Internal Report 2012–11 June 2012

Universiteit Leiden
Opleiding Informatica

Developing an Ant System with visual support

Erik Zandvliet

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

1 Introduction

Ant Colony Optimization – ACO for short[3] – is the name of a group of algorithms inspired by the foraging behaviour of ants. Real-life ants tend to find the shortest path from their nest to a food source, by using stigmergy[1]: they deposit a trail of pheromones, which other ants – at a later point in time – use to make a decision on which of the available paths is most attractive to follow. The algorithms falling under this group – for example simple ACO (sACO)[3], Elitist Ant System[6] and ASrank [2] – can be used on virtually any problem that can be rewritten into a problem of finding the shortest path in a graph.

Current algorithms take as their input a weighted graph, and perform calculations on this two-dimensional model. However, when we look back at the origins of the algorithms – nature – it is clearly visible that for real-life ants the problem of finding the most optimal path consists of more challenging aspects in addition to distance as well. This paper proposes an extension of the normal algorithms for taking into account one of these aspects: the fact that nature does not provide perfectly flat surfaces. This results in an addition of a directionality in the graph, while still keeping the 2 directions of a path related: $i \leftrightarrow j$ becomes $i \rightarrow j$ and $j \rightarrow i$. This directionality is not found in the classic algorithms, and therefore requires some modifications to the algorithms.

Altough this addition seems unusual at first glance, the theory of combinatorial landscapes[5] – graphs augmented with height values – has various applications in for example bioinformatics and combinatorial optimization. By turning the problems into a multi-objective optimization problem – in which we try to optimize as well the path length as the amount of effort needed – researchers gain an added possibility of adding fuzzy constraints to a search-space.

Other possibilities this research opens are in the domain of geographic information systems[4]. Pathfinding in three-dimensional environments might for example be used for making a better planning for a hiking trip through a mountainous area, in which we do want to get to our destination by walking a certain distance, but also we might not want to climb over all hills, if there actually is a path somewhere between the mountains that is more easy to use.

The paper is organized as follows: First, we will discuss some prelimary knowledge in Section 2. Some of the basic algorithms – the ones we have used in our case study – will be explained in more detail in Section 3. A detailed description about translation of the given algorithms to a 3D environment will be given in Section 4. In Section 5 we describe a case study, and the results hereof, to end with our conclusions and some ideas on future research and possibilities in Section 6.

2 Preliminaries

Since ACO algorithms rely heavily on graph theory, we present in this section some definitions that we will need later on to describe the modifications we made to the algorithms.

In classical ACO algorithms, a problem is represented as a combination of edges and vertices

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

in which $\mathcal{V}$ is a set of vertices, or nodes, and $\mathcal{E}$ is the set of edges of our graph, which we represent as ordered tuples of vertices. To simplify notations, we enumerate our vertices from 1 to n, and represent these as $\mathcal{V}_i$.

We can then define the elements of $\mathcal{E}$ to be of the form

$$\mathcal{E}_{ij} = (\mathcal{V}_i, \mathcal{V}_j)$$

where $(1 \geq i, j \leq n) \wedge (i \neq j)$

Normally, the problems are represented as undirected graphs, e.g. $\mathcal{E}_{ij} = \mathcal{E}_{ji}$. However, since we will be using the directionality later on in the modificated algorithms, we will use directed graphs, and add some constraints to the edges, one of these being

$$\mathcal{E}_{ij} \in \mathcal{E} \iff \mathcal{E}_{ji} \in \mathcal{E} \tag{1}$$

Other – similar – constraints will be presented in the explanation of the used algorithms.

See Figure 1 as a small example of a graph.

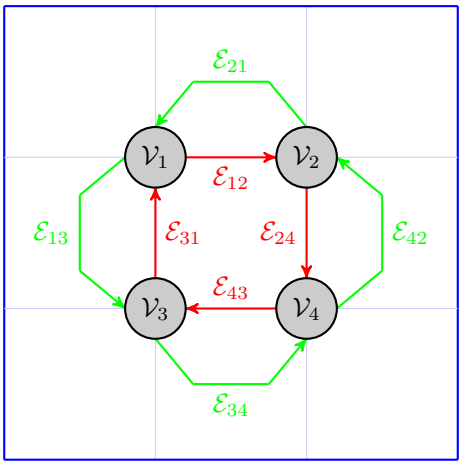

Figure 1: A small example graph

Next, we introduce the concept of neighbourhood for a vertex. This neighbourhood, denoted by $\mathcal{N}_i$ for a vertex i, contains all nodes for which there exists an edge connecting this vertex with vertex i. This gives us the following definition:

$$\mathcal{N}_i = \{\mathcal{V}_j \in \mathcal{V} \mid (\mathcal{E}_{ij} \in \mathcal{E}) \vee (\mathcal{E}_{ji} \in \mathcal{E})\} \tag{2}$$

However, since we added the constraint in (1), we know that if two vertices are connected, they are connected in both directions. This makes our final definition of neighbourhood the following:

$$\mathcal{N}_i = \{\mathcal{V}_j \in \mathcal{V} \mid \mathcal{E}_{ij} \in \mathcal{E}\} \tag{3}$$

Following this definition, for example, $\mathcal{N}_2 = \{\mathcal{V}_1, \mathcal{V}_4\}$.

We will also introduce the concept of a path between two vertices. A path between two vertices exists if and only if there exist edges connecting them, optionally through the use of intermediate vertices. This gives rise to the following:

$$\mathcal{E}_{ij} \in \mathcal{E} \iff \mathcal{V}_i \to \mathcal{V}_j \tag{4}$$

in which $\to$ represents a step in the path. Next, we make our definition of a path transitive:

$$(\mathcal{V}_i \to \mathcal{V}_j) \wedge (\mathcal{V}_j \to \mathcal{V}_k) \implies \mathcal{V}_i \sim> \mathcal{V}_k \tag{5}$$

In the remainder of this paper, we will show multiple paths, and denote them through their steps, writing $\mathcal{V}_i \to \mathcal{V}_j \to \mathcal{V}_k$ instead of $(\mathcal{V}_i \to \mathcal{V}_j) \wedge (\mathcal{V}_j \to \mathcal{V}_k)$. The paths we use are normally denoted as T.

The length of a path in most ACO algorithms is normally given by the number of edges, but since we will be using height as our main factor, we decided to use Euclidian distances between the coordinate points of the connected nodes. These lengths in a three-dimensional space can be calculated through the following:

$$|\,\mathcal{E}_{ij}\,| = \sqrt{(\mathcal{V}_{i_x} - \mathcal{V}_{j_x})^2 + (\mathcal{V}_{i_y} - \mathcal{V}_{j_y})^2 + (\mathcal{V}_{i_z} - \mathcal{V}_{j_z})^2} \tag{6}$$

where $\mathcal{V}_{i_x}$,$\mathcal{V}_{i_y}$ and $\mathcal{V}_{i_z}$ denote the x, y and z coordinates of $\mathcal{V}_i$, respectively. Since we use the squares of the difference between each coordinate-pair, we can set the following as a second constraint:

$$|\,\mathcal{E}_{ij}\,| = |\,\mathcal{E}_{ji}\,| \tag{7}$$

We also define the length of a path T as the sum of all its edges:

$$|T| = \sum_{\mathcal{E}_{ij} \in T} |\,\mathcal{E}_{ij}\,| \tag{8}$$

This length is also important for the algorithms later described, so we denote it by L.

3 Algorithms

In this section we will explain the three algorithms we adapted for use in three-dimensional environments. Algorithms in the ACO field of research are loosely based on the behaviour of real-life ants, with various ways of improvements in

the different algorithms. However, all algorithms share a common structural base.

Ants get converted to agents, which is a widely used term for describing entities that perform certain tasks in an environment. The task to be performed by each single agent, is to find a (possibly non-optimal) path between the nest and the food-source.

Converting the ants to agents, gives the possibility to differ from nature by giving the agents for example a memory in which they store the path followed (up till their current location). This memory turns out to be a factor of crucial importance in the optimization of most of the algorithms.

Let us assume that we have the amount of pheromone available as a weight on the edges of our graph, and denote this weight by the symbol $\mathcal{E}_{ij}^{\tau}$

The neighbourhood defined in (3) is used by the agents to determine which moves they can make, and we can derive a weighted probability out of this by means of the following equation:

$$\forall \mathcal{V}_j \in \mathcal{N}_i : \mathcal{P}(\mathcal{E}_{ij}) = \frac{\mathcal{E}_{ij}^{\tau}}{\sum_{k \in \mathcal{N}_i} \mathcal{E}_{ik}^{\tau}} \tag{9}$$

Next, we use a 'random' number between 0 and 1 to determine which move the agent will make in the next time step.

Each time step, also a pheromone-update takes place, determined by the following function:

$$\mathcal{E}_{ij}^{\tau} = (1 - \rho) \cdot \mathcal{E}_{ij}^{\tau} + \sum_{x} \Delta_{ij}^{x} \tag{10}$$

Where $\rho \in [0, 1)$ is a parameter for the speed of evaporation, the subterm $\sum_{x} \Delta_{ij}^{x}$ sums over all agents, and Δ_{ij}^{x} is specific for each algorithm, and typically a value depending on whether agent x moved over $\mathcal{E}_{ij}$ or not.

As a last item, we introduce two more notations, used in the artificial ants' memories. For an agent x, we use T_k^x to denote the path followed by this agent, and L_k^x to denote the length of this path.

The following three subsections will shortly describe the algorithms we have used for our case study.

3.1 Natural Behaviour Based Algorithm

The first algorithm implemented in our case study, is one that we developed ourselves. It is created to simulate the natural behaviour of real-life ants, and to serve as a benchmark for our application. The agents in this algorithm deposit a fixed amount of pheromone at each time step, e.g. unaffected by the quality of the solution (length of the found path), or the direction they are moving in. When an agent encounters a vertex of the graph, the agent chooses the next path with a probability corresponding to the amount of pheromone on each edge, as shown in formula 9 above.

Agents that have found the food source, will walk all the way back, again updating pheromone, and trying to find a new path, for the agents in this algorithm do not have any memory.

Pheromone updates follows the update-rule defined in equation 10, with Δ_{ij}^x defined as follows:

$$\Delta_{ij}^x = \begin{cases} \frac{1}{Q} & \text{if } \mathcal{E}_{ij} \text{ is the last edge traversed by agent } x \\ 0 & \text{Otherwise} \end{cases}$$

Where Q is a parameter that we can tweak to control the amount of pheromone that gets deposited on each timestep. The higher we set Q, the lower the amount of pheromone deposited on a timestep.

3.2 Simple ACO

This algorithm is a relatively simple one, and very useful to show the benefits of having agents over more realistically modelled agents. The agents are equipped with a memory, in which they store the edges taken to get to the food source. When agents finds a food source, they walk back over the same path and deposit pheromone according to the quality of the solution obtained.

Our implementation of the algorithm first lets the agents run over the graph to find a solution, keeping track of their followed path. In these paths, so-called cycles can occur – the appearance of a single node more than once in a path, the agent in this case has walked in a circle around this node –, and because these only lead to over-expression of pheromones on the involved edges, we eliminate such cycles from the paths – as described in Appendix B – before updating the pheromone amounts.

Updating the amounts of pheromones happens in one step, after all paths have been found and we have eliminated all cycles, using the following definition for Δ_{ij}^x:

$$\Delta_{ij}^x = \begin{cases} \frac{1}{L_k^x} & \text{if } \mathcal{E}_{ij} \in T_k^x \\ 0 & \text{if } \mathcal{E}_{ij} \notin T_k^x \end{cases} \tag{11}$$

3.3 ASrank

The third algorithm we used for our case study, elaborates a little further on the sACO algorithm. The ants are still equipped with a memory, and they still deposit pheromones only on their way back from the food source, but in this algorithm only the best 'w' agents deposit pheromone, antiproportional to their rank. This puts emphasis on shorter solutions, by not updating the – some of the – longer solutions found. On top of this, we deposit pheromone on the edges belonging to the best solution found so far, to serve as a reinforcement and aid in further emphasizing shorter paths. It also makes this algorithm unrealistic

to happen in nature, for real ants can't be selected and/or ranked by a global mechanism.

To accomodate for this, we have to adjust some of the update rules. Where for the previous 2 algorithms, equation 10 was enough, we will now have to adjust it so accomodate the updates of the best solution so far. The new equation will be the following:

$$\mathcal{E}_{ij}^{\tau} = (1-\rho)\cdot\mathcal{E}_{ij}^{\tau} + \sum_{r=1}^{w-1}(w-r)\cdot\Delta_{ij}^{x} + w\cdot\Delta_{ij}^{bs} \tag{12}$$

This formula looks different from the one in equation 10, but in practice isn't as different. All agents in this algorithm first find a solution. Then, from these solutions, all possible cycles get eliminated – same as in the Simple ACO algorithm – before we rank these agents.

Agents get assigned a rank r according to their found solution, with the agents that have found a better solution receiving a higher rank. Thus, we sort all agents according to the quality of their solution found, and assign them ranks 1 through m, where m is the amount of agents. Next, we have a parameter w which can be used to determine the amount of solutions that we want to take into consideration.

We then use this rank r, in combination with the amount of solutions w we take into consideration, to determine how much weight a specific solution gets in the updates. $(w-r)$ yields a higher value for higher ranks, and thus a higher weight.

Δ_{ij}^{x} is defined the same as in the Simple ACO algorithm, described in equation 11. Δ_{ij}^{bs} – where bs stands for 'best solution' – is defined analogous:

$$\Delta_{ij}^{bs} = \begin{cases} \dfrac{1}{L_k^{bs}} & \text{if } \mathcal{E}_{ij} \in T_k^{bs} \\ 0 & \text{if } \mathcal{E}_{ij} \notin T_k^{bs} \end{cases} \tag{13}$$

Here we define T_k^{bs} and L_k^{bs} as the path and pathlength of the best solution found so far, respectively.

4 Three-dimensional environments

As stated in the introduction, the real environment of ants consists of more elaborate – and therefore more complex – structures than the two-dimensional graph representations of the environment which is used in general literature to describe the behaviour of artificial ants. The goal of our research, was to find out if there is a way to resemble the environment better, and to see what kinds of research could possibly profit from a three-dimensional representation of the environment and adapt algorithms to cope with the different problems that arise when translating the environment.

One of the first questions one should ask when adapting such algorithms is what the added value could be, and in our case specifically what the added

third dimension would represent. Since the x- and y-axis together already give us a complete overview of the possible paths and routes the ants could take, adding a z-axis would not per definition yield added value if we take the edges along this axis to be more paths, because these could be represented in the two-dimensional fashion as well already.

The idea is to create what is called a "Combinatorial Landscape"[**?**], in which the third dimension represents the height of an area. When real ants traverse their environment, one could easily imagine them rather taking a path going down with a little slope, than going up. This concept can be represented by adding a factor 'effort', and stating that it will take an ant more effort to move up along an edge, and less effort when going down. We can compare this with for example biking up or down a mountain, going down costs you less effort for gravity also provides for a part of the total effort needed.

Along with this decision the question arises how we should model an ants "preference" for slopes that go slightly down, and "dislike" for slopes that move up. For this we adjusted the making of decision to not only taking into account the amount of pheromone on edges, but also the slope of the edge. There are various solutions possible for this problem, and we chose two – relatively – simple and easy to understand approaches.

As we focused our research on the effect of translation into the three-dimensional space, our test cases and their results will depend heavily on the design decisions made when making adjustments. Therefore, in Section 4.1 we will now define the changes in the algorithms that we propose to take into account height.

4.1 Modifications of the basic algorithm

When trying to incorporate height into one of the algorithms, we first needed to decide what sort of effect the height would have on the algorithm. Our first approach, was to let the angle of a path with regards to the current vertex decrease or increase the likelihood of the agents taking this a certain path.

As we have seen in Section 3, the normal algorithm for deciding the probability of an edge being taken, is as follows:

$$\forall \mathcal{V}_j \in \mathcal{N}_i : \mathcal{P}(\mathcal{E}_{ij}) = \frac{\mathcal{E}_{ij}^{\tau}}{\sum_{k \in \mathcal{N}_i} \mathcal{E}_{ik}^{\tau}}$$

If we define our landscape to be a "regular", four-connected, grid with heights – sometimes also referred to as a height map – we can define each vertex on the grid with a unique combination of three coordinates: x,y and z. For a specific vertex i, we will from now on denote these as i_x, i_y and i_z. We can now easily calculate the angle of our edges as well.

Consider an edge $\mathcal{E}_{ij}$, connecting nodes i and j. Now, if we know the coordinates of these vertices, we can calculate the angle between these two vertices. We can make some simple assumptions when calculating this angle. Because we defined our grid to have a *von Neumann* neighbourhood, that is, each vertex has four connected neighbours – except the ones on the borders of the

landscape – we can assume that for two vertices connected by an edge, either $i_x = j_x \wedge |i_y - j_y| = 1$ or $i_y = j_y \wedge |i_x - j_x| = 1$ holds, with $|i_x - j_x|$ denoting the absolute value of the substraction.

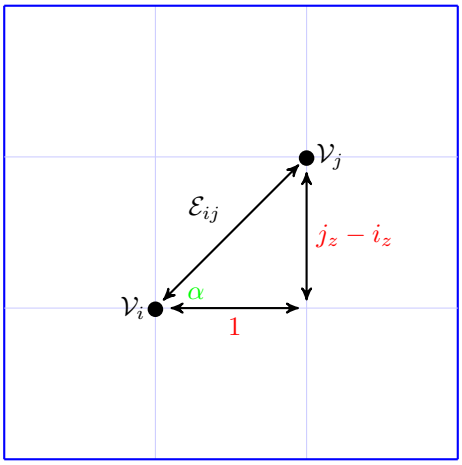

Figure 2: Explanation for formula 14

From trigonometry it is known that the tangent of an angle can be calculated by dividing the opposite side of a right triangle by the adjacent side. In our case, this means that we can get the angle of $\mathcal{E}_{ij}$ by dividing $j_z - i_z$ by 1, the length of either $|i_y - j_y|$ or $|i_x - j_x|$. This means that our angle is equal to the arctangent of the difference in height. Note, that this gives us the angle in radians, and that we apply a conversion to obtain the angle in degrees.

For an angle α as shown in Figure 2, the following equation holds:

$$\tan(\alpha) = \frac{(j_z - i_z)}{1} \tag{14}$$

Subsequently, we can use the arctangent to compute α itself. This, together with conversion from radians to degrees, gives us the following expression:

$$\alpha_{\mathcal{E}_{ij}} = \frac{\arctan(j_z - i_z)}{\pi} \cdot 180° \tag{15}$$

Using this angle, our first approach was to purely change the way the ants chose an edge, and increase the chance of agents taking a certain path, when it goes "down" from the current location. For this, we changed the function that calculates the probability of an edge being taken to be the following:

$$\forall \mathcal{V}_j \in \mathcal{N}_i : \mathcal{P}(\mathcal{E}_{ij}) = \frac{(1 - (\varphi \cdot \frac{\alpha_{\mathcal{E}_{ij}}}{90})) \cdot \mathcal{E}_{ij}^{\tau}}{\sum_{k \in \mathcal{N}_i} (1 - (\varphi \cdot \frac{\alpha_{\mathcal{E}_{ik}}}{90})) \cdot \mathcal{E}_{ik}^{\tau}} \tag{16}$$

Where $\varphi \geq 0$ is a parameter for "effort", this way we can change the effect an angle has on the algorithm. An increase in the angle or in effort, decreases the probability that an agent will walk a certain edge. Even though the algorithm also can handle negative values for φ, this would 'reverse' all slopes in the landscape, resulting in – probably – unwanted behaviour.

This turned out not to be as 'successful' as we had hoped it to be. Agents still somehow evaded the paths with a slope, most probably because of the fact that we use euclidean distances, causing paths with a slope to be a little longer than the ones that stay level. To exploit this idea, we decided to try and move the usage of this angle from the 'decision' part of the algorithms towards the update functions.

Recall from Section 3 that this is our original update function:

$$\mathcal{E}_{ij}^{\tau} = (1 - \rho) \cdot \mathcal{E}_{ij}^{\tau} + \sum_{x} \Delta_{ij}^{x}$$

If we also recall our definition of $\mathcal{E}$, where we stated there that $\mathcal{E}_{ij} \in \mathcal{E} \iff \mathcal{E}_{ji} \in \mathcal{E}$. This holds for all properties of the "original" algorithm, but when we translate the algorithm to work on three-dimensional landscapes, the angle is another detail we need to take into account. That is, we need to work with the direction on which the agent has traversed the edge, so that we can update the pheromone accordingly. Note that this adds an asymmetry in the graph update function. To achieve this, we change the definition of Δ_{ij}^{x} to the following:

$$\Delta_{ij}^{x} = \begin{cases} (1 - (\varphi \cdot \frac{\alpha_{\mathcal{E}_{ij}}}{90})) \cdot \frac{1}{L_k^x} & \text{if } \mathcal{E}_{ij} \in T_k^x \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

The most important property of this modification is that it preserves the normal functionality when we don't include any hight-properties in our testcase – that is, $\forall \mathcal{E}_{ij} \in \mathcal{E} : \alpha_{\mathcal{E}_{ij}} = 0$. In this case, the first part of equation 17 comes down to the following:

$$(1 - (\varphi \cdot \frac{0}{90})) \cdot \frac{1}{L_k^x} \implies \frac{1}{L_k^x}$$

A linear function on the angle to increase / decrease the amount of pheromone that is updated, might not be the most appropriate function. To solve this, we have taken two functions that both evaluate to -1 at $-90°$, 0 at $0°$ and evaluate to 1 at $+90°$. The functions we chose are $\frac{\alpha_{\mathcal{E}_{ij}}}{90}$ and $\sin(\frac{\alpha_{\mathcal{E}_{ij}} \cdot \pi}{180})$. The corresponding plots are shown in figure 3, in red and black respectively.

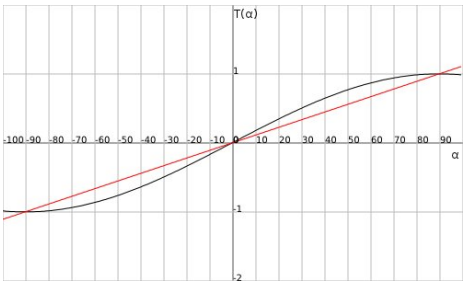

Figure 3: A plot of the two used functions

We expect to find that the non-linear sinusoid function will show a slightly different behaviour from the linear function, as we will investigate this difference in the experimental study.

5 Case Study

To support the hypothesis that adjustments to the algorithm to make them height-compatible, yield "added value" to the ACO class of algorithms we will now present a case study. We have divided our research into two parts.

First, we want the algorithms to still show the same behaviour on the original two-dimensional structures. To accomplish this, we describe two structures in Section 5.1, the same as introduced in **??** to show that the three-dimensional improvements do not adjust the behaviour on planar testcases.

Second, we want to show that our adjustments on the algorithm indeed yield added value. For this purpose, we designed three test landscapes, described in Section 5.2.

In Section 5.3 we describe our approach, and in Section 5.4 we will show the results of our research, and add some discussion.

5.1 Two-dimensional Structures

To test the behaviour of the algorithms on "normal", planar input, we chose two problems found in literature. The first problem we use, is the classic "double-bridge"-experiment, and we chose an extended version of this experiment to show that the principle also holds on more difficult problems.

Double Bridge

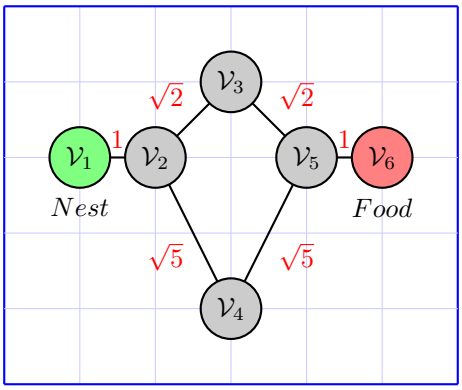

Figure 4: Double-Bridge

For the double-bridge experiment, we use Figure 4 as input. In this graph, there are two paths that do not involve cycles: $\mathcal{V}_1 \to \mathcal{V}_2 \to \mathcal{V}_3 \to \mathcal{V}_5 \to \mathcal{V}_6$ and $\mathcal{V}_1 \to \mathcal{V}_2 \to \mathcal{V}_4 \to \mathcal{V}_5 \to \mathcal{V}_6$, with path length $2 + 2\sqrt{2} \approx 4.83$ and $2 + 2\sqrt{5} \approx 6.47$ respectively. It can easily be seen that since the "weights" on all edges – their length – is non-negative, adding any cycles creates a longer path than either of these 2, for example adding a cycle $\mathcal{V}_2 \to \mathcal{V}_3 \to \mathcal{V}_5 \to \mathcal{V}_4 \to \mathcal{V}_2$ adds $2\sqrt{2} + 2\sqrt{5} \approx 7.30$ to the length of the path.

Extended Bridge

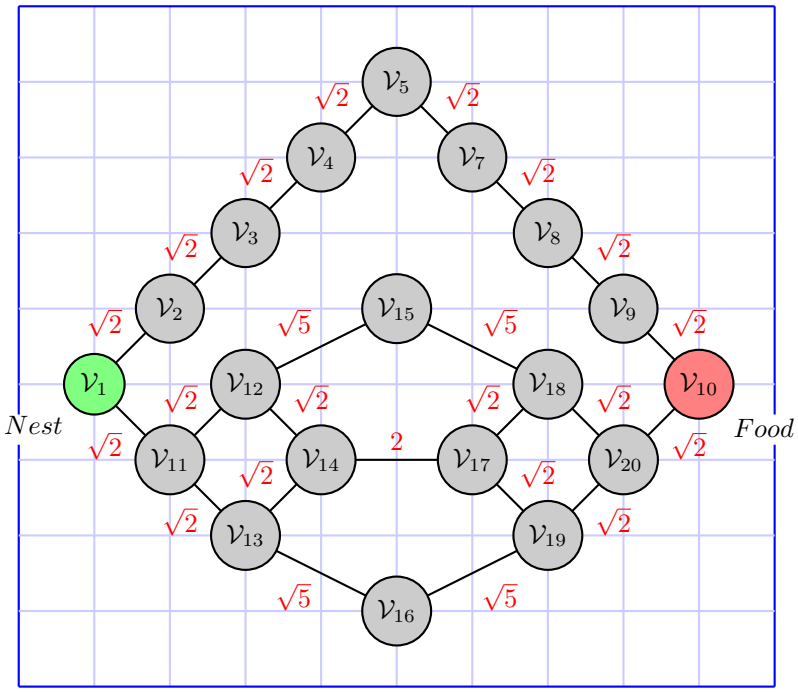

Figure 5: Extended-Bridge

Fig 5 shows an experiment based on the double bridge problem, with a slightly increased complexity. The agents can either choose an "easy", but sub-optimal path by moving towards $\mathcal{V}_2$ from the nest. This would result in the path $\mathcal{V}_1 \rightarrow \mathcal{V}_2 \rightarrow \mathcal{V}_3 \rightarrow \mathcal{V}_4 \rightarrow \mathcal{V}_5 \rightarrow \mathcal{V}_6 \rightarrow \mathcal{V}_7 \rightarrow \mathcal{V}_8 \rightarrow \mathcal{V}_9 \rightarrow \mathcal{V}_{10}$, with a path length of $8\sqrt{2} \approx 11.31$. The other option is, that they move towards $\mathcal{V}_{11}$ when starting at the nest, but this makes the task of finding a path much more difficult. It is fairly easy for the agents to find themselves caught in cycles, and with that find longer pathways than they would have found when taking the route to $\mathcal{V}_2$, but there also is a chance that they find shorter paths. One of these paths is, for example $\mathcal{V}_1 \rightarrow \mathcal{V}_{11} \rightarrow \mathcal{V}_{12} \rightarrow \mathcal{V}_{15} \rightarrow \mathcal{V}_{18} \rightarrow \mathcal{V}_{20} \rightarrow \mathcal{V}_{10}$, with a path length of $4\sqrt{2} + 2\sqrt{5} \approx 10.13$.

5.2 Three-dimensional Structures

5.2.1 Mountain

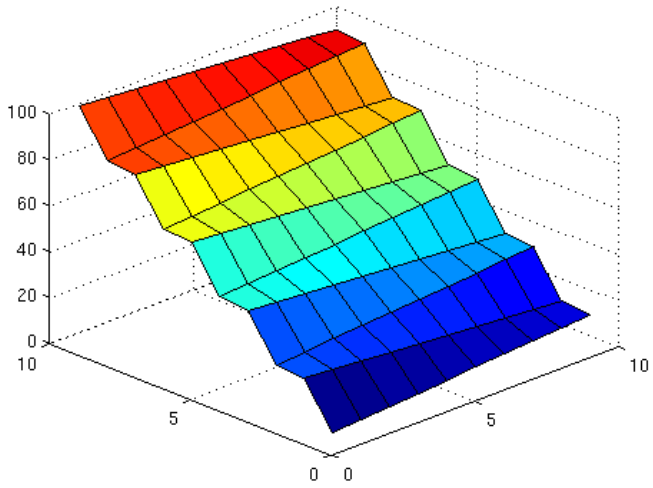

Figure 6: A landscape representing a Mountain Side

Our first three-dimensional test case describes the slope of a mountain, and a graphical representation can be found in Figure 6. The landscape can be described by the following height map:

	1	2	3	4	5	6	7	8	9	10
1	9.9	9.8	9.7	9.6	9.5	9.4	9.3	9.2	9.1	9.0
2	8.0	8.1	8.2	8.3	8.4	8.5	8.6	8.7	8.8	8.9
3	7.9	7.8	7.7	7.6	7.5	7.4	7.3	7.2	7.1	7.0
4	6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9
5	5.9	5.8	5.7	5.6	5.5	5.4	5.3	5.2	5.1	5.0
6	4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9
7	3.9	3.8	3.7	3.6	3.5	3.4	3.3	3.2	3.1	3.0
8	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9
9	1.9	1.8	1.7	1.6	1.5	1.4	1.3	1.2	1.1	1.0
10	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9

For the mountain side landscape, we can define 2 "special" paths: We can define the shortest path, given by the following vertices: $\mathcal{V}_{10,1} \to \mathcal{V}_{9,1} \to \mathcal{V}_{8,1} \to \mathcal{V}_{7,1} \to \mathcal{V}_{6,1} \to \mathcal{V}_{5,1} \to \mathcal{V}_{4,1} \to \mathcal{V}_{3,1} \to \mathcal{V}_{2,1} \to \mathcal{V}_{1,1}$ – with a path length of $5\sqrt{1^2 + 1.9^2} + 4\sqrt{1^2 + 1^2} \approx 14.76$ – and we can define a path that costs the

least effort: the path following the height indices with stepsize 1 – with path length $99\sqrt{1^2 + 0.11^2} \approx 99.49$. Because this path is 99 steps long, we will not write it down in full.

5.2.2 Waves

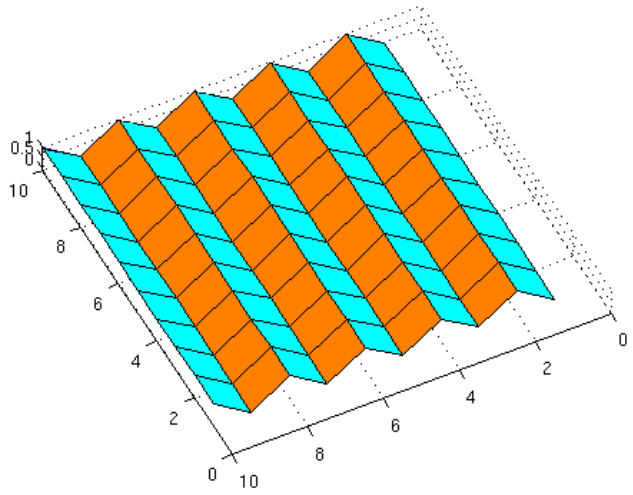

Figure 7: A 3D-Landscape representing a Wave pattern

For this test case we use the landscape represented in Figure 7. The corresponding height-map is as follows:

	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1	1	1
4	0	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	0	0	0	0	0
7	1	1	1	1	1	1	1	1	1	1
8	0	0	0	0	0	0	0	0	0	0
9	1	1	1	1	1	1	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0

For this test case there are multiple shortest paths, all consisting of 9 steps in the x-plane, and 9 steps in the y-plane, leading up to a path length of $9 \cdot 1 +$

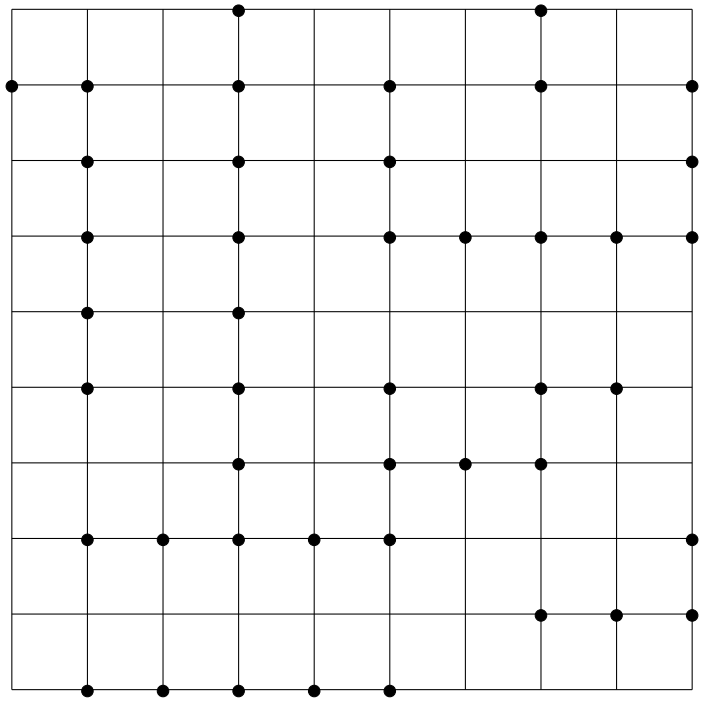

Figure 8: A 3D-Landscape representing a Maze

$9\sqrt{1^2+1^2} \approx 21.73$. When adjusting φ we expect the agents to find a longer path, following the lower area's until they find come to a split where they have to move up, move down immediately again and follow the lower area back. The expected path length for this situation would be: $45 \cdot 1 + 9\sqrt{1^2+1^2} \approx 47.73$

5.2.3 Maze

This last test case – shown in Figure 8 – represents a maze, of which the 'black' dots are the walls, and the other intersections represent the floor. The idea is that if we give the walls a height of one, we create boundaries surrounding a few paths, and the agents should not cross these boundaries. The height map in Table 5.2.3 illustrates this better.

	1	2	3	4	5	6	7	8	9	10
1	**0**	**0**	**0**	1	**0**	**0**	**0**	1	**0**	**0**
2	1	1	**0**	1	**0**	1	**0**	1	**0**	1
3	0	1	**0**	1	**0**	1	**0**	**0**	**0**	1
4	0	1	**0**	1	**0**	1	1	1	1	1
5	0	1	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**
6	0	1	**0**	1	0	1	0	1	1	**0**
7	**0**	**0**	**0**	1	0	1	1	1	**0**	**0**
8	**0**	1	1	1	1	1	**0**	**0**	**0**	1
9	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1
10	0	1	1	1	1	1	0	0	0	0

Table 1: The height map of the maze

The shortest path through the maze, if we see the 'walls' as normal hills, would be $\mathcal{V}_{1,1} \to \mathcal{V}_{1,9} \to \mathcal{V}_{1,8} \to \mathcal{V}_{1,7} \to \mathcal{V}_{1,6} \to \mathcal{V}_{1,5} \to \mathcal{V}_{1,4} \to \mathcal{V}_{1,3} \to \mathcal{V}_{1,2} \to \mathcal{V}_{1,1}$,with a length of $5 \cdot 1 + 4\sqrt{1^2 + 1^2} \approx 10.66$

The path we would want the agents to find when introducing φ, is the path indicated in bold markings in the height map – the solution of the maze. This path has a length of 43, but would require the least effort, as we move neither up nor down. The given path is the only instantation of a path without any climbing.

5.3 Test setup

As we have divided our case study into two parts – correspondence on the original inputs, and yielding an added value on 3-dimensional structures – we will also take this into account with our test set-up.

For the first part of our study, we want the adjusted versions of the algorithms to – in general – correspond with the "normal" versions of the algorithms on the 2-dimensional test subjects. We will run all algorithms on the 2 test-problems, and will register the shortest path length found.

As we are not interested in a single run of the algorithm but in the overall comparison, we will run each algorithm 100 times with different random seeds, and then report the arithmetic mean of the path length to be able to compare the different strategies.

For the second part, we would like to see that our adjusted versions find an "effort-balanced" path to the food source than their original version. The parameter we need to set in the adjusted algorithms is φ, to regulate the effect of the of a path angle on the probability of this path being taken. Furthermore, we need to define for a problem what kind of path we want the algorithm to find. If this is purely the shortest path, we want to set φ to 0, but if we want to find the path that takes the least effort, we will be setting φ to a larger value, to "prohibit" the ants from moving up with too high angles at a time. We have also taken this into account while defining our test cases to represent some nice

effects we can get by this.

The questions we want to get an answer for in the 3-dimensional environments, are which value we need to give φ such that the agents find the "optimal balanced" paths described above, and what the differences are between the adjusted algorithms with φ set to 1, and their original counterparts. Since these questions require more calculations after the algorithms are done computing, we will show these results for a single random seed.

5.4 Results

5.4.1 Comparison

As already described in Section 5.4, we start our case study with a small test on correspondence of our adaptations in the algorithm with the original versions. The key point in the original algorithms, is that the shortest path in a graph is found and followed by most of the agents. As we want our adaptations to only take place when height comes into play, we want these adapted version to also give rise to this property.

The tests we took to describe their behaviour are the "Double Bridge" experiment – as described in Section 5.1 – and an extended version hereof, the "Extended Bridge" experiment – as described in Section 5.1.

In the table below, a small overview of our results is given. We have used per algorithm 100 different random seeds – meaning that we have used 100 different, completely indepedent runs of the algorithm – and registered for each of these runs the shortest path that the agents would find.

			Simple ACO		ASrank	
		Shortest Solution	Linear Gradient	Sinus Gradient	Linear Gradient	Sinus Gradient
Double Bridge	μ	13.66	13.66	13.66	13.66	13.66
	σ	0	0	0	0	0
Extended Bridge	μ	9.66	9.66	9.66	9.66	9.66
	σ	0	0	0	0	0

Table 2: A comparison of our algorithms on two input problems

As we can see, each of our adapted algorithms has found the shortest path within the amount of iterations we let it run. Ofcourse, this data means nothing without the parameters we have used for the experiments:

Parameter		Value	
m	(Amount of Agents)	8	
ρ	(Evaporation factor)	0	
φ	(Effort factor)	0	(Not of any influence)
w	(ASrank Cutoff rank)	10	
Amount of iterations		1000	

This show, amoung other things, that we only need a relatively small amount of agents to let the algorithm converge to the optimal solution. It also shows, that no pheromone evaporation is need in order to make the algorithm converge, a factor that has been shown to be crucial on more difficult problems – such as for example Traveling Salesman Problem-instances.

5.4.2 Tuning φ

The biggest part of our research has been dedicated to examining the influence of φ on the algorithms, regulating the strength in which the angle of an edge is accounted for in the updating of pheromone on that edge.

The following three paragraphs show the results we have attained, followed by a small discussion.

Mountain Slope

The table below is a small overview of the results we have gained from letting the algorithms run on the Mountain Slope-testcase. We show merely the results of all runs on one specific random seed, due to computational factors – but the results for different random seeds are similar.

			Shortest	Optimal	1	2	4	8	16
Simple ACO	Linear Gradient	Length	14.76	100.5	14.76	17.23	17.40	17.40	17.40
		Correct Edges	4	100	4	4	3	3	3
	Sinus Gradient	Length	14.76	100.5	17.23	14.76	14.76	17.40	17.40
		Correct Edges	4	100	4	4	4	3	3
ASrank	Linear Gradient	Length	14.76	100.5	17.73	14.76	16.22	14.76	14.76
		Correct Edges	4	100	3	4	4	4	4
	Sinus Gradient	Length	14.76	100.5	14.76	14.76	14.76	14.76	14.76
		Correct Edges	4	100	4	4	4	4	4

Table 3: Results of the algorithms on the Mountain Slope-testcase

We show for each algorithm-gradient combination the path the algorithm converged into after 1000 iterations, with eight agents. From this path, we show both the length of this path, as well as the edges contained in the path that correspond with the "most optimal" solution described above – "Correct

Edges". An edge is only counted as correct if it is being traversed in the same direction as the optimal solution.

As we can see, the maximal amount of edges that the agents have found corresponding to the optimal solution doesn't exceed the amount of "Correct Edges" already found if we take the shortest path through the graph – and in some of the cases the solution found is equal to this shortest path.

Maze

For the maze result, our results – as shown below – are a little more interesting. Again, we let all the algorithms run 1000 iterations on the problem, and retrieve from this the most emergent path.

			Shortest	Optimal	1	2	4	8	16
Simple ACO	Linear Gradient	Length	9.66	43	15.49	11.66	9.66	9.66	9.66
		Correct Edges	5	43	2	6	5	5	5
	Sinus Gradient	Length	9.66	43	15.49	13.49	9.66	11.66	9.66
		Correct Edges	5	43	4	3	5	6	5
ASrank	Linear Gradient	Length	9.66	43	13.49	15.90	18.31	23.14	41.63
		Correct Edges	5	43	3	3	6	0	5
	Sinus Gradient	Length	9.66	43	13.49	13.49	13.49	13.49	13.49
		Correct Edges	5	43	4	4	4	4	4

Table 4: Results of the algorithms on the Maze-testcase

As we can see in these results, we have a couple of runs on which the algorithm finds more "Correct Edges" than the shortest solution already gives us, and at two places even at only a small increase of the length. As we can also see, we have one solution that has not found any edges that we denote as "Correct", but this does not mean the adaptions are useless on this specific run. The path found has little overlap with the shortest solution, showing that our agents do indeed explore other possibilities as well.

Wave Pattern

Our last testcase is a wave pattern on which we would like the agents to "evade" the waves, staying as long as possible on the 'lower' set of edges. As shortest solution we took one of the solutions that has an optimal amount of "Correct Edges", but there are also shortest solutions that have absolutely no overlap with the most optimal solution defined before.

			Shortest	Optimal	1	2	4	8	16
Simple ACO	Linear Gradient	Length	21.73	57.73	23.73	26.56	21.73	21.73	23.73
		Correct Edges	13	54	11	8	8	12	10
	Sinus Gradient	Length	21.73	57.73	21.73	21.73	21.73	21.73	24.56
		Correct Edges	13	54	12	2	12	12	4
ASrank	Linear Gradient	Length	21.73	57.73	21.73	26.56	26.56	26.56	26.56
		Correct Edges	13	54	4	6	6	4	6
	Sinus Gradient	Length	21.73	57.73	23.73	23.73	23.73	23.73	23.73
		Correct Edges	13	54	3	3	3	3	3

Table 5: Results of the algorithms on the Wave Pattern-testcase

The results on this testcase are very diverse. For one algorithm – ASrank with Sinus Gradient – the algorithm kept converging into the same path, no matter what value for φ we fed into the algorithm, while for other algorithms the value of φ heavily controlled the path in length as in correct edges. However, none of the results exceeds the 13 correct edges found in the shortest path.

Discussion

Even though taking the shortest path is a computationally cheaper and therefore a more optimal solution to the problem, the adaptions suggested for the algorithms are not useless. The adaptions when placed into the algorithms do actually influence the way the algorithm behaves, though not yet in the way we would like them to.

One of the things that we think to be a factor in the algorithm not yet behaving as we would like, are the random fluctuations that are needed for the algorithms to function properly – the algorithm needs an exploration phase in which all edges contain approximately the same amount of pheromone on each edge. A solution we think, might be found in combining the 2 solutions tried for incorporating height, and during the algorithm slowly switch from one to another.

Another problem might be, that the edges that are chosen in the first few iterations of the algorithm are reinforced to strongly. When we increase φ, we also increase the amount of pheromone update on edges that are pointed downwards. However, this may lead to agents approaching this edge from the other side, give a higher probability to this edge being part of an optimal solution – resulting in unwanted behaviour. This problem we want to try and counteract by slowly increasing φ during the iterations, instead of taking a static number. Even though this increases exploration possibilities of the algorithm, it also makes the problem more dynamic, a problem for which we would need to find a solution as well.

6 Conclusions and Future Research

Even though the results shown in this paper are not as impressive as we would have liked them to be, our even as qualitative as one would want, it still opens possibilities for further research. One of these ideas is to use more realistic testcase. For the adaptions to have any use at all, we need to test them on realistic problems, and see what we get out of this.

Anothe possibility to make the algorithms more realistic, is to use a Moore-neighbourhood, in which each node is connected to eight others. Other functions for the gradient are possible, as well as the usage of phsysics for this – for example gravity.

The incorporation of height in Ant Colony Optimization algorithms opens up a lot of possibilities and is very promising – if we can find a way to get them to work in the way we want them to.

References

[1] Bonabeau, E. Editor's introduction: Stigmergy. *Artificial Life – Special issue on Stigmergy* (1999).

[2] Bullnheimer, B., Hartl, R., and Strauss, C. A new rank based version of the ant system: A computational study. *Central European Journal for Operations Research and Economics* (1999).

[3] Dorigo, M., Caro, G. D., and Gambardella, L. Ant algorithms for discrete optimization. *Artificial Life* (1999).

[4] Heywood, I., Cornelius, S., and Carver, S. An introduction to geographical information systems.

[5] Reidys, C., and Stadler, P. Combinatorial landscapes. *SIAM Review* (2002).

[6] White, T., Kaegi, S., and Oda, T. Revisiting elitism in ant colony optimization.

A JSON

For reading input into the simulator, we have chosen to use the "JSON" fileformat. JSON is an abbreviation of JavaScript Object Notation, but the format is more widely usable than just in Javascript. One of the key properties for using this fileformat, is it readability in combination with the compact structure. We have used 2 different formats for defining a problem, and also a format for defining the settings.

A.1 Testcases

A.1.1 2-Dimensional

```
{
  "Nodes":[
    [0,2,0],
    [4,2,0],
    [6,0,0],
    [6,6,0],
    [8,2,0],
    [12,2,0]
  ],
  "Edges":[
    [0,1],
    [1,2],
    [1,3],
    [2,4],
    [3,4],
    [4,5]
  ],
  "Nest":0,
  "Food":5
}
```

Figure 9: An example of a 2-dimensional structure: the "Double Bridge Experiment"

This dataformat can be used for creating both two- and three-dimensional problems. First, we have a list of nodes, in the format $[X, Y, Z]$, representing the X, Y and Z coordinates of each node. After that, we have a list of edges, using the nodes, with a zero-index.

A.1.2 3-Dimensional

```
{
  "Landscape":[
    [0,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1]
  ],
  "Nest":0,
  "Food":63
}

```

Figure 10: An example of a 3-dimensional landscape: our "Waveboard" Testcase

This format can be used to easily create landscapes. Instead of having to manualy define each node and every connection, we use a height map as input. This implicitly defines all of the nodes, and the edges between them, as we defined to use a Von-Neumann neighbourhood.

A.2 Settings

```
{
  "Name":"Test",
  "Ants":50,
  "Iterations":1,
  "Evaporation":1,
  "Q":1,
  "W":10,
  "Effort":1,
  "Timesteps":1000,
  "Seed":0,
  "Autocalc":1
}
```

Figure 11: An example of an inputfile

This file format speaks for itself, the settings file is very readable, and every setting can be easily input into the algorithms.

B Loop Elimination

If we define the ant's memory as an array of integers, the pseudocode for the first 2 'flavours' of our loop elimination algorithms are as follows:

Forward Loop Elimination

```
for(i = 0; i < memory.size(); i++) {
  read memory[i];
  for(j = memory.size() -1; j > i; j--) {
    if (memory[i] = memory[j]) {
      break;
    }
  }
  erase elements memory[x] where i+1 <= x <= j ;
}
```

Backwards Loop Elimination

```
for(i = memory.size() -1; i >= 0 ; i--) {
  read memory[i];
  for(j = 0; j < i; j++) {
```

```
      if (memory[i] = memory[j]) {
        break;
      }
    }
    erase elements memory[x] where j <= x <= i-1 ;
  }
```

The 'Optimal Loop Elimination' algorithm actually performs both on a copy of the memory, calculates which solution is the most optimal one, and stores that solution in the memory of the ant.

B.1 Why multiple options

To justify the implementation of multiple algorithms for the same purpose, following are a few examples of paths that one can choose, with the corresponding graph, and the length of the solutions.

Nodes			Heights		
1	2	3	0	0	0
4	5	7	0	1	0
7	8	9	0	0	0

With 1 as the nest node, and 9 as the food node.

Path 1: No difference

Path:
1 2 5 4 7 8 5 2 3 6 9
Forward Loop Elimination:
1 2{5 4 7 8 5 2}3 6 9 $\rightarrow$ 1 2 3 6 9
Backward Loop Elimination:
1{2 5 4 7 8 5}2 3 6 9 $\rightarrow$ 1 2 3 6 9

As we can see above, the resulting paths are exactly the same, even though the loop has been eliminated on a different spot (between { and }), thus 'Optimal Loop Elimination' does not have to make a choice. This is the type of solution that also holds when we have a path without loops, for when there are no loops to eliminate at all, the resulting memory will ofcourse be the same as it was. However, just the fact that the paths are equal, we will still be wanting to calculate the length of the solution to compute how much the algorithm 'profits' from this technique:

Path:
$1 + \sqrt{2} + \sqrt{2} + 1 + 1 + \sqrt{2} + \sqrt{2} + 1 + 1 \rightarrow 5 + 4\sqrt{2}$
Loop Elimination:
$1 + 1 + 1 + 1 \rightarrow 4$
Speedup:

$$\frac{1 + 4\sqrt{2}}{5 + 4\sqrt{2}} \approx 62\%$$

Path 2: Forward Loop Elimination Advantage

Path:
1 2 5 4 7 8 5 2 1 4 7 8 9
Forward Loop Elimination:
1{2 5 4 7 8 5 2 1}4 7 8 9 → 1 4 7 8 9
Backward Loop Elimination:
1 2 5 4 7{8 5 2 1 4 7}8 9 → 1 2 5 4 7 8 9

In this example, we see a difference in the amount of nodes on the path, Forward Loop Elimination in this case provides a path with less edges to taverse. However, just stating this is not enough to tell us that the solution found this way is also the shortest path, we actually need to determine the length to find the optimal solution. The solutions with their corresponding speedups:

Path:
$1 + \sqrt{2} + \sqrt{2} + 1 + 1 + \sqrt{2} + \sqrt{2} + 1 + 1 + 1 + 1 + 1 \rightarrow 8 + 4\sqrt{2}$
Forward Loop Elimination:
$1 + 1 + 1 + 1 \rightarrow 4$
Speedup:

$$\frac{4 + 4\sqrt{2}}{8 + 4\sqrt{2}} \approx 71\%$$

Backward Loop Elimination:
$1 + \sqrt{2} + \sqrt{2} + 1 + 1 + 1 \rightarrow 4 + 2\sqrt{2}$
Speedup:

$$\frac{4 + 2\sqrt{2}}{8 + 4\sqrt{2}} \approx 50\%$$

We can now easily see that Forward Loop Elimination in this case provides a bigger speedup, thus the 'Optimal Loop Elimination' algorithm would decide on storing that solution into the ants memory.

Path 3: Backward Loop Elimination Advantage

Path (Length):
1 2 3 6 5 2 1 4 5 6 9 [$5 + 4\sqrt{2}$]
Forward Loop Elimination:
1{2 3 6 5 2 1}4 5 6 9 → 1 4 5 6 9 [$2 + 2\sqrt{2}$]
Backward Loop Elimination:
1 2 3{6 5 2 1 4 5}6 9 → 1 2 3 6 9 [4]

In this case, it is clear that we had to calculate the lengths to find the optimal solution. Both solutions have the same amount of nodes to pass, but because node 5 is located higher, the length of this solution becomes greater than the other solution. Corresponding speedups:

Forward Loop Elimination:

$$\frac{3+\sqrt{2}}{5+4\sqrt{2}} \approx 55\%$$

Backward Loop Elimination:

$$\frac{1+4\sqrt{2}}{5+4\sqrt{2}} \approx 62\%$$

Looking at the above examples, we can see that just implementing Forward Loop Elimination, or just Backward Loop Elimination does not guarantee the optimal solutions. However, we did implement all versions, for analytical purposes.