular patterns are considered in the next section with the grid world.

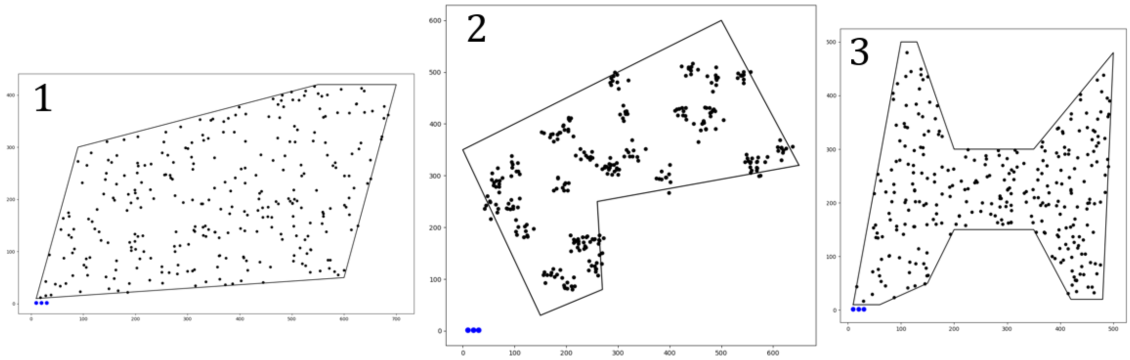

Figure 4.5: Fields for the UAV tests with randomly scattered points of interest.

Three shapes were developed, and are shown in figure 4.5. All shapes fit with a square of length 0.7km. The first shape is a simple concave shape similar to a rectangle. The other two are more complex convex shapes. Table 4.2 shows relevant properties for these fields.

Table 4.2: Properties of the fields for the UAV tests.

Field	Shape	Area A (ha)	Perimeter P (km)	IQ
1	rectangle	19.98	1.9	0.70
2	L	16.19	1.93	0.55
3	H	12.66	2.2	0.33

The isoperimetric quotient IQ is given by [16]:

$$IQ = \frac{4\pi A}{P^2} \tag{4.4}$$

where A is the area and P is the perimeter. It measures how close a field is to a circle. For reference, it is 1.00 for a circle, and 0.79 for a square.

The parameters were scaled to match real agriculture farms. The agents always started near the origin at (0,0)m. Their detect radius was set to 1m, and their maximum velocity to 10m/s (36km/h). The world was incremented in time steps of 0.2s.

The chosen speed is easily achievable speed for commercial UAVs. However, sometimes tasks require the UAVs to go much slower. For example, Matese et al. used a speed of 4m/s (14.4km/h) in their experiments on a 2.5ha field [10]. In those experiments, the UAV was used to capture images for NDVI analysis.

4.2.3 Grid world agriculture fields

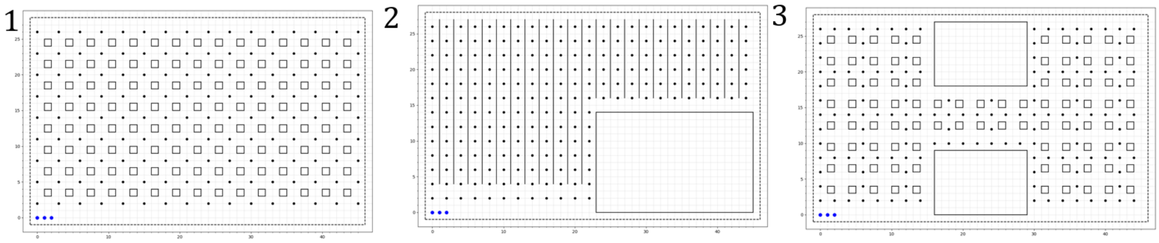

Figure 4.6: Grid fields for the UGV tests (scale is 1 units:3m).

The UGV tests used the grid world particle environment with obstacles. The purpose of these tests was to plan paths through regular patterns that mimic real agriculture fields. Three shapes of fields were used, which are grid versions of the UAV fields. These are shown in figure 4.6 and properties are listed in table 4.3. Two types of obstacles were used: lines, to represent crop rows, and squares, to represent crops that are planted in bushes or trees. Points were uniformly distributed throughout the shape to force the CPP problem.

Table 4.3: Properties of the grid fields for the UGVs tests

Field	Shape	A (unit^2)	P (unit)	IQ	Obstacle	Number	A total (unit^2)
1	rect.	1288	148	0.74	Lines	22	528
					Squares	120	480
2	L	928	148	0.53	Lines	22	396
					Squares	88	352
3	H	1008	188	0.36	Lines	23	416
					Squares	90	360

For visual purposes and to reduce run times, the fields were kept small and the grid density was kept low. Because the patterns in the fields are very regular, it is assumed that results will hold for larger fields. No scaling parameters were set in the code. However, based on the spaces between crops in real agriculture fields, each grid unit represents $2.5m$ [10]. The fields fit in a grid of 46×28 units, which therefore scales to $115m \times 70m = 0.805\ ha$. This is about 15-24 times smaller than the fields designed for the UAVs. It was assumed that because the geometry of the fields is regular, these results can be scaled up to larger fields.

Chapter 5

Results

The following settings are relevant to each algorithm:

- For the optimal solution, the cost function was set as equation 3.4.
- The parameters for ACO for the continuous world were: $\alpha = 3, \beta = 5$ and and $\rho = 0.01$. These were set because they performed well with ad hoc experiments. Since $\beta > \alpha$, this is more biased towards the NN heuristic than random exploration. Only the ACO shared method was used. The cost function was set as equation 3.4. For the grid world fields, β was changed to 4 because the results were slightly better (see appendix A.5).
- For most graphs, only the best performing combination of clustering and VRP heuristic is shown. This is referenced in the legend. Generally, NN with clustering performed worse than Christofides with clustering.
- Occasionally a single z value was chosen to reduce experiments. For these, $z = 0.5$ was chosen as a compromise between aiming for a minimum distance and aiming for a minimum time.

Note: NN ranked has no settings.

5.1 Randomly distributed points

Only the results for the continuous environment are presented in this section. This is because the results for the continuous and grid environments were similar, despite the more complex challenges of obstacles in the grid environment. The grid world results are in appendix A.5.

5.1.1 Trends with the optimal solutions

Although the optimal solution was limited to working with less than 23 points, it is useful for assessing trends. There is no noise in the solution, as with ACO or Q-learning, or local optimums as with all the approximation algorithms. But these trends do hold for the other algorithms, with more points, with obstacles, and with different point distributions, as is evidenced in the other sections.

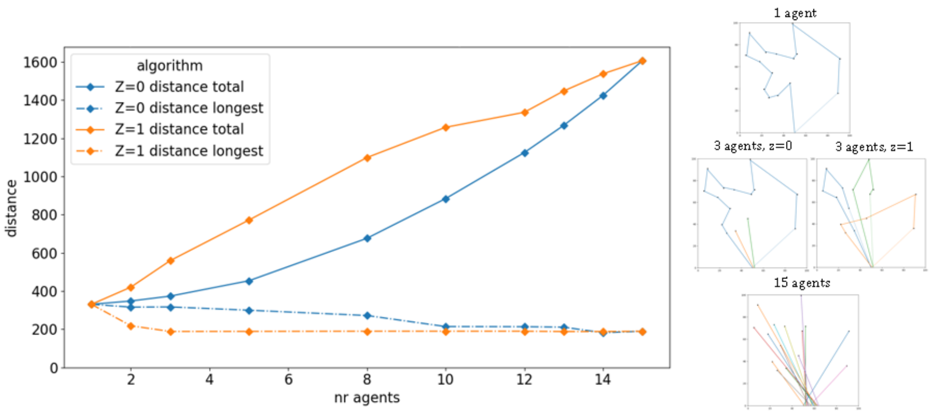

Figure 5.1: Optimal solution for one set of 15 uniformly random distributed points (left). Examples of solutions (right).

Figure 5.1 shows the variation of the solution with the number of agents for a random distribution of 15 points. Solutions are plotted for two parameters of z in equation 3.4: $z = 0$ and $z = 1$. Two parts of each solution are shown: the total distance D with solid lines and the maximum distance T with dash-dot lines.

The solutions in the graph go from one agent doing all the work, to dividing it between multiple agents, to the impractical case of one agent for each point. For each solution (blue or orange), the total distance increases with the number of agents, while the maximum distance decreases. The difference between the blue and orange graphs shows that optimising for only the total distance ($z = 0$) results in a longer maximum than when optimising only for the maximum, and vice versa. Therefore, as discussed in section 3.1.3, time and distance cannot be minimised simultaneously.

If they cannot be minimised simultaneously, the next question is how best to balance them. This was investigated with figure 5.2, which shows the variation of z in equation 3.4 with a fixed number of agents and points. As z increases, the total distance steadily increases. The maximum distance decreases sharply until $z = 0.4$, then decreases slowly until $z = 1$. From this it can be concluded that it is better to set $z \leq 0.6$ or even $z \leq 0.4$, because there is little gained from a higher z value. ACO and Q-learning can both be used with the same cost function, so this result is

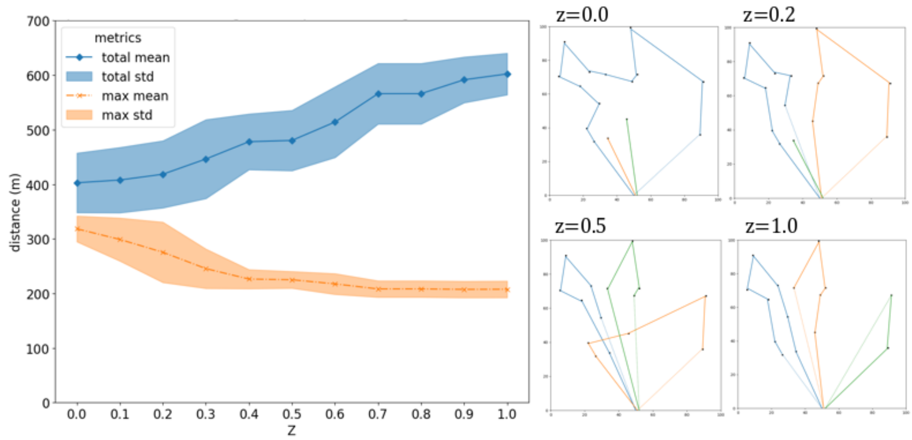

Figure 5.2: Effect of z on the optimal solution, for 3 agents for 15 points, averaged over 10 sets of points (left). Examples of solutions (right).

relevant for them.

5.1.2 Comparison between algorithms, uniformly random points

Figure 5.3 compares the algorithms with a fixed number of agents over a varying number of points. For each algorithm except the optimal solution, the value for both the total distance of the agents and the maximum distance between the agents is shown. For the optimal solution, only the more important metric is shown. For $z = 0$, this is the total distance, and for $z = 1$, it is the maximum distance.

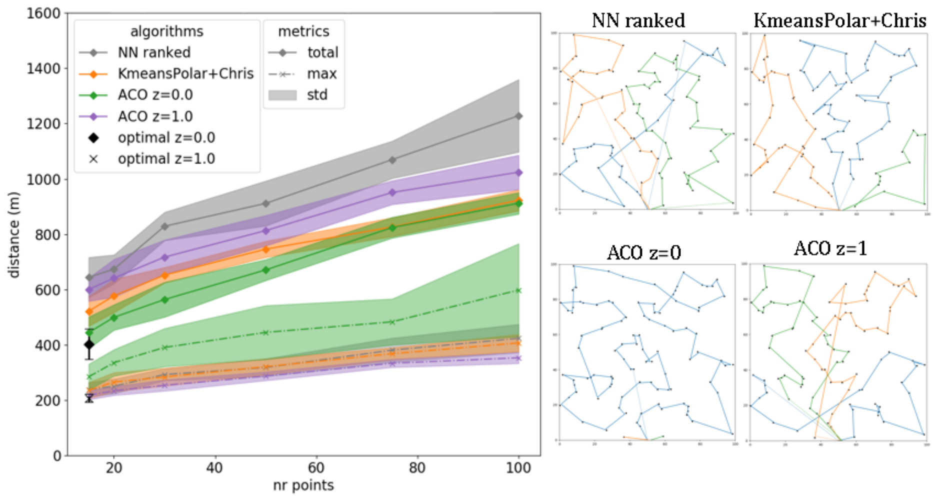

Figure 5.3: Comparison between algorithms for 3 agents, averaged over 10 runs. Examples right.

For all algorithms, the cost increases with the number of points. This was to be expected. None of the algorithms perform as well as the optimal solution for $n = 15$ points. Generally, NN ranked - the simplest and fastest algorithm - performs the worst, and ACO performs the best. However the ACO solution tends to have many crossing paths between agents. If it is desired to have paths that do not cross, it might be better to use the clustering method with Christofides algorithm. Compared to ACO, Christofides has a total distance that is up to 20% longer, and a maximum distance that is up to 6% longer.

Also, as can be seen by the examples in figure 5.3, ACO tends to drop agents when $z = 0$. This is because it finds that using less agents results in a lower total distance, as concluded in section 5.1.1. ACO is also capable of self-clustering with higher values of z. For the example in the figure, the points are almost in three distinct clusters, with some overlaps. However, Christofides with clustering performs relatively well, and its clusters have no overlaps. This avoids collisions between the agents.

5.1.3 Comparison between algorithms, clustered points

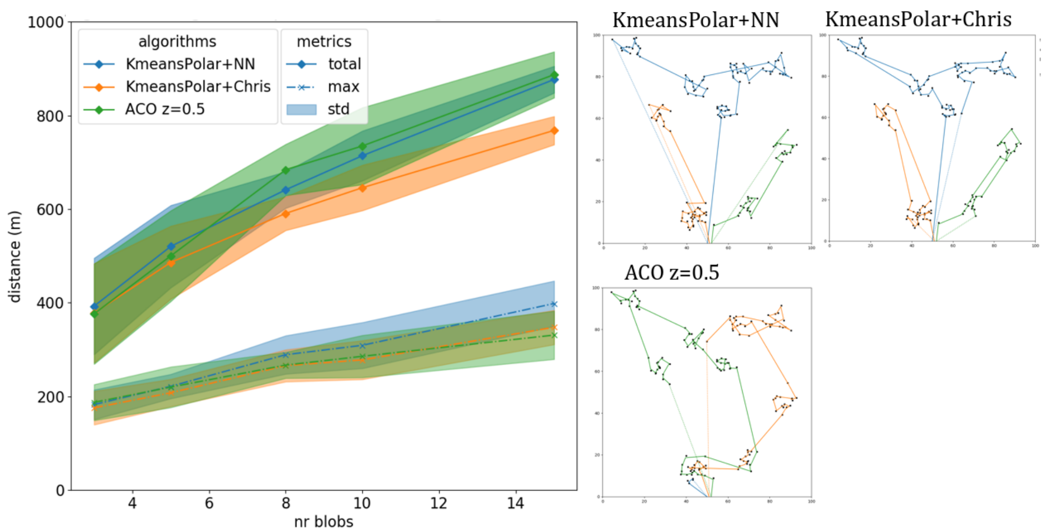

Figure 5.4: Comparison between algorithms for 3 agents with 15 points per blob, averaged over 10 runs. Examples right.

Tests were run with blobs[1] with 15 points each. The NN algorithm, Christofides' algorithm and ACO with $z = 0.5$ were tested. Figure 5.4 shows the results. As expected, the total and maximum distance both increase with the number of blobs. For the total distance, the Christofides'algorithm performs the best. Even though

[1]'Blobs' is used instead of 'clusters' to differentiate between the clusters made with the clustering methods.

ACO is not only minimising for time, it performs slightly better than the other two algorithms on this.

The NN algorithm performs worst on the quantitative metrics, but there are still qualitative advantages to it. For example, in figure 5.4, the NN solution has nice properties: there is no overlap between different agents' paths, and all the points in each blob are visited successively. In ACO, blobs are split between agents and in Christofides, the agents visit some blobs twice. Considering that in an agricultural field tasks will need to be done at each blob, NN might be the most practical algorithm.

5.1.4 Run times

The aim of this thesis was to develop efficient algorithms, because finding the optimal solution is extremely difficult. This can be quantified with the run time of each algorithm. This was tested on a Intel i7-8550 CPU with 1.80GHz laptop and 8.00GB of RAM. The code was written in Python and every effort was made to ensure it was efficient as possible.

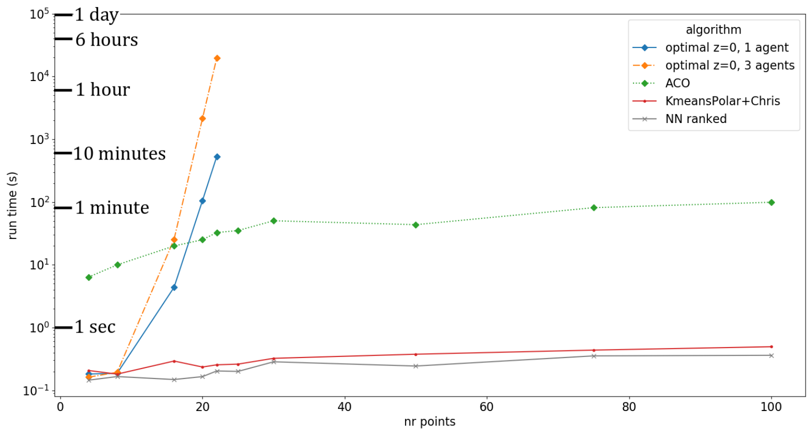

Figure 5.5: Run times for a random set of points in the continuous environment with 3 agents. The run time for BHK with 1 agent is also shown.

Run times are shown in figures 5.5 and 5.6. These are based on scenarios in the continuous world, which is determined almost exclusively by the type of algorithm. The grid world requires extra time to calculate the internal distance matrix with Dijkstra's algorithm and to place points[2]. This run time was not investigated and was not significant for small grid words ($< 100 \times 100$ nodes) anyway.

Figure 5.5 keeps the number of agents constant and varies the number of points. The time scales are very different for each algorithm:

[2]It has to check if it needs to move or remove points which are on obstacles.

- NN ranked and Christofides' with any clustering method run in under a second. This increases slightly with the number of agents.
- ACO runs in a few minutes. The bottleneck is running simulations for 100-300 iterations, because this algorithm is implemented in real time. More points increases the time to complete these simulations.
- The BHK run time grows exponentially from seconds to minutes and then hours before the computer hardware limits are reached. Already for 22 points and 3 agents, the run time is 5 hours 26 minutes ($10^{4.3}s$).

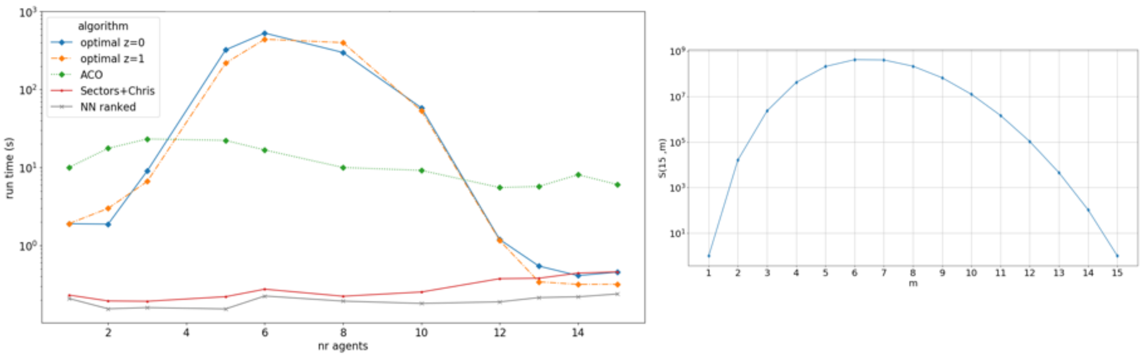

Figure 5.6: Run times for a random set of 15 points in the continuous environment (left). Stirling number of the second kind for $n = 15$ (right).

Figure 5.6 keeps the number of points constant and varies the number of agents. Again the behaviour is different for each algorithm:

- NN ranked and Christofides' have more operations to do with more agents, and therefore the run time increases slightly with the number of agents.
- ACO tends to run faster with more agents because they complete the simulations faster.
- The optimal algorithm searches through partitions of points, and therefore closely follows the Stirling number curve. At the two extremes with one agent ($m = 1$) and one agent per point ($m = n$) there is only one partition and the algorithm runs in seconds. The worst case is in the middle ($m \approx \frac{n}{2}$) with 420 million possible partitions. Here the algorithm takes about 9 minutes ($10^{2.7}s$).

5.2 Scaled fields for UAVs

Altogether there were six type of tests: three field shapes and two types of distributed points (random and blobs) per field. For each field, the following algorithms were used:

1. Clustering with Christofides' aglorithm, with all three clustering methods.
2. Clustering with the NN algorithm, with all three clustering methods.
3. ACO with $z = 0.5$.

For conciseness, and because most trends were similar, only some results are presented here. The number of points was kept fixed, while the number of agents was varied. For all results for all fields, see appendix A.5.

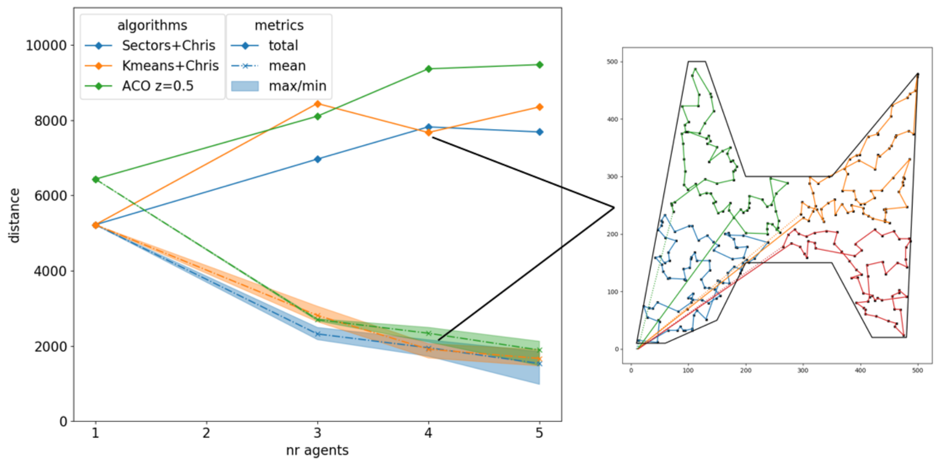

Figure 5.7: Comparison of solutions for field 3 with 300 randomly placed points.

The first test was to randomly place 300 points throughout each field. Only one set of points per field was tested. The number of agents was varied from 1 to 5. Figure 5.7 shows the results for fields 3. This graph shows the total distance with a solid line. A summary of the spread in agent distances is given by the shaded plots: the upper limit is the longest agent distance, the lower limit the shortest, and the dash-dot line is the mean agent distance. A narrower shaded region indicates that all agents travelled similar distances.

Christofides' always performed better than NN; hence the latter is not shown. For field 3, it performed better with sectors clustering than with K-means. This was different for the other fields. Interestingly, a lower total distance was achieved with 4 agents than with 3 agents. This however appears to be an exception, as this did not happen in the other fields or tests (see appendix A.5).

ACO performed worse than Christofides': both the total distance and the maximum distance was 15% to 25% longer for the H-field. This is perhaps due to the high number of points, which decreases the efficiency of this search algorithm.

Figure 5.8 shows the results for clustered points (blobs) on field 2 (L shape). Again Christofides with K-means clustering again appears to be the best algorithm. Although as stated in the previous section, NN does tend to give 'nicer' paths.

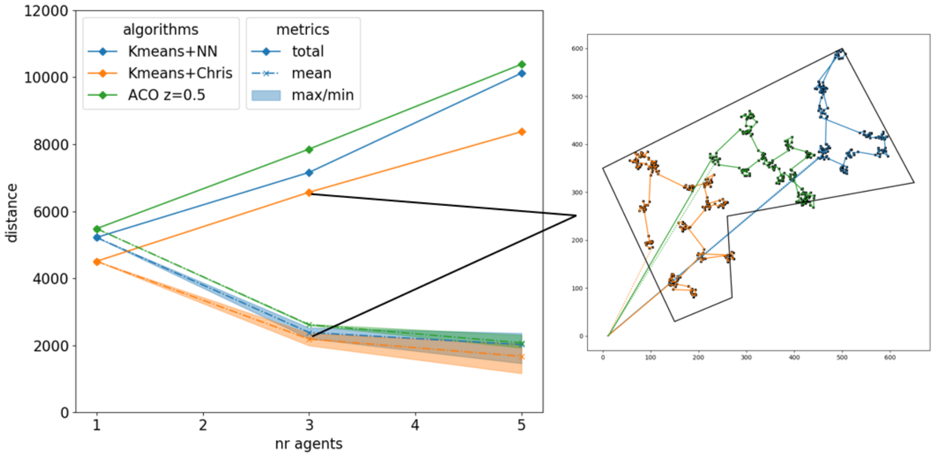

Figure 5.8: Comparison of solutions for field 2 with 40 blobs with 15 points in each blob.

For all algorithms for all fields, the maximum distance ranges from 2.1km with 5 agents to 7.2km with 1 agent. Assuming a constant speed of 5m/s (18km/h), this is a flight time of 7 minutes to 24 minutes. If a UAV only has a flight time of 25 minutes, then it is essential that the tasks at each crop do not take much time. It is probably better to use more than one UAV, because otherwise a single UAV will not be able to complete the tasks at each crop.

5.3 Grid fields for UGVs

Again there were six type of tests: three field shapes and two types of obstacles per field. For each field, the following algorithms were used:

1. Clustering with Christofides' algorithm.
2. ACO with $z = 0.0$, $z = 0.5$ and/or $z = 1.0$.

Again, only some results are presented here, and more results are in appendix A.5. For all tests, points were distributed uniformly around the field, and the number of agents was varied. As with the tests for the UAVs, only one set of points was tested for each field and obstacle type, and the shaded region in the plots represents the spread in the agent distances.

Figure 5.9 shows the algorithms for field 1, which was the rectangular field. As with the continuous case, with ACO with $z = 0$ the total distance is minimised, but this is done by using one agent to do most of the work. ACO with $z = 1$ does minimise the maximum distance, but for two and four agents this value is lower with $z = 0.5$. Also, for one and two agents, K-means polar clustering with Christofides

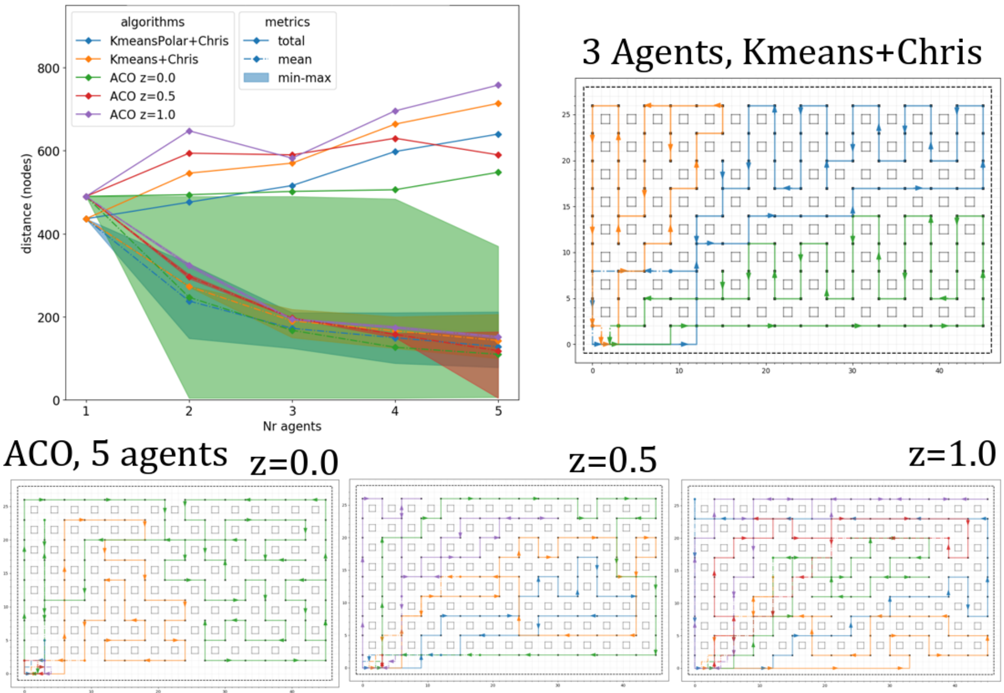

Figure 5.9: Comparison of ACO solutions for field 1 varying z, with square obstacles with 144 uniform points. The solutions for 5 agents are shown underneath.

has both a lower total distance than ACO with $z = 0.0$ and a lower maximum distance than ACO with $z = 1.0$. These problems might arise because ACO, which is built on top of the NN heuristic, is not optimised for uniform fields. The NN heuristic takes random actions when all points are equidistant from each other, as is often the case in a uniform grid. Lastly, with $z = 0.5$, the fifth agent is ignored, which suggests that there is no benefit to having more than 5 agents for this particular field topology.

Figure 5.10 shows an example solution for the more complex topology of field 3, also with square obstacles. Again ACO is ony superior for three or more agents, and the solution with $z = 0.0$ mostly ignores all but one agent. The ACO solutions with $z = 0.5$ also ignores the fourth and fifth agent, which suggests only three agents are needed for this field topology. This solution also very nicely clusters the region into three distinct clusters, as can also be seen in figure 5.10. However, the result is still marginally worse than K-means polar clustering with Christofides, which performs better than ACO $z = 0.5$ on both metrics.

With the line obstacles, clustering with K-means proved ineffective. It tended to divide rows: e.g. one agent will get the upper half and the other agent the lower half. A much more effective and simpler clustering method was to divide the grid evenly into rectangles. This clusters all rows together, unlike the various versions of ACO which are again more chaotic, as shown in figure 5.11. Despite this, ACO does

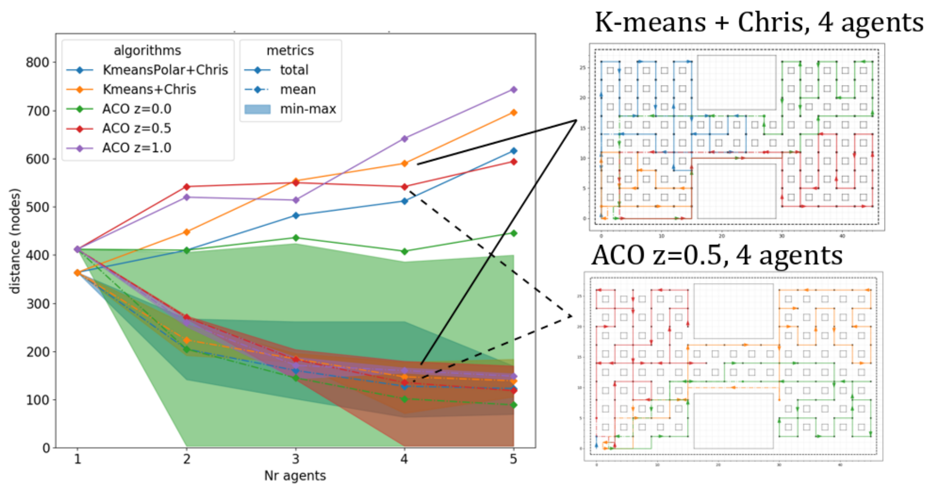

Figure 5.10: Comparison of solutions for field 3 with square obstacles with 120 uniform points.

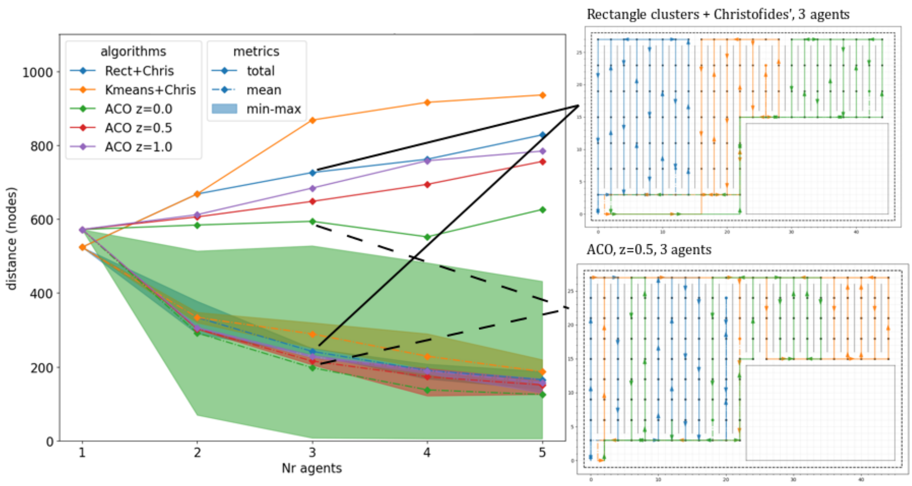

Figure 5.11: Comparison of solutions for field 2 with line obstacles with 163 uniform points.

perform better on the metrics: ACO with $z = 0.0$ does minimise the total distance except for one agent, and for most cases the metrics for ACO with $z = 0.5$ and $z = 1.0$ are lower than the corresponding Christofides solutions.

With the current simulations, it is not possible to conclude if grouping rows is a more desirable trait than lower distance and time metrics. If it is not, the UGVs should be designed with sufficient obstacle detection to avoid each other while following the ACO paths.

Chapter 6

Conclusions and recommendations

6.1 Conclusions

This research investigated algorithms for path planning for multiple robots in agriculture fields. It cast the problem as a type of VRP, and focused on approximation algorithms for solving it. Two types of agriculture field models were made: an environment with no collisions, which was used to simulate UAVs and a grid world environment with basic obstacle navigation for UGVs. The VRP algorithms were tested in three geometries for each situation. For the most part, the results obtained held in both environments. This was because all the obstacles were known at the start and did not change during the simulation, and thus paths could be planned around them with Dijkstra's algorithm.

The two most important metrics were the total distance and the total time. The total distance is directly correlated to energy and therefore cost. The total time is constrained by the slowest agent, which is assumed to be the agent that travels the longest distance. It is of primary importance to minimise the total distance; however minimising time is desirable, and in the case with UAVs with limited power supply, necessary. Ideally, both of these metrics can be minimised simultaneously but this is often not the case. In general, the trend was that more agents increase the total distance - by up to 600% - but decrease the total time - by up to 50%. A significant contributing factor to this is duplications of the journey to and from the origin point. If the agents each started at separate locations within the field, these trends might not be as prominent but this is not as practical.

For agriculture fields, it is stressed that as few agents should be used as possible. One agent will always minimise the distance. However, it is faster to use more agents, and this reduces the distance and therefore workload on each agent. This might improve each robot's life span. As an example, the maximum distance in the experiments was 7.2km for a single agent on the 20ha rectangular field. With three agents, this maximum distance could be reduced by 56% to 4.6km, with the cost of increasing the total distance by 52% to 11.0km. Later investigations need to take into account more variables in order to determine which is the best option.

Table 6.1: Suggested algorithm for different scenarios

	Point distribution	Obstacles	Best algorithms
1	Uniform/Random	None (grid or cont.)	ACO
2		Lines (grid only)	Rectangles + Christofides
3		Squares (grid only)	K-means (polar) + Christofides
4	Clustered	None	K-means (polar) + NN

The following algorithms were investigated: NN ranked, clustering with Christofides' algorithm or NN, ACO and Q-learning. NN ranked was the fastest to process, running in less than a second, but the results were not as good as the other algorithms. Regrettably, Q-learning with small states did not work well, and there was not enough time to investigate using it with larger states and more complex state functions. Otherwise, clustering and ACO both performed well on the three agriculture fields.

Of the last two algorithms, it is not immediately obvious which is better. ACO can be set to optimise for only distance ($z = 0$) or only time ($z = 1$), and then it outperforms the other algorithms. ACO will also tend to drop agents, especially with the grid fields with $z < 1$. It can therefore be used to estimate how many agents should be used. However, the solution is chaotic and is therefore only recommended for irregular geometries - the random points. Clustering results in paths that should be easier to use in real, practical agriculture scenarios. The best type of clustering algorithm depends on the geometry; table 6.1 gives suggestions for the geometries tested in this research. The main advantages over ACO are distinct regions for each agent and simpler trajectories. They can perform as well or better than ACO on the metrics of distance and time.

Clustering with NN is recommended only for the clustered points (blobs). This is because with NN each agent visits each blob in one go. This will be desirable if tasks are to be done at each blob. Otherwise Christofides always performed better on the metrics than NN. Furthermore, because of the mathematical properties of Christofides, it is known that the optimal path within each cluster will be at most 1.5 shorter than the Christofides' path. It therefore gives possible bounds on the optimal solution, which is useful even if another search algorithm is developed instead to find better paths.

6.2 Recommendations

This research approached the problem from a very high level. Hopefully it has explained the fundamental properties of the problem, and further research should work on more detailed simulations and more challenging conditions. For example, a more powerful simulation program could use more detailed 3D robot models and model sensor data. The tasks at each crop could also be modelled - e.g. waiting times to take pictures, or more complex crop interactions. The information could be dynamic e.g. the obstacles themselves move, like animals or vehicles. These results should then be validated with experimental data with robots on real fields.

This research assumed global observability - one master computer knows the entire layout of the field and can plan all the paths. Partial observability can be assumed instead, where each robot only knows their section of the field. They would compensate by communicating with the other robots and sharing information. These conditions would make most of the approximation algorithms invalid, except for NN ranked and Q-learning, so new algorithms would need to be developed. This however would be better for autonomous control, which is an important feature of swarm robotics. Deep Q-learning, which uses machine learning combined with Q-learning, is particularly promising for this type of problem.

For the current problem, there are still many more algorithms which can be investigated. Other well known VRP algorithms include simulated annealing, k-opt, Clarke-Wright, and Lin-Kernighan-Helsgaun. The latter three algorithms have a more refined search than ACO, and might be able to produce better results. Also, it is suggested to investigate plane cutting (branch and bound algorithms) so that optimal solutions can be obtained for more points. Plane cutting still has a very high time cost, but it has a far more manageable space cost than BHK (it is easier to wait for time than to physically increase CPU memory). Finally, a bespoke algorithm could be developed for some well known agriculture field geometries.

Finally, the grid world code has been written to be very flexible with different types of grids. Therefore more complex farms and geometries can be developed and investigated.

Bibliography

[1] Wikipedia. Precision agriculture. [Online]. Available: https://en.wikipedia.org/wiki/Precision_agriculture

[2] C. Carbone, O. Garibaldi, and Z. Kurt, "Swarm robotics as a solution to crops inspection for precision agriculture," *ESTEC Conference Proceedings*, vol. 2018, no. 6, pp. 552–562, 2018.

[3] T. Oksanen and A. Visala, "Coverage path planning algorithms for agriculture field machines," *Journal of Field Robotics*, vol. 26, no. 8, pp. 651–668, 2009.

[4] G. Laporte, "The Traveling Salesman Problem : An overview of exact and approximate algorithms," *European Journal Of Operational Research*, vol. 59, pp. 231–247, 1992.

[5] ——, "The Vehicle Routing Problem : An overview of exact and approximate algorithms," *European Journal Of Operational Research*, vol. 59, pp. 345–358, 1992.

[6] K. Braekers, K. Ramaekers, and I. Van Nieuwenhuyse, "The vehicle routing problem: State of the art classification and review," *Computers and Industrial Engineering*, vol. 99, pp. 300–313, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.cie.2015.12.007

[7] Areobotics. Welcome to Aerobotics - crop potection for your farm. [Online]. Available: https://www.aerobotics.com

[8] senseFly. How agriculture drones are changing the industry. [Online]. Available: https://www.sensefly.com/industry/agricultural-drones-industry/

[9] S. C. Hassler and F. Baysal-Gurel, "Unmanned aircraft system (UAS) technology and applications in agriculture," *Agronomy*, vol. 9, no. 10, 2019.

[10] A. Matese, P. Toscano, S. F. Di Gennaro, L. Genesio, F. P. Vaccari, J. Primicerio, C. Belli, A. Zaldei, R. Bianconi, and B. Gioli, "Intercomparison of UAV, aircraft and satellite remote sensing platforms for precision viticulture," *Remote Sensing*, vol. 7, no. 3, pp. 2971–2990, 2015.

[11] Clearpath Robotics. Husky Unmaned ground vehicle. [Online]. Available: https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/

[12] S. Bonadies and S. A. Gadsden, "An overview of autonomous crop row navigation strategies for unmanned ground vehicles," *Engineering in Agriculture, Environment and Food*, vol. 12, no. 1, pp. 24–31, 2019. [Online]. Available: https://doi.org/10.1016/j.eaef.2018.09.001

[13] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics : a review from the swarm engineering," *Swarm Intelligence*, vol. 7, no. 1, pp. 1–41, 2013.

[14] I. A. Hameed, D. D. Bochtis, C. G. Sørensen, and M. Nøremark, "Automated generation of guidance lines for operational field planning," *Biosystems Engineering*, vol. 107, no. 4, pp. 294–306, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.biosystemseng.2010.09.001

[15] H. Seyyedhasani and J. S. Dvorak, "Using the Vehicle Routing Problem to reduce field completion times with multiple machines," *Computers and Electronics in Agriculture*, vol. 134, pp. 142–150, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.compag.2016.11.010

[16] T. Machl, A. Donaubauer, H. Auernhammer, and T. H. Kolbe, "Shape and Ergonomics: Methods for Analyzing Shape and Geometric Parameters of Agricultural Parcels," in *EFITA-WCCA-CIGR Conference "Sustainable Agriculture through ICT Innovation"*, no. June 2013, 2013, pp. 24–27.

[17] R. J. Palmer, D. Wild, and K. Runtz, "Improving the efficiency of field operations," *Biosystems Engineering*, vol. 84, no. 3, pp. 283–288, 2003.

[18] A. Barrientos, J. Colorado, J. del Cerro, A. Martinez, C. Rossi, D. Sanz, and J. Valente, "Aerial remote sensing in agriculture: A practical approach to area coverage and path planning for fleets of mini aerial robots," *Journal of Field Robotics*, vol. 28, no. 5, pp. 667–689, 2011.

[19] D. Bochtis, H. W. Griepentrog, S. Vougioukas, P. Busato, R. Berruto, and K. Zhou, "Route planning for orchard operations," *Computers and Electronics in Agriculture*, vol. 113, pp. 51–60, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.compag.2014.12.024

[20] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*, 3rd ed. Princeton University Press, 7 2006.

[21] S. Gorenstein, "Printing press scheduling for multi-edition periodicals," *Management Science*, vol. 16, no. 6, pp. B373–B383, 1970. [Online]. Available: https://www.jstor.org/stable/2628724

[22] T. Bektas, "The multiple traveling salesman problem: An overview of formulations and solution procedures," *Omega*, vol. 34, no. 3, pp. 209–219, 2006.

[23] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2014, a Bradford Book. [Online]. Available: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

[24] R. E. Bellman, "Dynamic programming treatment of the travelling salesman problem," *RAND Corporation*, 1961.

[25] M. Held and R. Karp, "A dynamic programming approach to sequencing problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196–210, 1962. [Online]. Available: https://doi.org/10.1137/0110015

[26] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson, "Solution of a Large-Scale Traveling-Salesman Problem," *Operations Research*, vol. 2, pp. 393–410, 1954.

[27] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, "An analysis of several heuristics for the traveling salesman problem," *SIAM J. Computing*, vol. 6, no. 3, pp. 563–581, 1977. [Online]. Available: https://pdfs.semanticscholar.org/2081/25449f697c46d02a98eceb18b8c4622384c5.pdf

[28] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," *Technical Report 388*, 1976.

[29] G. Clarke and J. Wright, "Scheduling of vehicles from a central depot to a number of delivery points," *Operations Research*, vol. 12, no. 4, pp. 568–581, 1964. [Online]. Available: https://www.jstor.org/stable/167703

[30] M. Dorigo and T. Stützle, *Ant Colony Optimization : Overview and Recent Advances*. Springer, 2019.

[31] H. H. Hoos and T. Stützle, "Improvements on the Ant-System : Introducing the MAX-MIN Ant System Improving the Ant System : A Detailed Report on the MAX – MIN Ant System," January 1996.

[32] P. Junjie and W. Dingwei, "An Ant Colony Optimization Algorithm for," in *First International Conference on Innovative Computing, Information and Control-Volume I (ICICIC'06)*, 2006, pp. 210—-2013.

[33] B. Yu, Z. Z. Yang, and B. Yao, “An improved ant colony optimization for vehicle routing problem,” *European Journal of Operational Research*, vol. 196, no. 1, pp. 171–176, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.ejor.2008.02.028

[34] W. Liu, S. Li, F. Zhao, and Z. Aiyun, “An Ant Colony Optimization Algorithm for the Multiple Traveling Salesmen Problem,” in *2009 4th IEEE Conference on Industrial Electronics and Applications*. IEEE, 2009, pp. 1533–1537.

[35] D. Silver. UCL Course on RL: course notes. Online video lectures available at: https://youtu.be/2pWv7GOvuf0. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

[36] L. M. Gambardella and M. Dorigo, “Ant-Q: A Reinforcement Learning approach to the traveling salesman problem,” *Machine Learning Proceedings 1995*, vol. 5625, pp. 252–260, 1995.

[37] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, “Neural Combinatorial Optimization with Reinforcement Learning,” in *ICLR 2017*, 2016, pp. 1–15. [Online]. Available: http://arxiv.org/abs/1611.09940

[38] M. Nazari, A. Oroojlooy, M. Takáč, and L. V. Snyder, “RL for Solving the Vehicle Routing Problem,” *Nips*, no. NeurIPS, pp. 9861–9871, 2018. [Online]. Available: https://papers.nips.cc/paper/8190-reinforcement-learning-for-solving-the-vehicle-routing-problem.pdf

[39] Wikipedia. Taxicab geometry. [Online]. Available: https://en.wikipedia.org/wiki/Taxicab_geometry

[40] E. W. Weisstein, “Stirling Number of the Second Kind,” *MathWorld-A Wolfram Web Resource*, 2019. [Online]. Available: http://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html

[41] Wikipedia. Dijkstra’s algorithm. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[42] ——. A* search algorithm. [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm

[43] scikit-learn developers, “2.3 Clustering,” Online, 2019. [Online]. Available: https://scikit-learn.org/stable/modules/clustering.html

[44] R. S. Johansen, “A primer on repeatable random numbers,” Online, 1 2015. [Online]. Available: https://blogs.unity3d.com/2015/01/07/a-primer-on-repeatable-random-numbers/

[45] B. C. Rennie and A. J. Dobson, "On Stirling Numbers of the second kind," *Journal of Combinatorial Theory*, vol. 7, no. 2, pp. 116–121, 1969.

[46] StackOverflow. Calculate set partitions with specific subset sizes. [Online]. Available: https://stackoverflow.com/questions/22519470/calculate-set-partitions-with-specific-subset-sizes

[47] Z. Galil, "Efficient algorithms for Finding Maximum Matching in Graphs," *ACM Computing Surveys*, vol. 18, no. 1, pp. 23–38, 1986.

Appendix A

Appendix

A.1 Counting paths

Exact formulas for these counting problems are given here. Approximation formulas can be found in section 3.1.2.

A.1.1 Allocating cities to salesmen

If there are n distinct elements, these can be partitioned into sets with known sizes $p_1, p_2, ..., p_k$ as follows:

$$\begin{gathered} P(n, p_1, p_2, ..., p_k) = \frac{n!}{(r_1![p_1!]^{r_1})...(r_k![p_k!]^{r_k})} \ , \\ \sum_{i=1}^{k} r_i p_i = n, \ p_i \neq p_j \text{ for } i \neq j, \ p_i \in \mathbb{N} \end{gathered} \tag{A.1}$$

where r_i is the number of repeated partitions of this size in the permutation.

The number of ways to partition a set of n distinct elements into k non-empty sets is given by the Stirling Number of the second Kind [40]. It is the sum of all valid variations of equation A.1 for a given (n, k) pair:

$$S(n, k) = \sum_{p \in p(n,k)} \frac{n!}{(r_1![p_1!]^{r_1})...(r_k![p_k!]^{r_k})} \tag{A.2}$$

Where $p(n, k)$ is a set of valid of integer partitions for the (n, k) pair. These partitions can be found with algorithm 6 in section A.2.3.

It can also be calculated with the following sum [45]:

$$S(n, k) = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k - i)^n \tag{A.3}$$

Or the following recursive formula:

$$\begin{aligned} S(n,k) &= k \cdot S(n-1,k) + S(n-1,k-1) \\ S(0,0) &= 1,\ S(n,0) = S(0,k) = 0 \end{aligned} \tag{A.4}$$

Reasoning: if only the nth element is remaining, either there are $n-1$ elements that have already been put into k partitions, and there are thus k options to put this nth element into, or the nth element must go into its own singleton set, and the other $n-1$ elements have already been partitioned into $k-1$ sets. Adding the counts for these two situations gives the total answer.

A.1.2 Paths with 1 home city

Each set of p_i items has $p_i!$ permutations of paths, assuming reverse paths are different. Using equation A.2, the number of paths is:

$$\begin{aligned} N(n,k) &= \sum_{p \in p(n,k)} \frac{n!}{(r_1![p_1!]^{r_1})...(r_k![p_k!]^{r_k})}[p_1!]^{r_1}...[p_k!]^{r_k} \\ &= \sum_{p \in p(n,k)} \frac{n!}{r_1!..r_k!} \end{aligned} \tag{A.5}$$

This can also be calculated with a recursive formula:

$$\begin{aligned} N(n,k) &= (n-1+k) \cdot N(n-1,k) + N(n-1,k-1) \\ N(0,0) &= 1,\ N(n,0) = N(0,k) = 0 \end{aligned} \tag{A.6}$$

Reasoning: if only the nth element is remaining, either it can be placed in front of $n-1$ elements, or at the back of k permutations, or the nth element must go into its own singleton set, and the other $n-1$ elements have already been placed into $k-1$ sets. Adding the counts for these two situations gives the total answer.

If the reverse paths are the same, N must be divided by 2^k. However, singleton sets already have reflective symmetry and the following correction term needs to be added back for each partition with r_1 repeat singleton sets ($p_1 = 1$):

$$N^* = \left(-\frac{1}{2^k} + \frac{2^{r_1}}{2^k}\right)\frac{n!}{r_1!r_2!..r_k!} \tag{A.7}$$

A.2 Additional information on Bellman-Held-Karp

This section provides more details on the Bellman-Held-Karp dynamic programming in section 3.3.1. It describes implementation issues and also the the multi-agent algorithms. See appendix A.4 for the code source.

A.2.1 Formal description

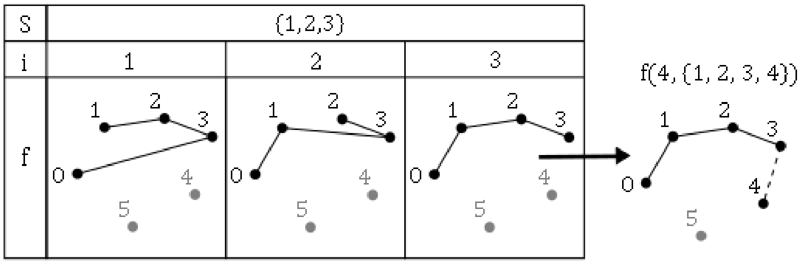

Figure A.1: Visualisation of the calculation of $f(4, \{1, 2, 3, 4\})$ for a set of points.

Let S be a subset of points and choose a point $i \in S$. Define $f(i, S)$ as the lowest cost path that starts at point 0, goes through all the other points in S and ends at point i. Then if a new point j is introduced:

$$f(j, S \cup \{j\}) = \underset{i \in S}{min}(f(i, S) + c_{ij}) \tag{A.8}$$

where c_{ij} is the cost from point i to point j. Figure A.1 visualises this calculation for a single step. The final solution is equivalent to $f(0, \{1, ..., n, 0\})$. It is found by iterating equation A.8 through all combinations of S for all layers of size $|S| = 1, ..., n$. This requires saving each layer, which drives the space requirements in the next section.

A.2.2 Space requirements

Each layer of size $|S|$ has $|S|\binom{n}{|S|}$ entries. The largest two layers are at the integers nearest $\frac{n}{2}$ and each has approximately $2^n\sqrt{\frac{n}{2\pi}}$ entries [24]. Proof: if n is even, then using Stirling's approximation:

$$\begin{aligned}
\frac{n}{2}\binom{n}{n/2} &= \frac{n}{2}\frac{n!}{((n/2)!)^2} \\
&\approx \frac{n}{2}\frac{\sqrt{2\pi n}(n/e)^n}{\pi n(n/2e)^n} \\
&= 2^n\sqrt{\frac{n}{2\pi}}
\end{aligned}$$

So space requirements grow exponentially.

The code was written in Python. Each entry was stored in a dictionary $dict[(j, S)] = [(f(j, S),\ P)]$. The keys are two `integers`: the point j and the unordered set S encoded in binary. E.g:

$$S = \{0, 2, 3, 5\} \rightarrow 10110_2 = 2^{5-1} + 2^{3-1} + 2^{2-1} = 22$$

The values are the `float` $f(j, S)$, defined by equation A.8, and an `integer` for the ordered set P encoded in base n. E.g.

$$P = \{0 \rightarrow 3 \rightarrow 5 \rightarrow\ 2\} \rightarrow 3n^0 + 5n^1 + 2n^2$$

Note that the key has to use the unordered set S for efficient lookups of the set in each iteration. Therefore for each entry, this requires:

$$\begin{aligned} \text{mem(entry)} &= [2 \times \text{mem(tuple)} + 3 \times \text{mem(int)} + \text{mem(float)}] \\ &= [2 \times 64B + 3 \times 28B + 28B] \\ &= 240\ B \end{aligned}$$

The dictionary itself also uses space of approximately 30B per entry on average. Therefore on average the total is 270B per entry. This limited n to 23 for a computer with 8GB of memory because:

$$8GB = 2^{33}B < 2(270B)(2^{24})\sqrt{\tfrac{12}{\pi}}$$

A more efficient programming language like C++ would probably enable a slightly larger n.

A.2.3 Set partitioning for multiple agents

For multiple agents, the optimal solution is found by searching through allocations of n points to m agents.

First, during the single agent iterations, the loops should be closed and the best values saved. Define $g(S)$ as the lowest cost path which starts at point 0, goes through all the points in S and ends at point 0. Similar to equation A.8 for $f(i, S)$, it is found with:

$$g(S) = \min_{i \in S}(f(i, S) + c_{i0}) \tag{A.9}$$

It is an order of magnitude smaller than f, so it increases space requirements by a relatively small amount:

$$|g| = \sum_{k=1}^{n-m+1} \binom{n}{k} \leq 2^n < 2^{n+1}\sqrt{\tfrac{n}{2\pi}} \approx |f|$$

The allocations are done in two steps. First an integer partition is found. This has a form $P(n, m) = p_1 + p_2 + ... + p_m = n$ e.g

$$P(5, 3) = 3 + 1 + 1 = 2 + 2 + 1$$

Second, the partitions for each integer partition are then listed. E.g for $P(5, 3) = 2 + 2 + 1$, the 15 partitions are:

$$[1, 2|3, 4|5], [1, 2|3, 5|4], [1, 2|4, 5|3], [1, 3|2, 4|5], [1, 3|2, 5|4],$$
$$[1, 3|4, 5|2], [1, 4|2, 3|5], [1, 4|2, 5|3], [1, 4|3, 5|2], [1, 5|2, 3|4],$$
$$[1, 5|2, 4|3], [1, 5|3, 4|2], [2, 3|4, 5|1], [2, 4|3, 5|1], [2, 5|3, 4|1]$$

Partitions are skipped in either step if the cost exceeds the best cost found so far. E.g. if $8 = 4 + p_2 + p_3$ and $g(\{1, 2, 3, 4\}) > cost_{min}$, then partitions of $\{5, 6, 7, 8\}$ are skipped for both $8 = 4 + 3 + 1$ and $8 = 4 + 2 + 2$.

Algorithm 6 List all integer partitions of n of size m

result $\leftarrow \emptyset$
P(n, m, n) ▷ Start iterations
procedure P(n, m, u)
    if $m = 1$ **then**
        if $n \leq u$ **then**
            return result
    else
        $u \leftarrow min(u, n - (m - 1)1)$ ▷ $n = (1)[u] + (m - 1)[1]$
        for $i = u, u - 1, ...1$ **do**
            return result+i+P$(n - i, m - 1, i)$

The integer partitioning is done with algorithm 6. It uses backtracking and recursion. It is based on the recursive formula:

$$P(n, m) = i + P(n - i, m - 1) \tag{A.10}$$

But this alone is not enough because it produces many duplicates. E.g.

$$P(8, 2) = 5 + P(3, 1) = 3 + P(5, 1)$$

These duplicates are excluded by adding an extra parameter u in $P(n, m, u)$, where u is an upper bound on allowed numbers.

The set partitioning is done with algorithm 7, which also uses backtracking and recursion. This StackOverflow discussion was very helpful for developing it: [46]. The explanation is as follows: for each partition k try all combinations of p_k numbers

Algorithm 7 List all partitions of an integer partition

Initialise $p \in P(n,k)$, g, $cost_{min} \leftarrow (\infty$ or $cost_{best\ guess})$
FIXED_PARTITIONS$(1, \{1,...,n\}, \emptyset, 0)$ ▷ Start iterations
procedure FIXED_PARTITIONS$(k, U, used_first, cost)$
  if $k = m$ **then** ▷ Is this the last set?
    if not $used\_first[p_k, U_1]$ **then**
      $cost \leftarrow cost + g[U]$
      if $cost < cost_{min}$ **then**
        $cost_{min} \leftarrow cost$ ▷ Also save the path and partitions
  else ▷ Partition remaining numbers
    for $first \in U$ **where not** $used\_first[p_k, first]$ **do**
      $used\_first[p_k, first] \leftarrow true$
      for all $S \in$ COMBINATIONS$(\{u|u \in U$ and $u \geq first\}, p_k)$ **do**
        $cost' \leftarrow cost + g[S]$
        if $cost' < cost_{min}$ **then** ▷ Skip otherwise
          FIXED_PARTITIONS$(k+1, U \setminus S, used_first$.copy$, cost')$

from the remaining numbers. To avoid duplicate permutations within partitions - e.g. $[0,2|1]$ and $[2,0|1]$ - each partition must be in ascending order. If each partition size is distinct, that is $p_1 \neq ... \neq p_m$, then this is sufficient to find all combinations. The number of partitions is:

$$N = \binom{n}{p_1}\binom{n-p_1}{p_2}\binom{n-p_1-p_2}{p_3}...\binom{p_m}{p_m} = \frac{n!}{p_1!...p_m!} \quad \text{(A.11)}$$

However, if any of the partition sizes are repeated - such as for $P(5,3) = 2+2+1$ - then this is not enough. If p_k is repeated r_k times, there will be $r_k!$ duplicate permutations of it. The number of partitions is instead defined by equation A.1. To avoid these, the following check is done: if the first number in partition k was used in partition $j < k$ with the same size, then this partition should be skipped. E.g. if $[1,2|3,4]$ was used, then $[3,4|*,*]$ does not allow $[1,*]$.

The final code actually combines algorithm 6 and 7 into one algorithm. In the highly unlikely worst case, this code will list all $S(n,m)$ partitions where $S(n,m)$ is a Stirling Number of the Second Kind [40]. However it will most likely only check a much smaller fraction of $S(n,m)$.

A.3 Christofides' algorithm

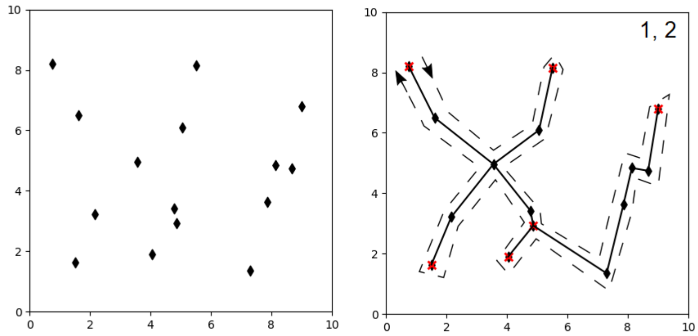

Figure A.2: Random points (left), and their MST (right). A walk around the tree is shown with dashed lines. Odd vertices are shown with red crosses.

Christofides' algorithm is based on the minimum spanning tree (MST) [28]. A spanning tree is a collection of edges that connect all points without creating loops and the MST is the spanning tree with the shortest total length. An example of an MST is shown in figure A.2. The following two statements are self-evident:

1. The optimal tour without the final edge is a spanning tree that is longer than the MST.
2. A walk around the spanning tree is a TSP tour (ignoring repeated vertices), but it is not optimal because each edge is used twice.

This can be summarised as:

$$\text{length(MST)} < \text{length(optimal)} < 2 \cdot \text{length(MST)}$$

The main innovation of Christofides' algorithm is to connect pairs of vertices to create bridges around the MST. These are used to make a TSP tour with the following bounds [28]:

$$\text{length(optimal)} \leq \text{length(Christofides')} \leq 1.5 \cdot \text{length(optimal)}$$

This is the best bound that has been obtained for a TSP algorithm. (But in practice, search algorithms perform better.)

The algorithm is as follows: for n points, define a graph G with $n(n-1)$ edges for each point i to each other point j. Each edge must have a weight w_{ij}. For the bounds to hold, the weights must be symmetric ($w_{ij} = w_{ji}$) and the triangle equality must hold between any three ($w_{ij} + w_{jk} \geq w_{ik}$). Both are guaranteed when the weights are lengths on a Euclidean plane. Then do the following:

1. Create a minimum spanning tree (MST) T of G.
2. Find all the odd vertices O in T.
3. Find a minimum-weight perfect matching M of the edges of O.
4. Combine the edges of M and T to form a multigraph H.
5. Form an Eulerian circuit in H.
6. Shortcut repeated vertices to obtain a Hamiltonian circuit.

Examples of the steps are shown in figure A.2 and figure A.3.

The following is a more detailed explanation: The MST T can be found in $\mathcal{O}(n\,log(n))$ time with Prim's algorithm or Kruskal's algorithm (step 1). Its odd vertices are the vertices with an odd number of edges (step 2). There will always be an even number of odd vertices.

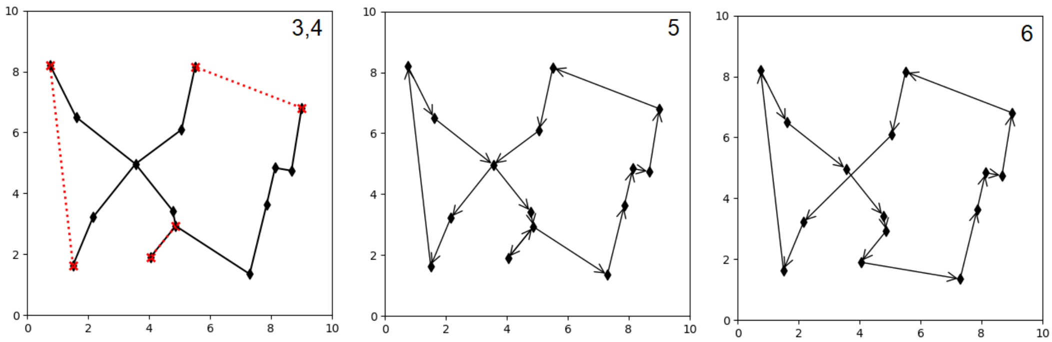

Figure A.3: Minimum-weight perfect matched pairs (left), the Eulerian circuit (middle) and Christofides approximation (right).

Link m pairs of the $2m$ odd vertices so that the total length of the added pairs is minimal (step 3). This is called the minimum-weight perfect matching M. This task is itself non-trivial. There are $\frac{(2m)!}{m!2^m}$ different sets of m pairs. Thankfully the Blossom algorithm can solve this in $\mathcal{O}(m^3)$ time [47].

Combine M and T to form a multigraph (step 4). "Multigraph" means that any overlapping edges must be kept as separate edges. The shortest pair in figure A.3 makes overlapping edges. An Eulerian circuit can then be found: this is a tour that passes through each edge once (step 5). Note the double arrow heads over the overlapping edges.

The final step is to shortcut repeat vertices on this tour so that each vertex is only visited once (step 6). This is called a Hamiltonian circuit and is also a TSP tour. Marginal improvements can be made by choosing when to shortcut a repeat vertex e.g. the first time the vertex is on the tour or the first time it is repeated.

The overall complexity is $\mathcal{O}(n^4)$, mostly because of step 3.

A.4 Code sources

There was much custom code written for this thesis. Code was also used from many different sources. The main sources are listed in table A.1.

Table A.1: Code sources

	Author	Code	Description	Link
1	NumPy	numpy package	Maths, arrays, pseudo-random generators.	`https://numpy.org/`
2	scikit-learn	make_blobs, k_means	Clustered datasets and clustering. See section 4.2.2 and **??**.	`https://scikit-learn.org/`
3	Matplotlib	pyplot, tri, path	Plotting, triangulation and polygons.	`https://matplotlib.org/`
4	OpenAI	gym package	RL framework.	`http://gym.openai.com/`
5	LCAS	deep_rl_for_swarms package	Particle swarm environments and multi-agent RL.	`https://github.com/LCAS/deep_rl_for_swarms`
6	Carl Ekerot	Held_Karp.py	Optimal solution. See section 3.3.1 and A.2.	`https://github.com/CarlEkerot/held-karp`
7	Andrew Zhuravchak	Christofides.py	Christofides' alogrithm. See section A.3.	`https://github.com/Retsediv/ChristofidesAlgorithm`
8	Joris van Rantwijk	mwmatching.py	Maximum weight matching. See section A.3.	`http://jorisvr.nl/article/maximum-matching`

Source 5 inspired the framework, and some classes and functions were copied from it. Sources 6 and 7 were modified as follows:

6. Held_Karp.py
    (a) Encode path in base n.
    (b) Store only two layers of the dictionary at a time.
    (c) Multiple agent code which searches set partitions, as explained in appendix A.2.
7. Christofides.py

(a) The minimum weight matching is done with mwmatching.py (code source 8) by passing negative weights. The original code used a sub-optimal greedy matching.

(b) Different inputs and outputs.

A.5 Additional Results

A.5.1 Random environments for the grid world

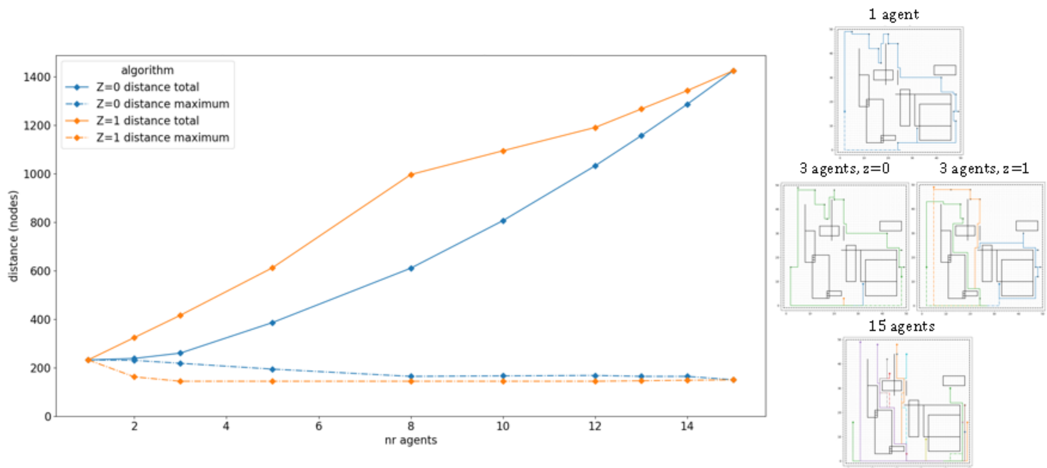

Figure A.4: Optimal solution for one set of 15 uniformly random distributed points on a grid (left). Examples of solutions (right).

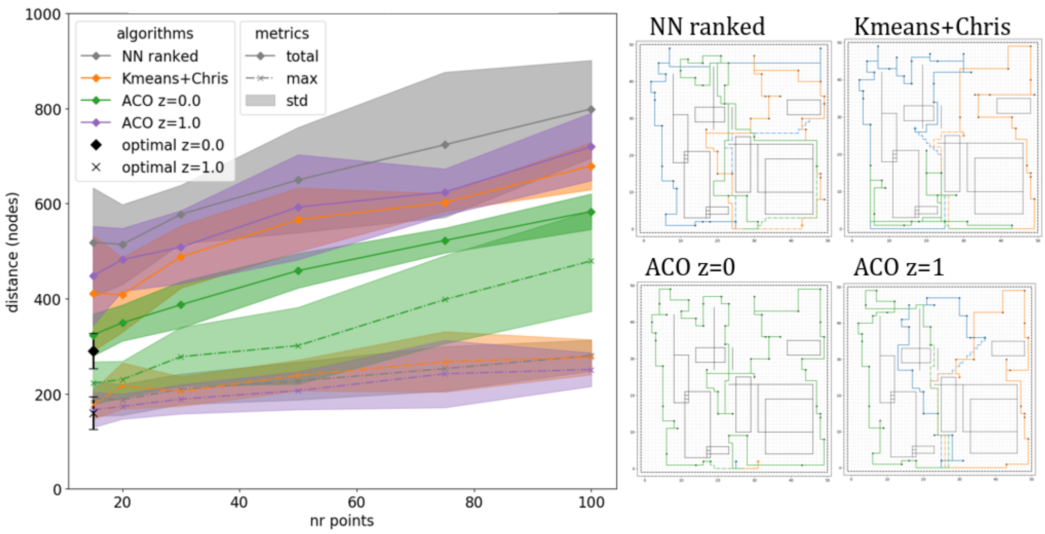

Figure A.5: Comparison between algorithms for 3 agents, averaged over 10 runs.

A.5.2 Scaled farms for UAVs

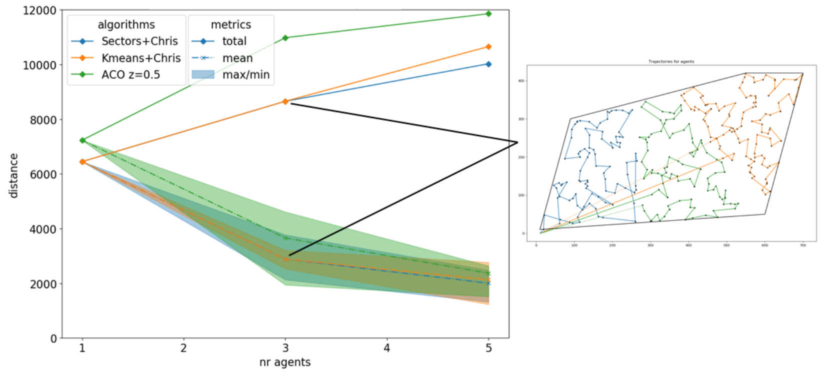

Figure A.6: Comparison of solutions for field 1 with 300 randomly placed points

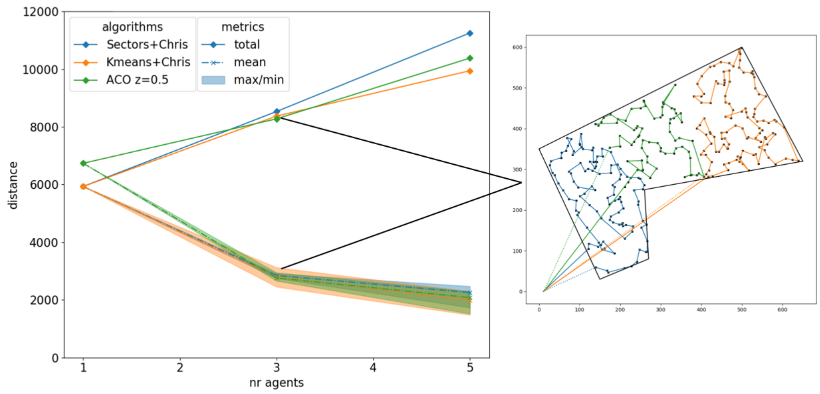

Figure A.7: Comparison of solutions for field 2 with 300 randomly placed points

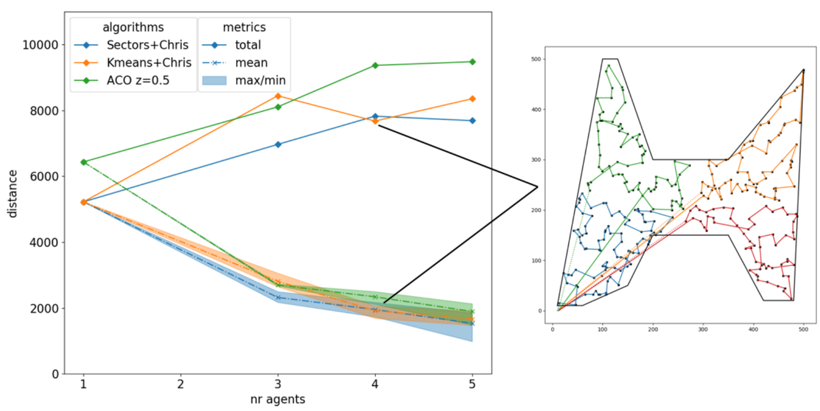

Figure A.8: Comparison of solutions for field 3 with 300 randomly placed points.

A.5.3 Grid farms for UGVs

Square obstacles

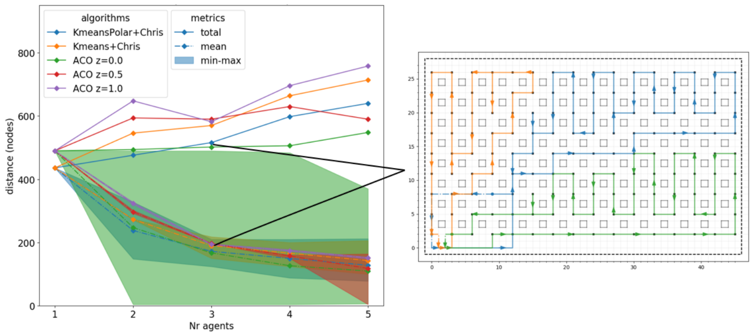

Figure A.9: Comparison of solutions for field 1 with square obstacles with 144 uniform points.

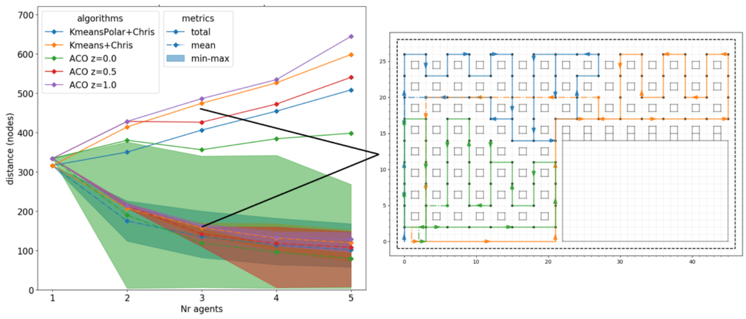

Figure A.10: Comparison of solutions for field 2 with square obstacles with 104 uniform points.

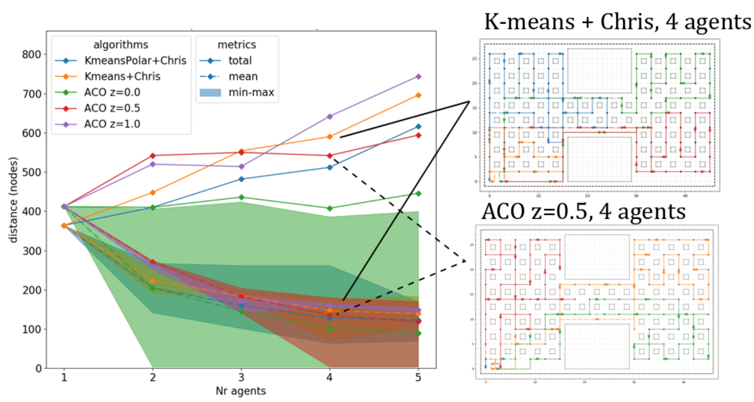

Figure A.11: Comparison of solutions for field 3 with square obstacles with 120 uniform points.

Line obstacles

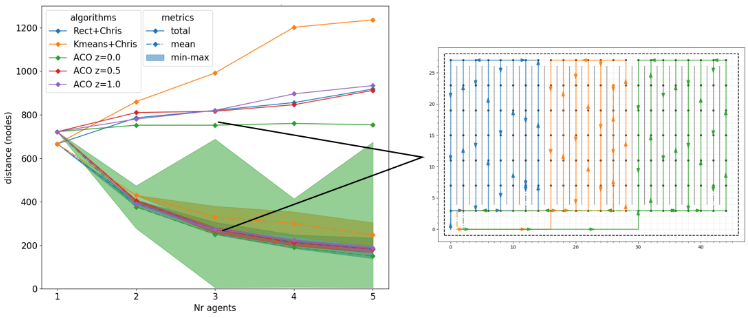

Figure A.12: Comparison of solutions for field 1 with line obstacles with 161 uniform points.

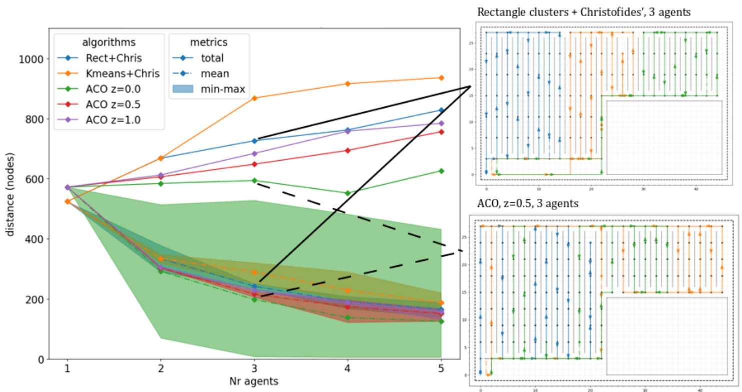

Figure A.13: Comparison of solutions for field 2 with line obstacles with 163 uniform points.

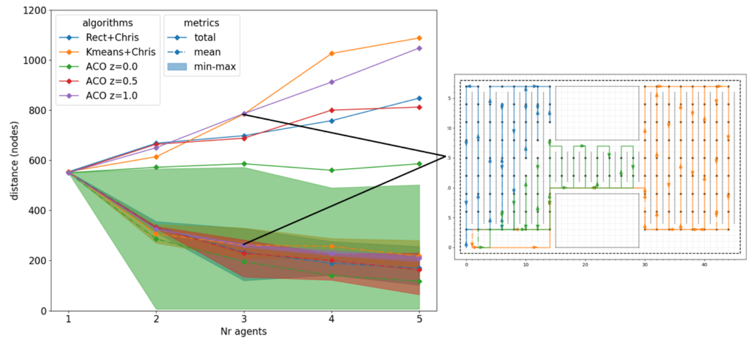

Figure A.14: Comparison of solutions for field 3 with line obstacles with 158 uniform points.

There is a variation of the above results with different parameters of ACO. Figure A.15 shows that in the rectangular field the value of the cost function is slightly lower

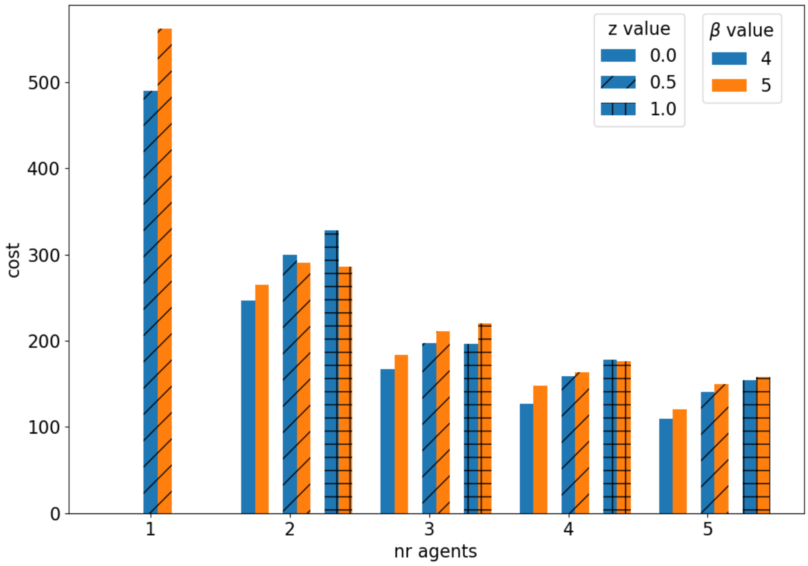

Figure A.15: Value of the cost function for field 1 with different parameters of ACO. $\alpha = 3$, $\rho = 0.01$

for $\beta = 4$ than $\beta = 5$ in most cases. Therefore $\beta = 4$ was used to obtain the grid field results presented in section 5.3.