

Visualization nature-inspired meta-heuristics for optimization problems

Hans Erik Nielsen

DTU



Kongens Lyngby 2012

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary (Danish)

Målet for denne afhandling er at visualisere og implementere forskellige metaheuriske algoritmer og derved vise at disse algoritmer kan bruges i praksis. Metaheuriske algoritmer er inspireret af naturen, og indeholder en vis tilfældighed, men på trods af de opfører sig tilfældigt kan de bruges til at få resultater, som optimale bit-strenger eller korteste vej rundt i en graf. Algoritmerne $(1+1)$ EA og RLS er visualiseret på den måde at spredningen af 1-taller i en bit-streng afgør positionen af et punkt i en rund figur. SA og ACO med MMAS er visualiseret i en 2-d graf, hvor problemet med Travelling Salesman Problem løses ved hjælp af disse algoritmer. Visuelt bliver rundturen i grafen kortere og kortere, jo længere algoritmerne kører. Der bliver beskrevet, hvordan algoritmerne er gået fra teori til en software implementation og hvilke problemer, der har opstået i forbindelse med implementationen. Desuden beskrives forskellige videreudviklinger af programmet, som kunne implementeres i en ny version af programmet. Algoritmernes effektivitet i softwaren er testet og analyseret. Testene beskriver, hvilken af $(1+1)$ EA og RLS, der opnår den bedste effektivitet i udviklingen af den stærkeste bit-streng. Testene ved Simuleret Annealing og Ant Colonization Optimazation tester deres effektivitet og sammenligner denne med brute force. Deres effektivitet sammenlignes desuden med hinanden. Der reflekteres over deres effektivitet og der gives et bud på deres effektivitet i større data-mængder.

Preface

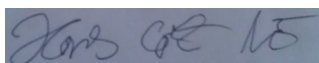
This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an B.Sc. in Informatics.

The thesis deals with Visualization of nature-inspired meta-heuristics for optimization problems. In which some meta-heuristics algorithms are investigated and later implemented into a software framework to show they are working and also that they are working efficient.

The thesis consists of a description of the used algorithms, software implementation and test/analysis.

The references are in usual L^AT_EX typesetting notation, details about the source is available in the Bibliography list.

Lyngby, 09-May-2012

A handwritten signature in blue ink, appearing to read 'Hans Erik Nielsen', on a light blue background.

Hans Erik Nielsen

Acknowledgements

I would like to thank my supervisor, Carsten Witt, for advices and answers for my many questions during the duration of the project and also supporting me with lots of material and implementation ideas. I would also like to thank my family and friends for understanding my priority of the time between the project and the time being with them. My workplace also deserves a great thanks for being flexible when it comes to my work time. Last I want to thank my girlfriend, Sara, for supporting me through out the project and waking me up in the morning.

Contents

Summary (Danish)	i
Preface	iii
Acknowledgements	v
1 Introduction	1
2 Basic Graph Theory	3
3 Nature-inspired algorithms	7
3.1 Evolution algorithms	7
3.2 Fitness functions	8
3.3 Comparing (1+1)EA and RLS	9
3.4 2-opt	9
3.5 Simulated annealing	10
3.6 Ant colonization optimization	11
3.7 Comparing SA and ACO	12
4 Software	15
4.1 Choice of platform	15
4.2 Requirement Analysis GUI	15
4.3 Graphical User Interface	16
4.4 Implementation of the algorithms	17
4.4.1 (1+1)EA and RLS	17
4.4.2 Permutations	18
4.4.3 File Load	18
4.4.4 Simulated Annealing	19
4.4.5 Ant Colonization Optimization	20

4.5	Problems during implementation	21
4.5.1	Technical problems	21
4.5.2	Algorithmic	22
4.6	Possible extensions	23
4.6.1	Algorithmic	23
4.6.2	Technical	24
5	Analysis and Results from the program	25
5.1	(1+1)EA and RLS	25
5.1.1	Goal of the test	25
5.1.2	Test	26
5.2	Simulated Annealing	26
5.2.1	Goal of the test	26
5.2.2	Test	27
5.2.3	Conclusion on SA	28
5.3	ACO	28
5.3.1	Goal of the test	28
5.3.2	Test	29
5.3.3	Conclusion on ACO	30
5.4	Comparing ACO to SA	30
5.4.1	Comparing results	30
5.5	Conclusion on the permutation test	30
6	Conclusion	33
A	Tables and Graphs	35
	Bibliography	43

CHAPTER 1

Introduction

Meta-heuristics algorithms are methods to solve problems, where these methods are based on things in the nature. Meta-heuristic algorithms are often used in optimization problems, where they evolve into a better solution over time. Most of the research of meta-heuristic algorithms is relative new[10] and the algorithms are therefore not used in many applications, in this thesis I will show that meta-heuristic algorithms can work in an application. To give a better on understanding how they work, the algorithms are implemented visually, the framework of this thesis is also implemented in a way, where it is easy to extend the program. More specific I will show that an Evolution Algorithm can be implemented visually and how it can evolve over a string generation over generation. I will show how Simulated Annealing and Ant Colonization Optimization can be visually implemented to find short paths in a graph. The thesis consists of 6 chapters, where the first is this introduction. The second chapter describes a bit graph theory used in the thesis. The third chapter is a review of the theoretical algorithms used in the program. The fourth chapter is a description on how the algorithms has gone from theory into the actual implementation. The fifth chapter covers testing of the program in regards to the algorithms and also contains a bit of analysis of the performance. The sixth chapter contains a small summary of the thesis and conclusion.

CHAPTER 2

Basic Graph Theory

In order to explain the algorithms and data structures used in this thesis, a bit basic knowledge of graph theory is needed. In this thesis a graph (G) consists of a finite number of vertices called (v) and finite number of edges called (e). An edge (e) connects two vertices to each other and the two vertices relationship with each other is neighbours N . Each edge has a weigh, one can define this as the cost of travelling on the specific edge. In this thesis and implementation this cost is always the Euclidean distance, i.e. the length between the two vertices. A cycle in graph theory is when you travel from a vertex (v) in (G) along a path that consists of distinct edges and you end up in the same vertex (v) as you began. An example on a cycle is drawn in Figure [2.1](#).

A Hamiltonian cycle, the definition is based on the web-page [\[9\]](#), is a path were all vertices are connected to each other in that way that if you start at a random vertex (v) in the graph (G) and travel along all the edges, you will at some point end up in the starting vertex and also have visited all the other vertices in (G). An example of a Hamiltonian cycle is given in Figure [2.2](#).

Some of the definitions are a bit free, not all edges have to be travelled to form a Hamiltonian cycle, the Hamiltonian cycle can be formed by a sub-part of the graph as long as all the vertices are contained in the path. This extra definition of a Hamiltonian cycle will be used in the implementation of the ACO algorithm. The definition of a Hamiltonian cycle is the fundamental idea behind

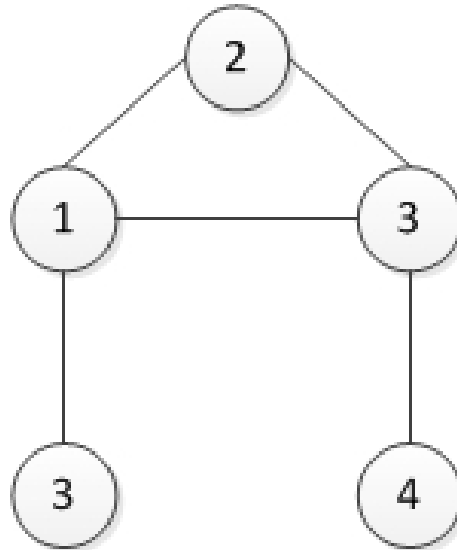


Figure 2.1: The points 1,2,3 together form a cycle

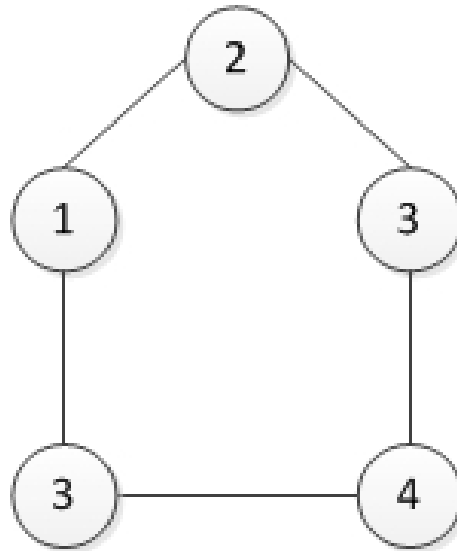


Figure 2.2: All the points in the graph(1,2,3,4,5) together form a Hamiltonian cycle

the travelling salesperson problem(TSP). To find a solution for TSP, one must find a Hamiltonian cycle in the graph. However TSP is not only about finding

a solution, it is also about finding a good solution. A good solution means that finding a Hamiltonian cycle with a short distance and it is preferable to find the shortest cycle of them all. But finding the shortest cycle is not the most simple thing in the world. TSP is a so called NP-complete problem, which means all combinations have to be tried before it can be verified which is the shortest cycle. Using this a simple brute force algorithm would find the shortest path by going through all the combinations one by one. This will however take a lot of time and is not very efficient, if there are more than 4 vertices. This thesis and implementation will use meta-heuristics algorithms instead of brute force to make faster and efficient solutions to the TSP. These solutions might not find the shortest path, but they will however give a good estimate on the shortest path.

CHAPTER 3

Nature-inspired algorithms

The meta-heuristic algorithms that are visualized in the program are described in this section. There is a description of each of the algorithms both in pseudo-code and text. The sections also holds theory for the non-meta-heuristic algorithms used in the implementation.

3.1 Evolution algorithms

Evolution describes the changes from a former generation to a newer generation. The changes can both be good and bad, but is made to overcome a problem. Eg. this can be seen in the evolution of laptops which has evolved to smaller devices to make them more mobile. A theory, that uses evolution as it's backbone, is survival of the fittest or natural selection. The theory predicts that the individuals that are best to adapt, has the highest chance to survive. Evolution algorithms are based on this idea. The algorithm implemented in the program is based on a the algorithm (1+1) EA, which is described in [7]. The algorithm 'Random Local Search' is described in the same book, which has also been implemented, this algorithm is not a meta-heuristic, however it is similar to (1+1) EA and it is therefore good to use this algorithm to compare with the (1+1) EA. The two algorithm is (1+1) EA Algorithm 1 and RLS Algorithm

2. The algorithm 1 finds the strongest individual of the parent and child. The strongest of the two survives while the other one dies. This means that a parent can live for many generations before a child becomes stronger than its parent and take over the role as parent. To determinate who is the strongest, a fitness function is used. Each bit in the string has a $1/n$ chance of flipping its value. The RLS Algorithm 2. has the same principals as (1+1) EA Algorithm 1. But instead of each bit having $1/n$ chance of flipping. RLS chooses a random bit and flips it. In RLS one bit will always flip, but also at most one bit will flip, while (1+1) EA potentially all bits can flip, but also no bits at all.

Algorithm 1 (1+1)Evolution Algorithm (1+1) EA_b

1. Choose $s \in 0, 1n$ uniform at random.
 2. Produce s' by flipping each bit of s independently of the other bits with probability $1/n$.
 3. Replace s with s' if $f(s') \leq f(s)$
 4. Steps 2 and 3 forever.
-

Algorithm 2 Algorithm 1 Random Local Search(RLS)

1. Choose $s \in 0, 1n$ uniform at random.
 2. Choose $i \in 1, \dots, n$ uniform at random and flip the i 'th bit of s .
 3. Replace s with s' if $f(s') \leq f(s)$.
 4. Steps 2 and 3 forever.
-

3.2 Fitness functions

This part describes the fitness functions used in the program. A fitness function can in a few words be described as a rule, which decide whether the child generation is stronger than it's parent. In the both algorithm 1 and 2 this is the step 3. The fitness functions used in the program are OneMax and LeadingOnes. OneMax chooses the bit-string containing the most ones. LeadingOnes is similar, it chooses the bit-string containing the most consecutive serie of 1's.

3.3 Comparing (1+1)EA and RLS

As mentioned before it makes sense to compare (1+1) EA to RLS. The basic idea behind both algorithms are almost identical. They have almost the same running time as both of them in average changes one bit per generation, however as pointed out in e.g book [3] RLS is a bit better than (1+1) EA, this theory will be tested using the implementations of the algorithms. The investigation and results are described later on in section 5.1.

3.4 2-opt

2-opt is not a meta-heuristics algorithm. But mere an algorithm used to switch 2 edges [8]. The original idea is to switch edges that crosses with edges that does not cross. In this thesis it is used to switch the edges for a meta-heuristic algorithm, namely Simulated Annealing(SA). SA will be described in section 3.5. The implementation used in this thesis also makes sure that the Hamiltonian circle is not broken. That is when 2 edges switch the graph will still be one connected graph. To solve this problem, all the edges consists of a 'To' and a 'From' vertex. The graph has a flow in one direction one could say. This can be used to make sure that the graph stays connected. When two edges are deleted, figure 3.2, and are about to switch for two other edges, the two 'To' vertices will connect to each other and vice versa for the 'From' vertices, this will make sure that the graph stays connected. The flow in the graph will then be messed up, figure 3.3. So the flow has to be re-ordered to be ready for another 2-opt step. This is done by taking a vertex in the graph and finding one of it's connected neighbour, make the flow go that way, then find the neighbour's neighbour and make the flow go that way. This is repeated until you end up in the start vertex, see figure 3.4. The flow is then sorted and is ready for another 2-opt. This whole description is illustrated in figures 3.1, 3.2, 3.3 and 3.4. .

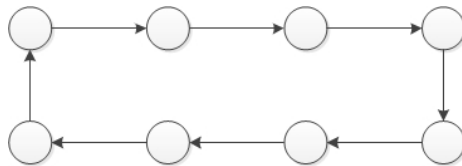


Figure 3.1: How the graph is connected at the start point

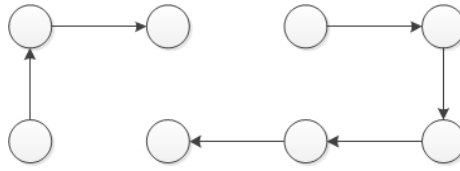


Figure 3.2: Two edges is deleted to be replaced by two new ones

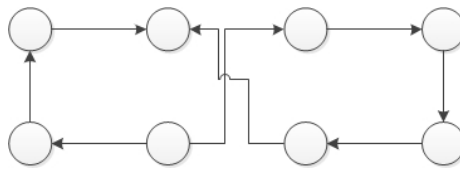


Figure 3.3: The new edges connects with a partner with the same flow eg. out to out

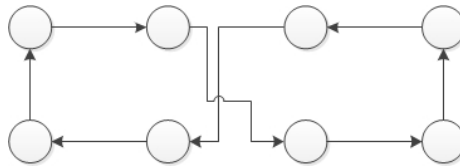


Figure 3.4: Edges with opposite flow gets redirected

3.5 Simulated annealing

SA is based on the natural process that happens when heated iron gets cooled. The small crystals in the iron goes from jumping around in a random pattern to jump around in a strict pattern. The algorithm 3 is loosely based on the algorithm in article [6]. The algorithm 3 chooses two edges and switches them, this step is done by 2-Opt in the implementation, if the new solution has a shorter path it will switch to the new solution. If the new solution instead has a longer path, it has a chance to switch to this solution. The chance for switching to the new solution is based on the current temperature. The current temperature is

calculated using the current generation of the run. The temperature starts at a high degree and goes towards lower degree. A high temperature means a good chance to do a switch to a worse solution. And the lower the temperature gets the lower the chance of switching to a worse solution will get. The temperature section of the algorithm has been made to avoid getting into local minimums and therefore not finding a good solution.

Algorithm 3 Simulated Annealing (SA)

Require: $t := 1$

Require: $T(0) = |E|^3$

while true **do**

 Choose randomly two distinct edges e' and e'' in the graph G , e' and e'' must not have any vertices v in common

 let $(v_i, v_{i+1}), (v_j, v_{j+1}), i < j$ be the chosen edges p with weight sum $s1$. $(v_i, v_j), (v_{i+1}, v_{j+1}), i < j$ is the two edges p' after a 2-opt move and their summed weight is $s2$.

 Get the current temperature $T(t)$ and a random value r between 0 and $T(0)$, to calculate probability to accept a worse solution.

if $s2 < s1$ **then**

 change p to p'

else if $r < T(t)$ **then**

 change p to p'

else

 keep p

end if

$t + 1$

end while

3.6 Ant colonization optimization

ACO is an algorithm based on ants abilities to find the shortest path to their food sources from their nest, the first algorithm was proposed in 1992 by Marco Dorigo [1]. In this thesis and program the ACO algorithm known as ACO MMAS(Max-Min Ant System) has been implemented and investigated, the algorithm is described in the article [5]. The pseudo code is displayed in algorithm 4, algorithm 5 and algorithm 6. The algorithm is described in a similar way in book [7], however the algorithm described in article [5] seemed more simple and it is the one that is used in this thesis. The algorithm constructs a path in the graph based on a chance to find the next vertex. When a path has been

constructed, the algorithm checks whether the new path is shorter than the previous found path. The shortest path updates pheromone on its edges, meaning the chance to select the same edges again increases and this continues as long as wanted. The Max-Min name in the algorithm is an reference to the update function 6, which makes sure that edges chances does not become too small or too big. This step is made to make the algorithm more effective and prevent the algorithm to be locked at a given path.

Algorithm 4 The algorithm MMAS *.

Require: 1. **function** MMAS * on $G = (V, E)$ is $\tau(e) \leftarrow 1/|V|$, for all $e \in E$;
 2. $x' \leftarrow \text{construct}(\tau)$;
 3. $\text{update}(\tau, x')$;
while *true* **do**
 4. $x \leftarrow \text{construct}(\tau)$;
if $f(x) < f(x')$ **then**
 5. $x' \leftarrow x$;
end if
 6. $\tau \leftarrow \text{update}(\tau, x')$;
end while

Algorithm 5 The algorithm construct

Require: 1. **function** construct based on $\tau, \eta, \alpha, \beta$ is
for $k = 0$ to $n - 2$ **do**
 2. $R \leftarrow \sum_{y \in N(e_1, \dots, e_k)} \tau(y)^\alpha \times \eta(y)^\beta$;
 3. Choose one neighbour $z \in N(e_1, \dots, e_k)$ where the probability of selection
 of any fixed $z \in N(e_1, \dots, e_k)$ is $\frac{\tau(z)^\alpha \times \eta(z)^\beta}{R}$
end for
 4. Let e_n be the (unique) edge completing the tour;
 5. return (e_1, \dots, e_n) .

3.7 Comparing SA and ACO

SA and ACO does not have much in common other than they can be used to solve the same problems. However they form a similar problem as RLS compared

Algorithm 6 $\tau' = \text{update}(\tau, x)$

```

 $\tau' =$ 
if  $e \in E(x)$  then
     $\min((1 - p) * \tau(e) + p, \tau_{max})$ 
else
     $\max((1 - p) * \tau(e), \tau_{min})$ 
end if

```

to (1+1) EA did. One of the algorithms, SA, tries to change it's solution for every generation, but it can only do one change. While the other algorithm, ACO, only has a chance to change it's solution per generation, but can make as many changes as there are edges. This comparison will be investigated in section [5.4](#).

CHAPTER 4

Software

This chapter contains descriptions and thoughts on how the program design and algorithms has been implemented in a framework.

4.1 Choice of platform

For the implementation .NET with C# and WPF(Windows Presentation Foundation) is used. The reasons why I choose to work on the .NET platform rather than JAVA or other technologies is that it is easy to make a good looking graphical user interface but also to get some more experience in working with the .NET technology. The disadvantages to use .NET rather than JAVA is that it is only compilable on windows based machines. The book [2] has been used as a reference from time to time, when implementing various elements.

4.2 Requirement Analysis GUI

Before starting the actual coding of the program, an outline of the programs feature was made. Before even taking a look at the algorithms, all the basic

program features has to be implemented. Since the program is about visualization, this requirement analysis deals with the Graphical User Interface(GUI) and not algorithms that is to be implemented. The program is designed to hold one window with 3 panels. A menu panel in the top, a canvas panel in the left and middle and a tool-panel to the right. The panels has the following features.

1. Canvas, panel visualizing the graph
 - (a) Displaying vertices
 - (b) Displaying edges
2. Menu
 - (a) Load, able to load a TSP file
 - (b) Close, closes the program
3. Tools panel
 - (a) Button for generating a graph
 - (b) Drop-down box to choose size of search space
 - (c) Radio-buttons able to choose algorithm
 - (d) Radio-buttons able to choose search-space, eg. bit-string/permutations
 - (e) Text box to display statistics about the running algorithm, like generation, distances
 - (f) Button to make the algorithms run/stop

4.3 Graphical User Interface

The GUI is intended to be as simple as possible. There are no features that are hidden away in sub-menus etc. The radio buttons are designed to hold features for both bit-strings and permutations to save space, i.e. one radio button has two uses. It was into consideration making non supported features blank out and not visible, but there was not a simple way to do this. There were several more things that were into consideration and it will be addressed in section 4.5. For features that need a value a combo-box is used with a limited number of choices for the value. The limitation helps to keep the GUI simple, but this could easily be replaced with more choices or another mean of input-box like a text-box. All the combo-boxes has a default value to avoid errors, but also to be able to use the program without tinkering with too many settings.

4.4 Implementation of the algorithms

This section will cover how the algorithms have been implemented into the program. The algorithms are both described how they are visualized and how the data structure is made. The software methods will not be addressed directly, but a class diagram is located in the appendix in figure [A.5](#).

4.4.1 (1+1)EA and RLS

This section will cover both (1+1)EA and RLS as both the visualization and data structure are almost identical.

4.4.1.1 Visualization

The evolution algorithm (1 + 1EA) and the similar RLS algorithm has been implemented the following way. The user has to be in 'Bit-String mode', this mode is chosen by a radio-button. There is another mode called permutation, used for permutation problems, this mode will be described later. When the user is in 'Bit-String' mode an red circle drawing will appear in the canvas. The user can choose how big a bit-string that should be generated in a combo-box field, default is 8, but the user can choose to generate 16,32 or 64 bits instead. In another drop-down field the user chooses to use either OneMax or LeadingOnes as the fitness function, OneMax is marked as default. The user has to click on the button 'generate' to make a random bit-string. Depending on the spread of the 1's and 0's in the bit-string a dot will be displayed inside the drawing. The dot is located in the bottom of the circle, if there are more 1's than 0's and likewise the dot will appear in the top of the circle if there is a majority of 0's. The dot will appear to the right if the spread of the 1's in the bit-string more intensive in the right section of the bit-string compared to the left and vice versa. In both the fitness functions OneMax and LeadingOnes the dot will appear in the bottom of the circle, when the bit-string is optimal. The coordinate of the dot is calculated by using the locations of the 1's in the string, i.e. what number the bit is in the row of bits, e.g. for the string 0001, the 1 is at location 4. Using the size of the string and the sum of the locations where the 1's are located, the x-coordinate for the dot is made. The y-coordinate is just calculated using the percentage of 1's in the whole string. When one of the algorithms are run it stops when it is optimal. The button 'Run' is used to begin the algorithm. One can clear the graph with the button clear.

4.4.1.2 Data Structure

The random bit-string, that is used as a search-space, is generated simply by using the build-in random function in `c #`. The classes for the fitness functions, `LeadingOnes` and `OneMax`, both contains methods to check if the new generation is stronger according to the fitness function. Using the bit-string the coordinates for the dot gets calculated. Each bit uses its position in the string as a value. These values are counted together and the mean value is found and the mean is used to calculate the position on the x-axis. The position on the y-axis is calculated using the ratio of 1's in the string. Both the implementations of (1+1)EA and RLS take a string as an input and randomly calculates a new string. (1+1)EA goes through the whole bit-string and each bit has a chance to flip, the chance to flip is $1/n$ where n is the length of the bit-string. RLS randomly picks a bit and flips it.

4.4.2 Permutations

To run the permutation algorithms, the user has to be in permutation mode. To be in permutation mode, the radio button permutation should be checked. The structure and visualization for the permutation algorithms are a lot different than the bit-string algorithms. It consists of a graph, i.e. vertices and edges. The vertices has several properties, e.g. X-, Y-coordinates. There are actually two X-,Y-coordinate properties, because the vertices are glued onto the canvas in the point that is in the top left corner of the vertex, the vertices also has X,Y-properties to hold the middle of the vertices. These middle points of the vertices are used to connect edges on, else the edges would connect to each other in the top-left corner of the vertices. The edges only has the X-,Y- coordinates, which they use to know where to be drawn. There are two ways of making a searchable graph. One is by hitting file and load. The edges are made by the order they are loaded into the program. The other way of making a searchable graph is with the generate button. The graph uses the size of the 'search space combo-box' to set the number of vertices to be generated. The graphs generated randomly with generate button also have randomly generated edges. In both cases the vertices and edges form a Hamiltonian cycle.

4.4.3 File Load

A load button has been implemented with a load function. The program is able to load TSP-files from [4] with the Euclid 2-d format, like berlin52. The

load button only works as long as the user is in permutation mode. There will be displayed an error message, if any attempt of loading a file while in bit-string mode. The function checks if the file has EUC2D in it to state whether it is a legal file or not. Afterwards it removes the text in front of the coordinates. Thereafter it makes the sub-string containing the coordinates to actual coordinates and it inserts them onto the canvas. For files containing coordinates with a coordinate value over 800, it will scale down all the coordinates by a factor calculated by taking the largest number and 800. The calculation is $scaledown = 800/largestnumber$. For graphs where the largest number is under 400, a constant(scaled up) is multiplied to the coordinates to make the graph look bigger. To avoid the distance being meshed up, a ratio is calculated using scaled down/scaled up, the $ratio = 1/scaledown \text{ or } 1/scaledup$. This ratio is multiplied to the distance, so the distance will be displayed correctly. Parameters to use with the loaded file should be made before opening it.

4.4.4 Simulated Annealing

The algorithm Simulated Annealing has been implemented using permutations as the search space. Therefore the implementation is very different than (1+1)EA and RLS. SA uses 2-opt to switch edges. The 2-opt class could be used with other algorithms if needed.

4.4.4.1 Visualization

The graph redraws every time the graph get updated, i.e. when a 2-opt move is made. The algorithm runs as long as the number of specified generations steps are chosen in the combo-box. When the graph is redraw it looks like only two edges shifts, but in reality all the edges gets redrawn even though only two edges get new data. This is due to the fact all the edges gets reordered every time a 2-opt move occurs. The re-ordering is explained in the part about 2-opt at [3.4](#).

4.4.4.2 Data Structure

To make the cycle for SA, two classes has been made. One for the algorithm SA and one for two-opt, the two classes has been made to make the program easier to extend with other algorithms. SA in the program works the following way. It takes the current graph and does a two-opt move, however it does not signal the drawing mechanism, which means it will not get drawn at this

moment. The class SA has a method to check whether the new solution is better than the previous one, if it is the case the move gets 'legalized' and the program signal to draw the new solution. If the new solution is worse than the previous one, the move will only get legalized by a chance, which is calculated by the temperature. If the move is dropped the former solution will be remade. The temperature is calculated using the following parameters. The starting temperature is calculated $t(0) = e^3$ where e is the number of edges, Alpha is $1 - 1/e$. This alpha determinates how fast the temperature drop. The current temperature is calculated using the following $t(g) = \alpha^{g-1} \times t(0)$ where g is the current generation. To check whether a new worse solution should be accepted, a random number is generated between 0 and $t(0)$. If the generated number is smaller than $t(g)$ the new solution get accepted and finally drawn in the canvas.

4.4.5 Ant Colonization Optimization

Visually ACO is similar to SA, but there are some visual differences. The data-structure is also very different than SA.

4.4.5.1 Visualization

As mentioned above the visualization is similar to SA, but ACO does not have a limit on how many edges switch places at a time. There is a chance that all the edges will switch in the same generation. This is however must unlikely to happen often as each travelled edge gets a larger chance of being used again, if it already has been a part of the solution. Therefore the visualization tends to look the same as SA.

4.4.5.2 Data Structure

The data structure of ACO is a bit more complex than SA and way more than any of the other implemented algorithms. This is because the algorithm contains a few more steps, but also because these steps require a lot of data manipulation. To help calculating the cycle, two 2-dimensional arrays/matrices are set up. One array contains all the distances between the vertices. This is used to fast look up the new distance for the path and the data is not updated while the algorithm is running. An example on the distance matrix is given in [A.1](#). The other matrix contains the chance of going to the next vertex. The matrix containing the chances will get updated every time a new generation is made and the chances

are updated in relation to the best path. This update step is step 6. in the algorithm 4 and the update it self is described in 6. This chance matrix keeps track on the probabilities of each vertex travelling to any other vertex. The chance matrix consists of sorted values, which is used as intervals. To calculate which vertex to choose as the next in the path, a random number from 0-1 is made. The vertex with the right interval is the next vertex, which is the vertex that is the first vertex that is bigger than the random value. An example on the chance array is given in A.2 and how the values look like after an update is given in A.3. There is however more to it than this as each vertex is visited, their chance for being visited again has to be deleted. An array keeps track on the vertices current in the path, i.e. the already visited vertices, and another array keeps track of the vertices that have yet to be visited. When a looking for the next vertex, these vertices arrays makes sure that the intervals of the already visited vertices are spread equally between other unvisited vertices. When a cycle has been found, the cycle is looked up in the distance matrix to form the new cycle's distance. A function checks whether the new cycle is better than the old. The best of the two cycles updates the pheromone chances in correlation with the update rules in algorithm 6. There were a few problems with this, which will be addressed in section 4.5.2. At this point the GUI needs to redraw itself and to do this a signal is used. A double changes its value, which signals the GUI to redraw. This double changes value every time a generation has been passed. The algorithm runs as long as the maximum generation combo-box is set to.

4.5 Problems during implementation

When working on a larger project which contains complex algorithms and technical features. There is a big chance of running into problems. While making the program I encountered both algorithmic and technical problems.

4.5.1 Technical problems

There was a technical problem with the drawing of the graph while calculating the Hamiltonian path. The problem was caused due to the lesser experience with WPF/C# and it was the difference compared to JAVA that gave some headaches. In WPF when you want to update the GUI while calculating, you have to use a specific tool called the dispatcher. The dispatcher gives the permission to update the GUI, while a calculation is under way. Each edge has a dispatcher to notify that it needs to be drawn again. The same goes with

the node running in the circle for bit-strings. However these dispatchers gave more than one problem. Because the search space size is user-selected and does not a fixed size, the numbers of dispatchers cannot be fixed either as it has to follow the right amount of edges. However a dispatcher is a special class and it cannot use for-loops values in the normal way, the class is simply made to ignore these values. The dispatcher has to get a local value passed to it. To solve this problem a local integer should be assigned to the iteration in the for-loop and this local value can be passed to the dispatcher. The solution is rather simple, but annoying as it does not follow regular programming standards.

4.5.2 Algorithmic

Some of the problems that occurred during the implementation of the algorithms, was caused due to the initial design was flawed. This happened at the design of the two-opt algorithm. Where after the testing of the algorithm the Hamiltonian cycle could brake, i.e. make two unconnected graphs. Therefore the reordering of the graph was made, this reordering is described in figure 3.1 to 3.4. Another problem that occurred during the implementation was how to decide when the algorithms should stop when running permutations. For bit-strings the algorithm ends when it is optimal, this is easy and fast to do, it just involves counting the current number of 1's and compare this number to the size of the bit-string. However it is not easy to find the most optimal solution for permutations. Originally there was a counter that counted generations without changes and after a specific number was reached the algorithm would end. But this feature was dropped, as it was hard to find a good calculation to determinate on, how many consecutive generations without any changes should be allowed. This is especially a problem on larger graphs, where the algorithms can run through many generations without changing anything. The algorithm was instead set to run after a specific number of generations. Algorithmic wise the ACO algorithm is rather complex to implement even though the algorithm itself is pretty simple. The data-structure to hold the chances of the edges gave a few problems. As mentioned earlier the chances of finding the next vertex is found using intervals. It is important to keep these intervals within the right range. Because a lot of parameters have influence on the chances and therefore a lot can go wrong. The intervals go from 0-1 and vertices not available to move to has the value -1. One must make sure that these -1 values does not get updated, but also that the other values does not use a too large interval. There were some trouble with some chance-values got past the 0-1 interval. There were several bugs which caused this. Some was caused by small casting errors, which could easily be fixed. One of these was a pheromone bug, the string containing the pheromone value did not paste the right value as it ignored any 0's in front of it, e.g it ignored the zero in 0.1 and made it into 1. This large pheromone caused

some odd values and therefore make the chance matrix bugged. This could lead to a exception and a shut down of the program. The most annoying thing about these intervals was however the Max-Min step. This step makes the algorithm much more complex, when making an edge pheromone the minimum value, the percentage must be taken from another place and it is the same with the maximum value, where the percentage must be given to other edges. In start of the implementation of the ACO algorithm, this was not done and the edges would fast get negative values, which would make them invisible to other vertices and therefore impossible to visit. To solve this problem two things was implemented, one to solve each of the two problems. To ensure that the minimum edge value was kept, the edge with the largest chance would give some of it's percentage to these edges that needed more chance. To avoid the intervals getting mesh up by the maximum edge value, the surplus of the percentage from a maximum edge gets assigned to the edge going out from the first vertex in the list. By doing this one would assume that the vertices would always have a bigger chance of going to this vertex. This is however avoided by always starting the tour from this vertex and it's pheromone chance will instead be evenly handed out to the other vertices, when finding the next vertex.

4.6 Possible extensions

As this program only has a few features a lot of extensions could be made. Here is a few of the ideas considered in the making of the project.

4.6.1 Algorithmic

Other meta-heuristic algorithms would of course make sense to implement. The two-opt class makes it easy to implement new algorithms using a two-opt switching function. This could be the (1+1)EA algorithm or similar. SA could also have been implemented for bit-strings to compare it with (1+1)EA. I would have liked to implement a more complex swarm algorithm like the firefly algorithm [11], but the schedule of the project made it to difficult to start implementing it.

4.6.2 Technical

At the start of the project I had an idea to make the algorithms and graph dynamic. Which means that you could manipulate with the graph while it ran through the algorithm. The graph is dynamic and can be manipulated during a run, however the algorithms are not implemented to work with the dynamic graph. This could however be a extension to the program. Another tool to implement would be the ability to make your own graph vertex by vertex. An early version of the program had this feature, but dropped as there was not a straightforward design for the edges to reconnect the graph without having the dynamic graph. There could also be extensions for the GUI. It would have been cool, if when a search-space was chosen the rest of the GUI would switch to the features used for that search-space, and it would hide the features that was unusable. This would have been great for the usability, but as priority of features to implement got bigger this would seem a small extension to use precious time on.

CHAPTER 5

Analysis and Results from the program

The program can be used to visualize several meta-heuristic algorithms like (1+1)EA, Simulated Annealing and Ant Colonization Optimization. However the program can also be used to show how the different kind of algorithms perform efficiently.

5.1 (1+1)EA and RLS

5.1.1 Goal of the test

For testing the bit-string algorithms, 8 times 10 tests have been conducted. The tests are made to get an approximate number of generations it will take before a random bit-string is at it's optimal state according to it's fitness function. This way it is easy to get an overview, which is the fast algorithm and also whether the fitness function has any influence on run-time of the algorithm.

5.1.2 Test

The tests consists of 10 tests on a random 8-bit/16-bit string using (1+1) EA and the fitness function OneMax, 10 tests on a random 8-bit/16-bit string using RLS with OneMax, 10 test on a random 8-bit/16-bit string using (1+1)EA with LeadingOnes, and finally 10 tests on a random 8-bit/16-bit string using RLS with LeadingOnes. The data from the tests is both displayed with all the 10 tests and without the two highest and lowest values to eliminate coincidences. The generations counts are displayed in the tables [A.4](#), [A.5](#), [A.6](#), [A.7](#). As the tables show (1+1)EA is much slower than RLS. This is also suggested in the theory part in section [3.3](#).

5.1.2.1 Conclusion on (1+1)EA and RLS

(1+1)EA has never been intended as an fast algorithm, but as an simple meta-heuristic algorithm based on evolution. It is a good introduction algorithm used to explain the principles of a nature-inspired algorithms. The tests showed that it seems the difference between the performance between RLS and (1+1)EA gets even bigger as the size of the search-space increases, this is seen in the tables [A.4](#), [A.5](#), [A.6](#), [A.7](#) and in the 'summary' table [A.8](#). It is worth noticing that both (1+1)EA and RLS does much worse running LeadingOnes compared to OneMax. This is however not surprising as LeadingOnes only chooses the longest string of consecutive 1's and therefore at some point will have to wait for 1 or 2 specific bits to flip before proceeding. Where OneMax just have to wait for 0's to flip. The fitness function does not affect runtime of the any of two algorithms more than the other, this is seen in table [A.8](#), where the difference is almost the same whether the fitness function is LeadingOnes or OneMax. The short conclusion to this must be that it is better to always flip 1 bit than have a chance to flip each bit.

5.2 Simulated Annealing

5.2.1 Goal of the test

SA has been implemented on permutations, where it finds a short Hamiltonian cycle. SA makes use of the 2-opt switch edges feature. This feature is one of the algorithm's weak points. Because it always only switches two edges, one can assume that the chance of finding the best cycle in a big graph is way harder

than finding a good cycle in a small graph. Therefore the following hypothesis 5.1. will be investigated.

HYPOTHESE 5.1 *The effectiveness of SA decreases as the search-space size increases.*

To see whether the hypothesis holds or it is rejected several tests with different search-spaces are run using SA. If the theory holds an estimate on the critical size of the search space is made, i.e. where the algorithm loses its effectiveness. The test will also compare the number of generations to the number, it would take to find the most optimal cycle using brute force. As one cannot be sure to know when the most optimal route is reached, it will be decided visually by the rules that no edges may cross and no extreme edges can be in the graph, i.e. there is visually a 2-opt move that will make the path shorter. The TSP file, berlin52, will also be used for testing. The advantage for testing with files from the TSPLib is that the shortest path distance is known and therefore the algorithm can easily be evaluated. The TSP test is used to test the efficiency of algorithm and see how fast and close it gets to the shortest path in the graph. The algorithm will run for 15000 generations and the results will be noted. The distance the algorithm is aiming for is 7542, which is the shortest path in the file.

5.2.2 Test

Starting with finding whether 5.1 holds. Like the tests with bit-strings, 10 tests of each type were tested. Here types are 8, 16 and 32 vertices search-space. The test results are displayed in table A.9. Where they are displayed with and without the two extreme maximum and minimum values. Just by looking at the values, it seems that the needed generations for finding a good path increases a lot when the search-space gets bigger. In table A.10 the generation per vertex has been calculated. Table A.10 shows that the hypothesis 5.1 holds. The value per vertex increases more than the double when the search-space gets doubled. This is however pretty good compared to brute force, where the running time is $n!$, SA gives a decent answer in a running time that is comparable to n^2 for small search-spaces, when the search-space gets larger, i.e. 32+ vertices, the running time gets more comparable to n^3 , the correlation between number of vertices and generations can be seen in graph A.1. This estimate is given from tests in the program and not theoretical analysis of the algorithm. The algorithm works pretty good even though it has some drawbacks at bigger graphs. Because it only switches 2 edges, it will take a long amount of time to get a good solution. This leads to the test of efficiency of the algorithm, where the test on

berlin52 was made. However with the suggested running time of the algorithm before finding a very good solution, the algorithm will have to run for about $52^3 = 140608$ generations before coming up with a really good answer to the berlin52 file. In this test with berlin52 only 15000 generations is run, which still should give a decent answer. The test was run and after 15000 generations it ended with a distance at 13355, which is rather bad compared to 7542. However given the estimated running time of at least 140608, the distance after the first 15000 generations is pretty good. The distance is not that bad either when looking at the visualized graph [A.2](#). There are no crossed edges and by simply looking at the graph it is hard to find many pair of edges to switch to make the graph shorter.

5.2.3 Conclusion on SA

The algorithm works pretty well and is much better than brute force, even though it does not always give the shortest answer. There is however also some problems with the algorithm, especially on larger graphs the temperature becomes less practical even though it starts on a higher temperature, it gets too low too fast, which may cause the graph getting into local minimums and stuck at larger distances than the shortest path. The algorithm could properly be faster, if the temperature-step was designed in another way than this implementation in this program. But it seems it will always perform better than brute force.

5.3 ACO

5.3.1 Goal of the test

ACO will run a similar test as SA. This will give some insight on how efficient the algorithm actually is, but this will also give the chance to compare ACO to SA. The comparison will however be covered in the next section [5.4](#). Whereas in this section will look on ACO's performance compared to brute-force and it's scalability. The same hypothesis that was set for SA is used for ACO.

HYPOTHESE 5.2 *The effectiveness of ACO decreases as the search-space size increases.*

ACO is not bound only to switch two edges at a time and could in theory switch all the edges in one move. However the chances for switching gets smaller and smaller and therefore it might only rarely switch. To avoid getting stuck too often all tests are performed with the pheromone value set to 0.1, as a higher value will reach τ -max and -min faster.

5.3.2 Test

The test results gave some surprises, during the first test with 8-vertices. The ACO algorithm performed really bad. The problem is that the τ -max and -min values are too high and too low. When using the τ from [5], which is displayed in algorithm 6. The result of the test with the ACO algorithm using 0.1 in pheromone level on a 8 vertices search space is displayed in table A.11. The hypothesis 5.2 seems to hold as the algorithm did not produce any really good results within 15000 generation when doubling the search-space to 16. Which again leads to ACO has some scalability issues, when it comes to large files. However it will must likely outperform brute force on all graphs. But the bad test lead to some analysis and rethinking on the implementation of the ACO algorithm and by studying various runs, it seemed that when the max- τ value was reached for most of the edges, the algorithm rarely shifted edges and when it did, it was only 2 edges, like SA. Because of the algorithm's bad performance, some tuning was attempted by improving the max-,min- τ . The τ value was assigned a strength value, the strength value makes the maximum τ lower and minimum higher when the strength is high and vice versa when the strength value is low. This will force the algorithm to shift more often if the strength is high. The ACO algorithm was tested with a strength value set at 5 and again with pheromone set a 0.1 and 10 new tests were conducted. τ is calculated by $\tau - max = 1 - (1/n * s)$ and $\tau - min = 1/n^2 * s$. The results from the test with $s = 5$ are listed in table A.12. The table suggest that making the edges shift more frequently ACO will perform better. However the test did not show a major improvement in finding a shorter cycle and therefore needed more investigating. To test whether the value of τ -max and -min affects the runtime and efficiency of ACO, a test on the TSPfile berlin52 was made to test the efficiency on a larger graph. The two graphs can be seen at A.3 and A.4. The algorithm with the strength set to 5, clearly outperformed the algorithm with the standard strength of 1. Where ACO with $s = 1$ had a distance of 16554 after 15000 generations, ACO with $s = 5$ had 14163.

5.3.3 Conclusion on ACO

MMAS ACO works, however the standard implementation uses too many generations without doing anything, which is a resource and time waste. By manipulating with the τ values and make it shift more often, the algorithm seemed to perform better. This is because of less wasted generations. The little change to the τ values could increase the performance of the algorithm. It could be interesting to investigate if there is an optimal $max - \tau$ value from an analysis point of view. An optimal value would not waste many generations, but also it would not switch too many edges per generations as too many switches will make the algorithm less effective and if the values get too big it would just be a random path algorithm.

5.4 Comparing ACO to SA

It is natural to compare ACO to SA as they are both solving the same problem and they are both meta-heuristic algorithms. Visually they perform similar and it is hard to spot the difference, if one does not know which algorithm is running.

5.4.1 Comparing results

By looking at the findings in the SA and ACO tests, it is easy to conclude that the test results for SA is much better than ACO. The ACO algorithm performed better with the change to the τ and the algorithm can properly be tuned to perform similar to SA.

5.5 Conclusion on the permutation test

The test showed that both SA and ACO are better than brute force. When ACO's chance matrix is saturated, all vertices has a favourite neighbour with a τ close to the maximum τ , the changes that happen most common only consists of a move similar to 2-opt/SA. But where SA in every generation tries to make a new solution ACO only has a chance to make a new solution and it will often end up doing the same solution multiple times in a row and therefore becoming less efficient compared to SA. And just like the final words on the test and

analysis on bit-string, the same thing applies to SA versus ACO. It is better to change 1 thing every time than have a chance to change many.

Conclusion

To summarize the project findings, all the algorithms implemented (1+1EA) 1, RLS 2, SA 3 and ACO MMAS 4 all are pretty simple in theory and (1+1)EA, RLS and SA are also very simple to implement. ACO is a bit more complex as it has a lot of parameters. But in the end all the algorithms has been successfully visually implemented and they are all solving their problem. For the permutations algorithms, SA and ACO, their efficiency seemed far better than Brute Force. SA performed really good in all the tests and it could be used in a real application. ACO did not perform that good in the test, however after a simple tuning of the algorithm, it performed better and with some more tuning it might perform better or similar to SA. For the bit-strings RLS performed better than (1+1)EA and there is no parameters to change in (1+1)EA and therefore it has the worse efficiency of the two. For both permutation and finding optimal bit-strings, the algorithms that changed every generation was best. RLS for bit-strings and SA for permutations. There are many other meta-heuristic algorithms and features that could have been implemented into a the software framework. This program and thesis only covers a few, however the program could easily be extended with other algorithms and features in the future. Other algorithms might change the my picture of the efficient algorithms being the ones that ensure change.

APPENDIX A

Tables and Graphs

This appendix contains a few graphs displaying the data from the tests.

-	1	2	3	4
1	-1	4	7	7
2	4	-1	3	3
3	7	3	-1	5
4	7	3	5	-1

Table A.1: Example on the distance matrix

-	1	2	3	4	5	6	7	8	9	10	11
1	-1	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
2	0.1	-1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
3	0.1	0.2	-1	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
4	0.1	0.2	0.3	-1	0.4	0.5	0.6	0.7	0.8	0.9	1.0
5	0.1	0.2	0.3	0.4	-1	0.5	0.6	0.7	0.8	0.9	1.0
6	0.1	0.2	0.3	0.4	0.5	-1	0.6	0.7	0.8	0.9	1.0
7	0.1	0.2	0.3	0.4	0.5	0.6	-1	0.7	0.8	0.9	1.0
8	0.1	0.2	0.3	0.4	0.5	0.6	0.7	-1	0.8	0.9	1.0
9	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	-1	0.9	1.0
10	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	-1	1.0
11	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	-1

Table A.2: Example on a starting chance matrix with 11 vertices, 11 has been chosen to ensure a simple example

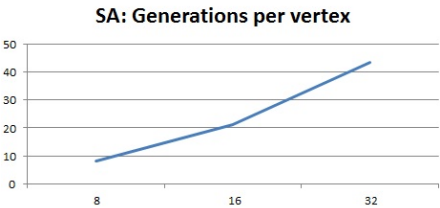


Figure A.1: The graph displays the correlation between number of vertices and the performance of SA

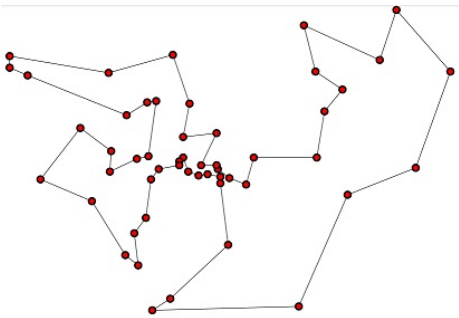


Figure A.2: The berlin52 graph after 15000 generations running SA. Distance is 13355

-	1	2	3	4	5	6	7	8	9	10	11
1	-1	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
2	0.19	-1	0.28	0.37	0.46	0.55	0.64	0.73	0.82	0.91	1.0
3	0.09	0.28	-1	0.37	0.46	0.55	0.64	0.73	0.82	0.91	1.0
4	0.09	0.18	0.37	-1	0.46	0.55	0.64	0.73	0.82	0.91	1.0
5	0.09	0.18	0.27	0.46	-1	0.55	0.64	0.73	0.82	0.91	1.0
6	0.09	0.18	0.27	0.36	0.55	-1	0.64	0.73	0.82	0.91	1.0
7	0.09	0.18	0.27	0.36	0.45	0.64	-1	0.73	0.82	0.91	1.0
8	0.09	0.18	0.27	0.36	0.45	0.54	0.73	-1	0.82	0.91	1.0
9	0.09	0.18	0.27	0.36	0.45	0.54	0.63	0.82	-1	0.91	1.0
10	0.09	0.18	0.27	0.36	0.45	0.54	0.63	0.72	0.91	-1	1.0
11	0.09	0.18	0.27	0.36	0.45	0.54	0.63	0.72	0.81	1.0	-1

Table A.3: Example on the first updated chance matrix with a pheromone level set at 0.1 and the path 1,2,3,4,5,6,7,8,9,10,11

Table A.4: In this first experiment with a 8 bit-string using the OneMax fitness function, the experiment showed that RLS is almost twice as efficient compared to (1+1)EA. Without the extremes the data shows the same pattern.

8-bit OneMax			
10 tests		without extremes	
RLS	(1+1)EA	RLS	(1+1)EA
13	32	13	32
1	29	-	29
10	38	10	38
12	27	12	27
19	44	19	-
14	11	14	-
24	18	-	18
14	43	14	-
11	2	-	-
21	20	-	20
13,9	26,4	13,67	27,33

Table A.5: Using LeadingOnes the pattern is the same as before even though the difference gets a bit smaller.

8-bit LeadingOnes			
10 tests		without extremes	
RLS	(1+1)EA	RLS	(1+1)EA
58	61	-	61
57	44	57	44
23	44	23	44
23	44	23	-
17	60	17	60
61	50	-	50
13	56	-	56
26	27	26	-
16	64	-	-
45	66	45	-
33,9	51,6	31,83	52,5

Table A.6: The difference becomes even larger when using a longer bit-string in the OneMax experiment.

16-bit OneMax			
10 tests		without extremes	
RLS	(1+1)EA	RLS	(1+1)EA
24	104	24	104
19	27	-	-
21	34	21	-
15	93	-	93
44	150	44	-
24	46	24	46
26	86	26	86
24	129	24	-
26	66	26	66
26	99	-	99
24,9	83,4	24,16	82,33

Table A.7: In the last experiment where a 16-bit string with LeadingOnes was used, the difference is also pretty large. It was the only experiment where removing the extremes had a big influence on the average generation count.

16-bit LeadingOnes			
10 tests		without extremes	
RLS	(1+1)EA	RLS	(1+1)EA
25	328	-	328
16	180	-	180
95	146	-	146
83	332	-	-
43	194	43	194
64	119	64	-
65	136	65	136
48	470	48	-
56	108	56	-
56	146	56	146
55,1	215,9	55,33	188,33

Table A.8: By comparing the average run time of RLS and (1+1)EA, it is easy to see RLS is better, but also the fitness function does not affect one algorithm over the other

Comparing (1+1)EA to RLS			
Test name	RLS	(1+1)EA	Difference
8-Onemax	13,67	27,3	1,997074
8-LeadingOnes	31,83	52,5	1,649387
16-Onemax	24,17	82,33	3,406289
16-LeadingOnes	55,33	188,33	3,403759

Table A.9: SA performed well in all the test cases

Simulated Annealing					
10 tests			without extremes		
8	16	32	8	16	32
96	144	1416	96	-	1416
21	317	965	-	317	-
144	404	1350	-	404	1350
104	345	1534	-	345	1534
98	317	1126	98	317	1126
55	282	1117	55	282	1117
26	384	969	26	384	-
25	583	1917	-	-	-
34	208	1834	34	-	1834
78	426	1999	78	-	-
68,1	341	1428,1	64,5	341,5	1396,17

Table A.10: Increase in runtime per vertex

Simulated Annealing					
generation per vertex			without extremes		
8	16	32	8	16	32
8,51	21,31	44,63	8,06	21,34	43,63

Table A.11: The stopping generation was set to 2000 and 4 tests reached this mark, therefore the average runtime at least 900+

ACO ph = 0.1, strength=1	
generation per vertex	without extremes
8	8
97	-
921	921
124	124
2000	2000
188	188
45	-
2000	-
2000	-
144	144
2000	2000
951,9	896,7

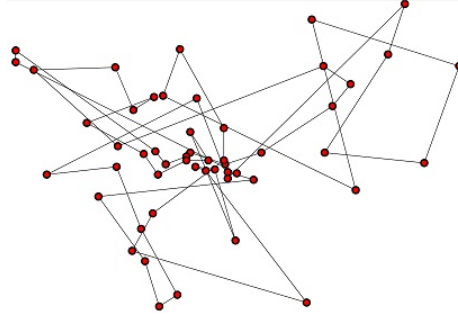


Figure A.3: The berlin52 graph after 15000 generations running ACO with pheromone 0.1 and strength 1. Distance is 16554

Table A.12: The strength value force the algorithm to shift more frequently

ACO ph = 0.1, strength=5	
generation per vertex	without extremes
8	8
990	990
34	-
2000	2000
50	50
2000	-
2000	-
1400	1400
33	-
54	54
44	44
860,5	756,33

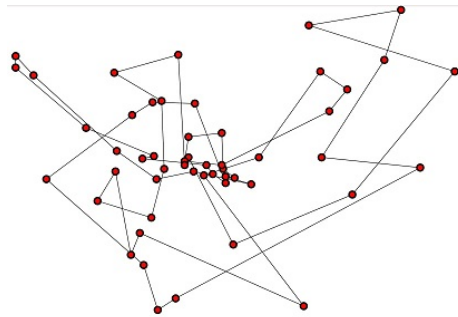


Figure A.4: The berlin52 graph after 15000 generations running ACO with pheromone 0.1 and strength 5. Distance is 14163

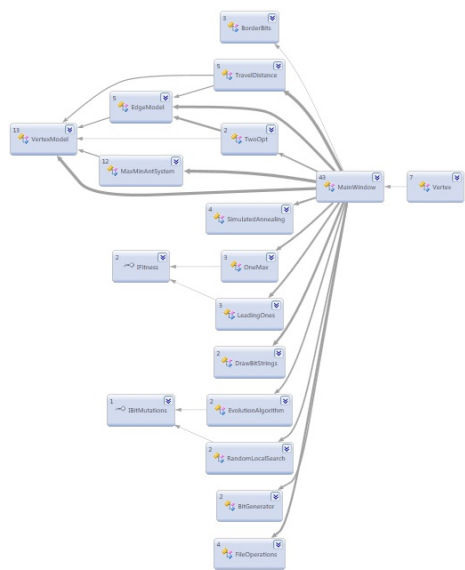


Figure A.5: Overview of the program

Bibliography

- [1] Marco A. Montes de Oca. Aco history.
- [2] Arlen Feldman and Maxx Daymon. *WPF in action with visual Studio 2008*. Manning, Greenwich, CT 06830, 2009.
- [3] International Society for Genetic and Evolutionary Computation. *Genetic and Evolutionary computation - GECCO 2004*. Springer, Berlin Heidelberg, 2004.
- [4] Rubrecht-Karls-Universitat Heidelberg. Tsplib.
- [5] Timo Kötzing, Frank Neumann, Heiko Röglin, and Carsten Witt. Theoretical analysis of two aco approaches for the traveling salesman problem. *Swarm Intell*, 6:1–21, 2011.
- [6] Klaus Meer. Simulated annealing versus metropolis for a tsp instance. *Information Processing Letters*, 104:216–219, 2007.
- [7] Frank Neumann and Carsten Witt. *Bioinspired Computation in Combinatorial Optimization - Algorithms and Their Computational Complexity*. Springer, DTU, Denmark, 2010.
- [8] Wikipedia’s users. 2-opt wiki,<http://en.wikipedia.org/wiki/2-opt>.
- [9] Wikipedia’s users. Hamiltonian path, http://en.wikipedia.org/wiki/hamiltonian_path.
- [10] Wikipedia’s users. Metaheuristic history, <http://en.wikipedia.org/wiki/metaheuristic>.
- [11] Xin-She Yang. Firefly algorithm,http://en.wikipedia.org/wiki/firefly_algorithm.