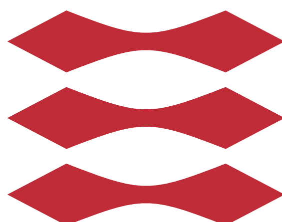
Visualizing and Evaluating the Working Principles of Nature-Inspired Optimization Metaheuristics

Emil Lundt Larsen (s153214)
Supervisor: Carsten Witt
B.Sc. Software Technology
DTU Compute

June 11, 2018



Abstract

The purpose of this study is to visualize and evaluate the working principles of nature-inspired metaheuristics for the solution of combinatorial optimization problems, most notably the traveling salesperson problem. To this end, a framework has been made that implements the algorithms, (1+1) EA, simulated annealing and Max-Min Ant System that can be run on different problems while visualizing the way they construct solutions to these. Further, key figures from the run of said algorithms are extracted that are used to benchmark their performance with different parameters. The study concludes that the algorithms achieve good results on the traveling salesperson problem in practice and that many of the tests support the theoretical bounds on the expected optimization time for the algorithms on the pseudo-boolean functions OneMax and LeadingOnes. The experiments made also demonstrate different advantages and disadvantages of each algorithm: The (1+1) EA is remarkably simple and the performance is not heavily dependent on good parameter settings. Simulated annealing on the other hand is for a carefully defined cooling schedule incredibly efficient on OneMax, LeadingOnes and the traveling salesperson problem and is also very fast in terms of running time. The Max-Min Ant System is the best performing algorithm with respect to optimization time for the traveling salesperson problem, but has a significantly higher running time than the (1+1) EA and simulated annealing.

Contents

1 Introduction **1**

2 Background **2**
2.1 Combinatorial Optimization 2
2.2 Search Spaces 3
2.2.1 Bit Strings 5
2.2.2 Permutations 12
2.3 Evolutionary Algorithms 15
2.3.1 (1+1) EA 18
2.4 Simulated Annealing 23
2.4.1 Defining a General-Purpose Cooling Schedule 24
2.4.2 Simulated Annealing on OneMax 27
2.4.3 Handling of Pseudo-Boolean Function Obstacles 28
2.5 Ant Colony Optimization 29
2.5.1 Pseudo-Boolean Functions 32
2.5.2 TSP 37
2.6 Related Work 38

3 Analysis **40**

4 Design **42**
4.1 Program Structure 42
4.2 Graphics 43
4.2.1 Program Flow 43
4.2.2 Boolean Hypercube 48
4.2.3 Graph for TSP Instances 49
4.3 Testing 50

5 Implementation **51**
5.1 Algorithm Specifics 51
5.2 Graphics 54
5.2.1 Animation 55
5.3 Statistics 56
5.4 Optimizations 57
5.4.1 (1+1) EA 57
5.4.2 Random Number Generation 59
5.4.3 Ant Colony Optimization 60

6 Evaluation and Results **61**
6.1 Evolutionary Algorithms . . . 63
6.1.1 OneMax . . . 63
6.1.2 LeadingOnes . . . 65
6.1.3 BinVal . . . 66
6.1.4 TSP . . . 67
6.2 Simulated Annealing . . . 70
6.2.1 OneMax . . . 70
6.2.2 LeadingOnes . . . 72
6.2.3 TSP . . . 73
6.3 Ant Colony Optimization . . . 81
6.3.1 OneMax . . . 81
6.3.2 LeadingOnes . . . 86
6.3.3 TSP . . . 88
6.4 Comparing the Algorithms . . . 92
6.4.1 OneMax . . . 93
6.4.2 LeadingOnes . . . 94
6.4.3 (1+1) EA and SA on Jump_k and f_2 . . . 96
6.4.4 TSP . . . 98

7 Future Work **105**
7.1 Graphics and Visualization . . . 105
7.2 Content . . . 106
7.3 Analysis . . . 107
7.4 Optimizations . . . 108

8 Conclusion **110**

9 References **111**

10 Appendix **114**
10.1 Miscellaneous . . . 114
10.2 Program User Guide . . . 115

Emil Lundt Larsen (s153214)
11/6-17

1 Introduction

Nature has always been a great source of inspiration for various ideas, inventions and principles and has also lead to several successful algorithmic approaches. These so-called nature-inspired algorithms show remarkable performance on a wide range of problems in practice, including computationally hard and complex optimization problems. Furthermore, the algorithms themselves are quite simple and are easily adapted to suit specific needs and setting. However, because the algorithms rely heavily on randomization and complex stochastic processes they are often difficult to analyze and their theoretical foundation not nearly as profound as their practical success may otherwise warrant. Recently a body of theory has developed with the goal of changing that and the theoretical results achieved thus far are encouraging. An essential part of this is through analyzing the working principles of the algorithms on sample problems and derive characteristics and patterns in behavior based on this that can then be used to reason about the algorithms in a more general setting.[22]

The aim of this project is to develop a framework that implements and visualizes the working principles of nature-inspired metaheuristics for the solution of combinatorial optimization problems. To this end, selected evolutionary algorithms, simulated annealing and ant colony optimization algorithms will be analyzed, implemented and applied to different problems in the search spaces of bit strings and permutations; the problems that will primarily be analyzed are called OneMax, LeadingOnes and the traveling salesperson problem (TSP). A framework will be made that makes it possible to visualize the constructed solutions and extract key figures and results from a run of a given metaheuristic. Based on these results, the performance of the algorithms for different kinds of problems will be analyzed and compared with the expected performance suggested by the theoretical background of the algorithms in question.

The report is structured as follows: It starts off with the preliminaries of the project, where an overview of the problems and metaheuristics relevant for this project is given as well as a description of the three primary algorithms, namely the (1+1) EA, simulated annealing and Max-Min Ant System in further detail. A short comparison to two similar projects is also made. Following this, an analysis and delimitation of the requirements and problem statement is given. The design and noteworthy parts of the implementation of the program are then presented. At the end of the report the results of running the algorithms on the different problems is shown along with an evaluation and discussion of said results. The report is completed with a number of ideas for future work with the project and topic in general and a conclusion of the entire project.

References and an appendix with a user manual for the framework is found at the end of the report.

2 Background

2.1 Combinatorial Optimization

An important type of problems in computer science is that of combinatorial optimization problems, which are concerned with assigning values to discrete variables in order to achieve an optimal solution with respect to a given objective function. Formally an instance of a combinatorial optimization problem is a triple (S, f, Ω), where S is the search space, f is the objective function and Ω is a set of constraints. S gives rise to a set of candidate solutions to the problem, which are typically called search points. The subset of these that satisfy Ω will be valid solutions to the problem. By taking search points from S as input, the objective function f returns a value called fitness that determines how good this solution is relative to other solutions. Depending on the problem in question the goal may either be to find a valid solution with lowest fitness among all valid solutions or find the one with the highest fitness. The problem is classified as a minimization problem in the former case and as a maximization problem in the latter. Two well known examples of combinatorial optimization problems are the traveling salesperson problem (discussed in section 2.2.2) and the minimum spanning tree problem, which are a minimization and maximization problem, respectively.[22, 6]

Although the traveling salesperson problem and minimum spanning tree problem are both combinatorial optimization problems their computational complexities differ significantly with the traveling salesperson problem being much harder. It is therefore important to know that there are many different classes of combinatorial optimization problems with respect to their computational complexity. One such complexity class is the class NP, which contains a subset of problems known to be NP-complete. No efficient algorithm is known for these problems and they are therefore said to be computationally hard. The notion of efficiency in this context is defined as an algorithm being efficient if it runs in polynomial time with respect to the input size. Because of this, these problems quickly become infeasible to solve for large problem instances, where one in principle must resort to look at all valid solutions in order to find the optimal solution and guarantee its optimality. Nevertheless, there are still many NP-complete problems that are of high importance and interest in finding good solutions for. Fortunately, at the expense of optimality it is in many cases possible to use algorithms that find solutions that are good approximations of the optimal solution in polynomial time by various means.[8, 6]

One of the ways to do this is through the use of metaheuristics, which can concisely be described as follows: A metaheuristic is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. While a heuristic is a problem dependent way of finding approximative

solutions that utilizes features of the underlying problem, a metaheuristic can be thought of as a general-purpose heuristic designed to define good problem-specific heuristics. The power of a metaheuristic lies in its ability to be adapted to many different problems with relatively few modifications, which is made possible by its abstract notion and treatment of the problems to which it is to be applied as black-boxes. Yet, it is able to mitigate some of the issues that may arise with conventional heuristic methods: Typically heuristics can be categorized as either constructive or local search methods depending on if they generate solutions by iteratively adding solution components to an initially empty solution until a solution is completed or if they start with an initial solution which they then try to improve through local changes, respectively. The issue with these methods if they are not carefully designed is that they are either only able to generate a limited number of different solutions being the case for the former, or if they stop at poor quality local optima (to be defined later), which is the case for the latter method. Metaheuristics can alleviate this by providing a strategy for how the heuristics can be tweaked and better applied. For example a metaheuristic could define a scheme for how a local search method can be rerun several times to avoid it getting stuck at local optima.[6]

Now, it should be mentioned that we will not in the following sections be pedantic about the formal use of the terms metaheuristic, heuristic and algorithm and their differences. Since the notion of an algorithm covers a lot of aspects as well, and because one may argue that a metaheuristic is a heuristic in itself and that a heuristic is a kind of algorithm, we will allow ourselves to denote the metaheuristics in this project as algorithms. A closely related term also worth mentioning in this context is randomized search heuristics, which emphasizes the randomness aspect of the heuristic.

Combinatorial optimization problems may be either static or dynamic, but we will only be focusing on static problems in this project. For static problems the parameters and characteristics of the problem do not change during the process of running an algorithm on them. Dynamic problems on the other hand are defined based on quantities whose value are set by the system meaning the problem characteristics change during run time. Algorithms applied to dynamic optimization problems therefore need to be able to adapt online to changes in the problem environment. A prominent example of a dynamic optimization problem is a networking routing problems, where data traffic and structure varies with time.[6]

2.2 Search Spaces

In order to better reason about the search space for the given optimization problem at hand it may be useful to keep the notion of a state-space landscape at the back of one's mind: A state-space landscape in short, is a way of mentally picturing the

state space as a landscape with locations and elevation, which are based on the current search point and by the objective function respectively. The elevation at a certain location is a measure of the fitness of the search point that corresponds to being at that location in the search space. If the problem is a maximization problem the goal is to maximize the objective function, which in the analogy means reaching a location with highest possible elevation and similarly for a minimization problem reaching a location with minimum elevation. Such locations are denoted as global maxima and minima, respectively. We may also define local maxima and minima as locations where the neighborhood around it has lower or higher elevation, respectively and plateaus as flat areas of locations with equal elevation. How the neighborhood is defined depends intrinsically on the problem and the algorithm used, so within the analogy we typically just assume that a location adjacent to another means that all search points of that location are in the neighborhood of all search points of the other. Naturally, it is of importance if we were to actually draw the state-space landscape in practice, but for many problems this is usually not feasible anyway, and the term is mostly used as a way of thinking about the underlying mechanisms of the problem and how the algorithm solves it. For many of the pseudo-boolean considered in this project, however, it can be quite useful to draw a simple version of the state-space landscape, which is shown further down below. Formally, we define a neighborhood and optima as follows:[24]

Definition 2.1. *A neighborhood structure is a function $N : S \mapsto 2^S$ that assigns a set of neighbors $N(s) \subseteq S$ to every $s \in S$. $N(s)$ is also called the neighborhood of s.[6]*

Definition 2.2. *A local optimum for a minimization problem (location minimum) is a solution s such that $\forall s' \in N(s) : f(s) \leq f(s')$. Similarly, a local optimum for a maximization problem (a local maximum) is a solution s such that $\forall s' \in N(s) : f(s) \geq f(s')$.[6]*

Figure 1 shows a depiction of an example of state-space landscape with both a global and local maximum drawn as well as two kinds of a plateau, here denoted as a shoulder and a flat local maximum, respectively.

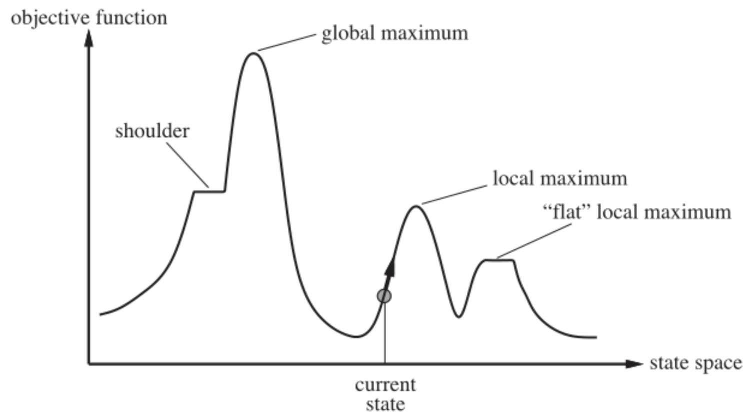


Figure 1: An illustration of a one-dimensional state space with plateaus and local and global maxima.[24]

Because local optima often look like hills or valleys when pictured like this it is not strange that local search strategies are often called hill-climbers for maximization problems or gradient-descent for minimization problems. The terms above give a nice reference point when talking about combinatorial optimization problems and algorithms by making up the fundamental goals of them. Moreover, they give ways to reason about the issues an algorithm might face for a particular problem. For example, if a local search method does not accept search points in its neighborhood of equal fitness then it might be difficult or even impossible for it to tackle plateaus.

2.2.1 Bit Strings

One of the search spaces with problems we will be working with in this project is that of bit strings in which all search points are bit strings of a dimension n. There are a myriad of different problems where the search points can be represented this way as a bit can represent the choice of including an element in a set or not. This way the solutions to a problem like the minimum spanning tree problem for example could be bit strings, where each bit has an associated edge and a one means that the edge is part of the minimum spanning tree and a zero means that it is excluded. Note that some search points may give rise to trees that are not spanning the graph or to subgraphs that contain cycles and are therefore not trees, but this does not void the applicability of the representation; the set of constraints Ω of the combinatorial optimization problem take care of this by defining what search points are valid solutions for the problem and which are not.[8]

In this project we will focus on a special type of bit string problems called pseudo-boolean functions, which are mathematical functions mapping all bit strings

to a fitness value. These are interesting as they provide many simple functions that can be used to benchmark algorithms on and more easily study their behavior. Before presenting the problems some more preliminaries relating to this search space are given, which makes it possible to better classify the different problems. We start with the definition of linear functions:

Definition 2.3. *A function $f : \{0,1\}^n \to \mathbb{R}$ is called linear if there exist weights $w_0, w_1, \dots, w_n \in \mathbb{R}$ such that:[10, 30]*

$$f(x) = w_0 + \sum_{i=1}^{n} w_i x_i, \forall x \in \{0,1\}^n \tag{1}$$

Alternatively (1) may be written as $f(x_n, \dots, x_1) = w_n x_n + \dots + w_1 x_1 + w_0$. Thus, a linear function is a function that can be written as a weighted combination of the bits such that each bit contributes a certain amount to the overall fitness. Formally, if we perceive $\{0,1\}^n$ as a vector space of bit vectors and the weights $w_0, w_1, \dots, w_n \in \mathbb{R}$ being scalars, then the function must be able to be written as a linear combination of vectors to be linear. The definition of linear functions is important, since there are many linear pseudo-boolean functions of theoretical interest and because the definition has made it possible to derive some strong results for the optimization time for many different metaheuristics (see e.g. Theorem 2.1).

Another class of functions is the set of unimodal functions:

Definition 2.4. *A function $f : \{0,1\}^n \to \mathbb{R}$ is called (strictly) unimodal if there exists for each non-optimal search point x a hamming neighbor x' where $f(x') > f(x)$.[21]*

A hamming neighbor to a search point in the context of bit strings is another search point that differs in only in the value of a single bit. We typically make a distinction between a function being strictly unimodal and just unimodal depending on if a greater than or equals sign is used instead of a greater than sign or not in the expression $f(x') > f(x)$, although this is not always done in the literature. The definition above defines the term strictly unimodal. At first glance, strictly unimodal functions are intuitively easy to optimize, since they ensure that it is always possible to increase fitness by flipping one of the bits unless the optimum is reached. For many strictly unimodal functions it is indeed also the case that they can be optimized in polynomial time by e.g. the (1+1) EA presented in section 2.3.1. It is important to mention, however, that it is not always the case: Droste, Jansen and Wegener based on previous work by Rudolph define a (strictly) unimodal function called Path_k, where k is a parameter and prove that choosing $k = \sqrt{n-1}$ makes the expected running time of the (1+1) EA $\Theta(n^{3/2} 2^{\sqrt{n}})$ i.e. exponential.[7]

Lastly, we define smooth integer functions as follows:

Definition 2.5. *A pseudo-boolean function $f : \{0,1\}^n \to \mathbb{R}$ is called smooth integer, if $f(x) \in \mathbb{Z}$ for all $x \in \{0,1\}^n$ and if $|f(x) - f(y)| \leq 1$ for all $x, y \in \{0,1\}^n$ with $H(x,y) \leq 1$, where $H(x,y)$ denotes the Hamming distance between x and y.[12]*

The term smooth integer stems from the observation that a smooth integer function does not have drastic changes in fitness between hamming neighbors, since the difference is at most 1. Unlike strictly unimodal functions it is not difficult to see that smooth integer functions can be difficult to optimize, since a smooth integer function may have many local optima and plateaus.

We now present the different pseudo-boolean functions of interest in this project. All the problems are in the following formulated as maximization problems, but many of them could easily have been defined as minimization problems instead. We start by defining OneMax:

Definition 2.6. *The fitness function OneMax: $\{0,1\}^n \to \{0,1,\dots,n\}$ is defined by:[8]*

$$OneMax(x_1, \dots, x_n) = x_1 + \cdots + x_n \tag{2}$$

It is easy to see that OneMax simply counts the number of ones in the bit string, hence the name. This problem is one of the most well known and investigated problems in the field of metaheuristics and is interesting because it is arguably the easiest linear function both in definition and ability to be optimized. Nevertheless, it turns out that for some algorithms like the (1+1) EA (see section 2.3.1), the upper bound on the expected optimization time obtained for OneMax actually also holds for all linear functions up to lower-order terms for this algorithm. This is just one example that motivates the study of the performance of an algorithm on OneMax, since despite the problem being simple it can still help to highlight key aspects of the workings an algorithm and the results may also generalize to other problems as well.[30]

Another very important pseudo-boolean function is called LeadingOnes and is defined as follows:

Definition 2.7. *The fitness function LeadingOnes: $\{0,1\}^n \to \mathbb{R}$ is defined by:[2]*

$$LeadingOnes(x) = \sum_{i=1}^{n} \prod_{j=1}^{i} x_j \tag{3}$$

The name comes from the observation that the function value is exactly equal to the number of leading ones. As a consequence a bit string consisting of all ones except the very first bit will have a fitness of 0 making it no better than the all zeros bit string in terms of fitness. Thus, LeadingOnes differs from OneMax in that the position of the bits are of importance. For this reason LeadingOnes is unlike OneMax not a linear function and harder to optimize for many algorithms.

A function with some of the characteristics of both OneMax and LeadingOnes is called BinVal:

Definition 2.8. *The fitness function BinVal:* $\{0,1\}^n \rightarrow \{0,1,\ldots,2^n-1\}$ *is defined by:[8]*

$$BinVal(x) = \sum_{i=1}^{n} 2^{n-i} x_i \tag{4}$$

We see that BinVal computes the integer value of a bit string with x_1 being the most significant bit. This means that bits further to the left contribute more to the fitness than bits to the right, which is somewhat reminiscent of LeadingOnes. That said, there is a major difference between them in that BinVal is actually a linear function because it satisfies definition 2.3 contrary to LeadingOnes. For this reason BinVal turns out to be easier to solve for optimality for many algorithms. One example of this is seen for the (1+1) EA in section 2.3.1.

We now turn to some functions that are interesting when examining how algorithms handle obstacles. The first function is called Jump$_k$ and is defined below:

Definition 2.9. *The fitness function Jump$_k$:* $\{0,1\}^n \rightarrow \mathbb{R}$ *is defined by:[4]*

$$Jump_k(x) = \begin{cases} k + OneMax(x) & \textit{if } OneMax(x) = n \textit{ or } OneMax(x) \leq n-k \\ n - OneMax(x) & \textit{otherwise} \end{cases} \tag{5}$$

Note that we have utilized the definition of OneMax in this definition, since it gives an easy way to concisely refer to the number of ones in search point x. Jump$_k$ is sometimes also called Jump$_{n,m}$ e.g. in [7], where m is equivalent to k in the definition given here. Jump$_k$ behaves like OneMax up until the number of ones being $n-k+1$ where the fitness suddenly drops significantly and declines as the number of ones increases until the number of ones is n which gives rise to the optimum. Intuitively, Jump$_k$ defines a kind of gap in the state-space landscape of OneMax, which an algorithm must "jump" over in order to reach the optimum, that is multiple bits have to be flipped at once in order to not accept a very high loss of fitness. The point just before the gap is a local maximum where an algorithm may get stuck.

A function similar to Jump$_k$ is Trap, which also behaves like OneMax for most values of the number of ones. Only the fitness of the search point consisting of only zeros is different from OneMax and is the optimum:

Definition 2.10. *The fitness function Trap:* $\{0,1\}^n \rightarrow \mathbb{R}$ *is defined by:[8]*

$$Trap(x) = \begin{cases} OneMax(x) & \textit{if } OneMax(x) \geq 1 \\ n+1 & \textit{otherwise} \end{cases} \tag{6}$$

The name of the function comes from the observation that most algorithms would see an increase in the number of ones as a good thing, since the fitness increases, but once the local maximum of all ones has been reached the algorithm is trapped, since it needs to flip all the bits in order to reach the optimum.

A very hard linear pseudo-boolean function is called Needle, because there are no clues as to how an algorithm should find the optimum, due to all search points except the all ones-bit-string having a fitness higher than 0:

Definition 2.11. *The fitness function Needle:* $\{0,1\}^n \rightarrow \{0,1,\ldots,n\}$ *is defined by:[21]*

$$Needle(x) = \begin{cases} 1 & \text{if } n = x_{OPT}, \\ 0 & \text{otherwise}, \end{cases} \tag{7}$$

where x_{OPT} *is the unique global optimum.*

Solving the problem is like finding a needle in a haystack, which is where the name orignates from.

Relating the problems just defined with the three classifications of pseudo-boolean functions we see that OneMax and BinVal are linear, strictly unimodal and smooth integer functions, while LeadingOnes is also strictly unimodal and a smooth integer function, but is not linear.

For many of these problems only the number of ones in the bit strings affect the fitness, so for these problems we can draw a simplified state-space landscape as seen in figure 2. For LeadingOnes and BinVal this is not readily done in a similar manner, but can be done using a boolean hypercube representation (see figure 3).

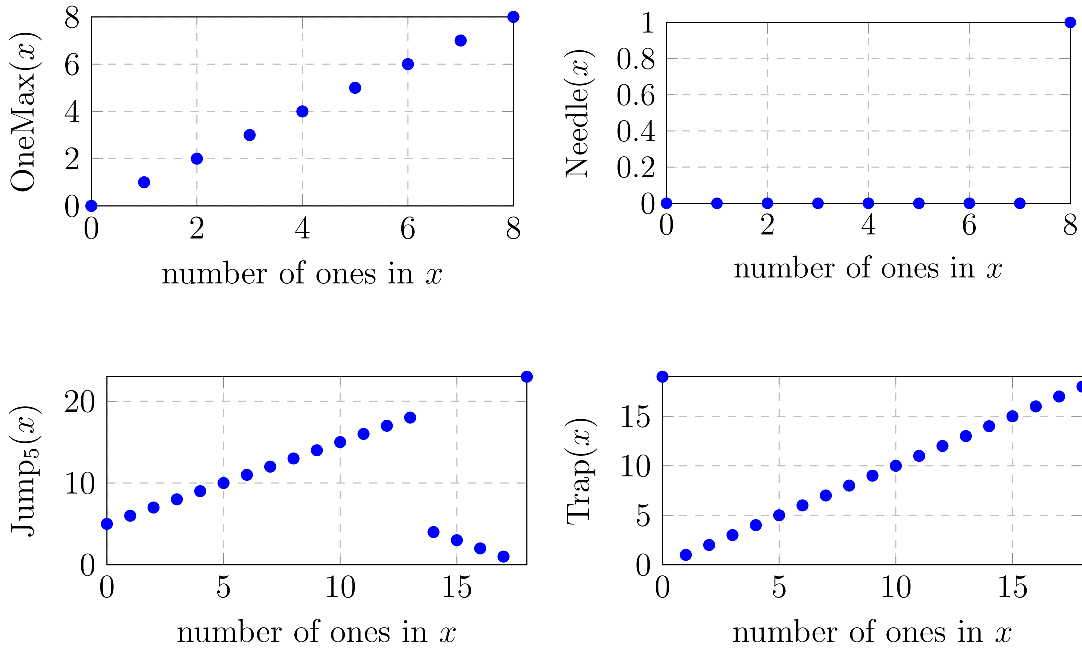


Figure 2: Fitness landscape for some sample problems

Some example values of n have been chosen for each problem; the parameter k in Jump$_k$ has been set to five in the example. It is important to stress that adjacent points are not necessarily the only points in the neighborhood of one another, since some algorithms may flip more than one bit at once. The adjacency of the points in the figures are here Hamming neighbors. Nevertheless, the figures give a nice layout of the problems and a good indication of the difficulties that an algorithm may face when attempting to find the optimum.

In order to study the performance of the (1+1) EA and simulated annealing (refer to section 2.3.1 and 2.4, respectively) on obstacles two more functions are defined. Since they mostly serve to illustrate special properties of said algorithms and are not common in the literature they are just denoted by f_1 and f_2:

Definition 2.12. *The fitness function $f_1 : \{0,1\}^n \rightarrow \mathbb{R}$ is defined by:[12]*

$$f_1(x) = \begin{cases} OneMax(x) & \textit{if } OneMax(x) \neq n-1 \\ 0 & \textit{otherwise} \end{cases} \tag{8}$$

Definition 2.13. *The fitness function $f_2 : \{0,1\}^n \rightarrow \mathbb{R}$ is defined by:[12]*

$$f_2(x) = \begin{cases} 2^n \cdot n + 1 & \textit{if } x = 0^n \\ 2^n \cdot n - (n-i) & \textit{if } x = 1^i 0^{n-i}, i \in \{1,\dots,n\} \\ 2^n \cdot OneMax(x) & \textit{otherwise} \end{cases} \tag{9}$$

Note that f_1 is akin to Jump_k, for $k = 2$ and contains a one-wide gap between the optimum and the rest of the search space. f_2 has optimum 0^n like Trap and also leads towards the all ones bit string. Unlike Trap, however, the second case of the function gives a kind of "escape-route" that leads to the optimum. An algorithm that accepts a slight worsening in fitness may incrementally flip one bits to zeros from the right of the bit string until the all-zeros bit string is reached.

In order to draw a visual representation of the problems LeadingOnes, BinVal and f_2 we draw a boolean hypercube as seen in the figure below.

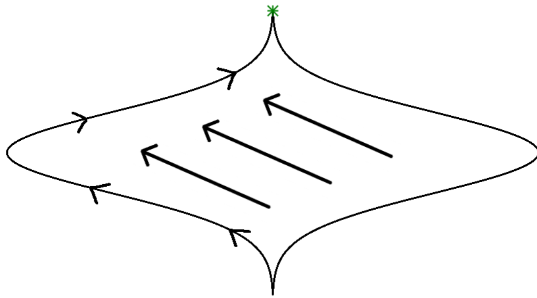


Figure 3: LeadingOnes and BinVal

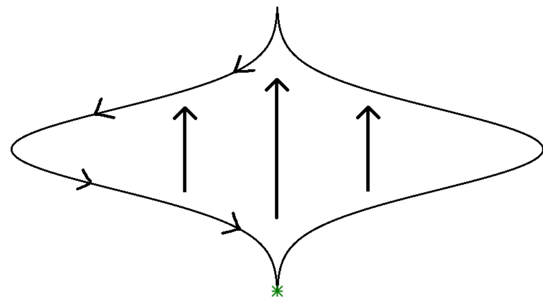


Figure 4: f_2

In short, these figures show how the fitness increases depending on how many ones there are in a search point and the placement of these. The placement along the vertical axis is based on the number of ones with more ones meaning that it is placed further up; the all-ones bit string is thereby the single search point in the very top. The positioning along the horizontal axis is done in a way such that one-bits to the left of the middle of the bit string contribute to moving it to the left in the boolean hypercube and one-bits to the right contribute to moving it to the right. For a more precise description of the calculation of this please refer to section 5.2 with the details of the implementation of a boolean hypercube as a means to visualize bit string search points in the framework. The arrows in the figure show the general direction of where the fitness grows. The green marker shows the optimum.

In the figure 3 for LeadingOnes and BinVal the fitness will generally tend to increase with a higher number of ones and the ones being further to the left although this does not hold in general. For LeadingOnes a search point with many

one-bits far to the left can still have equal or worse fitness than a search point with very few ones if it does not have a leading one. For figure 4 we see that the fitness increases towards the all ones bit string, but the optimum is in fact the all zeros bit string, which the second case of f_2 helps the algorithm to realise by making the fitness increase along the left most edge. For this reason the problem may be easier to solve for optimality than Trap for some algorithms; it turns out that this is the case for simulated annealing unlike the (1+1) EA (see section 2.4.3).[12]

2.2.2 Permutations

As with bit strings, many kinds of problems involve determining a permutation of a number of elements that is optimal in some sense. An example could be the 8-queen problem in which 8 queens need to be placed on a chess board such that as few queens are attacking each other as possible. The solutions to this problem can be represented as a permutation by letting the index in the permutation be the column and the value be the row under the assumption that the queens are placed in a unique row. The most well known problem in the search space of permutations, however, is arguably the traveling salesperson problem (abbreviated TSP for short), which will be the subject of further study in this project in this search space: Given a number cities and the distance between each pair of cities the task is to construct the shortest possible tour visiting all the cities once. Formally we are given a weighted complete graph with n vertices and the goal is to determine a permutation of the vertices such that following the vertices in this order makes for the tour of shortest length.[8]

The traveling salesperson problem has been proved to be NP-complete meaning no algorithm exists that can compute the optimal solution in polynomial time in the input size unless $P = NP$. Another important known result is that in general it is not possible to give a polynomial time algorithm that computes a solution that is guaranteed to be a factor α from the optimum either. If an α-approximation algorithm did exist, that is, an algorithm that runs in polynomial time, returns a valid solution and the solution is within factor α from the optimum, then this algorithm could be used to solve (i.e. find the optimal solution) to the hamiltonian cycle problem in polynomial time as well; since the hamiltonian cycle problem is also NP-complete this is not possible unless $P = NP$.[27]

If the problem is relaxed or constraints to the problem instances are introduced, however, then it may be solvable or approximative in polynomial time. The TSP instances that we will introduce below and analyze in section 6 use the Euclidian metric for the distance between two points and is therefore a metric TSP instance. For metric TSP instances approximation algorithms exist and it could therefore be interesting to analyze how well the solutions of the metaheuristics of this project compare to the theoretically achievable bounds of these known approximation

algorithms. Two well known algorithms for the metric TSP are the double tree algorithm and Christofides' algorithm with approximation factors of 2 and 3/2, respectively. An important property of metric TSP instances that is used in these approximation algorithms is that the weight of a minimum spanning tree is a lower bound on the optimum tour, since if an edge in the optimal tour is removed it becomes a spanning tree, which cannot have lower weight than the minimum spanning tree.[3]

Another interesting value to compare the metaheuristics with is a tour generated by the nearest-neighbor heuristic, which is a greedy algorithm that chooses the closest city as the one to visit next. Generating a solution using this strategy can be done very fast in terms of constant factors in the running time and is simple to implement as well. The greedy algorithm will in most cases not be able to find the optimum even after letting it run for a long time because it is only able to explore a very small part of the search space and so only serves as a target value to beat.

The TSP instances used as part of this project have been downloaded from the TSPLIB library [9] as .tsp files with the coordinates of all cities. The main TSP instance of interest is called berlin52, which has 52 locations in Berlin that must be visited. Technically these locations are not cities, but since the problem remains the same we will not disallow ourselves to denote them as cities nonetheless. The figures below show the 52 locations and the optimal tour, which is also specified on the TSPLIB website. The framework of this project presented in section 4 has been used to render the locations and to draw the optimal tour.

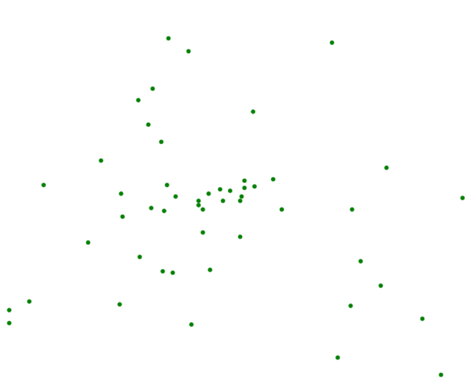


Figure 5: berlin52.tsp

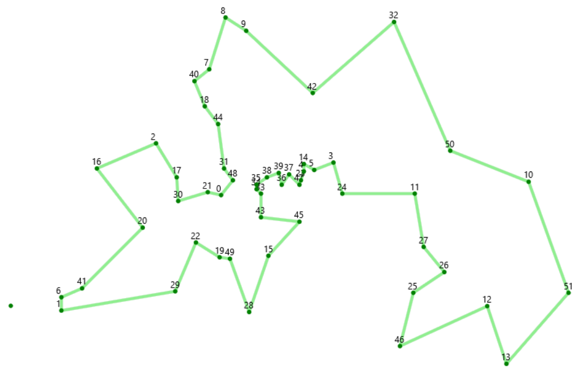


Figure 6: berlin52.tsp with the optimal tour drawn

The library has many other TSP instances of various sizes and layouts that may be interesting to run algorithms on as well. Below are four selected TSP instances with a number of cities of 127, 225, 318 and 1379, respectively, which are incorporated into the program as well.

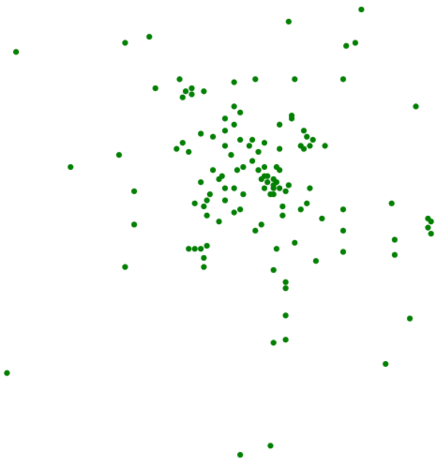


Figure 7: bier127.tsp

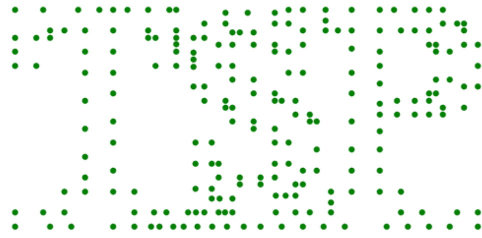


Figure 8: tsp225.tsp

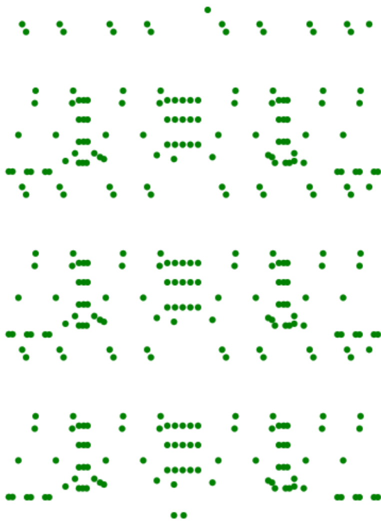


Figure 9: lin318.tsp

Figure 10: nrw1379.tsp

Another TSP instance that is also featured in the program is called d15112 and is a massive TSP instance with 15112 cities in Germany. Figure 58 in the appendix shows this instance, where the radius of the dots have been decreased in order for it to better display. This instance is not meant to be used with the program normally and the program has not been designed to work well with this very big TSP instance. It has been included in the program nonetheless as a way to stress-test the application and the choice was made to keep it out of interest and for potential future testing.

The optimal tour lengths for five instances above are given by the documentation of the TSPLIB library and are shown in table 1.

TSP Instance	Optimal tour length
berlin52	7542
bier127	118282
tsp225	3919
lin318	42029
nrw1379	56638

Table 1: Optimal tour lengths[9]

Observe that the optimal tour length is not necessarily increasing with the number of cities in the TSP instance. We see that the optimal tour for bier127 is the longest, which is because the cities are further apart for the bier127 tsp instance compared to e.g. nrw1379.

It is also important to mention that the documentation stipulates that the length between two cities with coordinates $(x[i], y[i])$ and $(x[j], y[j])$ be calculated the following way:

```
xd = x[i] - x[j];
yd = y[i] - y[j];
dij = nint( sqrt( xd*xd + yd*yd) );
```

where `nint(x)` is defined as `(int) (x+0.5)`. This is to ensure consistency between different implementations so the lengths of the tours are calculated the same way and can be compared appropriately.[9]

2.3 Evolutionary Algorithms

The first metaheuristic we will be looking at is that of Evolutionary algorithms (EAs for short), which are inspired by biological evolution. EAs come in different variants, but the standard evolutionary algorithm scheme works by maintaining a population of individuals from which new individuals and populations are evolved the following way: At each iteration the EA selects two individuals x and y from the current population, applies some crossover operator that takes some traits from each of these to give an offspring z; the search points x and y are called parents and z the child. A mutation operator might then afterwards be applied to z to alter it slightly. Doing this a number of times with different members of the population evolves the population to a new one and the algorithm continues like this until a certain stopping criterion is met.[22]

As reflected upon in a paper by Duc-Cuong Dang et al. [4] it is usually crucial to strive for a high diversity among the individuals in the population in order to avoid premature convergence. If the individuals are too much the same, crossover may be ineffective and the algorithm may not achieve good solutions even if it is run for a long time. Various measures to avoid low diversity exist, but are beyond the scope of this project.

The way individuals are selected and how they produce offspring and how it is mutated can be implemented in many different ways depending on which kinds of selection-, crossover-, and mutation operators are used, which is what makes this algorithm highly adaptable to different problems. It is also possible to only let crossover or mutation happen with a certain probability or drop it entirely. The following is a scheme for the generic EA, which incorporates all of said operators:[8]

Algorithm 1 Generic EA

1: Initialise population P_0 of size μ
2: $t \leftarrow 0$
3: **while** stopping criterion not fulfilled **do**
4: $\quad P_{t+1} := \emptyset$
5: $\quad$ **for** $i \leftarrow 1, \ldots, \mu$ **do**
6: $\quad\quad$ Choose two individuals x and y from P_t by applying some
7: $\quad\quad$ selection operator
8: $\quad\quad$ Create z by applying some crossover operator to x and y
9: $\quad\quad$ Create z' by applying some mutation operator to z.
10: $\quad\quad$ Add z' to P_{t+1}
11: $\quad$ **end for**
12: $\quad t \leftarrow t + 1$
13: **end while**

How well an EA performs often depends crucially on which operators are used. Not all operators are applicable for all search spaces either. Below we briefly describe some of the most prominent selection, crossover and mutation operators relevant for bit strings and permutations.

Selection: In order to create a new generation a number of individuals from the previous population are chosen by some selection strategy. Intuition would say that a good strategy is to make it more likely for individuals with high fitness to be chosen rather than individuals with low fitness. This is exactly what the selection operator called *fitness-proportional selection* does. Another such strategy is the *tournament selection* operator, where an individual is selected by first picking k individuals among the population uniformly at random and then choose the one with highest fitness. Both fitness-proportional selection and tournament selection can be used for bit string problems as well as for permutations.

Crossover: For bit strings a crossover operator could be the *uniform crossover* which chooses the bits of the child uniformly at random from the parents. Crossover operators for permutations are a bit more involved, since they must ensure that the child is still a valid permutation. Uniform crossover is therefore not applicable in its basic form just described. Nevertheless, many crossover operators for permutations specifically designed with the TSP problem in mind exist. Some operators for permutations include the *order crossover* (OX) and the *partially matched crossover* (PMX). OX tries to preserve the order in which cities are visited in one of the parents as much as possible while PMX aims to preserve the absolute positions in which cities are preserved to a large extent. A more illustrious crossover, however, which has achieved great results for the TSP in practice is the *generalized partition crossover* (GPX). The idea is to make partitions based on the common edges of the union graph of the two parent tours and optimize the tour in each component. In a way it can be said to incorporate both OX and PMX, since it has the two aforementioned structural properties. GPX can be implemented in linear time in the number of cities $O(n)$ and is one of the most efficient crossover operators known for the TSP. Because it is not easy to implement efficiently it will not be part of the framework of this project though.

Mutation: An example of a simple mutation operator for bit strings is the bit flip mutation, which picks a bit at a random position and flips it. Another example is the swap mutation, which swaps the elements at two positions randomly picked. For permutations a well known and well performing mutation operator is called *2-OPT*: Given a permutation $(i_1, \dots, i_n)$ it chooses two indices $k, l \in \{1, \dots, n\}$, $k + 1 < l$ uniformly at random and produces the following new tour: $(i_1, \dots, i_k, i_l, i_{l-1}, \dots, i_{k+1}, i_{l+1}, \dots, i_n)$. The figure below shows an example of 2-OPT applied to the tour $(1, 2, 4, 3, 5)$ with $k = 2$ and $l = 4$.

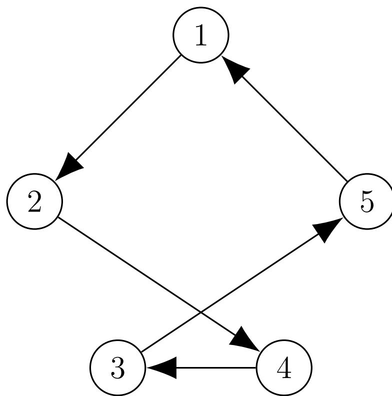


Figure 11: Before 2-OPT mutation

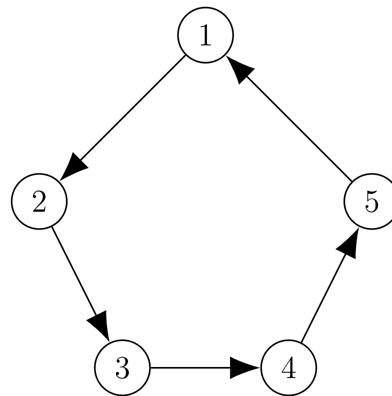


Figure 12: After 2-OPT mutation

Thus, 2-OPT takes out the edges (i_k, i_{k+1}) and (i_l, i_{l+1}), reverses all the edge

orientations of the edges of the sub-tour $(i_{k+1}, \ldots, i_l)$ and then reconnects everything by adding the edges (i_k, i_l) and (i_{k+1}, i_{l+1}). The idea behind this operator is that it can help fix parts of the tour by straitening them out like it did for the tour in the example. Generally this seems like a good idea, since overlaps in practice rarely seem to make for short tours. This is also part of the reason why the convex hull of the graph is often interesting to examine for TSP, when looking at the structure of solutions produced.[10, 8]

2.3.1 (1+1) EA

A simple and yet remarkably well performing evolutionary algorithm is the (1+1) EA. In its standard form it only focuses on a single individual, which it attempts to mutate a number of times at each iteration, though variations of it exist with a population of more than one individual. A key property of the (1+1) EA is that it only accepts the newly mutated individual generated at a given iteration if it has a higher or equal fitness than the current search point. At first glance this might not seem like a good strategy with the issues of local maxima already described in mind. However, because the (1+1) EA may make multiple mutations in a single iteration it is able to get away from a suboptimal solution and this way avoid getting stuck. For this reason it is less prone to get stuck at local optima and obstacles than the very similar EA called RLS (Randomized Local Search), which like the (1+1) EA only focuses on a single search point; the difference is that RLS mutates a specific number of times, typically only once in each iteration and so does may not have the chance to mutate enough times to escape suboptimal solutions.

For the search space of bit strings the algorithm is defined as follows:[8]

Algorithm 2 (1+1) EA for Pseudo-Boolean Functions

1: $t := 0$. Choose $x \in \{0,1\}^n$ uniformly at random
2: **repeat**
3: $y := x$.
4: For each $i \in \{1, 2, \ldots, n\}$: With probability $1/n$ flip bit i.
5: **if** $f(y) \geq f(x_t)$ **then**
6: $x_{t+1} := y$
7: **else**
8: $x_{t+1} := x_t$
9: **end if**
10: $t := t + 1$
11: **until** some stopping criterion is fulfilled

Note that the algorithm as presented here is only applicable for maximization

problems. The greater-than-or-equals sign in line 5 should be reversed if it is to be applied to minimization problems.

The formulation just given was for bit strings and does not seem readily adapted to other search spaces as well. However, it is possible to reformulate it in a way that makes it applicable for permutations while still capturing the essential working principles behind it. In order to do this we make the observation that the number of bit flips (mutations) follows a distribution which approaches the Poisson distribution with parameter $\lambda = 1$ as n goes to ∞; as long as n is reasonably high the approximation of using the Poisson distribution is very good. Thus, by substituting the bit flip mutation with a mutation operator applicable for TSP and applying this a number of times determined by the Poisson distribution we get an algorithm that works very much the same way. The algorithm then becomes:[25]

Algorithm 3 (1+1) EA on TSP

1: $t := 0$. Let x be a random permutation of n cities.
2: **repeat**
3: $y := x$.
4: Choose s according to a Poisson distribution with parameter $\lambda = 1$ and perform sequentially s + 1 mutations on y.
5: **if** $f(y) \leq f(x_t)$ **then**
6: $x_{t+1} := y$
7: **else**
8: $x_{t+1} := x_t$
9: **end if**
10: $t := t + 1$
11: **until** some stopping criterion is fulfilled

Note that since the TSP is a minimization problem the greater-than-or-equals sign from before has been flipped accordingly. In the following we will be using 2-OPT as the mutation operator. The reason we add one to s is to ignore iterations when nothing would happen if we get 0 when drawing from the Poisson distribution. Clearly, such iterations and are not worth considering and can easily be avoided this way.

We note that Algorithm 3 can potentially make more than n mutations, which the (1+1) EA for pseudo-boolean functions cannot do. Following Scharnow and Wegener in their paper [25] we will not enforce a limit of n mutations, however, since the probability of this happening is very small as long as n is not too low.

We now turn to the optimization time of the (1+1) EA and start with a general result that holds for all linear functions:

Theorem 2.1. *On any linear function, the following holds for the expected optimization time $E(T_p)$ of the (1+1) EA with mutation probability p.[30]*

(1) If $p = \omega((\ln n)/n)$ or $p = o(n^{-C})$ for every constant $C > 0$, then $E(T_p)$ is superpolynomial.

(2) If $p = \Omega(n^{-C})$ for some constant $C > 0$ and $p = O((\ln n)/n)$, then $E(T_p)$ is polynomial.

(3) If $p = c/n$ for a constant $c > 0$, then $E(T_p) = (1 \pm o(1))\frac{e^c}{c} n \ln n$.

(4) $E(T_p)$ is minimized for mutation probability $p = 1/n$ (up to lower-order terms).

(5) No mutation-based EA has an expected optimization time smaller than $E_{1/n}$ (up to lower order terms).

This theorem is very powerful, since it holds for all linear functions and very precisely gives the connection between the mutation probability and the optimization time. Since OneMax is a linear function and setting $p = \frac{1}{n}$ is optimal for all linear functions, the theorem gives us the following corollary:

Corrolary 2.1. *The expected optimization time of the (1+1) EA on OneMax with mutation probability $p = \frac{1}{n}$ is:*

$$E(T_{1/n}) = en \ln n \pm O(n) = \Theta(n \log n) \tag{10}$$

Theorem 2.1 is proved in the paper by Witt [30] using multiplicative drift, which is a very useful analysis tool for randomized search heuristics that can be used when there is a multiplicative decrease in the distance between the current search point and the optimum. That is, when it is the case that the expected difference in fitness between the current search point and the fitness of the optimum decreases by a factor in each iterations.

If we limit ourselves to the case of the (1+1) EA on OneMax only it is also possible to reach the complexity $\Theta(n \log n)$ in corollary 2.1 in another perhaps more simple way. To show the upper bound, one can look at the steps that only flip a single zero-bit to a one-bit of the current search point. This has the probability $(n-i) \cdot \frac{1}{n} \left(1 - \frac{1}{n}\right)^{n-1}$ since we assume there are $n - i$ zero bits that could flip to a one-bit and multiply by $\left(1 - \frac{1}{n}\right)^{n-1}$ to ensure only a single bit is flipped. By applying a few inequality rewritings and utilizing that the fitness never decreases it is possible to derive the bound by summing over the expected time to go up by one in fitness starting from 0.

Having observed that the initial search point in expectation starts out with $n/2$ zero-bits, the aim in the proof of the lower bound is to show that with positive

probability it happens that at least one of the $n/2$ zero-bits is not flipped in $cn \ln n$ steps, where c is a constant, which can be done using that the mutation is independent for each bit.[10, 8]

For RLS it is even simpler to show that the expected optimization time is $\Theta(n \log n)$ by applying a waiting time argument to the probability of flipping a zero-bit to a one-bit, since only a single bit is flipped in each iteration. It turns out that RLS is actually faster by a factor of e at optimizing OneMax compared to the (1+1) EA, which is proved in a paper by Benjamin and Carola Doerr [5]. This is only a special case though; in general on other non-monotonistic functions, the RLS has a hard time competing with the performance the (1+1) EA due to the difficulties with the various types of obstacles touched upon in section 2.2.1, so we will not go deeper into the analysis of this result. Nevertheless, we will keep it in mind in the analysis of the (1+1) EA in section 6.1.

The lower bound of $\Omega(n \log n)$ for the (1+1) EA holds for all of the functions mentioned in section 2.2.1, which is expressed in the following theorem:

Theorem 2.2. *Let $f : \{0,1\}^n \to \mathbb{R}$ be a function with a unique global optimum. The expected optimization time of the (1+1) EA on f is then $\Omega(n \log n)$.[10, 8]*

Thus, we cannot hope to solve any of aforementioned pseudo-boolean functions faster than $\Omega(n \log n)$. This is not surprising considering the bound of $\Theta(n \log n)$ was tight for the (1+1) EA on OneMax, which is as stated earlier arguably one of the easiest functions to optimize among those with a unique global optimum. Note that we require that a unique global optimum exist in this theorem, because otherwise one may make up a pseudo-boolean function where all the search points are optima in which case the lower bound would clearly not hold.

Turning to LeadingOnes we cannot use the result of Theorem 2.1, since LeadingOnes is not a linear function. In a paper by Böttcher, Doerr and Neumann [2], however, they are remarkably able to achieve a precise expression of the optimization time based on the mutation probability p:

Theorem 2.3. *The expected optimization time of the (1+1) EA with fixed mutation rate p for LeadingOnes is:[2]*

$$T = \frac{1}{2p^2}\left((1-p)^{1-n} - (1-p)\right) \tag{11}$$

Observe that there is no O-notation used in the expression of this optimization time, which makes it possible to precisely calculate the optimization time for different values of p. Following the authors in order to determine the optimal value for p, we compute the derivative of T with respect to p and set it equal to 0. Doing this gives us an equation, which we solve numerically to get that the value $p = \frac{1.5936}{n}$ is the global minimum and hence the optimal value for minimizing the

optimization time. If we set $p = \frac{1.5936}{n}$ in the optimization time expression this gives us an expected optimization time of approximately $0.77n^2$, which is 16 % faster than the expected optimization time using the standard $p = \frac{1}{n}$ value.[2]

By flipping multiple bits in one iteration the (1+1) EA is able to jump over obstacles and solve the problem Jump_k, which is expressed in the bound of the following theorem:

Theorem 2.4. *Let $n \in \mathbb{N}$ and $k \in \{1, 2, \ldots, n\}$ be given. The expected optimization time of the (1+1) EA on Jump_k is then $\Theta(n^k + n \log n)$.[10]*

The term $n \log n$ stems from the usual optimization time of the (1+1) EA on OneMax. As discussed earlier typically the (1+1) EA will after $\Theta(n \log n)$ iterations reach the local maximum, where it will stay until it is able to flip all k remaining zero-bits in one step while not flipping any one-bits, which is where the term n^k comes in. It is not difficult to see that the expected number of steps before the (1+1) EA jumps across the gap is n^k through the following reasoning: The probability of flipping exactly the k remaining zero-bits must be the probability of not flipping any one-bits, $\left(1 - \frac{1}{n}\right)^{n-k}$, times the probability of flipping the k zero-bits, $\left(\frac{1}{n}\right)^k$. Using a well known inequality result this is greater than or equal to $\frac{1}{en^k}$. Using a waiting time argument, the expected time is then $\frac{1}{\frac{1}{en^k}} = en^k = \Theta(n^k)$. Formally, the theorem can be proven using the fitness-level method (refer to section 2.5.1 for a presentation and example of usage of this method).[10]

When studying the optimization time of the (1+1) EA we give one last result that one should keep in mind. Given any pseudo-boolean function, observe that no matter the optimum the (1+1) EA will have positive probability of flipping all the incorrect bits of the current search points in a single iteration. The probability of this happening is $(1/n)^n$ and is independent of the current search point. Using a waiting time argument the expected number of iterations before such a step happens is then $1/(1/n)^n = n^n$, which proves the following theorem:

Theorem 2.5. *Let $f : \{0, 1\}^n \rightarrow \mathbb{R}$ be arbitrary. The expected optimization time of the (1+1) EA on f is then at most n^n.[8]*

Obviously this upper bound on the optimization time is not particularly interesting. Nevertheless, it is good to keep in mind that the (1+1) EA is able to optimize all pseudo-boolean functions of a given dimension n with probability infinitely close to 1 for the number of iterations going to ∞, which is not the case for all algorithms. RLS for example is not able to make such a step meaning we cannot prove a similar upper bound.

2.4 Simulated Annealing

Simulated annealing (SA) is a metaheuristic inspired by the annealing process in metallurgy in which a solid (e.g. steel) is heated and then cooled slowly in order to achieve a perfect lattice structure with a minimum energy state. Like the real world annealing process, simulated annealing maintains an artificial temperature value, which influences the probability that new search points are accepted. Unlike the (1+1) EA, simulated annealing can accept worsenings in fitness of new search points, but as the temperature decreases the probability at which it accepts worse search points does so as well. The main motivation is to allow the algorithm to be more chaotic in the beginning and traverse many areas of the search space and as the temperature cools stay in areas of high quality solutions.[1]

The algorithm works by first choosing an initial search point x_0 from the search space uniformly at random. In each iteration it then chooses a new search point in the neighborhood of x_0. How the neighborhood is defined varies with both the problem at hand and the way the algorithm is implemented. Usually the neighborhood consists of the search points of the search space that can achieved by applying some mutation operator on x_0. The chosen search point y in the neighborhood is accepted with probability $\min\{1, e^{-\frac{f(y)-f(x_0)}{T(t)}}\}$, where $T(t)$ is the temperature at time t (time is typically equivalent to the number of iterations). The temperature is determined as by a function $T : \mathbb{N}_0 \to \mathbb{R}^+$ called the cooling schedule. If y is accepted this becomes the new current search point in the next iteration. After each iteration the time increases and the temperature is updated. We observe from the probability expression that the probability of accepting new search points decreases as the temperature decreases. Further, the probability will be lower for search points with low fitness relative to the current one. If the new search point has higher fitness than the current one the expression $e^{-\frac{f(y)-f(x_t)}{T(t)}}$ will be above 1, so the minimum of this is 1, so better search points will always be accepted. Below is the pseudocode for the algorithm.[1]

Algorithm 4 Simulated Annealing

1: Set $t := 0$. Choose $x_t \in S$ uniformly at random.
2: **repeat**
3: Choose $y \in N(x_t)$ uniformly at random.
4: With probability $\min\{1, e^{-\frac{f(y)-f(x_t)}{T(t)}}\}$ set $x_{t+1} := y$
5: else set $x_{t+1} := x_t$.
6: $t := t + 1$
7: **until** some stopping criterion is fulfilled

In the literature the acceptance probability is alternatively defined by $\alpha(t)^{-(f(y)-f(x_t))}$,

where $\alpha(t) = e^{1/T(t)}$, which is equivalent to the above expression. Note that $\alpha(t)$ and $T(t)$ are inverse proportional, meaning a high temperature corresponds to a low value of alpha and vice versa.

The cooling schedule plays a significant role in terms of how well the simulated annealing algorithm will perform. If it decreases too fast the algorithm might not have enough time to get away from a local optimum and will get stuck there. On the contrary if the temperature cools too slowly the algorithm might find a search point close to the optimum, but then could go far away from it by accepting a big worsening move. Another thing worth considering is the starting temperature, which should be high enough to allow the acceptance of worsening moves such that the algorithm can get over obstacles. We make the observation that if the starting temperature is set infinitely close to 0, then simulated annealing directly corresponds to RLS, since the probability of accepting a worsening move is $\lim_{T(t)\to 0} e^{-\frac{f(y)-f(x_t)}{T(t)}} = 0$. For simplicity we will just write $T(1) = 0$ to denote this though. As an interesting consequence this gives rise to a phase transition of SA to RLS if the algorithm is allowed to run for long enough with very low temperature. Furthermore, by running the algorithm with temperatures infinitely close to 0, it is possible to apply many of the theoretical results that hold for the expected optimization time of RLS to SA as well.

Very closely related to simulated annealing as defined above is the algorithm called the Metropolis Algorithm (MA). Sometimes in the literature this is defined as a special kind of simulated annealing and at other times vice versa. In the metropolis algorithm the temperature is fixed throughout the entire search process, meaning the algorithm behaves the same at later stages of the search as in the beginning. It is possible to show that simulated annealing can outperform the metropolis algorithm for some problems because of this. Wegener showed the first natural such results in the case of minimum spanning trees [28]. Based on his analysis it is shown in another paper by Klaus Meer [20] that for a specific TSP instance, simulated annealing with a carefully chosen cooling schedule is able to successfully find the optimal solution in polynomial time in expectation, while the metropolis algorithm is not no matter how the temperature is set for this. Looking at the converse question: Can the metropolis algorithm beat simulated annealing? Then this is indeed the case, since as mentioned above, it is easy to choose a bad cooling schedule for simulated annealing. However, since we can transform SA to MA by picking the same starting temperature and decreasing the temperature infinitely slowly, in general we tend not to think of MA as being better than SA.[1]

2.4.1 Defining a General-Purpose Cooling Schedule

It is not trivial to define an optimal cooling schedule, which usually depends intrinsically on the problem. Nevertheless, we now make an effort to try to

establish some general guidelines that seem to work well for many problems. We will base our cooling schedule on the results in the paper by Wegener [28] and the paper by Meer [20] mentioned above. Even though the problems differ, interestingly enough the authors use almost the same cooling schedule with success. The starting temperature is set to $T(1) := m^3$ for both problems, where m is the number of edges in the graph. The rate at which the temperature decreases is set to $T(t+1) = r \cdot T(t)$, where $r = (1 - \frac{1}{cm})$ for the MST problem and $r = (1 - \frac{1}{cm^2})$ for the TSP instance, for some $c > 0$. The important characteristic of these cooling schedules is that both the starting temperature and the cooling factor depend on the size of the problem expressed through m; for larger problems meaning higher values of m, the starting temperature is set to a high value and cooled more slowly as well. This seems like a good thing to do in a lot of cases, since the state space is generally increasing with the problem size, meaning the search should take longer to allow a properly thorough search. which is achieved by a higher starting temperature and slower cooling. Another thing worth noting is that the temperature decreases exponentially by a constant factor. Wegener mentions that there are other possibilities to "continuous cooling" as he calls the one used here, like phases with constant temperature or dynamic cooling schedules where the decrease in temperature depends on the success rate or the rate of fitness improvement steps. The simple continuous cooling schedule works fine for the most part though and is what we will be using for this project. [20]

With inspiration from these two sources we will use a very similar cooling schedule to the pseudo-boolean functions and TSP. Instead of m we will be using n, that is the dimension of the search space. Note that means we will not be using the number of edges for the TSP. For a complete graph of n vertices the number of edges is $n + (n-1) + (n-2) + \cdots + 2 + 1 = \Theta(n^2)$, which would make $T(1) = n^6$ if m^3 is to be used. However, neither the graph used by Wegener nor the TSP instance used by Meer has this many edges relative to the number of vertices; the number of edges is linear in the number of vertices (for the TSP instance in Meer's paper $m = 20n$). Since the MST problem is in the same search space as the pseudo-boolean functions, namely bit strings, we will, following Wegener, set the standard cooling factor to $1 - 1/cn$. For the TSP we will like Meer set it to $1 - 1/cn^2$, which because of the exponent of n will make the cooling factor increase a bit more slowly with the problem size.

Having settled on this cooling schedule we are still left with the constant c that does not depend on the problem size and needs to be set. First we observe that a higher value of c gives a slower cooling. Recalling our previous discussion on the issues with too fast and too slow cooling we remark that the value of c is of high importance when we benchmark the algorithm. When running a benchmark we must decide on the stopping criteria, where we will not allow the algorithm to run

indefinitely until the optimum is found; it is unfeasible to test without some kind of limit to the number of iterations or CPU time. For this reason it seems like a natural idea is to choose the constant c such that the temperature attains a certain value when the limit is hit. Look at the cooling schedule for TSP. The temperature at iteration t is then given by:

$$T(t) = T(1) \cdot r^t = n^3 \cdot \left(1 - \frac{1}{cn^2}\right)^t \tag{12}$$

Given how many iterations t we allow the algorithm to run for we can thereby calculate a value of c that makes the algorithm have a temperature of T_{final} when the time is up. The problem of deciding on the value of c then becomes solving the equation for a value of T_{final}. A good value for this in general seems to be 1 with the reasoning being that is simple and is neither too high nor too low. A value of 1 means that the probability of accepting a worsening move becomes $e^{-(f(y)-f(x_t))}$, thus only depending on the difference in fitness. For OneMax this would mean that the probability of accepting a move that flips one one-bit to a zero-bit so the fitness is one lower will be 0.4; since this is slightly less than 0.5 the algorithm will still have a stronger tendency to increase the number of one-bits rather than decreasing them. The plot below shows the probability of acceptance of worsening moves at this temperature.

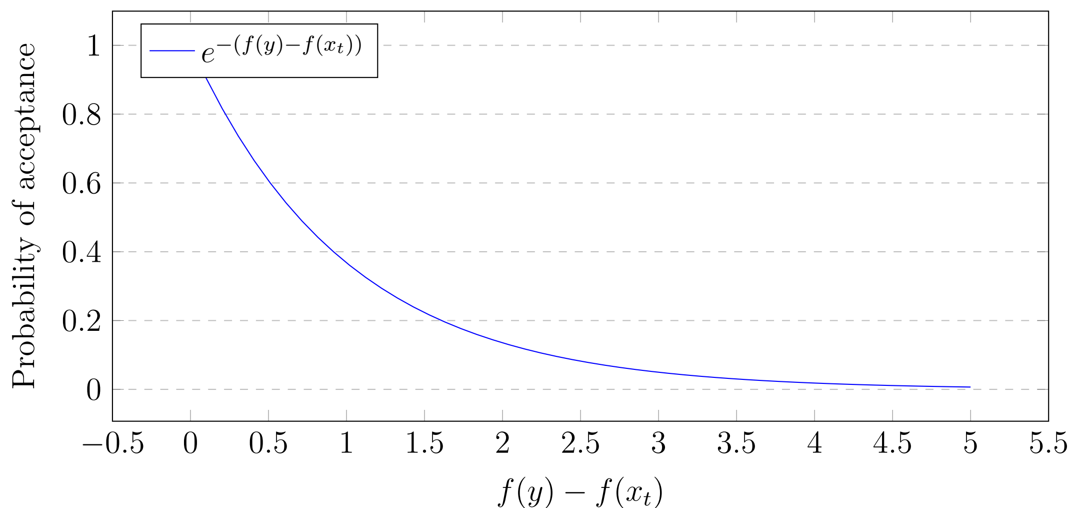


Figure 13:

Recalling that the lengths of the tours of the TSP instance differ quite a bit a temperature of 1 may seem very low if the difference in fitness between a neighbor and the current search point is much higher than 5. Still, choosing 1 is fair, since

this at least ensures that the algorithm is not moving too far astray from the good solutions at the last minute. Moreover, the difference between tour lengths are not necessarily higher just because the overall length of a tour is higher compared to another TSP instance.

2.4.2 Simulated Annealing on OneMax

In order to better reason about the performance of simulated annealing it is necessary to restrict it in some way. Simulated annealing may behave radically differently on the same problems depending on the cooling schedule chosen as discussed above. For this reason stating upper bounds for SA directly that hold in general is not always possible, because one may define a cooling schedule that makes it perform very poorly. Often the metropolis algorithm makes the analysis much simpler, so we may sometimes use this for the analysis. As stated earlier, because we can transform SA into MA by decreasing the temperature very slowly the results achieved for MA may still be very much relevant for SA as well. We start with some general results on OneMax.[12]

One may wonder if there exists a cooling schedule such that SA is able to optimize OneMax in time $o(n \log n)$. The following theorem shows that this is not possible:

Theorem 2.6. *For any cooling schedule, the expected optimization time of SA on OneMax is bounded from below by* $\Omega(n \log n)$*.[12]*

This means that SA cannot solve OneMax asymptotically faster than the (1+1) EA for example. That said, if the temperature is 0 and SA is equivalent to RLS then it will in fact be a factor e faster than the (1+1) EA.

The bound is tight if we choose the right cooling schedule for MA:

Theorem 2.7. *Let* $\varepsilon > 0$ *be a constant and* $\alpha \geq \varepsilon n$*. Then the expected optimization time of* $MA(\alpha)$ *on OneMax is bounded from above by* $O(n \log n)$*.[12]*

Recall that $\alpha(t) = 1/e^{T(t)}$. The theorem tells that if we choose α appropiately we can garuentee a similar worst case performance as the (1+1) EA. Again a temperature of 0 would also yield this bound. The same paper with the result of Theorem 2.7 also proves an interesting theorem that gives some insight into the behaviour of MA for high temperatures corresponding to α close to 1:

Theorem 2.8. *Let* $\alpha = n/\beta(n)$ *for some increasing function* β *where* $\beta(n) \to \infty$ *as* $n \to \infty$*. Then the expected optimization time of MA(*α*) on OneMax is bounded from below by* $\Omega(2^{\beta(n)/3} \cdot n/\beta(n))$*.[12]*

This theorem is in line with our intuition of higher temperatures meaning more chaotic search, which is generally not helping to optimize strictly unimodal functions like OneMax. Since the standard SA only flips a single bit in each iteration, flipping a one-bit to a zero-bit meaning going down one fitness level is not a good idea, since this only means we later need to flip a zero-bit to a one-bit only to get back to the same fitness level where we started and can then begin to try and flip more zero-bits to one-bits again.

Based on this theorem it is possible to define a phase transition from super-polynomial to polynomial optimization time bounds expressed in the following theorem:

Theorem 2.9. *The expected optimization time of MA(α) on OneMax is polynomially bounded if and only if $\alpha(n) = \Omega(n/\log n)$.[12]*

In a nutshell, this tells us that α needs to be above $\Omega(n/\log n)$ in order for the search to be efficient. Equivalently the temperature should be set sufficiently low.

2.4.3 Handling of Pseudo-Boolean Function Obstacles

Some very interesting results are known for the performance of SA on problems with some kind of obstacle in the fitness landscape. Informally two types of such obstacles in the context of pseudo-boolean functions are valleys and gaps. Take a look at the function Jump_k, an example for which was seen in figure 2 with $k = 5$. This function contains a gap between the optimum search point and the ordinary OneMax function. We already saw in section 2.3.1 that one way of solving this function with the (1+1) EA was to jump over the gap by waiting for the time that all remaining zero-bits were flipped. Because SA only makes one local change in each iteration SA is not able to make such a jump. If the temperature is not 0 it does have a positive chance of solving the problem still, however, because it is able to accept worsening in fitness. The only way for SA to optimize Jump_k is to instead of jumping over the gap, step into it, which means accepting a big worsening move. Assuming SA is run and the current search point is just before the gap it has to go one step to the right meaning a huge loss in fitness; then move further down in fitness to the right until it finally reaches the right side of the gap at which point it will accept the optimum search point. For the function f_1 we have an interesting result based on this observation given here: [12]

Theorem 2.10. *If $\alpha : \mathbb{N} \to [1; \infty[$ is (not necessarily strictly increasing) (i.e. the temperature is decreasing), then the expected optimization time of SA($\alpha(t)$) on f_1 is bounded from below by $\Omega(2^{0.257 \cdot n})$. [12]*

This theorem is concerned with the function f_1 which is like Jump_k but the gap is only one wide. We conjecture that Jump_k is even harder than this function for SA when $k > 2$.

This is a rather negative result for SA, but it makes up for this with strengths elsewhere. Look at the function f_2 instead. Contrary to Jump$_k$ the declination in fitness of the second case in the function definition is not too high; the fitness falls to a value that is relatively close to the ordinary OneMax function. For this reason we tend to denote this as a valley instead of a gap. Because the fitness in the valley is still relatively high, it is not too unlikely that SA is able to walk down the valley and up on the other side. This observation is critical for how SA is able to overcome obstacles. Furthermore, we make the observation that even if this valley is very wide, that is, if the top of the two hills is far apart, then SA may still have very good chances of going from one side of the valley to the other. For the (1+1) EA this is not the case at all; it must resort to jumps, which could take very long time if the valley is wide. Two theorems with the running times that show this performance difference are the following for the function f_2:

Theorem 2.11. *For $\alpha = n^{4/2^n}$, the expected optimization time of MA(α) on f_2 is bounded from above by $O(n^3)$.[12]*

Theorem 2.12. *The expected optimization time of the (1+1) EA on f_2 is $2^{\Omega(n)}$.[12]*

These results show an exponential optimization time difference between SA and the (1+1) EA and illustrate that accepting worsening moves can be a good strategy in some cases after all.

The results in the theorems above show that the difference in the way (1+1) EA and SA handle obstacles can lead to very different optimization times. The authors have also shown that despite this there are also functions where the (1+1) EA and SA have a very similar optimization time. Since the (1+1) EA had trouble with wide gaps and SA had trouble with deep gaps it would not be far-fetched to believe that an obstacle with gaps that are sufficiently narrow and not too deep would be something that both the (1+1) EA and SA would be able to overcome in a polynomial number of iterations. The authors define an obstacle, which they call a (k, d)-obstacle that is just that, where k and d are parameters for how wide and deep the gap is. On this function they prove that with the right relationship between k and d the (1+1) EA and SA can overcome this obstacle in polynomial time. We only state this result cursorily here without the theorem and exact details, since we will limit ourselves to only test the (1+1) EA and SA on the problems discussed above.[11]

2.5 Ant Colony Optimization

Ant Colony Optimization (ACO) is yet another nature-inspired metaheuristic, which is inspired by the foraging behavior of ants. Studies show that ants over time are able to find good solutions to the problem of determining the shortest

way to a food source from the ant colony. Despite each individual ant on its own not possessing much overview through the use of pheromone they are able to communicate with each other indirectly and together find a short path. Ants are able to lay out pheromone on their path and detect the level of pheromone already present at their location. The key mechanism is that the ants that find the shortest path to the food will be able to return home faster as well, meaning more pheromone is deposited along this path compared to paths by other ants, which the next ant starting at the ant colony will then base its decision on. More pheromone means the ant will more likely go that way. Over time and with enough ants searching, statistically the system as a whole will find a good solution.[22]

The ant colony optimization metaheuristic works by letting a number of artificial ant agents perform randomized walks on a so called construction graph, which is a completely connected graph $G_C = (C, L)$ whose nodes C are called components and the edges L are called connections. The construction graph models the problem in a way that makes it fit for a run of an ant colony optimization algorithm for which the solution can be interpreted as a solution to the original problem. The components and connections can have an associated pheromone trail and heuristic value (typically only the connections have one), which are the values influencing the choice an ant makes at a given component. The heuristic value is set based on some knowledge about the problem instance at hand and gives an estimate of how good adding the component or connection to the solution under construction really is. An example of this is in the TSP problem where an obvious choice for the heuristic value of a connection is the weight of the corresponding edge between the two cities that make up the two connected components by this connection. This information is useful, since it in the general case is a good choice to go to a city close by on a tour rather than one that is very far away and the edge to which has higher weight.[6]

The ant agents differ from real ants slightly in a few ways. One of these is that they do not generally do forward updating of pheromone meaning they typically only update pheromone on the path they have traversed when they go back again. This means they also need to have memory of exactly which path they have used to get to their destination. Contrary to a real world ant system there need not be a restriction that the ants be at the same starting position (the ant colony). In problems involving graphs, it is fine to let each ant start at a random places in the graph for instance. On a general level ant colony optimization algorithms work by doing the following steps:[6]

Algorithm 5 ACOMetaheuristicStatic

1: Set parameters, initialize pheromone trails
2: **while** stopping criteria not met **do**
3: ConstructAntSolutions
4: ApplyLocalSearch // Optional
5: UpdatePheromones
6: **end while**

Note that this is for static problems only. The local search step is optional and be seen as a kind of daemon action in that this step does not make much sense in the underlying ant colony analogy.

The `ConstructAntSolutions` steps works by each time step an ant chooses which component in its neighborhood $N(u)$ to go. The probability of the ant going to the component $v \in N(u)$ in the next time step is given by:[22]

$$p_v = \frac{[\tau_{(u,v)}]^\alpha \cdot [\eta_{(u,v)}]^\beta}{\sum_{w \in N(v)} [\tau_{(u,v)}]^\alpha \cdot [\eta_{(u,v)}]^\beta} \tag{13}$$

α and β are two parameters which determine the influence of the pheromone trail relative to the heuristic information. If β is greater than α this means more emphasis is put on the heuristic information making the algorithm more greedy as it is more likely to go to the component that is best at the current point in time and location.

After all ants have constructed a solution by means of traversing the construction graph, the pheromone trails on the connections are updated. Based on the so-called evaporation factor ρ, the pheromone on every connection is decreased by this factor. Given a connection (u, v) with pheromone value $\tau_{(u,v)}$ the pheromone becomes $(1 - \rho)\tau_{(u,v)}$ after the evaporation. In addition to the evaporation, pheromone is deposited on the connections that are part of the path the ant has taken and is increased by a value Δ_i. The specific value of Δ_i can depend on many different things about the run and also on which variant of ant colony optimization is used. The pheromone value $\tau'_{(u,v)}$ of edge (u, v) after the update becomes:[22]

$$\tau'_{(u,v)} = (1 - \rho)\tau_{(u,v)} + \sum_{i=1}^{k} \Delta_i \tag{14}$$

The idea behind the pheromone evaporation is that it helps the algorithm escape local optima and forget bad decisions previously made. Since the pheromone of a specific connection is decreased by a factor each round, we observe that it actually decreases exponentially in the number of iterations if no new pheromone is deposited.

The idea of ant colony optimization was originally proposed by Dorigo in the 90's and has since then been applied to a wide variety of problems and has been adapted to multiple different variants. An early ant colony optimization algorithm is called Ant System (AS), which has been the basis for different extensions and improvements. Initially there were three version of Ant System called ant-density, ant-quantity and ant-cycle. Ant-cycle has proved to be the best among the three, so when referring to AS one is technically referring to ant-cycle. Ant System specifies some guidelines for the way the parameters should be initialized and how the pheromone updating procedure (13) is to be carried out in practice.[6]

A very well-known and successful extension of AS is the Max-Min Ant System (MMAS), which was first described by Stützle and Hoos in 1997. It modifies AS in four main ways, which are listed below:[6]

- Only the iteration-best or the best-so-far solutions will be the subject of a pheromone deposit.
- Pheromone trail values are limited to the interval $[\tau_{min}, \tau_{max}]$, where $0 \leq \tau_{min} \leq \tau_{max} \leq 1$.
- Pheromone values are initialized to the upper pheromone trail limit.
- Pheromone trails are reinitialized each time the system approaches stagnation or no improved tour has been generated for a certain number of consecutive iterations.

These four modifications that are all quite simple have proven to have a substantial positive impact on the performance of the algorithm. Because of the success of this algorithm, we will mostly focus on investigating the performance of this algorithm in this project. Note that the four extensions stated above are not set in stone and may be changed a bit if necessary. In some papers it may be necessary to initialize the pheromone to a value different from the upper trail limit or drop the system stagnation prevention mechanism in order to prove certain results. The first two changes are the most important. [6]

In order to apply ant colony optimization to the problems in the search space of bit strings and for the TSP some things need to be detailed. This is discussed in the following.

2.5.1 Pseudo-Boolean Functions

For the problems with search points in the search space $\{0, 1\}^n$ we are interested in a construction graph in which a path corresponds to an assignment of bits. One such graph is the chain graph seen in the figure below, where the ants will start at vertex v_0.

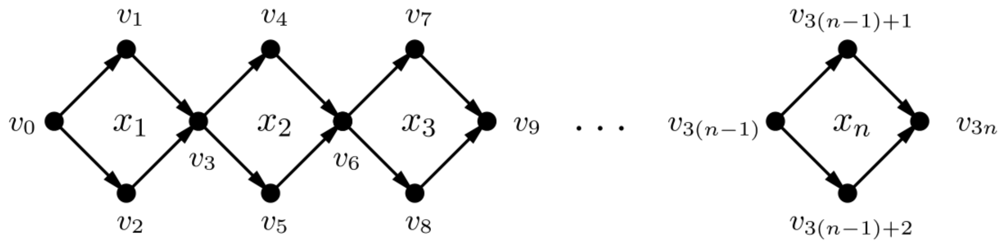


Figure 14: Construction graph for ant colony optimization on bit strings. Known as the chain graph.[21]

For each of the n bits we create two vertices with one above the other and connect these together as shown. We see that because of the direction of the edges at the nodes $v_0, v_3, v_6, \ldots$ there are two ways to get to the next node in this series and ants cannot move backwards. What we achieve is that for each of the bits the ants need to decide whether to go up or down, which will directly correspond to a decision of setting the corresponding bit to 1 or 0. Many other problems involving decisions on whether to include something or not also have a similar construction graph.[21]

Since the construction graph serves to capture that the algorithm must make n binary choices of whether to set the bits to 1 or 0 and there is no specific knowledge the algorithm has available about the problem a priori the heuristic information is not used and β is therefore set to 0 and α is simply set to 1. The probability of whether to choose an edge therefore simply becomes:

$$p_v = \frac{[\tau_{(u,v)}]}{\sum_{w \in N(v)} [\tau_{(u,v)}]} \tag{15}$$

This way the pheromone on an edge is effectively equivalent to the probability of choosing that edge.

As done in a paper by Neumann, Sudholt and Witt [21] we will restrict the pheromone on the edges to the interval $[1/n, 1 - 1/n]$. The reason for the choice of these specific values is inspired by the standard mutation probability in evolutionary algorithms, namely $p = \frac{1}{n}$ being the probability of correcting an incorrect bit. These values have also been used in other studies. Let $P(x)$ be the best-so-far or the iteration-best solution depending on which is used for pheromone deposit for MMAS. The pheromone rule (14) is then modified to be:

$$\tau'_{(u,v)} = \begin{cases} \min\{(1-\rho) \cdot \tau_{(u,v)} + \rho, 1 - 1/n\}, & \text{if (u,v)} \in P(x) \\ \max\{(1-\rho) \cdot \tau_{(u,v)}, 1/n\}, & \text{otherwise} \end{cases} \tag{16}$$

In the paper the authors list two version of MMAS, one where it it only accepts strict improvements in fitness and a variant where it also accepts equally fit solutions.

We will in this project limit ourselves to MMAS in its original form, where it only accepts strict improvements and call this MMAS. Note that in said paper this is denoted by MMAS*. We now introduce a number of theoretical results on the optimization time of the algorithm below.

First we recall the proof technique of fitness-based partitions and the fitness-level method. Although this method is mostly used in the analysis of evolutionary algorithms it can also be used to elegantly prove upper bounds for MMAS on OneMax and LeadingOnes. In a nutshell, a fitness-based partition is a partition of the search points in the search points into disjoint sets such that all elements in the same set have equal fitness; the sets are indexed with respect to their fitness level, so a set with higher index contains search points of higher fitness. Given a fitness-based partition $A_0, A_1, \ldots, A_m$ and an algorithm that does not accept worsenings in fitness then the expected optimization time is given by

$$\sum_{i=0}^{m-1} \frac{1}{s_i}, \tag{17}$$

where m is the number of fitness levels and s_i is the minimum probability of moving from a search point in fitness level $L_i, 0 \leq i < m$ to a search point in a higher fitness level $L_j, i < j \leq m$. Thus, our goal in the following will be to define an appropriate fitness-based partition for a problem, bound the probability of moving up in fitness level and find an upper bound for the sum $\sum_{i=0}^{m-1} \frac{1}{s_i}$.[21, 8]

Since MMAS keeps track of the best-so-far solution, which will never decrease in fitness during the course of running the algorithm, the fitness-level method can be used. We need to make a modification, however, to take into account the dynamic probabilities of moving from one search point to another caused by the pheromone changes. Assume the current best-so-far solution is not changed and define t^* to be the number of iterations it takes before before all pheromones have reached their borders (either τ_{min} or τ_{max}). We call t^* the *freezing time*. Assume that MMAS is currently in fitness level A_i and look at what happens during t^* iterations. We notice that there are two cases: Either MMAS found a search point of higher fitness during this time, which would mean that the best-so-far solution is updated and the algorithm advances in fitness level or no better search was found and the best-so-far solution was not updated; in the latter case, the pheromone values will all have frozen at one of the border values. Since the pheromone has now frozen the probability of constructing a better solution is static and must at least per definition be s_i. Hence, the expected time until the best-so-far fitness increases in this case is at most the time it takes for the pheromone to freeze, t^*, plus using a waiting time argument one over the probability, $\frac{1}{s_i}$, as in (17). Following Neumann, Sudholt and Witt [21] we use $\tau_{min} = 1/n$ and $\tau_{max} = 1 - 1/n$ as mentioned before and further that $\ln(1-\rho) \leq -\rho$ for $0 \leq \rho \leq 1$, which allows one to derive the

following upper bound on t^*:

$$t^* \leq \frac{ln(n)}{\rho}. \tag{18}$$

It is not difficult to show this, but we give no arguments for it here. Combining this with our bound on the probability of moving up in fitness level we arrive at the following modified bound of (17):

$$\sum_{i=0}^{m-1}(t^* + \frac{1}{s_i}) = \frac{m\ln(n)}{\rho} + \sum_{i=0}^{m-1}\frac{1}{s_i}. \tag{19}$$

Using this fitness-level method we are now ready to prove an upper bound on the expected optimization time of MMAS on OneMax:

Theorem 2.13. *The expected optimization time of MMAS on OneMax is bounded from above by $O((n\log n)/\rho)$.[21]*

Proof. Since the fitness of a search point for OneMax only depends on the number of ones, we can define a fitness-based partition of OneMax as $A_i = \{x|f(x) = i\}, 0 \leq i \leq n$. Assume the best-so-far solution is in fitness level A_i. For simplicity we may restrict our attention to the iterations where a single 0-bit is flipped to a 1-bit and all the other bits are not flipped, which is a pessimistic assumption, since the probability we derive will be smaller or equal to what it really is. Furthermore we also pessimistically assume that we always need to wait t^* iterations and that MMAS not during this time finds a better solution. This assumption ensures that the pheromone along all edges of the chain graph has frozen meaning that the edges that make up the current best-so-far solution will have pheromone value $1 - 1/n$ and the edges that are not will have a value of $1/n$. Look at a specific 0-bit in the best-so-far solution. In order to flip this bit and not any of the other bits an ant has to traverse all the edges with pheromone $1 - 1/n$ except when it reaches the vertex where the two out-going edges correspond to the choice of either setting that 0-bit to 1 (flipping it) or 0 (not flipping it) in which case it must take the edge that is not part of the best-so-far solution and therefore only has pheromone value $1/n$. Since the pheromone value of an edge is equivalent to the probability of assigning the corresponding bit to the value associated with traversing that edge, the probability of creating a new solution in an iteration where a 0-bit flips to a 1-bit while all the other bits remain the same is at least:

$$\frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \geq \frac{1}{en}$$

The inequality follows by using a well-known bound from probability theory. Since there are $n - i$ 0-bits that need to be flipped and the random events for all of these

are disjoint, the probability s_i in (19) will be given by $s_i \geq (n-i)/(en)$. Using (19) this implies that the expected optimization time becomes:

$$\frac{m\ln(n)}{\rho} + \sum_{i=0}^{n-i} \frac{1}{\frac{n-i}{en}} = \frac{m\ln(n)}{\rho} + \sum_{i=0}^{n-i} \frac{en}{n-i} = \frac{m\ln(n)}{\rho} + en\sum_{i=0}^{n-i} \frac{1}{i}$$

Since $en\sum_{i=0}^{n-i} \frac{1}{i} = O(n\log n)$, $\rho \leq 1$ and $m = n$ an upper bound on the expected optimization time of MMAS on OneMax is $O((n\log n)/\rho)$ when adding the terms in the equation above together, which completes the proof. □

The proof is inspired by Neumann, Sudholt and Witt[21]. Based on this theorem we see that the higher the evaporation factor ρ, the lower the optimization time with $\rho = 1$ giving the lowest optimization time, $O(n\log n)$. Notice that this optimization time is identical to that of the (1+1) EA on OneMax. This is not a coincidence since we make the important observation that with $\tau_{min} = 1/n$ and $\tau_{max} = 1 - 1/n$, then the pheromone has frozen for all edges in the chain graph, MMAS behaves very similarly to the (1+1) EA. With a high evaporation factor this will happen instantly meaning MMAS collapses to the (1+1) EA.

Theorem 2.13 was proved in 2009. In 2014 the bound was improved to the following, which we state here without proof:

Theorem 2.14. *The expected optimization time of MMAS on OneMax is $O(n\log n + n/\rho)$.[17]*

We now turn to LeadingOnes where we also know some bounds on the expected optimization time for MMAS. By using the fitness-level method in a way similar to the proof of Theorem 2.13 it is possible to derive the following theorem for LeadingOnes:

Theorem 2.15. *The expected optimization time of MMAS on LeadingOnes is bounded from above by $O(n^2 + (n\log n)/\rho)$.[21]*

Neumann, Sudholt and Witt improve the bound of this theorem through a proof that is more involved by showing that it is not necessary for MMAS to freeze the best-so-far solution in pheromone on every fitness level. They give two bounds that are better than the bound in Theorem 2.15 when $\rho = O(1/n)$ and when $\rho \leq n^{-(1+\Omega(1))}$, respectively. We only write the first bound in the theorem below that is better in the former case:

Theorem 2.16. *The expected optimization time of MMAS on LeadingOnes is bounded by $O(n^2 + n/\rho)$.[21]*

The authors also prove the following lower bound on the optimization time of MMAS on LeadingOnes:

Theorem 2.17. *Choosing $\rho = 1/poly(n)$, the expected optimization time of MMAS on LeadingOnes is bounded from below by $\Omega(\frac{n/\rho}{\log(2/\rho)})$.[21]*

We will in section 6.3 benchmark an implementation of MMAS with the aforementioned parameter setting and compare the empirical optimization time with the bounds in the theorems presented here.

2.5.2 TSP

For TSP instances the construction graph is simply going to be the same graph as in the problem itself. That is, the cities are the components and the edges between cities are the connections. Unlike the bit string problems the heuristic information plays a very big role for the TSP and we will be exploiting that it is generally better to go to a city close to the current one rather than one that is far away. It is important to stress that this can be seen as a kind of aid, which the (1+1) EA and simulated annealing do not get. We will keep this in mind for the analysis and comparison of the algorithm.

The parameters we are going to use for this problem will be based on the study by Dorigo and Stützle. They have carried out extensive testing and found some parameters that give good results for many different TSP instances. These values are shown in the table below: [6]

Algorithm	α	β	ρ	m	τ_0
AS	1	2 to 5	0.5	n	m/C^{nn}
MMAS	1	2 to 5	0.02	n	$1/\rho C^{nn}$

Table 2: Optimal parameter settings for AS and MMAS for the TSP

α and β are the two constants determining the relative influence of pheromone and the heuristic information. ρ is the evaporation rate. m is the number of ants. τ_0 is the initial pheromone on the edges in the construction graph. C^{nn} denotes the length of a tour generated by the nearest neighbor heuristic; that is, a a tour generated by a greedy strategy. This tour can be found quite quickly and the idea is that will help to tune the pheromone levels in a way that makes the initial explorative phase of the algorithm long enough to not converge to quickly.

For MMAS on TSP the trail limits are advised to be set to $\tau_{max} = 1/\rho C^{bs}$ and $\tau_{min} = \tau_{max} \frac{1-\sqrt[n]{0.05}}{(avg-1)\cdot\sqrt[n]{0.05}}$, where avg is the average number of different choices available to an ant at each step while constructing a solution. For simplicity we are just going to set this value equal to a fixed value of $\frac{n}{2}$. When we run tests for MMAS on TSP and refer to optimal settings we think of the settings above unless stated otherwise.[6]

2.6 Related Work

As mentioned part of the goal of this project is to develop a tool that allows for visualizing the working principles of nature-inspired metaheuristics. Other groups and people have developed similar frameworks that allow for this. One such framework is called FrEAK (abbreviation of Free Evolutionary Algorithm Kit) made by project group 427 supervised by Thomas Jansen and Ingo Wegener from University of Dortmund, Germany, in 2003 and is released under the GNU General Public License [13]. FrEAK is a a very extensive framework with a lot of functionality and features both in terms of visualization options and sample problems and algorithms.

FrEAK has been a great source of inspiration for my own project, since it showcases a lot of the things needed for my program and is also a Java application. I have primarily used this in the early stages of the project in the design of my own program and to a lesser extent with the implementation itself (e.g. for the boolean hypercube implementation). This is due to the scale of the FrEAK framework, which because it includes a lot more features also has class and functionality relationship that is more heavily interrelated than what is needed for my purpose. Further, it has been developed using the Swing graphical library instead of JavaFX, which was not available at the time, meaning all the graphic and visualization elements differ quite a bit from the way I would want to implement it. Studying the design and layout of the framework instead, however, has helped me think about how I wanted my application to look and how the user should be able to interact with the program. The details of my design are discussed in section 4.

Like in FrEAK I will be using two separate windows where the user can create a schedule and one where the user can run the algorithm and visualize the working principles. In FrEAK creating a schedule involves going through eight steps, each of which includes a list with options for a particular setting of the schedule. This way of creating a schedule is simple and intuitive, so I have chosen a similar approach, but focus only on the core parts of it, skipping steps in FrEAK like setting up the so-called population, geno-type mapper and choice of observers. A difference between FrEAK and my application is that in my case the user has to pick the search space, problem and algorithm etc. without being able to specify the specifics of these yet; this is done as part of the batch creation system instead. I am greatly inspired by the batch creation in FrEAK and have employed a similar idea and extended it. An advantage of letting the user specify the algorithm specific settings is that it allows for easily running the same algorithm on the same problem with many different settings by making each batch different.

The window in which the algorithm is run differs between FrEAK and my application in that I have allocated the left part of the screen to show a summary of the problem and algorithm chosen and statistics for the current batch; in FrEAK

this information can be accessed through a menu system instead. The controls in the bottom of the screen of my application similar those in FrEAK though, with the ability to start, pause and and change speed of running the algorithm. FrEAK uses a clever way of allowing the user to go back in time by having saved the seed of the random number generator used, which is not something I have considered looking into. The specifics of how the user wants to customize the visualization is done in the bottom menu in my application, whereas in FrEAK this was specified during the schedule creation process. Because my application is simpler and not as extensive the choice of embedding these settings into the window where the algorithm is run seems to be a good choice, since there are few enough to make accessible for the user at all times.

The problem statement, which this project is based on has also been used in past years meaning other students have made similar projects, which can be found in the database of the library at DTU. In the beginning of the project period I found and looked at a couple of other projects to primarily get an idea of what a final report could look like and what goes into a program that meets the project requirements. The student Alexander Dahl Juhl completed his bachelor thesis in 2017 based on the problem statement of this project and his report has helped serve as a good example of this [15]. Further, the experiments and benchmarks he has made can also be used as a reference point and the subject of drawing a comparison with.

An interesting aspect of his report is his results of running the (1+1) EA, simulated annealing and MMAS on the TSP instance, which he calls *simple graph* that is used in a paper by Zhou [31] and a paper by Kötzing, Neumann, Röglin and Witt [16], since I have not prioritized testing on this instance myself. Zhou has proved an upper bound of $O(n^6 + (1/\rho)n \ln n)$ for MMAS on this TSP instance for some specific parameters, most notably that β is set to 0; Alexander's results indicate that this bound holds as expected. His comparison between the (1+1) EA and SA with MMAS on this instance is also interesting when setting $\beta = 0$, since despite MMAS theoretically being able to solve this instance for optimality in polynomially time it is vastly inferior to both the (1+1) EA and SA in terms of optimization and CPU time. When setting $\beta = 1$ MMAS suddenly achieves the lowest optimization time among the three algorithms, however. I have tried to run MMAS with 50000 function evaluations allotted on berlin52 and also observe that the effectiveness of MMAS is radically lowered compared to run it with $\beta = 2$ (see figure 44); further, compared to the (1+1) EA and SA run with the same amount of functions evaluations the fitness of the solutions it achieves is significantly worse (refer to figure 28 and 35).

Alexander has also benchmarked the (1+1) EA, SA and MMAS on the berlin52 TSP instance as I have done in section 6. His results show that SA using a cooling

schedule like the one presented in section 2.4.1 of this report is the best performing algorithm among the three. His results for SA are very similar to the results I get when running SA for 5,000,000 iterations (see table 5 for my results). He must have defined what the constant c is in the cooling schedule differently though, since he uses the constant $c = 0.5$, which is very different from mine even though the algorithms achieve the same results. He has tried to run MMAS for the same amount of iterations as well with only a very low number of runs to average over, which I chose not to do since like his implementation found MMAS to run much more slowly than the (1+1) EA and SA. His results show that MMAS is not as efficient on berlin52 as SA is. His results differ from mine in this regard, since I found MMAS to be able to optimize berlin52 in all cases given enough iterations. I suspect that this is because of the difference in parameter setting, which shows that setting the parameters right in MMAS can make a very big difference in performance. The choice of parameters in his test is inspired by the analysis in the paper by Zhou [31] and the paper by Kötzing, Neumann, Röglin and Witt [16]. The parameters chosen here allowed the authors to prove some theoretical results on the optimization time of MMAS on some special TSP instances. I conjecture that these parameters will therefore not be as efficient in practice for TSP instances with radically different layout like berlin52, compared to parameters determined by extensive empirical testing, which is the case of the parameter settings found in table 2 attained by Dorigo and Stützle [6]. If Alexander had used these parameters instead he would most likely also have achieved more encouraging results for MMAS on berlin52.

3 Analysis

Looking at the requirements of the project it is clear that is important to make a framework with a good coherent structure. In order to make it flexible, modular and extensible the structure calls for a high level of procedural decomposition and decoupling. Furthermore, the class structure should be made as general as possible and make use of abstract classes and interfaces where applicable. This will make it possible to adapt the different algorithms more easily as well as extending the program with more types of problems and algorithms.

The minimum requirements demand that the program incorporate the search spaces bit strings and permutations and within these search spaces, the problems OneMax, LeadingOnes and TSP. Both of these search spaces are easily defined by their dimension and an array of bits and integers, respectively, though integers can be used to cover both cases. A permutation further stipulates that the integers be unique. Having implemented the basic support for problems within these search spaces it is possible to add many different problems to the program, since a lot of

problems exist in one of these two search spaces as mentioned earlier. To avoid the loss of focus it seems like it is best to focus on pseudo-boolean functions and the TSP only.

The program must make it possible to visualize the constructed solutions for first and foremost the TSP instances. One could choose to make a very simplistic graphic framework that does not include animation or the ability for the user to specify settings, but this seems too minimalistic. The nature of the project elicits the need for designing and implementing a functioning graphical user interface (GUI) that makes it possible to choose among the available problems and algorithms and incorporates dynamic visualization functionality, so the algorithms can be visualized while they run. The visualization will also make it possible to get better insight into the working principles of the algorithm and information about how the algorithm is constructing the solution. Apart from better satisfying the goal and aim of the project this will also make it much easier to evaluate and test the algorithms. Although it is not a requirement that the user have an easy way to specify the settings for the algorithms he or she wishes to run, implementing a basic interface for this also seems like a good idea; compared to the alternatives of for example doing this through the console and/or launch parameters, a graphical user interface for doing this has many merits. Because there are multiple different problems and algorithms featured in this project this will make it easier to get an overview of the possibilities. Moreover, because each of the algorithms can be tweaked in a lot of ways a user interface could allow for easily specifying how a given algorithm should be set for a particular run. Care must be taken, however, to strive for a satisfying delimitation. Otherwise the interface could get cluttered by the many options and it may also be cumbersome to implement all options.

In order to be able to evaluate and measure the performance of the different algorithms it is important that the framework records and extracts the different key figures from a run of metaheuristic. These should be either printed to the console, written to a file or displayed on the screen. The most prominent figures will be the number of function evaluations and the fitness of the best-so-far solution an algorithm achieves. Because the number of function evaluations is proportional to the number of iterations for a lot of the algorithms this may also be a viable alternative to show. The number of function evaluations and iterations is from a theoretical point of view of greater interest than the CPU time (i.e. how much actual time has it taken for the algorithm to generate a final solution), so this together with the fitness ought to always be displayed. The option to see the CPU time should also be implemented, however, but need not be displayed using the graphical display. It makes sense to disable the graphics when testing the CPU time to avoid the graphics having too much of an impact on the performance. Moreover, it does not make much sense to keep track of the CPU time if the user

changes the speed of the algorithm or pauses it at times. The number of function evaluations is not dependent on this. More problem and algorithmic specific figures can be implemented and displayed if there is more time available.

4 Design

4.1 Program Structure

To make the program flexible as discussed in the analysis of the requirements in section 3, the program has been organized and structured in a way that aims for a great level of abstraction. For this reason object instances are to a large extent up casted to a super class, so the various problems and algorithm can work with each other and the specifics be applied through the use of polymorphism. This way program can a structure with a good degree of decoupling, flexibility and and less code redundancy.

The program classes are primarily organized in three packages, `algorithms`, `graphics` and `problems`. The algorithm package includes a package for each of the three types of algorithms investigated in this project. It also includes a number of auxiliary classes which includes a set of operator classes. Each type of operator has its own class. This makes it possible to let a given algorithm such as simulated annealing work with different operators that may be parsed as a parameter at instantiation. The algorithms are made in a version that handles bit string problems and one that handles the TSP problem. It has been necessary to do this to take the problem specifics into account, but most of the non-problem specific parts are kept in a superclass that these algorithms inherit from. For SA for example all the logic is placed in a super class and the two classes `SABitStrings` and `SATSP` for bit string problems and TSP, respectively, only specify the optimal parameter settings and how to fetch a search point from the neighborhood of the current search point. The `graphics` package includes the FXML files and controller classes for the `CreateScheduleScene` and the `RunScene`, that are used in the graphical user interface to let the user create a schedule and run the algorithm, respectively. Refer to section 4.2 and 5.2 for more details on this. The `problems` package includes the different problems and classes representing the search points.

A class diagram showing the class hierarchy of the most important algorithm classes of the program is seen in figure 15.

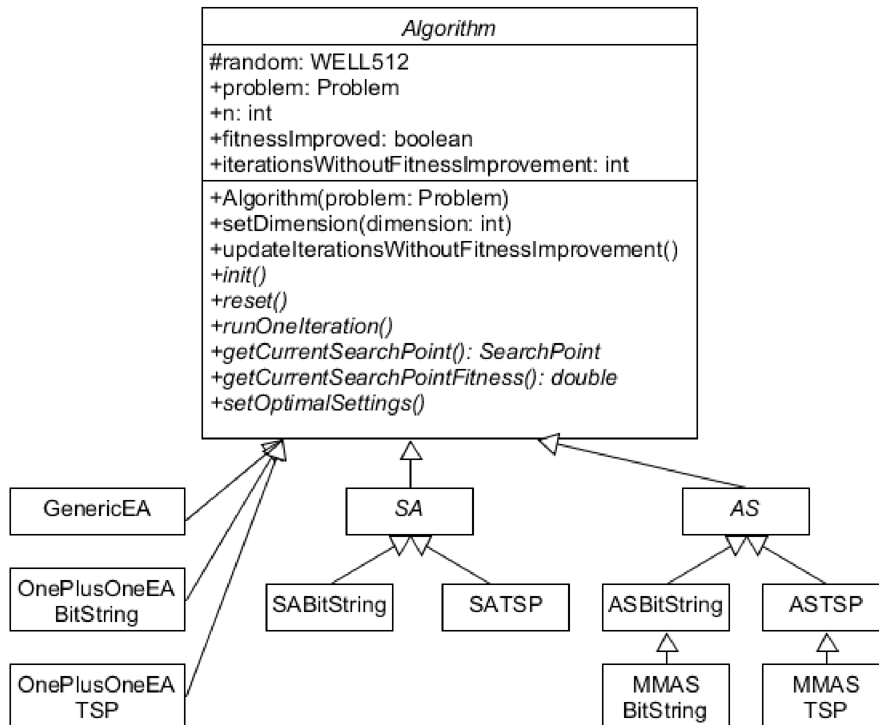


Figure 15: Class diagram of the algorithm classes. The program called UMLet has been used to draw the diagram.

This figure shows that the algorithms are primarily separated into three different groups, evolutionary, simulated annealing and ant colony optimization. It has been necessary to make a separate class for the two search spaces, since they entail a few vital differences. For the (1+1) EA which is simple and short it was not deemed worthwhile to make an abstract class like for SA and MMAS that contains all the common functionality. For SA the only difference between `SABitString` and `SATSP` is the generation of a neighbor and the optimal parameter settings. The rest of the functionality is in the abstract class `SA`.

4.2 Graphics

4.2.1 Program Flow

The graphic part of the application consists of two main windows: The create schedule window and the run window each of which are presented below. We will in this section quickly go through the menu for creating a schedule and the window where a schedule is run and show what the user can choose and explain some of

the ideas behind the aspects of the design. For a user guide that describes how the program is used in more detail, please refer to section 10.2. Since the windows are denoted as scenes within JavaFX terminology we will take the liberty to denote the windows as scenes in the following.

When the program is started the user is presented with the create schedule menu. The menu consists of five steps, each of which allows the user to specify some part of the schedule he or she wishes to run. The navigation is done using the buttons in the lower part of the screen. If clicking a button has no effect it will be rendered with lower opacity to indicate this, which should make the navigation more intuitive. The current step is shown in the left-hand side of the screen with text that is underlined and bold. When the user has completed all five steps the *finish button* can be clicked, which will exit the menu and present the user with the run schedule scene. At each step when the user clicks the *next button*, the available options in the next step are updated to match the previously selected settings e.g. the pseudo-boolean problems are only shown if the user selected the search space bit strings in the previous step. On the right hand side of the screen a text field is shown that contains some information about the currently selected search space, problem and algorithm. It will update automatically when a new item is selected. The design choice of including this short description was made to make each step of creating the schedule a bit more rich in content and to add a simple way to tell the user what the problems and algorithms are. Figure 4.2.1 shows a screenshot of the menu at the first step where the user has selected the *permutation* search space option. The five steps are listed below. Refer to section 10.2 for information about which problems and algorithms are available.

1. Search Space: Initially the user chooses the search space.

2. Problem: The available pseudo-boolean functions are shown if the user chose the search space bit strings in the previous step and the traveling salesperson problem if the permutations option was chosen.

3. Algorithm: At this step the user can choose among the algorithms in the program in a table as well as the operators to be used for that algorithm is applicable. Note that for the algorithms (1+1) EA, RLS and SA the mutation operator is always set to be *bit flip* for bit strings and *2-OPT* for TSP regardless of what the user chooses as mutation operator in the table.

4. Stopping Criteria: At this step the user can select between five stopping criteria by toggling them on and off with a checkbox and write a value for them in an adjacent input text field. The stopping criteria can all be combined with each other. By default the stopping criterion *optimum found* is selected meaning the algorithm stops as soon as the current best solution is

the optimum; this is often a setting the user wants to enable, so the choice to enable it by default was made. Note that the program does not sanitize the input, that is, no check is made for the input to be valid. A valid is in this context an integer. This means that if the user for example writes a literal and hits enter it will cause a `NumberFormatException`. The user is still able to type in a valid value afterwards though and the program can still proceed normally. The choice of not checking if the input is valid was made reasoning that the user knows that it does not make sense to type in invalid input in the text fields and the time it will take to sanitize the input could instead be invested elsewhere in the program.

5. Batches: At the final step the user can create batches of runs. A batch is represented as a table entry where the values in the cells can be changed. The values that make up a batch that the user can change depend on which problem and algorithm has been chosen. If the user does not want to configure the settings himself/herself, (s)he has the option to check the checkbox that says *use optimal settings*, which will make the program calculate the optimal parameters based on what the theoretical suggestions discussed in 2 for the chosen problem. Note that for SA this cannot be done optimally not knowing for how long it will be run, so a default cooling schedule is set. The program can be run without the GUI in the console instead if the other checkbox is checked. This option is primarily used for generating data for the benchmarks in section 6. Figure 4.2.1 shows a screenshot of the menu at the batch creation step assuming the user has selected the search space permutations, problem TSP and algorithm MMAS. By having the possibility to create batches with different settings it gives the opportunity to try out the algorithm in different ways more easily instead of having to go back to the menu to change the settings each time. Like for the text fields for the stopping criteria no check for valid input is made. Valid input for the various fields are integers and doubles.

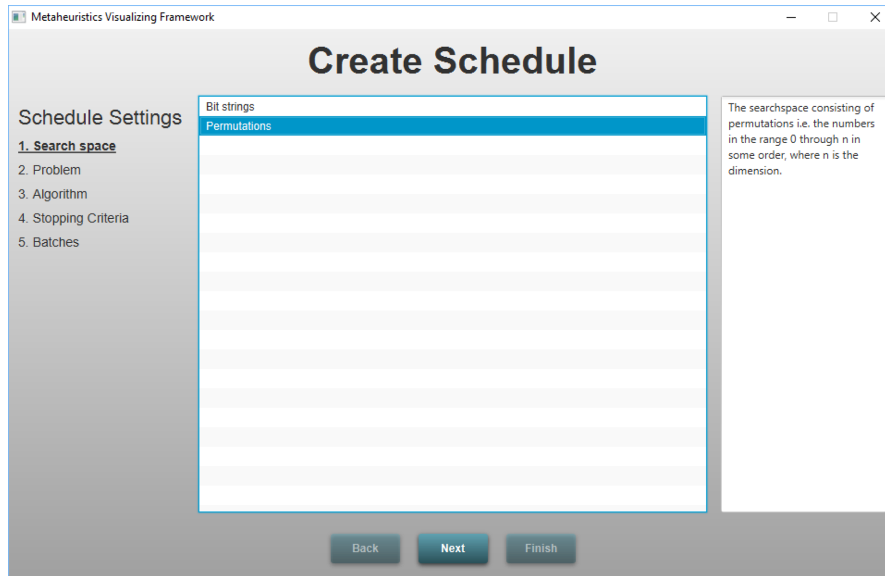


Figure 16: Step 1

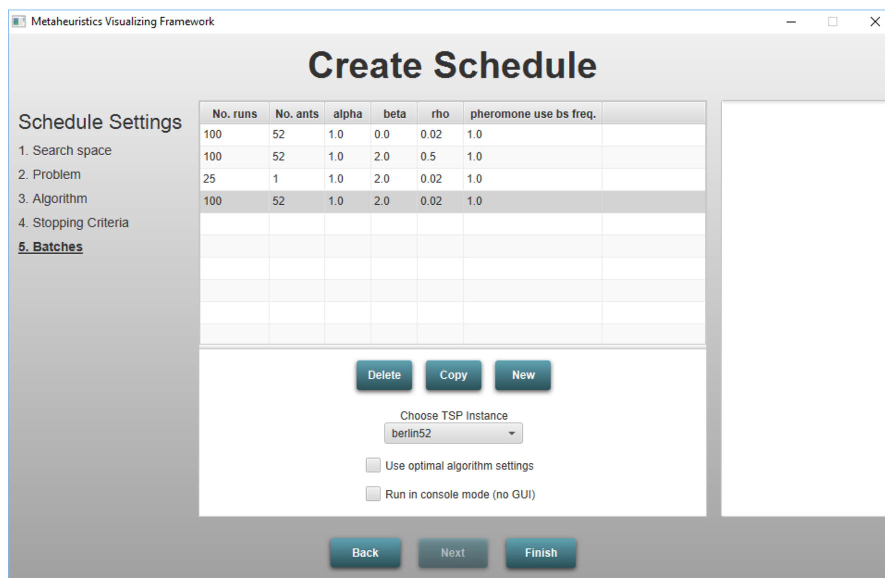


Figure 17: Step 5

After the user has created a schedule and pressed the *finish button*, the run scene is presented. This scene consists of three main areas containing controls and settings, an information panel and animation panel, respectively. Assuming the user has chosen the same settings as the ones used in the screenshots above the screen will look like what is shown in figure 4.2.1.

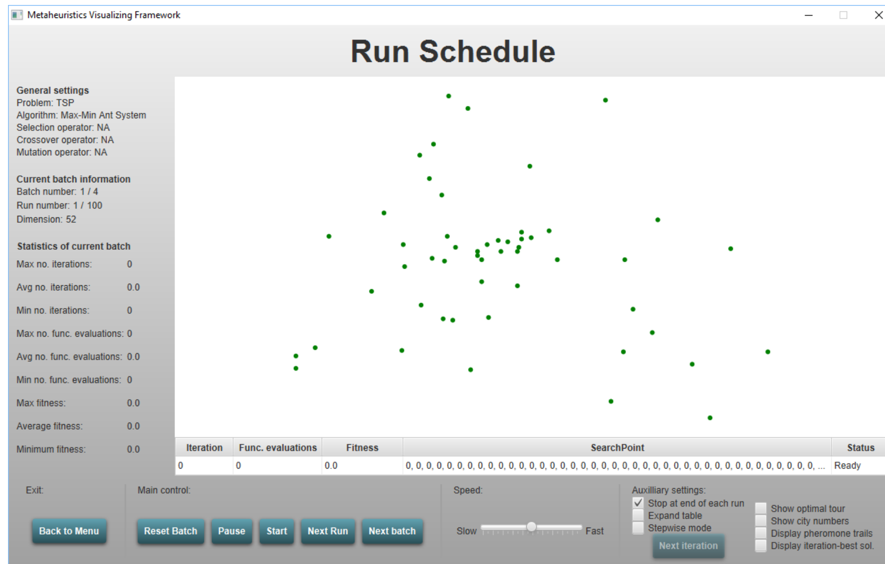


Figure 18: The run schedule scene as it is initially shown before the algorithm is started. For this example the TSP was chosen as problem and the Max-Min ant system as algorithm.

The controls are placed at the bottom and can be accessed at any time, both before an during the algorithm is running. To make it more organized they have been separated into four groups based on their functionality. The user has the ability to go back to the menu and create a different schedule and is able to start, pause and reset the algorithm at any time. The lower right hand side of the window consists of a slider for setting the speed of the algorithm and for toggling various display settings on and off (e.g. whether the pheromone trails should be displayed while running an ant colony optimization algorithm on a TSP instance or not).

The information panel is shown on the left-hand side of the screen. This includes a summary of the problem, algorithms and operators chosen and an overview of the current batch and run number. It also includes some of the most important statistics of running the program like the maximum, minimum and average number of iterations and fitness, respectively, which are updated each time a run of the algorithm completes. If the schedule is run in console mode these statistics will be prompted in the console as well as the average CPU time and the number of times the optimum was found.

The visualization area is in the middle of the screen and consists of a table and a canvas. The table has intentionally been resized such that only one row is visible. This is because most of the algorithms only maintains a single search point or it is only interesting to see the search point that is the best so far solution. If need be it is possible to expand the table by toggling the checkbox *expand table* on and

off, which is found under the auxiliary controls the in the bottom right corner of the window. The visualization of the solution construction for the algorithms is primarily done using the canvas. Depending on if the search space is bit strings or permutations either a boolean hypercube or a graph is used, each of which is described in the next two sections.

4.2.2 Boolean Hypercube

The boolean hypercube is a way to visualize the bit string search space. It does this by calculating the position of a search point based on the number and positions of 1-bits. The number of one bits determines the y-coordinate and the positions of the bits determine the x-coordinate. Ideally the boolean hypercube should assign a unique x-coordinate for each layout of the ones in a search point. However, an issue is that the number of ways the one bits can be positioned increases exponentially with the problem size: Given n, the number of different search points with $\frac{n}{2}$ bits is $\binom{n}{n/2}$. Note that the distribution is symmetric though, so going above $n/2$ would mean the possibilities decrease again. It would require too much CPU time to calculate for a specific search point what position it has within the ordering of the possible search points with the same amount of 1-bits as one would have to go through the possibilities until the right search point is found to find the position with respect to the ordering; and because there are so many possibilities this takes too much time.

A compromise has therefore been taken and is done instead using the same idea seen in the FrEAK framework [13]. Given a search point x with k 1-bits of size n, the sum of indices is calculated as $\sum_{i=0}^{n} i \cdot x_i$. This way the left-most bits account for less than the right most bits. This calculation takes linear time in the problem size, which is faster than time in the order of the number of permutations we would otherwise have had to go through. It is, however, at the expense of different search points potentially having the same value e.g. 00110 and 01001 both have the value 5. Having calculated this value, the minimum and maximum values for k 1-bits are calculated in order to determine a range in which the value of x lies. This is done by exploiting that we know that if the bits are all positioned as far to the left as possible it will give the smallest value possible and likewise as far to the right as possible for the greatest value. The difference between the greatest and lowest value then give the length of the range. Next, the position of x within this range is determined. After this only the necessary scaling is required to find the actual x-coordinate on the screen where the search point should be rendered. The y-coordinate on the screen is simpler to calculate as the number of 1-bits determines the position in the range of $[0, n]$ and then scaling can be done. In order to properly scale the search points so they lie within a suitable area a function is used to mimic the increase in the number of different search point values

as n increases (up to $n/2$). The function $e^{-\frac{x^2}{8}}$ is used, but this could have been any function that is symmetric and convex. This function was chosen even though the slope decreases as the top point is reached, which goes against the fact that the number of different search points increase exponentially with n as mentioned before. It is easier to see the different search points though with this function so it was used nonetheless. We remark that in the FrEAK framework they use a sine function, which does not increase exponentially around the middle either. The function is used to encompass an area by drawing the graph turned 90 and 270 degrees and put together. Based on this enclosed area the scaling of the search point coordinates is done such that the points fit within the area with the highest possible value attaining the position of the right border. The figures below show how the boolean hypercube looks in the program with a single search point and 20 search points, respectively. The color of the search point is calculated based on its fitness relative to the fitness of the optimum with a transition from the color blue to red. In the left figure the search point is close to the optimum and is therefore very red in color while the search points to the right have a fitness that is about half of the optimum fitness so they are both blue and red in color, which makes the color purple. Notice by the way that on the left figure the algorithm has been run for a while and as expected from figure 3 the search point is positioned along the left border, where the fitness is highest for the number of one-bits it currently has. The optimum is shown with a green mark.

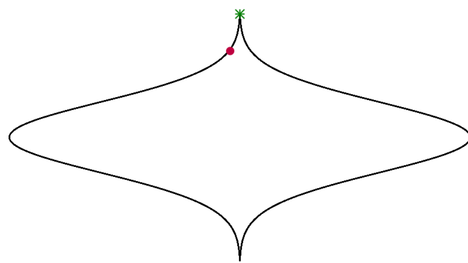


Figure 19: (1+1) EA on LeadingOnes, $n = 100$

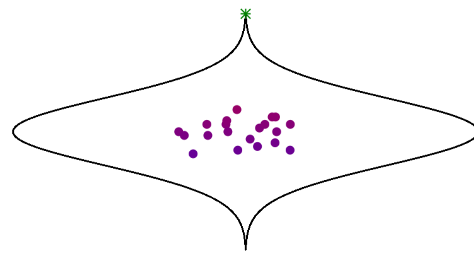


Figure 20: GenericEA on OneMax, $n = 64$, population size 20.

4.2.3 Graph for TSP Instances

For TSP instances the cities are laid out based on the coordinates read from the TSP instance file, but scaled scaled such that they fit within the canvas while still keeping their relative distances to each other the same. Since the graph is a complete graph i.e. there is an edge between every pair of cities, the choice of not showing the edges was made, which makes it is much easer to see the tour

generated by the algorithms, especially for large TSP instances. No information is lost by this, since we know that edges between cities exist. The tours generated by the algorithms have an orientation, but it was not deemed important to show this, so currently the only way to see the orientation is by looking at the actual tour in the table below the canvas if need be. On top of the graph various other visualization options can be applied if applicable for the algorithm in quesiton. For MMAS the user has the option to enable the rendering of the iteration-best solution and pheromone along the edges. Figure 4.2.3 shows MMAS run on berlin52 where the pheromone trails are shown with orange. The opacity of the pheromone lines shows how much pheromone is on the associated edge relative to the highest value of pheromone deposited.

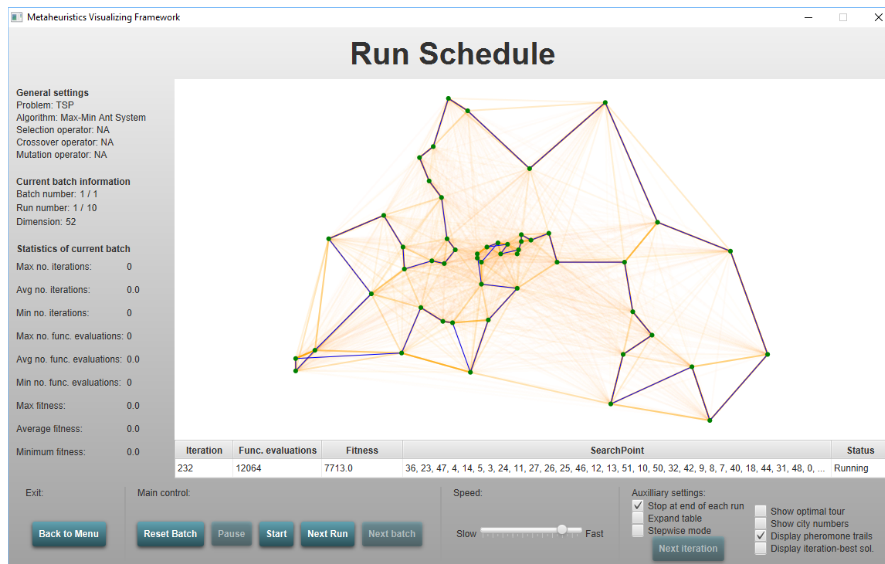


Figure 21: A run of MMAS with optimal parameters after 12000 function evaluations with pheromone trails shown.

4.3 Testing

The primary method of testing the program has been manual tests even though there are many disadvantages to this strategy. Heavy use of automatic tests and paradigms like extreme programming and test driven development can greatly reduce the risk of bugs in the program and allows the programmer to change things in the program without having to do a lot of manual testing again to verify that the change did not cause an error. The choice of not doing more automatic testing nonetheless is that many of the algorithms are randomized, which makes automatic tests for them more involved and time consuming. We note that it is possible

to make the algorithms run deterministically by choosing a specific seed for the random number generator, but one would still have to study what choices the algorithm makes with this seed in order to write tests for the expected behavior and performance of the algorithm. Some automatic tests (Unit tests) have been made using JUnit though still. The program features a test for all the problems implemented, some of the most important operators and for selected subroutines of the algorithms, primarily for MMAS. These tests are not extensive so it has been necessary to do manual testing of them as well.[26]

5 Implementation

5.1 Algorithm Specifics

As was touched upon earlier, because the algorithms are relatively simple in how they work, most of the algorithms have been implemented in a straightforward fashion. Moreover, utilizing the abstract classes of `Problem`, `SearchPoint`, `Algorithm` and the different kinds of operators much of the implementation is very similar to the pseudo code for the algorithm. The fact that this is the case in practice supports the claim that this is one of the advantages of metaheuristics. To illustrate how similar the pseudo code is to that of the actual Java code of an algorithm a code snippet of the main part of the `SA` class is shown here:

```
public void runOneIteration() {
  for (int i = 0; i < roundsPerIteration; i++) {
    SearchPoint neighbor = generateNeighbor(current);
    double neighborFitness = problem.fitness(neighbor);

    double delta = neighborFitness - currentFitness;
    delta *= problem.isMinimizationProblem ? -1 : 1; // flip the sign
        ↪ of delta to make the algorithm work with minimization
        ↪ problems
    if (delta > 0) {
      current = neighbor;
      currentFitness = neighborFitness;
      fitnessImproved = true;
    } else {
      double probability = getAcceptanceProbability(delta);
      double r = random.nextDouble();
      if (r <= probability) {
        current = neighbor;
        currentFitness = neighborFitness;
      }
    }
  }
```

```
    updateTemperature();
    updateIterationsWithoutFitnessImprovement();
}
```

The code is based on the pseudo code of Algorithm 4 [1] with some inspiration from [6] as well (e.g. the idea of making it able to do more than one round per iteration if enabled). The `generateNeighbor` method mutates the current `SearchPoint` using an instance of the type `MutationOperator` and returns either a `BitStringSearchPoint` or `PermutationSearchPoint`. The `updateTemperature` method decreases the temperature by a predefined factor. In order to make this method handle both maximization and minimization problems the sign of `delta` is flipped before checking if it is above 0 if it is a minimization problem. The code for the `getAcceptanceProbability` method was kept in separate method to ease reading the code and is implemented as follows:

```
public double getAcceptanceProbability(double delta) {
  if (calculateProbabilityWithRespectToAlpha) {
    return Math.pow(alpha, delta);
  }

  if (temperature <= 0) return 0;
  return Math.exp(delta / temperature);
}
```

which either returns the probability based on the usual calculation or with respect to α, which is practical for testing the algorithm in some cases in order to use values given by the theoretical foundation. Recall that $\alpha = e^{-1/T(t)}$.

Generally the code follows very closely the pseudo code and the places where they differ it is mostly because of the explicit code for e.g. the exponential function and for generating a random number that determines if a search point with lower fitness is accepted or not. One common property for all the algorithms implemented that tends to make the code a bit less readable is the inclusion of lines like `fitnessImproved = true` and `updateIterationsWithoutFitnessImprovement`, which are part of the functionality for one of the stopping criteria in the example above. At the expense of making the code less readable such lines have been deemed necessary in order to make the program adjustable by the user in various ways. It is also for this reason the class `AlgorithmRunner` at first glance seems quite verbose.

We now look at some of the details of the method `asDecisionRule` that decides which city to go to next in the tour, which is a critical part of the ant colony optimization algorithms AS and MMAS. The method is included in the `AntAgent` classes with the reason that it is the individual ants that need to make the choice and not the system. It can potentially also make it easier in the future to run the algorithm concurrently. For bit strings the method is very simple, since traversal

of the chain graph can be done in practice by letting each ant decide for each bit whether it should be set to 1 or 0; it is unnecessary to make the ants traverse an actual graph with vertex and edge objects. Nevertheless, the variable names are inspired by analogy of the chain graph with *up* meaning setting the bit to 1 and going *down* meaning setting it to 0. The code is shown below:

```
public void asDecisionRule(int step) {
  double selectUp = antSystemBitString.choiceInfo[2*step];
  double selectDown = antSystemBitString.choiceInfo[2*step+1];
  double sumProbabilities = selectUp + selectDown;

  double r = randomDoubleInInterval(0.0, sumProbabilities);
  if (r <= selectUp) {
    searchPoint.values[step] = 1;
  } else {
    searchPoint.values[step] = 0;
  }
}
```

The `step` passed as a parameter to the method gives the index of the current bit to set. In order to determine if the ant should go up or down in this step, a random double is drawn in an interval set by the probabilities of going up and down, which are equal to the utility of each edge stored in the `choiceInfo` matrix. For bit strings this is just the pheromone values of the edges, since $\beta = 0$.

The idea of using the probabilities to define an interval is a major part of the implementation of the `asDecisionRule` for the TSP. For ant colony optimization on the TSP there are more than two choices of where to go, so the probabilities define a kind of "circular roulette wheel", where the random number drawn between 0 and the sum of the probabilities will land in one of the probability intervals of the roulette. The implementation is inspired by Dorigo and Stützle who have shown how to implement the method in C-style notation [6]. The code for `asDecisionRule` for TSP is shown below.

```
public void asDecisionRule(int step) {
  int currentCity = searchPoint.values[step];
  double sumProbabilites = 0.0;
  double[] selectionProbability = new double[nn]; // probability for
      ↪ selecting each city

  // Assign probabilities for each city (corresponding to the
      ↪ incident vertex of an edge) based on pheromone and heuritic
      ↪ value
  for (int i = 0; i < nn; i++) {
    if (visited[nearestNeighbors[currentCity][i]]) {
      selectionProbability[i] = 0.0;
    } else {
```

```
            selectionProbability[i] = antSystemTSP.choiceInfo[currentCity][
                ↪ nearestNeighbors[currentCity][i]];
            sumProbabilites += selectionProbability[i];
        }
    }

    // If all nearest-neighbors have already been visited (
        ↪ sumProbabilities will be equal to 0) then choose the best
        ↪ next
    // Otherwise pick the city among the nearest-neighbors at random
    if (AuxTools.areDoublesEqual(sumProbabilites, 0.0)) {
        chooseBestNext(step);
    } else {
        // Randomly choose one of the cities based on their respective
            ↪ probabilities
        // This is done in a "circular roulette wheel" fashion
        double r = randomDoubleInInterval(0.0, sumProbabilites);
        int counter = 0;
        double probability = selectionProbability[counter];
        while (probability < r) {
            counter++;
            probability += selectionProbability[counter];
        }
        // Go to the city drawn
        searchPoint.values[step+1] = nearestNeighbors[currentCity][
            ↪ counter];
        visited[nearestNeighbors[currentCity][counter]] = true;
    }
}
```

In short, the method first assigns probabilities to the choice of each edge based on the edge utilities given by the `choiceInfo` matrix. It is important to stress that only edges to the `nn` closest cities are assigned probabilities, where `nn` is the nearest neighbor depth, that is the number of cities that will be considered in the `asDecisionRule` method. `nn` will be equal to n though for $n \leq 500$. It then chooses among these edges as just described. If all closest cities have been previously visited, however, which is the case when the probability sum is 0, it invokes the method `chooseBestNext(int step)`, which goes through all cities and finds the one closest. The `chooseBestNext` method will not be necessary when $nn = n$ though.

5.2 Graphics

The user interface is developed using JavaFX, which was introduced in Java version 8 and meant to replace Swing as the standard built-in graphic utility framework. JavaFX allows for creating graphical user interfaces using FXML, which is a

markup language based on XML. Using this the the graphics of the framework were implemented without having to create class instances, make method calls and set field values using conventional Java code syntax. Since a lot of the user interface is going to consist of static objects anyway, specifying it using FXML makes it much easier to manage. A popular tool called *Scene Builder* (the version compiled by Gluon has been used in this project), which can be integrated into different IDEs (Eclipse included) was used to make the user interface using a WYSIWYG (what you see is what you get): Elements of the user interface were dragged and dropped on an initially empty application place holder; after designing the interface the FXML code for this could be generated automatically. It is worth mentioning that writing the code in FXML rather than the standard Java code syntax does not make the final product different from one another, since the underlying scene graph that make up the graphics is the same. For this reason it is also possible to combine elements written using FXML and standard JavaFX syntax, which has been done in the controller classes where it made sense.[23]

The graphics written in FXML have been integrated with the model is through a number of controller classes. For each scene (the create schedule scene and the run scene) a controller class takes care of updating the view and handling user input from the GUI. This is to make it in accordance with the principles of the model-view-controller structuring of programs.

5.2.1 Animation

The animation is primarily done through the canvas object in the scene graph. Some updating of the labels on the left hand side is done after a new run or batch is initiated. The table in the middle of the screen under the canvas is also updated every time a change happens.

The animation is done using an `AnimationTimer` object, which is part of the `animation` package of JavaFX. The animation timer calls the `render` method of the `AlgorithmRunner` class in every frame and is run on the Java application thread. It is important that the animation in the canvas is only done by the animation timer and not on other threads than the Java application thread, since the updating of the canvas may not be consistent otherwise. The `render` method of the `AlgorithmRunner` class will call the `render` method in the `BooleanHypercube` class or the `GraphDrawer` class depending on if the search space is bit strings or permutations. It will also update the contents of the table.

A major part of the program of concern is the synchronization between the main JavaFX application thread and the thread controlling the `AlgorithmRunner` instance. To avoid potential race conditions the JavaFX application thread, after it has initialized the `AlgorithmRunner`, will stall and wait for the `AlgorithmRunner` to finish setting up everything before it starts the animation timer. The `synchronization`

keyword is used extensively throughout the AlgorithmRunner class, since the AlgorithmRunner thread needs to be able to wait on different wait conditions and react to actions by the user. For example, if the program is paused the algorithm runner has to wait for the user to press the start button before continuing. Different fields are used as guards and the system is protected against *spurious wake-ups* by using while-loops over said conditions. The notifyAll() calls can be made without concern of performance loss when many threads that are not relevant are awakened, since the program only has two threads running at a time (the main application thread and the AlgorithmRunner thread).[23]

5.3 Statistics

The program needs to in some way record the results of running a given meta-heuristic. First and foremost the key figures should be extracted and shown to the user of the program in some way. As mentioned earlier the program has a table and an information panel where some of this is shown to the user during the run of an algorithm.

The two classes RunTableData and AlgorithmRunData are used to handle the connection between the model and the view for the data in the table below the canvas and the statistics shown in the left-hand side of the screen, respectively. Instead of primitive types the data is defined as either an instance of the IntegerProperty, DoubleProperty or StringProperty classes part of JavaFX. This allows for a so-called binding of the data to an associated label part of the view and JavaFX will automatically make sure that the view is updated accordingly when the values of the data fields change. To update the contents of the table this functionality is also used. Since the data is bound to a label it must be updated by the Java application thread to work properly, however. Updating the statistics therefore has to be done using the following piece of code:

```
Platform.runLater(() -> {
  if (currentRun >= totalNumberOfRuns) runController.
      ↪ updateButtonStylingNextRun(true); // Make the NextRun button
      ↪ visually disabled
  runController.algorithmRunData.updateStatistics(iteration,
      ↪ numberOfFunctionEvaluations, fitness);
});
```

By doing the updating using Platform.runLater the Java application will take note of this pending action and act accordingly when it can. The code snippet below shows how the max, min and average iterations fields of an AlgorithmRunData instance, which are instances of IntegerProperty are bound to the corresponding labels.

```
statMaxIterations.textProperty().bind(algorithmRunData.
    ↪ getMaxIterations().asString());
statAvgIterations.textProperty().bind(algorithmRunData.
    ↪ getAvgIterations().asString());
statMinIterations.textProperty().bind(algorithmRunData.
    ↪ getMinIterations().asString());
```

This is done for all relevant fields. Because there are quite a few fields this part of the code is unfortunately a bit verbose, but having grouped the binding declarations together in the code it is not a major concern.

To record the results of a run of a metaheuristic more easily a secondary class `AlgorithmRunnerConsole` is made. By running the algorithms without the GUI it can run them as as fast as possible without the rendering impeding the speed. The results will also be written to a file rather than only be displayed on the screen. After each batch, the class invokes the method `storeBatchResultData(Batch batch)` in the `AlgorithmRunData` class with the current batch, which then writes the relevant statistic fields to a `csv` file. The `csv` file can then be used to draw the graphs seen in section 6.

Because the `AlgorithmRunnerConsole` class does not make use of the GUI, the class also allows for setting up custom batches, stopping criteria and algorithm settings quickly without needing to type it in through the GUI. Moreover, some settings that can not be changed in the batch creation step using the GUI can be tweaked in here. Most notably it allows different values for a stopping criteria for each batch which is not possible if the GUI is used to create the batches as of this moment, since this would require a major and more complicated restructuring of the stopping criteria and batch creation system. The possibility of setting the values in the stopping criteria to different values for the different batches has been used to generate the data in section 6.[23]

5.4 Optimizations

During the implementation of the algorithms and other parts of the program it has been clearly evident that many optimizations can be made. The most noteworthy of these optimizations are described in the following section.

5.4.1 (1+1) EA

If the (1+1) EA for bit string problems is implemented naively based on its description it would involve doing n random experiments at each iteration to test if each bit should be flipped or not. From a theoretical point of view this is not a major concern, since the fitness function evaluation is assumed to take the majority of the time and therefore limiting the amount of times the fitness function is

evaluated is the primary objective if the running time is to be lowered. However, clearly the running time with respect to actual CPU time is of concern as well and doing n random experiments is indeed rather costly in this regard. A way to mitigate this is to observe that the (1+1) EA usually flips far fewer bits than n; given a mutation probability p of flipping a bit, the algorithm will on average flip only $n \cdot p$ bits on average each iteration amounting to just a single bit flip on average each iteration for the standard mutation probability $p = \frac{1}{n}$. Based on this observation Jansen and Zarges described in a paper [14] how to randomly draw the position of the next mutation site (i.e. which bit should be flipped next) rather than n random experiments. The geometric distribution gives the probability of getting the first success in a series of Bernoulli trials in the the k'th trial. Checking if a bit should be flip is a Bernoulli trial so by using the geometric distribution to draw how many times we would have had to check if a bit should be flipped we get the index of the first bit that should be flipped immediately. We can then afterwards do the same thing for the rest of the bits. With inspiration from Jansen and Zarges we can generate the next mutation site based on a random draw of a double between 0 and 1 using the following function:

```
private int getNextPosition() {
  double randomNumber = random.nextDouble();
  int nextGeometric = (int) Math.floor(Math.log(randomNumber) / Math.
      ↪ log(1.0 - probabilityOfSuccess));
  return nextGeometric;
}
```

Using this function the index of the first bit that is to be flipped is calculated. Assuming a bit was flipped, the next bit to be flipped is computed by making a new random mutation draw using the function again. This new function value is added to the index of the last bit that was flipped. One is added as well so the same bit cannot be flipped more than once. If at any point the bit to be flipped is beyond the length of the bit string then the loop breaks. The loop is shown be below:

```
int currentIndex = getNextPosition();
while (currentIndex < n) {
  mutationBitFlip.flipBit(y.values, currentIndex);
  currentIndex = currentIndex + 1 + getNextPosition();
}
```

For large n this implementation should provide a substantial speed up in the CPU time. In order to test the effect of this optimization the (1+1) EA has also been implemented in the simple standard form in the class `OnePlusOneEASimplistic`.

To test the performance of the two implementation it has been run on OneMax and the results are shown below:

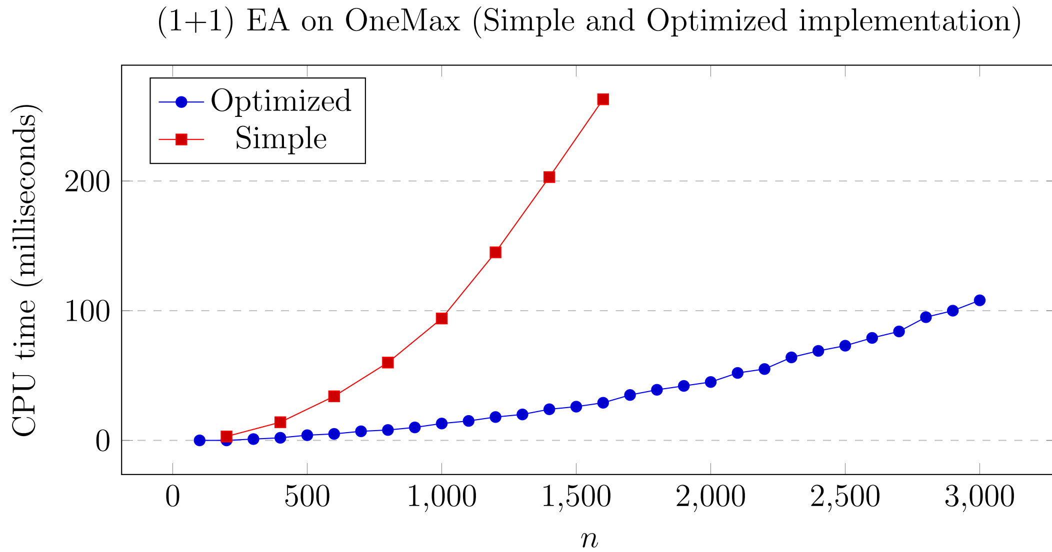


Figure 22: Averaged over 100 runs. Lower is better.

As we can see from this test, the optimized implementation is much faster than the simple version in practice. For OneMax it is not surprising that the implementation plays a heavy role for the CPU time, since the fitness function is not too expensive to evaluate. The specifications for the system the test was made on can be found in section 6.

5.4.2 Random Number Generation

Because all the algorithms in this project make use of randomization, a significant improvement in CPU time could potentially be gained if the generation of random numbers is done more efficiently. Java already has a built-in random number generator (RNG), but there are many alternatives to this available that may have better performance. On such alternative is called *Well Equidistributed Long-period Linear* (WELL), which is based on the successful random number generation scheme Mersanne Twister.[18, 19]

The performance of a random number generator can be said to be twofold: It must generate numbers in a way that approximates a truly random distribution. Further, it must be able to do so using a short amount of CPU time. We will not investigate the degree of randomness for WELL compared to the Java built-in RNG reasoning that both of these are good enough in this regard for our purpose and refer to the documentation of WELL [18, 19] for details on this. We will instead make a quick test to see how the RNGs compare with respect to CPU time: For each RNG we make 10^9 random trials calling by invoking the `nextDouble`

method, which generates a double in the range $[0, 1]$. The library for WELL features different variants with different periods, which concern their randomness degree. We disregard the importance of this and restrict our attention to the CPU time and choose to test instances of the classes `WELL512` and `WELL1024`. For the Java built-in RNG we test a simple instance of the `Random` class. We save the current system time using `System.currentTimeMillis()` before and after all the random trials and calculate the time it took based on this. This is done for all three objects separately. The results are shown in table 3. We see that in our test the `WELL512` instance was the fastest and only spent 16 % as much time as the built-in Java RNG, which is a considerable improvement. Based on this test the choice was therefore made to use the WELL502 class for random number generation.

RNG	Time (ms)
Built-in of Java	19691
WELL512	3234
WELL1024	3689

Table 3: CPU time used to complete 10^9 invocation of the `nextDouble| method.`

Please refer to the sources [18, 19] for further information about the library and the terms of usage that apply.

5.4.3 Ant Colony Optimization

An easy optimization to the ant colony optimization algorithms is based on the observation that the value $[\tau_{(u,v)}]^\alpha \cdot [\eta_{u,v)}]^\beta$ in equation (13) needs to be computed by all ants time could therefore be saved by avoid recomputing it for each of these. To this end, the matrix `choiceInfo` stores this value, which is then only updated during evaporation and depositing of pheromone. More precisely `choiceInfo[i][j]` stores the value of $[t_{ij}]^\alpha[\eta_{ij}]^\beta$.[6]

Although not costly with respect to the number of fitness function evaluations, deciding which city to go to next in AS or MMAS for TSP can take a lot of CPU time when the number of cities is high. To mitigate this Dorigo and Stützle suggest that instead of looking all possible cities, only look at the *nn* closest, where *nn* is the nearest neighbor depth set to a constant value. By using nearest neighbors the time to find the next city to visit is greatly minimized for large instances. Instead of looking at $O(n)$ possibilities it only looks at a constant amount. The main drawback of this is that is limits the tours that can be generated and could even potentially make it impossible for the algorithm to find the optimal solution. In most practical cases for large instances, however, it will be a substantial improvement of the algorithm.[6]

For small TSP instances the benefit of using the nearest-neighbor heuristic in the `acDecisionRule` is not as great, so the default value of nn is set to 500 as of the time of writing. This means that all cities will be considered if the number of cities is 500 or below for the instance in question.

6 Evaluation and Results

In order to study the performance of each of the algorithms on the different problems and how they compare, each algorithm will first be tested individually with different parameter settings; based on these results the algorithms will be compared with the parameter setting that gave the best performance. The problems that we will test the algorithms on will first and foremost be OneMax, LeadingOnes and the TSP in this order. Some of the algorithms will also be run on a few of the other problems from section 2.2.1 as well, since this may highlight some other aspects of interest.

The primary measure of computational cost when optimizing the problems will be the number of fitness function evaluations, which from a theoretical perspective is the operation in the algorithms that takes the longest and the number of invocations of this should therefore be kept as low as possible. Closely tied to this is the number of iterations, which is also worth considering when studying the performance. Most of the algorithms including the (1+1) EA and SA will only make a single fitness function evaluation each iteration, so for these algorithms the number of iterations and number of function evaluations are equivalent. MMAS on the other hand will make one function evaluation per ant agent to check the fitness of the tour it has constructed. The number of function evaluations and iterations are still related, however, since they are proportional, but it can make it misleading to look at the number of iterations in this case. Because MMAS makes more function evaluations per iteration the theoretical cost of an iteration is higher than for e.g. the (1+1) EA and SA, which is important too note. When analyzing the optimization time of the algorithms we will primarily be looking at the number of iterations it takes for the algorithm to find the optimum in the cases where the number of function evaluations is the same, since the notion of iterations seems easier to think about. For the experiments for MMAS we have to analyze it with respect to the function evaluations though.

From a practical perspective the CPU time, that is, the number of real world seconds used, is the cost that should be minimized as much as possible. Theory says that this is correlated with the number of function evaluations, but there may be subroutines in some algorithms that take a long time in practice, which would not be evident if we only looked at the number of function evaluations. Another thing is that for the problems considered in this project, evaluating the fitness

function is not much more time consuming that the other parts of some of the algorithms e.g. updating of pheromone in MMAS, which makes the CPU of the algorithms have a impact than just the number of function evaluations. We will therefore for OneMax, LeadingOnes and the TSP also benchmark the algorithms with respect to their CPU time. The system that the benchmarks will be made on uses an Intel i5-4690K CPU with 4 cores and a cache size of 6 MB and is running at a stock clock speed of 3.5 GHz; for the memory 16 GB of HyperX FURY DDR3 RAM with a speed of 1866 MHz are used. Other specifications of the system are not mentioned here, since they should not have too much of an influence on the benchmarks.

For the pseudo-boolean functions, OneMax and LeadingOnes, we will primarily analyze the optimization time by generating plots showing the number of iterations or function evaluations it takes for the algorithms to find the optimum as a function of the dimension n of the search space. In the following sections for all the plots where it says *iterations* along the y-axis this is to be understood as the number iterations used to find the optimum. For the TSP instances we will instead plot the fitness of the solution constructed by the algorithms after it has run a certain number of iterations or invoked the fitness function a certain amount of times. This is because the algorithms may not find the optimum within a reasonable amount of time, so we are instead interested in the quality of the approximations they generate. The way we will generate the data points will be done by creating a number of custom batches with the same parameter settings where only n (for pseudo-boolean functions) or the number of function evaluations allotted (for the TSP) will be different. Doing it this way allows us to use the already implemented functionality of the statistics of batches described in section 5.3 and the stopping criteria of the program. Instead of having to implement a new way to keep track of the performance of the algorithms while they run we just store the statistics of the current batch, which are updated each time a stopping criteria is met. A drawback of this approach is that it takes longer to complete the tests, since the algorithms are restarted for each data point.

We will carry out the benchmarks in two ways. For each algorithm we will run a series of tests where we stop the algorithms after a relatively small amount of function evaluations, which will make the testing faster and make it easier to test many different parameter settings that should give hints as to what works well. After this we will run a test where the algorithm is allowed to make many function evaluations; we will try with both 1,000,000 and 5,000,000 function evaluations allotted. For MMAS preliminary testing show that CPU time for MMAS is too high for it to be feasible to run the algorithm with this number of function evaluations unless the algorithm is allowed to stop if it finds the optimum before then; this is also documented in the testing to come (see e.g. figure 55). Fortunately it turns

out that because MMAS is very effective on e.g. the berlin52 TSP instance this is not a problem, since it will on average not do a high number of function evaluations anyway, which will become evident by the results in figure 6.4.4 in section 6.4.4.

Lastly we stress that when comparing the optimization time of the algorithms for the pseudo-boolean functions with the expected optimization time of the theorems in section 2 we must choose some values for the hidden constants and logarithmic bases hidden by the asymptotic notation in the bounds. Unless there is a good reason not to, we will primarily decide to use simple constants with value one and logarithmic base two or e. The implication of this is that the expected optimization lines drawn will not be entirely accurate. We will therefore have to keep this in mind when making observations and conclusions based on the comparison of the empirical data with the expected.

6.1 Evolutionary Algorithms

6.1.1 OneMax

We start off with the optimization time of the (1+1) EA on OneMax with mutation probability $p = 1/n$. Corollary 2.1 tells us that the expected optimization time is $en\ln(n) + O(n)$. To compare the empirical data with the expected we will disregard the lower-order terms in this expression and draw a line for the function $en\ln(n)$. We will also try to run RLS on OneMax, which we based on the paper by Doerr [5] expect to be a factor e faster. The figure below shows the optimization time for said algorithms, that is, the average number of iterations it took the (1+1) EA and RLS to find the optimum as a function of the dimension n. The expected optimization time for OneMax and RLS are shown as lines with a olive green and a green color, respectively.

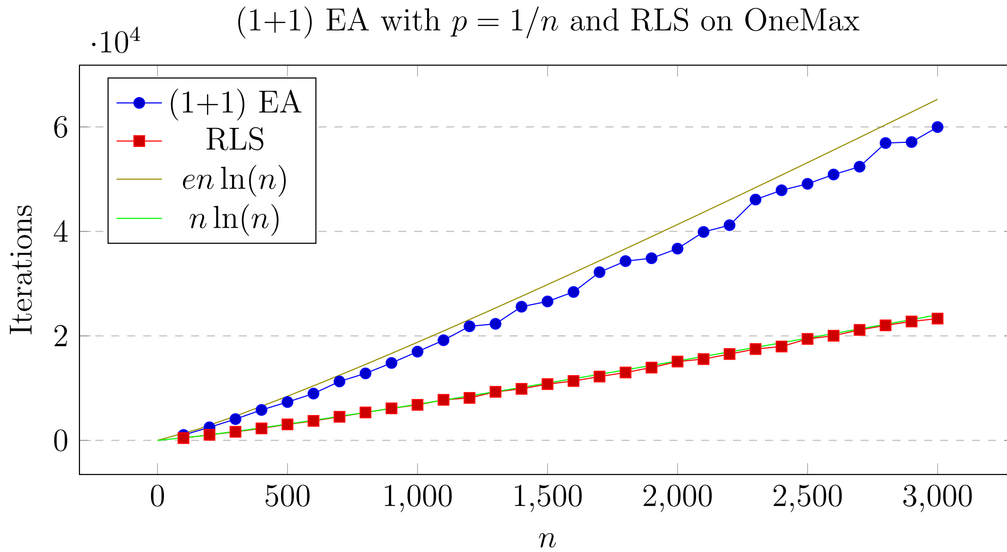


Figure 23: Averaged over 100 runs

The results seem to be as expected and the optimization time is quite close to the expected upper bounds when disregarding the lower-order terms. The optimization time seems to increase almost linearly, which is not surprising since the factor $\ln(n)$ increases very slowly compared to the factor n and the values of n used in this test are relatively high.

Comparing the (1+1) EA with RLS we see that the latter algorithm is indeed faster at optimizing OneMax in this test as expected. Furthermore, the difference in optimization time also seems to be about factor e faster, although slightly less in this test. This is seen by the line for $n \ln(n)$ being very close to data points for RLS while the line for $en \ln(n)$ is not too far above the data points for the (1+1) EA.

The figure above was using the standard mutation probability of $p = 1/n$. According to Theorem 2.1 this mutation probability should be optimal for all linear functions which includes OneMax. In order to test this theoretical result we now try running the (1+1) EA on OneMax again with different mutation probabilities close to $p = 1/n$ and see how this affects the optimization time (see figure 24).

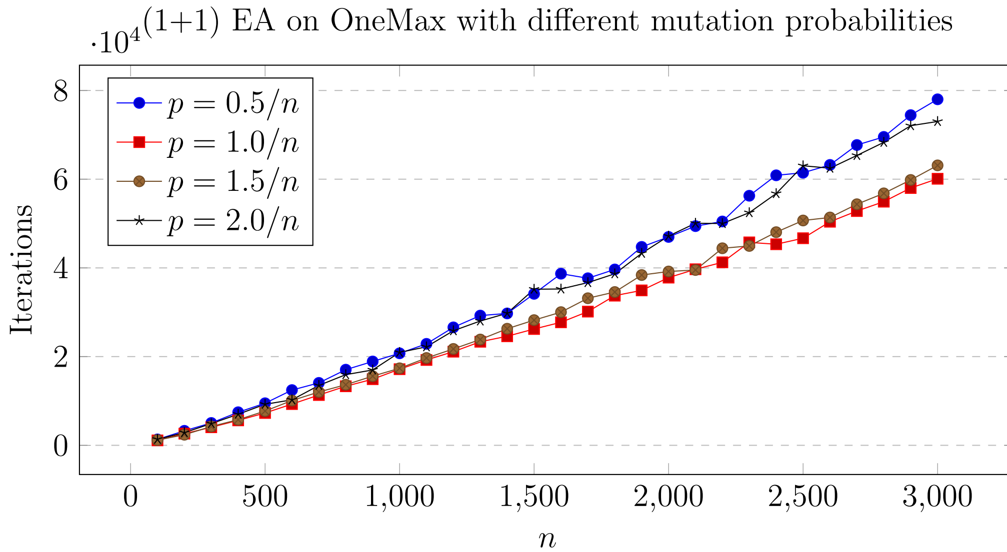


Figure 24: Averaged over 50 runs

As expected a mutation probability of $p = 1/n$ gives the lowest optimization time among the values tried. Increasing it slightly by a factor of 0.5 gives an optimization time taht is quite close though. An interesting observation is that decreasing and increasing the standard mutation probability by a factor two seems to give a similar optimization time seen by the data points being very close for these two values of p.

6.1.2 LeadingOnes

The expected optimization time of the (1+1) EA on LeadingOnes given by equation (11) contained no O-notation like we had for OneMax, so we expect to see a much smaller difference between the expected optimization time and the empirical data for this problem. Figure 25 shows the results of running the (1+1) EA with the default mutation probability of $p = 1/n$ and using the optimum value for LeadingOnes specifically found earlier in section 2.3.1 of $p = 1.5936/n$.

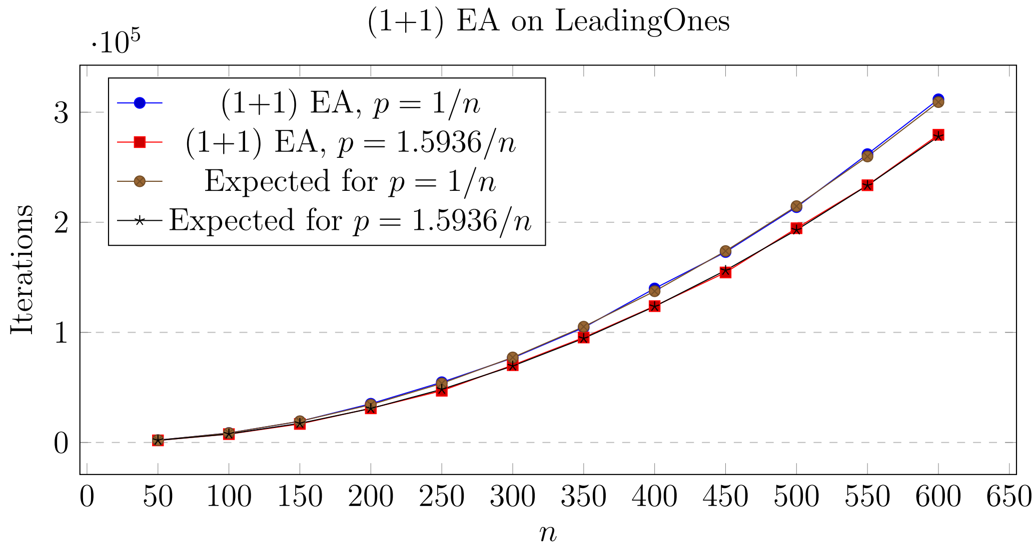


Figure 25: Averaged over 100 runs.

The results show that the expected value indeed is a very precise estimate of the empirical for both mutation probabilities, since the expected lines and the data lie directly on top of each other. Looking at the data point for $n = 600$ for the graph of $p = 1/n$ it took 311725 iterations and the expected number is calculated to be 309183. Thus, the difference is only 0.82 %, which is incredibly precise given the inaccuracies in the test. We also notice that a mutation probability of $p = 1.5936/n$ gives a lower optimization time compared to using $p = 1/n$ as expected. The standard mutation probability still seems to give a fair result though, which demonstrates the versatility of this mutation probability

6.1.3 BinVal

Since BinVal is linear function, Theorem 2.1 applies, which we will test now. Since a `double` in Java is represented by 32 bits, we cannot differentiate the fitness between search points if $n > 32$ and the search point is close to the optimum. We will therefore only carry out the test for very low values of n. The results are seen in figure 26.

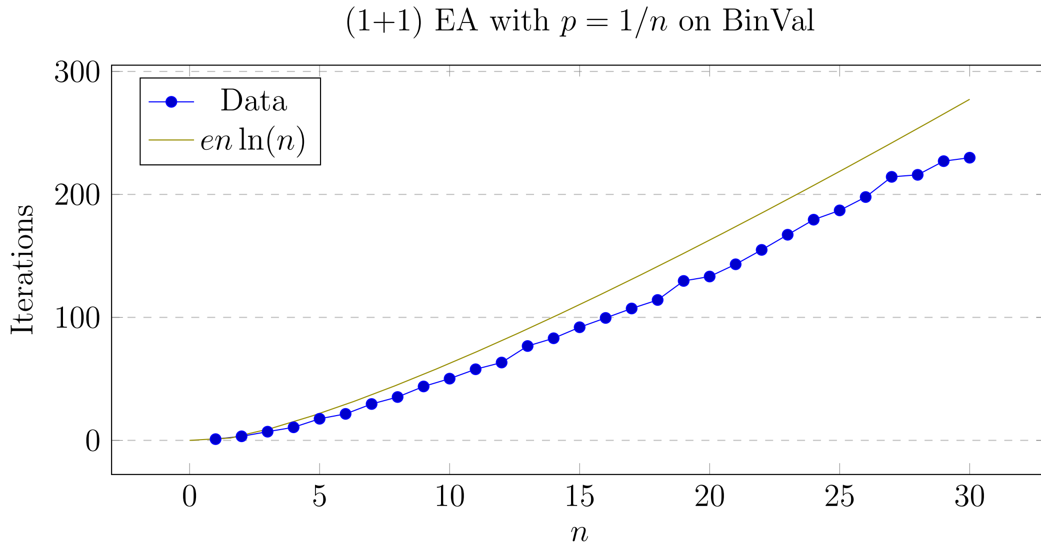


Figure 26: Averaged over 500 runs

The results of the test seem to agree with upper bound of Theorem 2.1. We also notice that the optimization time seems to be very close to the one for the (1+1) EA on OneMax as the Theorem tells us.

6.1.4 TSP

For the TSP we start off by trying to run the GenericEA with a few different settings. As touched upon the beginning of section 6 when testing the performance on the TSP we will show the fitness as a function of the number of function evaluations the algorithm has used. Note that for the GenericEA the number of function evaluations is not the same as the number of iterations. The results are shown below:

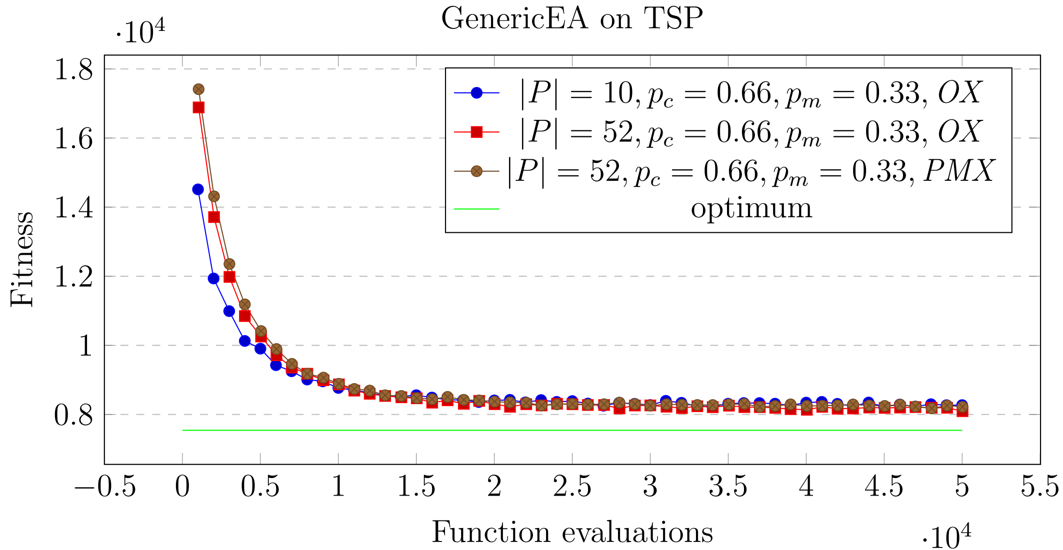


Figure 27: Averaged over 50 runs. p_c is the crossover probability and p_m is the mutation probability. Tournament selection with draw size $k = 10$. 2-OPT mutation. Crossover type is shown in the legends.

We observe that all three parameter settings give a solution with the same convergence behavior, which is somewhat surprising. It seems like increasing the population size from 10 to 52 does not have a visible effect in this test; we may need to run the algorithm on a bigger instance with a bigger difference in population in order to see a performance difference on this parameter. Even though the crossover operators OX and PMX work in different ways they both seem to generate solutions of the same fitness. We suspect that other operators may perform better, especially GPX should outperform these crossover operators by a fair margin.

It seems like the algorithm with all three parameter settings approaches stagnation quite quickly seen by the fitness not decreasing by much after about 15,000 function evaluations. This leads us to believe that the parameter settings used is perhaps not as good as it could be. We will not investigate this further, however, since the GenericEA is not the main focus in this project.

To run the (1+1) EA on TSP we will use the algorithm exactly as seen in Algorithm 3. One could change λ to a value different from 1 and analyze the effects, which indirectly corresponds to changing the mutation probability p for the pseudo-boolean functions. Reasoning that $p = 1/n$ was a good choice for linear pseudo-boolean functions and to a certain extent LeadingOnes, we will limit ourselves to only use the algorithm for TSP with $\lambda = 1$, though. The figure below shows the (1+1) EA on the berlin52 TSP instance with whiskers for each data point indicating the minimum and maximum fitness achieved while running the

algorithm 100 times for the data point in question; the data points themselves show the average fitness as before for the other tests. The optimum value of 7542 is shown as a green line.

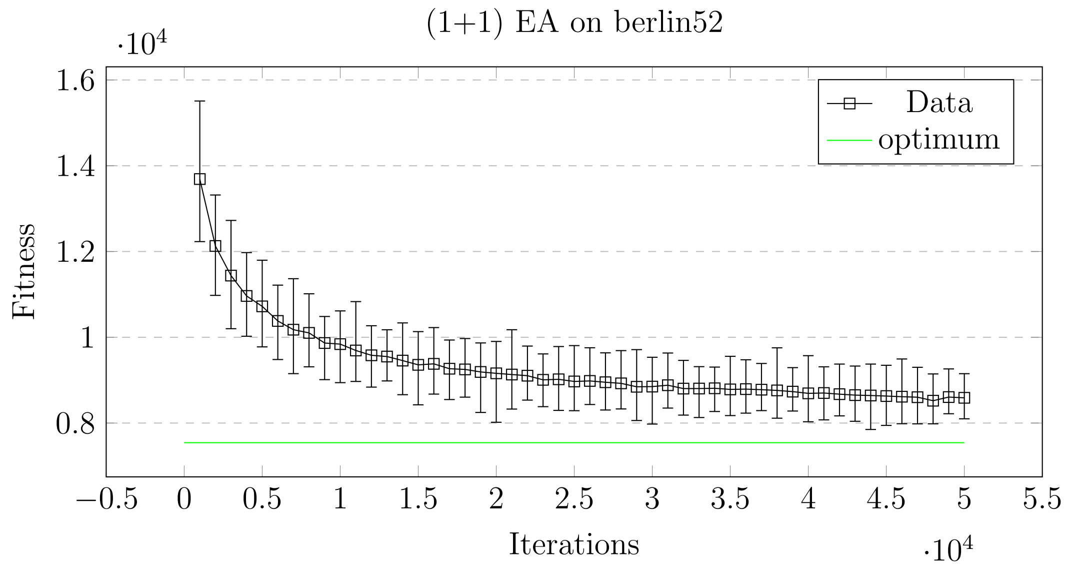


Figure 28: Averaged over 100 runs.

We see that the algorithm does not find the optimum within 50000 iterations during any of the runs, since the minimum whiskers are all above the optimum line. The average fitness clearly improves though. After 50000 runs the average fitness is 8628, which is much better than an average fitness of 13717 that it achieved with 1000 iterations allotted.

We see that the average fitness after 50000 iterations are not as good as the results for the GenericEA after 50000 function evaluations. This is not surprising considering the (1+1) EA is much simpler. Contrary to the GenericEA though it looks like the (1+1) EA is still capable of finding solutions with even better fitness if it is run for more iterations. It will therefore be interesting to look at the test with a high number of iterations. The table below shows the performance of the (1+1) EA for a very high number of iterations.

runs	100	100
iterations	1,000,000	5,000,000
avg CPU time	367	1836
max fitness	8399.0	8369.0
avg fitness	7928.92	7851.27
min fitness	7542.0	7542.0
no. optimum found	2	19

Table 4: (1+1) EA on berlin52

Indeed, the fitness of the solutions does decrease and goes below 8000 at 1,000,000 iterations. Furthermore, the algorithm manages to find the optimum in two out of the 100 runs after this many iterations and 19 times after 5,000,000 iterations. We observe that doing 5,000,000 iterations took around 1.8 seconds on average. Since this test suggests that the optimum is found in about every fifth run on average we expect that we would have to run the algorithm for 5,000,000 iterations five times to find the optimum once. Thus, on average we only expect to wait approximately 10 seconds. This is an incredible result considering the simplicity of the algorithm which implementation wise is not much more complex than implementing a brute-force search, which we conjecture would use far more time to find the optimum.

6.2 Simulated Annealing

6.2.1 OneMax

There are many possible cooling schedules for simulated annealing that we could try out. For OneMax though we already know that a temperature of 0 gives the best result, so we expect that the cooling schedule with starting temperature 0 corresponding to RLS gives the fastest optimization time. To also compare this to some other cooling schedules we will be trying out the schedule with $T(1) = n$ and $r = 1 - 1/n$. Note that we start with temperature $T(1) = 1$ rather than n^3, which we said in section 2.4.1 would be a good starting temperature in general. The reason for this is that on OneMax starting with $T(1) = n^3$ clearly only makes the algorithm much worse, since there are no obstacles where the temperature can help. We have also chosen the constant $c = 1$ to make it cool fast. We expect that the effect of this is that it will quickly collapse to RLS and only then start to make proper progress towards the optimum. We will also try out the metropolis algorithm for the α being $1 \cdot n$ and $0.5 \cdot n$, which according to Theorem 2.7 should be bounded from above by $O(n \log n)$. The results are shown in figure 29 below.

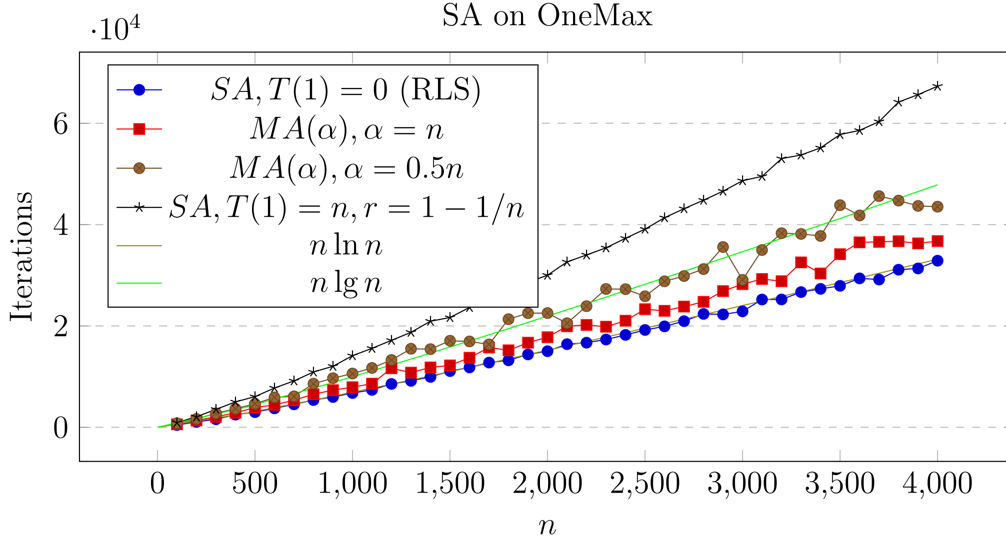


Figure 29: Averaged over 50 runs.

As expected $T(1) = 0$ gives the lowest optimization time of the different cooling schedules. Further, observe that the function $n \ln n$ is almost spot on with respect to the empirical data of this function. Since the line of the function $n \ln n$ was just as close for the RLS algorithm in figure 23, this supports our proposition of $T(1) = 0$ making SA identical to RLS.

The upper bound from Theorem 2.7 for MA(α) seems to hold as it is upper bounded by $n \lg n$ and follows the same asymptotic pattern. We observe that the data points for the optimization time of $MA(\alpha), \alpha = 0.5n$ have higher variance than for $MA(\alpha), \alpha = n$. This is in line with out intuition of lower alpha corresponding to higher temperatures makes the search more random.

Lastly looking at SA with relatively high starting temperature n and fast cooling, it is asymptotically very similar to the others and as expected has a higher optimization time than the one with starting temperature 0 (RLS). We see that the increase in optimization time is less chaotic than for MA, which we suspect is because the algorithm in the beginning will be very chaotic and not make much progress towards finding the optimum, but will eventually collapse to RLS at which point it will very quickly find the optimum. The chaotic search initially will be too short for it to make the data points deviate too much from each other like it does for e.g. MA with $\alpha = 0.5n$. Although SA with $T(1) = n, r = 1 - 1/n$ will eventually collapse to RLS the optimization time of MA for both values of α chosen in this test is lower. This suggests that SA is not able to reach a low enough temperature quickly enough before MA is able to find the optimum.

In continuation on the effect of different values of α, Theorem 2.9 states that

if alpha is too low then the function is no longer polynomially bounded and the optimization time will be exponential. To verify this phase transition we run now MA with $\alpha = \frac{n}{\lg n}$ and $\alpha = \frac{n}{\lg(n)^2}$ and compare the optimization time of the two versions of the algorithm.

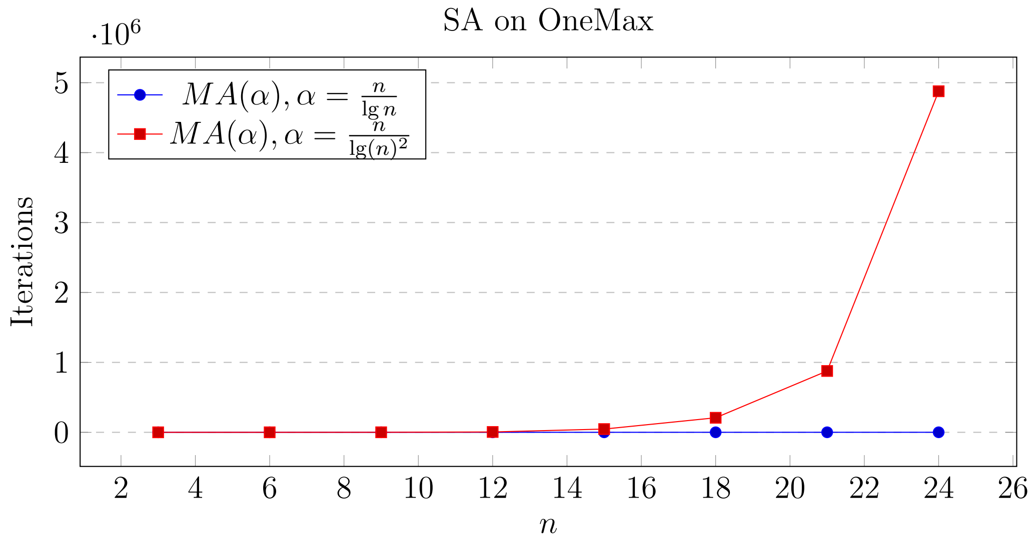


Figure 30: Averaged over 50 runs.

The results clearly show that $\alpha = \frac{n}{\lg(n)^2}$ makes the optimization time much higher than $\alpha = \frac{n}{\lg n}$. The increase in optimization time for the former value of α also looks like an exponential increase as expected. Only small values of n have been used to make the testing feasible.

6.2.2 LeadingOnes

Like for OneMax we expect a lower temperature to give a faster optimization time for LeadingOnes, since this function too is strictly unimodal and has no obstacles. As the following figure shows this is indeed also the case.

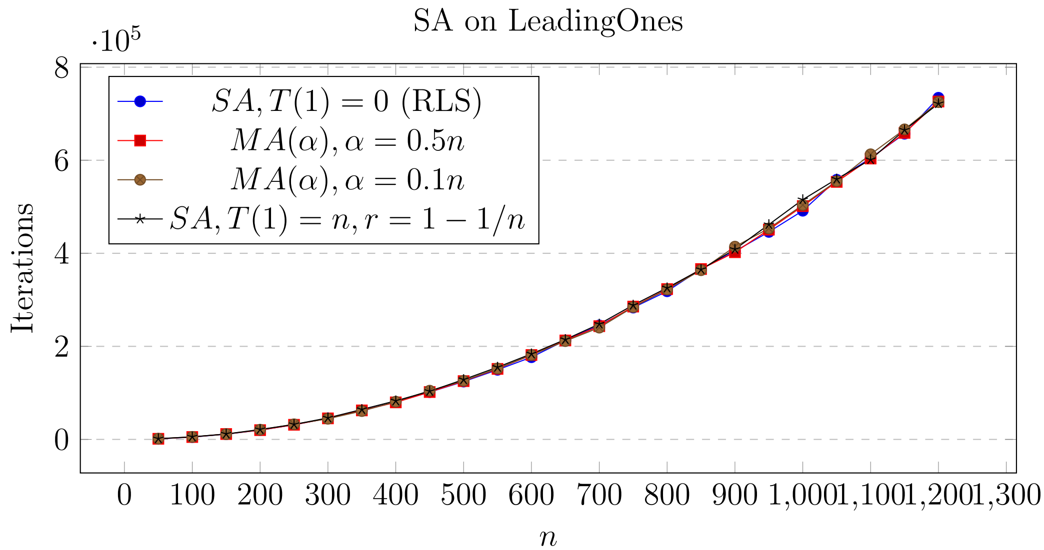


Figure 31: Averaged over 50 runs.

Interestingly enough the data points lie almost on top of each other for all the cooling schedules. SA with $T(1) = n$ was not nearly as effective on OneMax as SA with $T(1) = 0$, but for LeadingOnes they have quite similar performance with an optimization time of 721749 with $T(1) = n$ and 734303 for the latter. One should be wary of not letting the scale of the graphs deceive when making such observations though, since the graph for SA with higher temperature attains much higher values and therefore requires that the y-axis be higher than it otherwise would have to be had it not been shown. We conjecture that the the reason why SA with $T(1) = n$ performs much better compared to SA with $T(1) = 0$ is because LeadingOnes is harder to optimize, which means that the initial iterations where SA with $T(1) = n$ will not set it too far behind SA with $T(1) = 0$ until it reaches a very low temperature from which point on they will behave the same.

6.2.3 TSP

Turning to TSP the cooling schedule suddenly becomes a lot more important as there are many obstacles in the fitness landscape that the algorithm needs to overcome in order to reach the optimum. Since the algorithm only makes local changes it needs to accept search points with lower fitness in order to move away from local minima. For this reason we no longer expect a starting temperature of 0 to give the best results. To begin with, we will investigate the effects of different cooling schedules for primarily the berlin52 instance, where we initially will only allow a low amount of iterations (50000).

The cooling schedules we will try to use will primarily be based on the things discussed in section 2.4.1 meaning a starting temperature of $T(1) = n^3$, which will be 140608 for berlin52 with 52 cities; the cooling factor will be $1 - 1/cn^2$. For berlin52 the constant that makes the temperature equal to 1 at the end of 50,000 iterations is 1.57, but we will also try a constant a bit lower and higher to investigate if these will be better.

Since this benchmark is done for a relatively low number of iterations we may suspect that $T(1) = n^3$ means the cooling needs to be too fast to give good results for the temperature to reach 1 after 50000 iterations. For this reason we will also try to look at a variation of the cooling schedule with $T(1) = n$ instead for different constants. We suspect that this might be better for this number of iterations, since the cooling can be done more slowly this way while still reaching a low temperature by the end of 50000 iterations. With this starting temperature the constant that makes the temperature 1 at the end of 50000 iterations will be 4.7. We will also test some constant higher and lower than this to compare with.

Lastly it will be interesting to see how RLS compares to these cooling schedules, which can function as a baseline. We expect that most of the cooling schedules will beat RLS. Another interesting test to compare with as a baseline would be MA with some temperature. Note that we will write the number of function evaluations on the x-axis for these tests instead of the number of iterations even though they are equivalent. This is because we will in one of the tests use a variant of SA that performs more than one function evaluation per iteration. To not make comparing the results of the different tests confusing we choose therefore to use the number of function evaluations for all the tests.

Before proceeding with the analysis it is important to stress that to properly analyze the difference between a starting temperature of $T(1) = n^3$ and $T(1) = n$ we need to also run the algorithm on another TSP instance where n is different. If n is fixed the difference in starting temperature would be constant and not polynomial. We will therefore end this section with analyzing the performance on the bier127 TSP instance. We will invest more time in the analysis on berlin52 though, where a starting temperature of 140608 and 52, respectively, should still highlight some interesting aspects of the influence of the starting temperature.

The figure below shows the standard cooling schedule with high starting temperature.

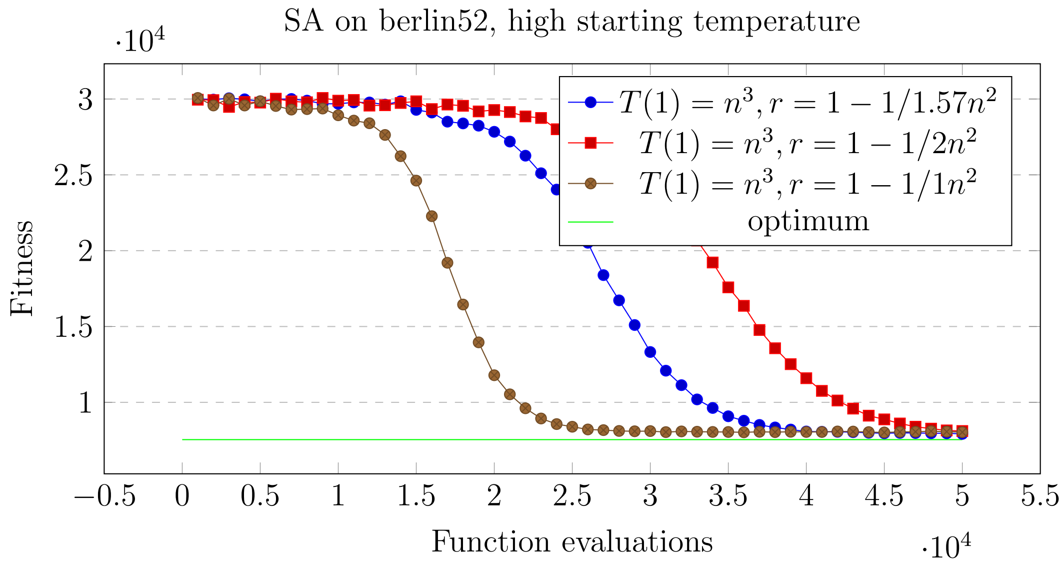


Figure 32: Averaged over 100 runs.

As expected the starting temperature of $T(1) = n^3$ makes the search very chaotic in the beginning seen by the very high fitness of the solutions generated. Because of this it is hard to see which cooling schedule achieves the best solution after 50000 iterations, so we now remove the first data points and only draw the last ones together with the other cooling schedules again. This gives the situation shown in the figure below.

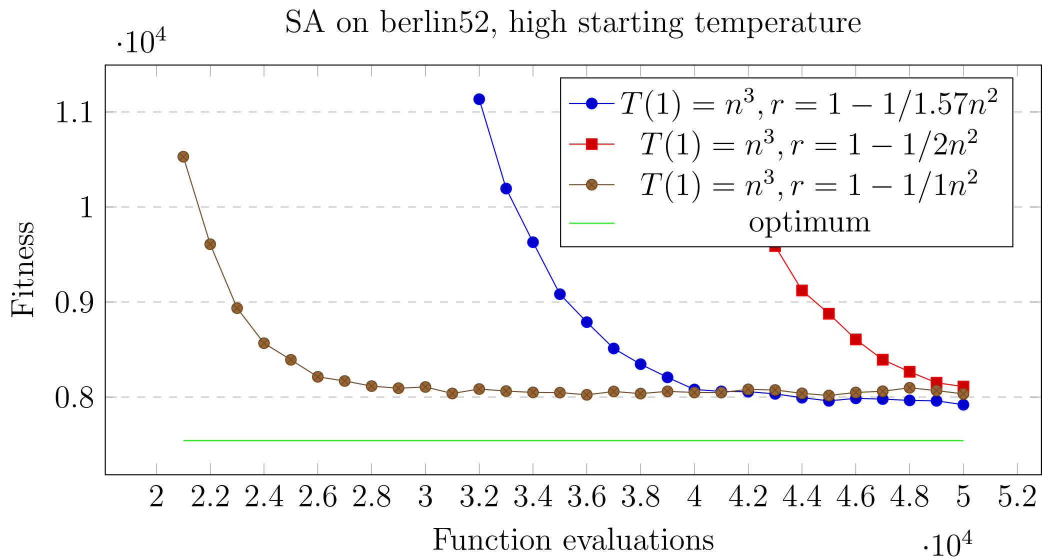


Figure 33: Averaged over 100 runs.

We observe that the best solutions at the end is generated by the cooling schedule with $c = 1.57$, so our idea that ending with a temperature of 1 when the algorithm is stopped seems to be a good idea.

We now try starting with a much lower temperature of 52. We now expect the convergence to happen very fast, since the temperature is low enough for the algorithm not to accept too big of a loss in fitness initially, even for a very slow cooling with c being set high. With a temperature of 52 the probability of accepting worsenings moves of e.g. 100 in fitness is equal to 0.15 which is much lower than with a temperature of 140608 like before. The figure below shows the fitness of the solutions generated as a function of the number of function evaluations for a few different cooling schedules.

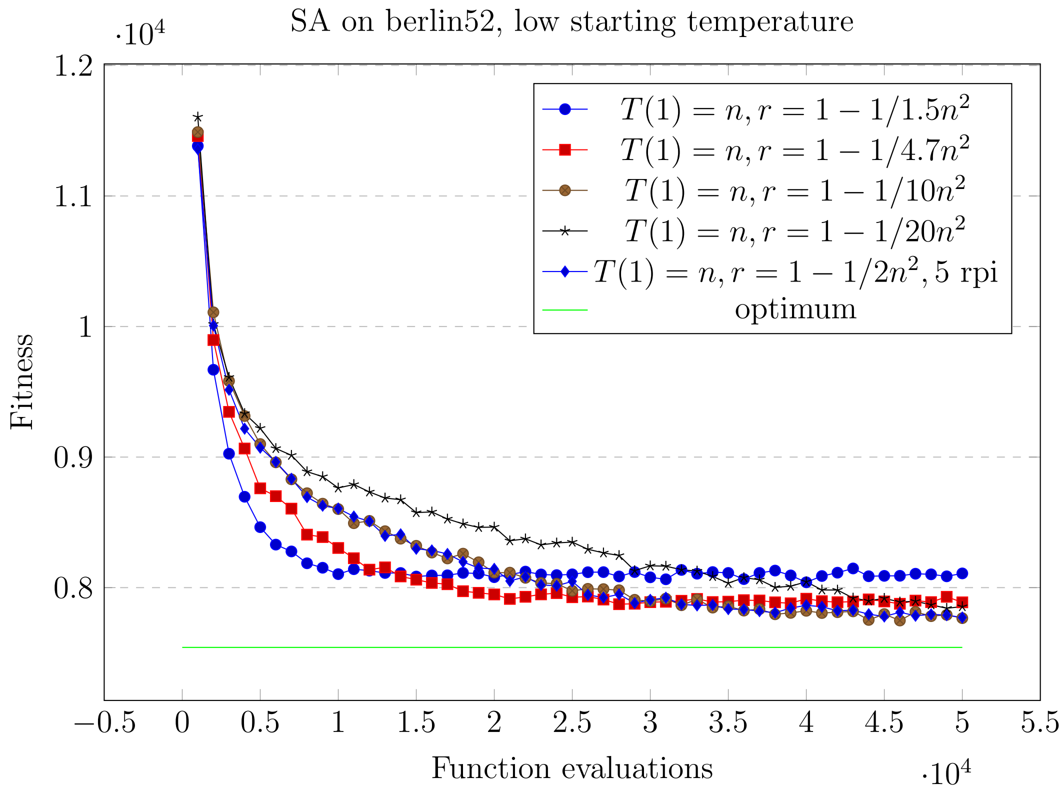


Figure 34: Averaged over 100 runs. We have used rpi as an abbreviation for *rounds per iterations* (in the batch creation step in the program this is called *iterations per temperature*).

We initially make the observation that the cooling schedule with $c = 1.5$ gives the worst solution after 50000 function evaluations among the cooling schedules tested. This result is expected since any value of the constant c lower than 4.7 will

make the temperature at the end of the 50000 iterations below 1 and a value of 1.5 will make it much lower than that. Hence, a very fast convergence to RLS in behavior, which will make it more susceptible to get stuck at local minima.

We see that setting the constant $c = 10$ rather than 4.7 gives rise to a better solution, however. This observation suggests that slower cooling at the expense of ending with a higher temperature than 1. This is different from what we saw when we started with a high temperature in figure 33. We may suspect that had we used a constant between 1.57 and 2.0 in the test in figure 33 we may have gotten a similar result though. Comparing $c = 4.7$ in figure 34 with $c = 20$ it looks like these get a solution with about the same fitness after 50000 function evaluations and we conjecture that increasing the constant c beyond 20 would therefore lead to more and more worse solutions.

The main difference between this test and the one in figure 33 with high starting temperature seems to be that with $T(1) = n$ the initial iterations are much more fruitful than for $T(1) = n^3$; this is seen by the very quick initial drop in fitness for the graphs with $T(1) = n$ in figure 34, whereas for $T(1) = n^3$ in figure 33 the decrease in fitness is not particularly great in the beginning. We suspect that for this reason it is better to spend more iterations with a low temperature. Combined with the observations made just before it seems like the iterations below 52 and above 1 are the most important with respect to lowering the fitness as much as possible.

An interesting result is that setting $c = 2$ and making the algorithm make 5 function evaluations before decreasing the temperature behaves almost exactly like the regular cooling schedule with $T(1) = n, c = 10$. This is not surprising at all, however, since $c = 2$ means that the temperature decreases by a factor of about five more quickly than for $c = 10$; by making five function evaluations per temperature decreasing step, however the two schedules will have the same temperature every five iterations. The slight difference between them does not seem to be significant with these parameter settings, but we may suspect that if we were to change five to 100 there would be a visible difference, where the standard cooling schedule would most likely be the best. The idea to try to let SA do more than one round per iteration is from the short description of simulated annealing by Dorigo and Stützle, but is also seen elsewhere in the literature.[6]

We now try to compare the solution quality for $T(1) = n^3$ and $T(1) = n$, when we take the best performing cooling schedules among each of them as well as some of the others that may be interesting to compare. The results are shown in the figure below along with RLS as a baseline for comparison.

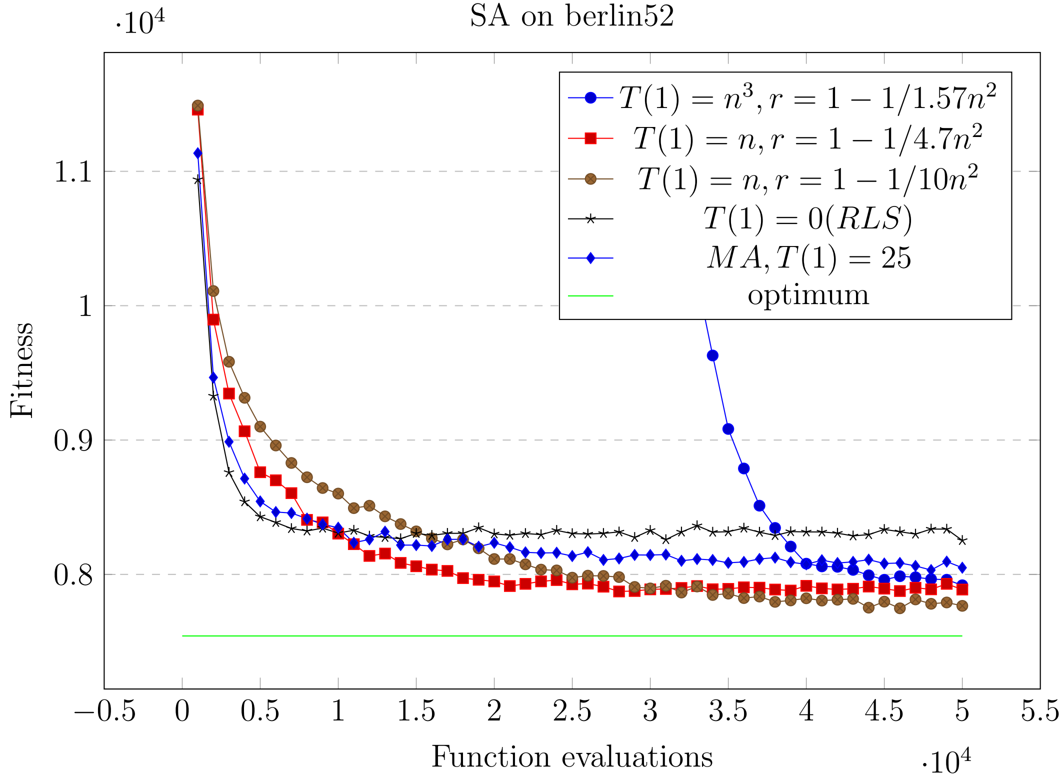


Figure 35: Averaged over 100 runs.

We immediately see that all the cooling schedules outperform RLS as expected by a fair margin.

We observe that the cooling schedule with $T(1) = n, c = 10$ is better than that with $T(1) = n^3, c = 1.57$ with respect to average fitness for this number of function evaluations allotted, which with the discussion on the importance of iterations with temperatures between 52 and 1 in mind is not surprising. Interesting in this regard is also to compare with MA with $T(1) = 25$, where the temperature will always be somewhere in said temperature interval. We see that MA does not perform as well as the other cooling schedules with decreasing temperature except RLS, but it is not that far away from the graph of $T(1) = n^3, c = 1.57$. The fact that it does well further supports the importance of iterations with temperatures in this interval.

It seems like at 50000 iterations the solutions constructed with a high starting temperature and $c = 1.57$ is about as good as that with low starting temperature and $c = 4.7$. If we were to make a proposition based on this test alone this would seem to indicate that starting with a higher temperature does not seem to give better results than starting with one that is considerably lower. We do not suspect this to be the case, however, so we will do more tests to investigate this further

with a higher amount of iterations. Initially we try to use the cooling schedule $T(1) = n, c = 10$ from before, which was the best performing for this number of iterations and compare it with the cooling schedule $T(1) = n^3, c = 10$ which will cool by the same rate but have a higher starting temperature. This way only the starting temperature will be different and allows us to show that in the long run the higher temperature will be beneficial and lead to better solutions. We note that the cooling schedule with $T(1) = n$ will reach a temperature below 1 faster than with the high starting temperature meaning that it after a certain point will collapse to RLS in behavior while the cooling schedule with $T(1) = n^3$ will still have sufficiently high temperature to utilize this to escape local optima. In order to still make a fair comparison we will also use the cooling schedule $T(1) = n$ with c chosen such that the temperature for this at the end also will be 1. To do the test we must estimate which number of iterations we should generate data points for. We expect that the cooling schedule with $T(1) = n^3, c = 10$, will not make much progress when the temperature is 1, so we will calculate the number of iterations when this happens. With the constant c this happens after about 320,000 iterations. We therefore choose to generate data points up until this number of iterations. This implies that the constant c should be set to 30 if the same should hold when starting with $T(1) = n$. Since the fitness of the solutions generated by the cooling schedule with $T(1) = n^3$ initially is much higher than that of the other two cooling schedules we disregard the first data points for this and only look at the last interesting ones so everything can more easily be compared by studying the plot. See the figure below for the results.

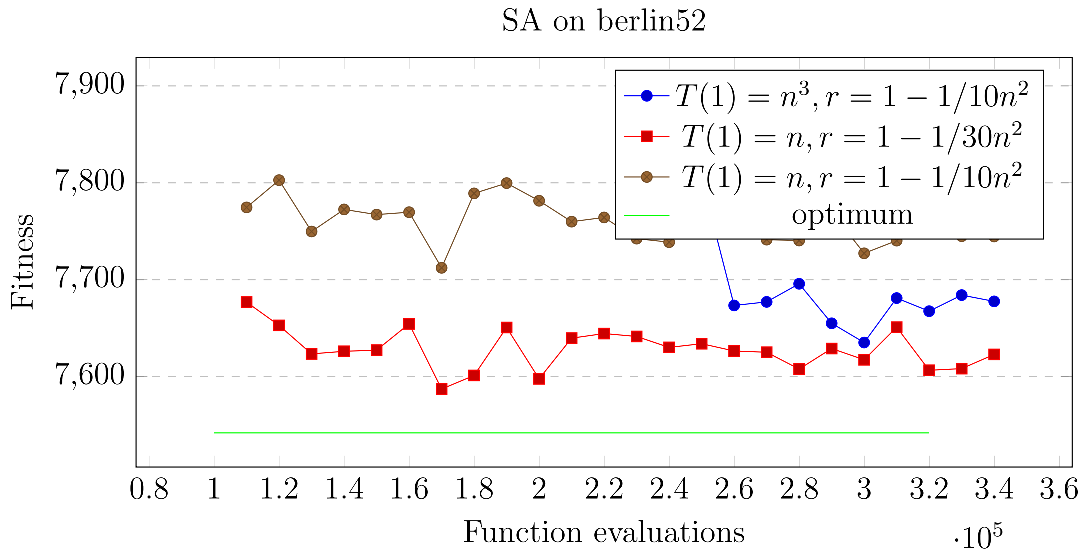


Figure 36: Averaged over 50 runs.

As expected the cooling schedule $T(1) = n^3, c = 10$ does beat $T(1) = n, c = 10$, which shows that if the temperature is decreased at a given rate then it can be beneficial to start with a higher temperature given enough iterations.

However, we see that the cooling schedule $T(1) = n^3, c = 10$ is not able to beat that of $T(1) = n, c = 30$, so we can still not based on empirical data argue that starting with a high temperature is better than a low starting temperature, since these two cooling schedules compete on even terms. We will make a test on bier127 further down below to examine what happens if we increase n. So far though, the results from the previous tests suggest that slower cooling is more important than higher starting temperatures given enough iterations. To test this further we now try to make a test with 1,000,000 and 5,000,000 iterations allotted, where we calculate c so the final temperature is approximately 1. The results of running them 100 times is shown in the table below.

Cooling Schedule	$T(1) = n^3$		$T(1) = n$	
c	31.2	156	93.6	468
runs	100	100	100	100
iterations	1,000,000	5,000,000	1,000,000	5,000,000
avg CPU time	219	1074	228	1207
max fitness	7962.0	7775.0	7816.0	7547
avg fitness	7593.0	7551.04	7551.0	7542.31
min fitness	7542.0	7542.0	7542.0	7542.0
no. optimum found	80	96	96	100

Table 5: Berlin52

Note that the reason why the number of times the optimum was found is 100 for the last cooling schedule despite the average being 7542.31 and max being 7547 is because the stopping criterion used in the test is only the number of iterations. For this reason, the algorithm can find the optimum and then wander astray afterwards. Had we set the final temperature to something lower than 1 the risk of this happening would be smaller. However, it seems like it is only one time where the algorithm found the optimum and left for a solution with fitness 5 higher, so this is not a major concern.

We see that even for this very high number of iterations the cooling schedule with low starting temperature is still the best performing; though they both perform very well. We now try to test the performance on bier127 with $n = 127$ for the cooling schedules with starting temperature $T(1) = n^3$ and $T(1) = n$ with the constant c calculated based on the number of iterations they will be run for such that the temperature will end at about 1. The results for this instance are shown in table 6.

Cooling Schedule	$T(1) = n^3$		$T(1) = n$	
c	4.27	21.4	12.8	64.0
runs	100	100	100	100
iterations	1,000,000	5,000,000	1,000,000	5,000,000
avg CPU time	381	1943	432	2170
max fitness	126752.0	122682.0	130217.0	129594.0
avg fitness	121709.36	120100.38	124430.91	122132.79
min fitness	118697.0	118313.0	119216.0	118423.0
no. optimum found	0	0	0	0

Table 6: Bier127

We now see that it is the cooling schedule with $T(1) = n^3$ that performs the best both for 1,000,000 iterations and for 5,000,000 iterations. This result suggests that it is not in general the case that starting with a temperature of n rather than n^3 is better.

Based on the results of this section we make the proposition that $T(1) = n^3$ is the best cooling schedule in general for larger TSP instances given a high amount of iterations, preferably at least 1,000,000. For small TSP instances it seems like a low starting temperature linear in the number of cities gives the best results. We also remark that if the algorithm will only be run for a low number of iterations then a low starting temperature may be better on larger TSP instances as well, since the cooling needs to be sufficiently slow in order to be effective.

6.3 Ant Colony Optimization

For the tests of MMAS on OneMAx and LeadingOnes we will be using $m = 1$ and $\tau_{min} = 1/n$ and $\tau_{min} = 1 - 1/n$ like is done in section 2.5.1 with the theorems on the bounds of the expected optimization time.

6.3.1 OneMax

Figure 37 and 38 below show the number of iterations used by MMAS to find the optimum on OneMax as a function of n with values of ρ that differ by a factor of 10. In the first of the two figures the expected upper bound of Theorem 2.13 with a default value for the hidden constant in the O-notation chosen to be 1 is shown; the second figure shows the improved upper bound of Theorem 2.14.

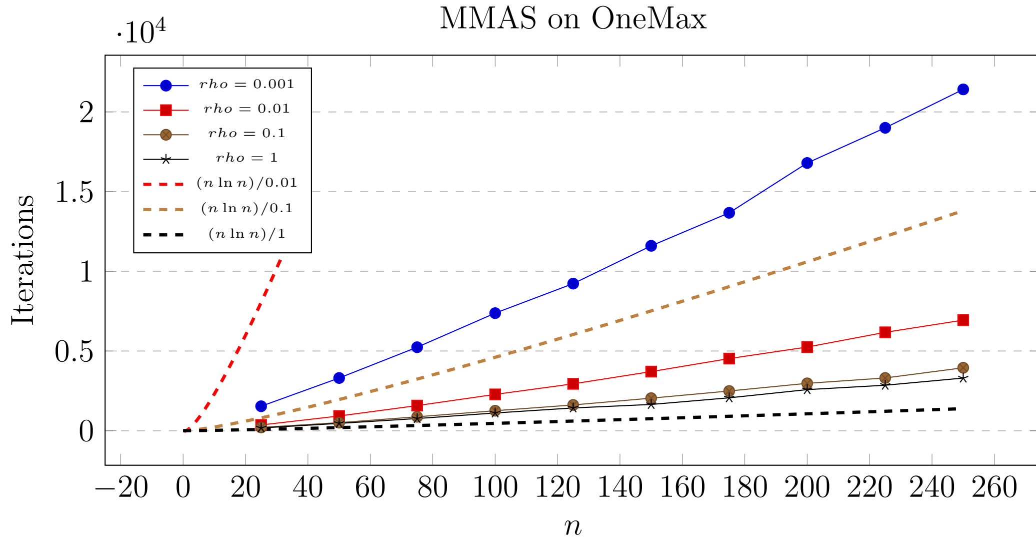


Figure 37: Averaged over 100 runs.

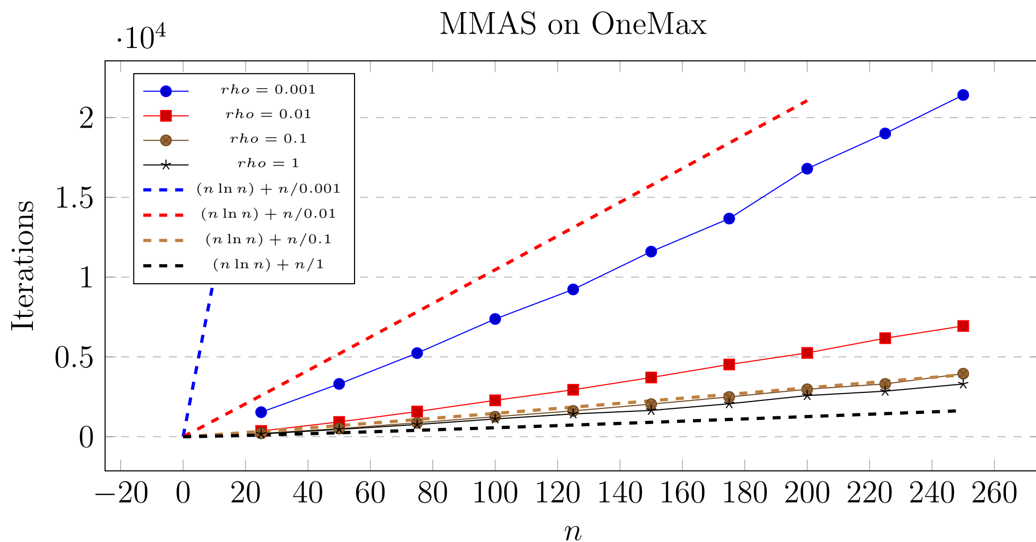


Figure 38: Averaged over 100 runs.

The results are at first glance not as expected in this test, since the expected upper bounds seem far from the empirical data - even for the improved upper bounds. This may be because the constants hidden in the O-notation and the base for the logarithm are far from the default values we have chosen for the expected optimization time. Furthermore, the values of n used here are very small, which makes the hidden constants and the choice of the logarithm base have a bigger

influence and not choosing them quite right may make the expected values seem further away from the empirical than they would be if n were higher. The increase in optimization time as n gets bigger seems to be very close to linear though. This is as expected, since for fixed ρ the function should behave like $n \log n$ asymptotically, which because the $\log n$ terms increases very slowly makes it seem very close to linear for not too low values of n.

An important observation, which is as expected is that the optimization time is lowest for $\rho = 1$ among the values we have tried. Apart from this being expected based on the fact that we divide by ρ in the optimization time expression, this also goes well with our intuition and discussion in section 2.5.1 with it being beneficial for the algorithm to hit the pheromone borders after which it will behave in a way similar to (1+1) EA, which as we saw earlier is good at optimizing OneMax.

Another observation we make is that it seems like decreasing ρ by a factor of 10 like we have done in the tests above with ρ being $1, 0.1, 0.01$ and $0, 001$, respectively, makes the optimization time for fixed n increase by a constant factor as well. Looking at number of iterations for the data point point for $n = 250$ we have the optimization times for ρ being set to 1, 0.1, 0.01 and 0.001 to be 3301, 3948, 6938 and 21414 respectively. Let E_ρ denote the optimization time of the algorithm with ρ and $n = 250$. We make the observation that:

$$\frac{E_{0.01} - E_1}{E_{0.1} - E_1} = \frac{6938 - 3301}{3948 - 3301} \approx 5.0$$

and

$$\frac{E_{0.001} - E_1}{E_{0.01} - E_1} = \frac{21414 - 3301}{6938 - 3301} \approx 5.6$$

These values are not far from each other, although the estimation with just this single search point is very imprecise. Based on theorem 2.13, we expect the constant to be about the same, since we decreased ρ by the same factor between the different tests, namely by a factor of 10, and because in the upper bound on the expected optimization time one divides by ρ.

For fixed n the function should behave like the function ρ^{-c} i.e. decreasing polynomially. To further test this, we now try to plot the optimization time as a function of ρ instead for fixed n. In order to test whether the optimization time decreases polynomially we make both axes logarithmic, which should make the graph linear if it is polynomial. The figure below shows the number of iterations it takes to optimize OneMax for different values of ρ.

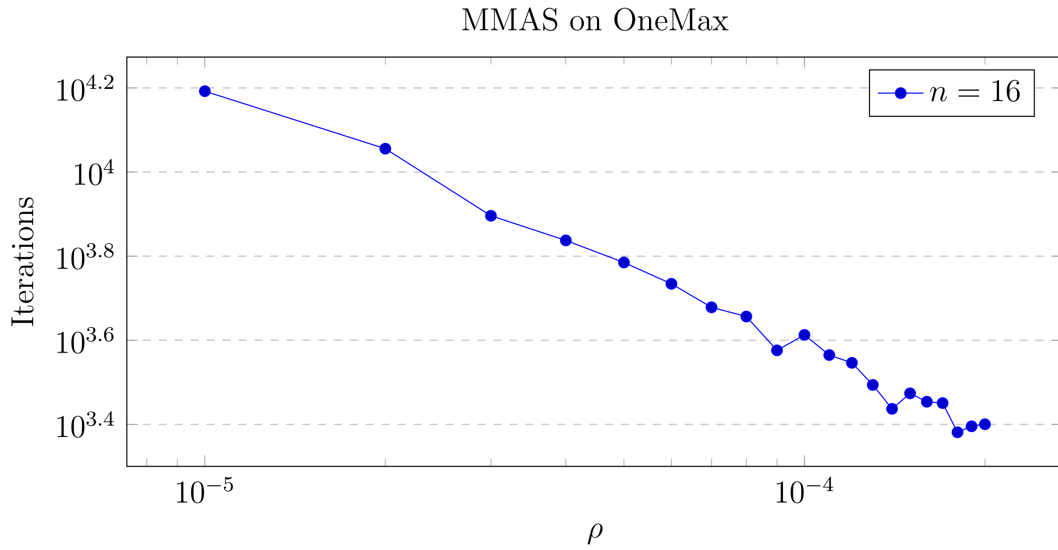


Figure 39: Averaged over 100 runs.

As suspected the decrease in optimization time seems to be linear with these axes, which supports our conjecture.

The plot above was only for relatively high values of ρ. In the paper by Neumann, Sudholt and Witt [21] the authors have benchmarked the running time of MMAS on OneMax as a function of ρ with very low values of ρ as well and they are able to observe an interesting pattern. Like in the paper we make a plot with exponentially decreasing values of ρ starting with $\rho = 2^{-19}$, which allows us to study the difference between this setting compared to high values of ρ:

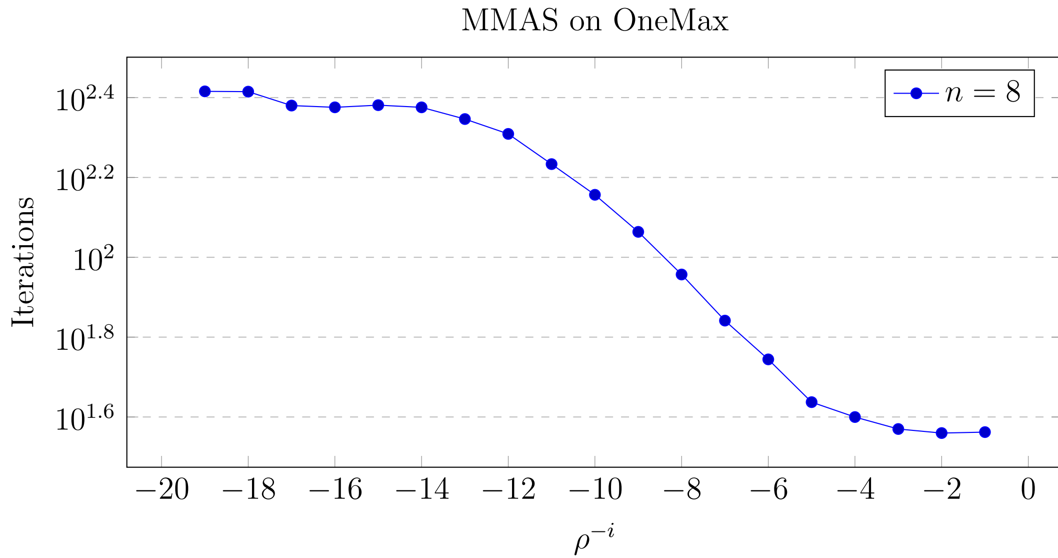


Figure 40: Averaged over 1000 runs. The numbers on the x-axis are the exponent i of ρ.

Note that the vertical axis is logarithmic. We observe that for very low values of ρ the optimization time does not decrease notably unlike for higher values of ρ. The authors get a similar result and write that for very low values of ρ the search will behave very similar to pure random search; this is expected since a very low value of ρ means that the pheromone does not have any significant influence on the choice of which bits to flip; the pheromone evaporation and deposit of pheromone on the bits that make up the best-so-far solution will happen so slowly that the algorithm by pure chance will find the optimum before it has a noticeable effect. In conclusion, for high values of ρ decreasing ρ seems to make the optimization time increase polynomially as we would expect from the upper bound of Theorem 2.13, but for very low values of ρ decreasing ρ further will not have much of an impact on the optimization time anymore.[21]

Overall the tests seem to confirm many of theoretical aspects and properties of MMAS on OneMax. Admittedly though the expected lines with the chosen default constants hidden by the O-notation are not as close to the empirical data as we would have hoped for. Further testing with higher values of n would be beneficial to strengthen the ground for the expected upper bound, but has not been done since the CPU time for running the tests is too high to make it practical to do so.

6.3.2 LeadingOnes

Before comparing the optimization time of MMAS on LeadingOnes with the expected upper bound from Theorem 2.16 and the lower bound from Theorem 2.17, we initially just plot the optimization time as a function of n with a few different values of ρ.

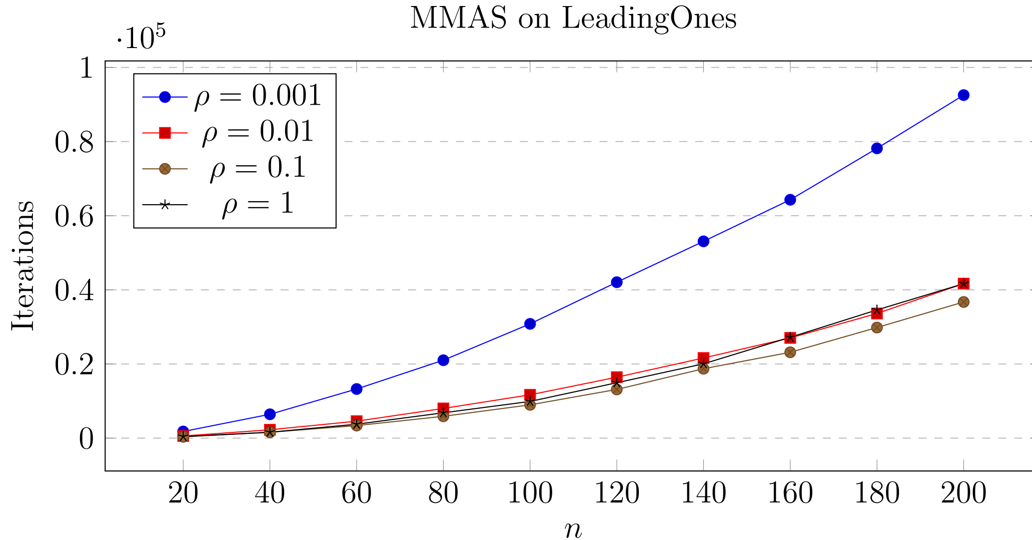


Figure 41: Averaged over 50 runs.

We clearly see that setting $\rho = 0.001$, which is the lowest value of ρ among the values used here, gives the highest optimization time. This is as expected, since the upper bound of $O(n^2 + n/\rho)$ and lower bound $\frac{n/\rho}{\lg(2/\rho)} = n$ increase when ρ decreases. Looking at the other data points we get a result that at first glance is unexpected: We see that the lowest optimization time is achieved with $\rho = 0.1$ and that the optimization time when setting $\rho = 1$ and $\rho = 0.01$ are almost the same; we would expect $\rho = 1$ to give the fastest optimization time.

It turns out that researchers in the literature also get seemingly unexpected results for high values of ρ for some pseudo-boolean functions, which suggests that there is not necessarily something wrong with our test above. In a paper by Kötzing, Neumann, Sudholt and Wagner [17] the authors carry out a benchmark of MMAS on BinVal, where they observe that $\rho = 0.5$ gives a better optimization time than $\rho = 1.0$. This result is not evident either from the upper bound on the expected optimization time. Although BinVal is not the same function as LeadingOnes it does have similarities with LeadingOnes and we believe that the fact that the authors of said paper do not get the result that $\rho = 1$ gives the best optimization time means that it is not so surprising that this is not the case in our

test on LeadingOnes after all from a pragmatic point of view.

We now try to plot the data points together with the expected upper and lower bounds with some default constant and base two for the logarithm.

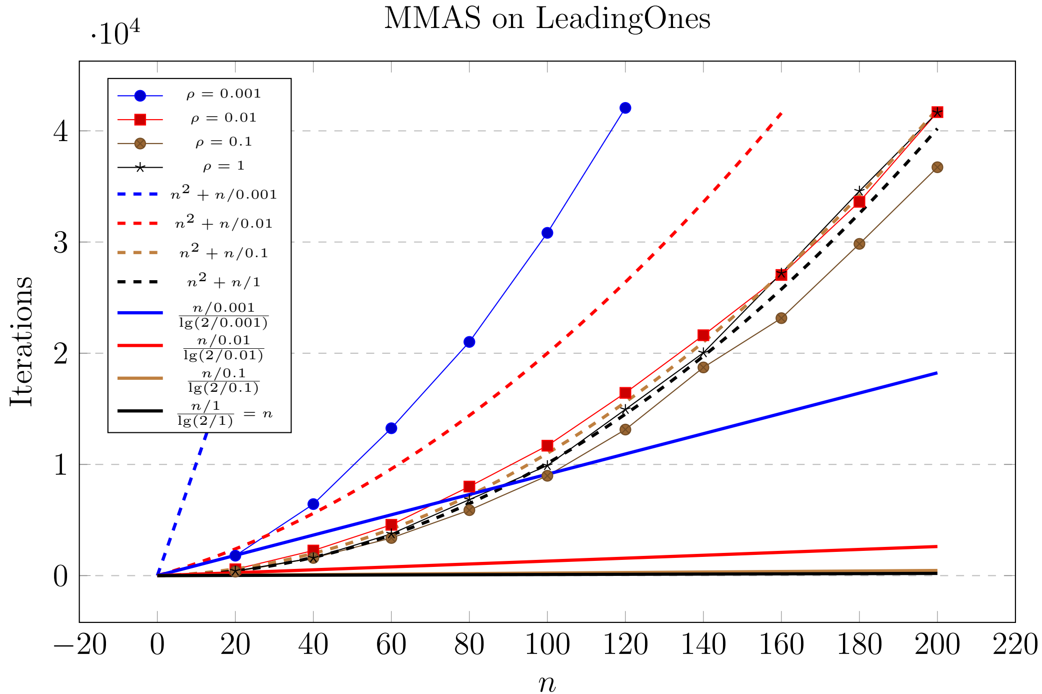


Figure 42: Averaged over 50 runs.

Looking carefully at the plot it does look like the optimization time increases asymptotically with n as expected with a curvature similar to that of the function n^2, which for fixed ρ is the dominating term in the upper bound. For $\rho = 0.001$, $\rho = 0.01$ and $\rho = 0.1$ the lines for the upper bounds are all above the data points as expected. For $\rho = 1$ this is not the case, although very close. For the lower bounds all the data points are above the lines as expected though. Note that like in the test for MMAS on OneMax we have only used relatively low values of n in this test, since the CPU of the algorithm makes it unpractical for higher values of n. For fixed n and values of ρ lower than 0.1 it seems like decreasing ρ will increase the optimization time polynomially like we observed for OneMax and is as expected. This is seen in the plot above by the big increase in optimization time when decreasing ρ from 0.01 to 0.001. It therefore only seems to be when ρ is in the interval $[0.01, 1]$ that the we are not able to see the decrease in optimization time when ρ decreases.

Contrary to the results we have achieved for the other algorithms thus far, we cannot readily confirm the upper bounds on the expected optimization time

based on this empirical analysis, since we get that optimization time when setting $\rho = 0.1$ is lower than setting $\rho = 1$. As a whole the test does not suggest that the upper bound is incorrect either, however, since we do observe the increase in optimization time when decreasing ρ when ρ is sufficiently low (lower than 0.1). Furthermore, the increase in optimization time for fixed ρ we observe is encouraging when comparing it to the expected upper bound. We also refer to the analysis carried out by Kötzing, Neumann, Sudholt and Wagner on BinVal where they observe a similar pattern [17]. In conclusion, this test has given some interesting insight into the optimization time of MMAS on LeadingOnes and shows that there are some interesting fluctuations in the optimization time for high values of ρ, which are not precisely captured by the upper bounds in section 2.5.1.

6.3.3 TSP

For MMAS there are many different parameters to tweak when going for an optimal parameter setting, which makes for even more combinations to analyze than for simulated annealing looked at in section 6.2. Because we already know some very good parameter settings that are based on extensive testing by Dorigo and Stützle (see section 2.5.2)we will therefore try to start with these parameters and change each of them to see what effect it has. We remark that this approach may not show all the important aspects of the influence of the different parameters, since the effect of one parameter may be heavily dependent on the value of another. It is possible that changing one parameter may be very beneficial, but will not become evident in the test if a specific other parameter is not set accordingly prior to the change. Knowing that each change should increase the performance in some way not observing an improvement may also be interesting in itself though.

Unless otherwise stated the optimal parameters are exactly those found in section 2.5.2 with τ_{max} and τ_{min} being dynamically adjusted as better solutions are found. The initial pheromone on the edges are also based on a tour generated on the nearest-neighbor as advised. Furthermore, we the pheromone trails on all the edges is reinitialized when the system approaches stagnation, that is, when no improvement in fitness has been made for a certain amount of consecutive iterations. We have set this number to 1000 in all the following tests. Using the visualization framework it seems like this does not have much of an impact on the performance, however, and may only help on very large TSP instances. As a rule we the pheromone deposit will be on the edges part of the iteration-best solution in each iteration for the berlin52 TSP instance, since the number of cities is low; we will also in a test try to use the best-so-far solution instead though, but this will be clearly stated in the legend of the figure in question.

Figure 43 shows MMAS on berlin52 with the optimal parameter setting and with the evaporation rate increased to 0.5 and with only a single ant agent. Figure

44 shows MMAS with $\beta = 0$, which is shown in a separate figure due to the relatively low quality of solutions compared to the other of said parameter changes.

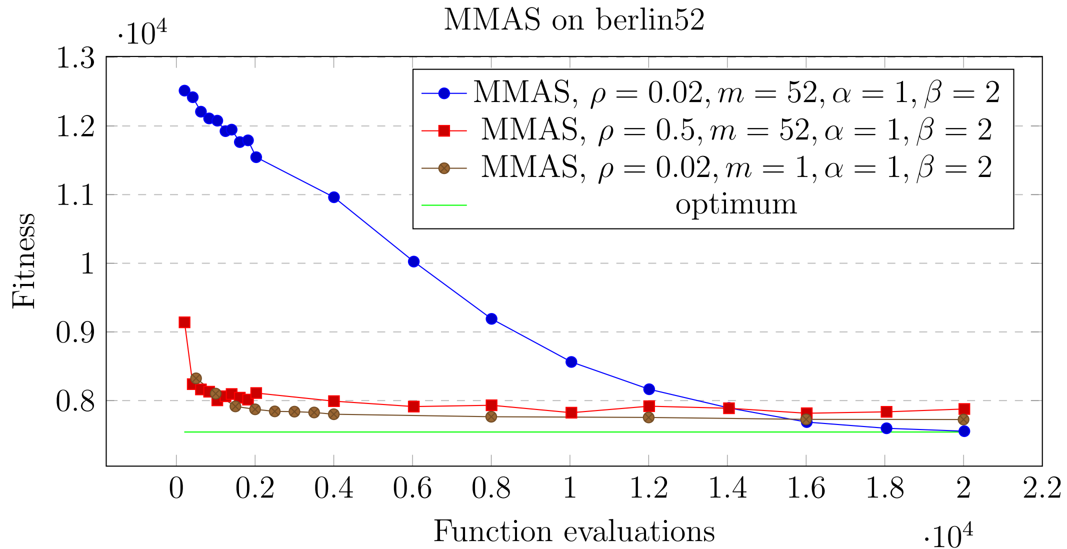


Figure 43: Averaged over 25 runs

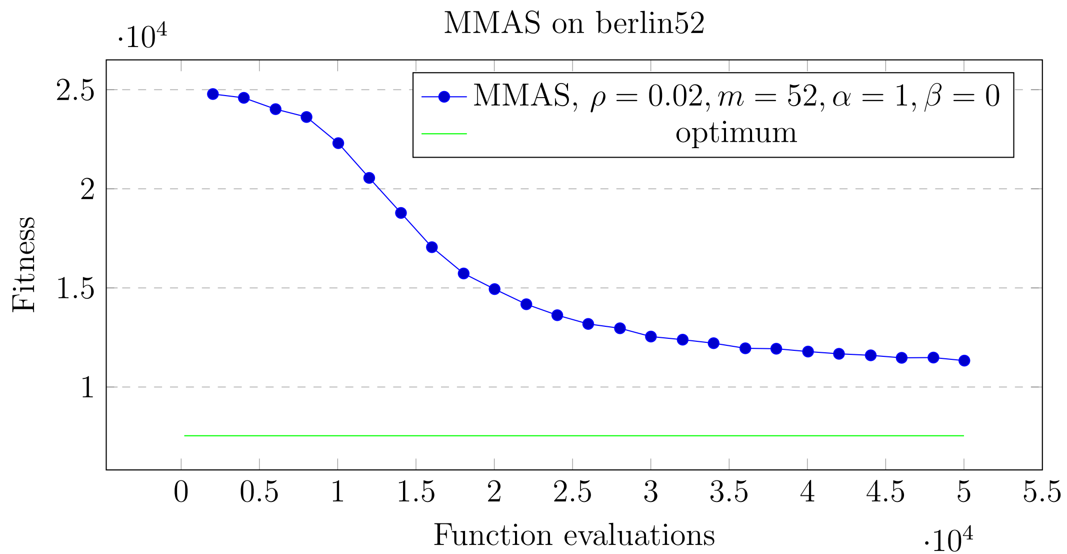


Figure 44: Averaged over 25 runs

We immedtiately notice that changing β from 0 to 2 helps immensely in achieving good solutions. This is no surprising result at all, since β was the parameter that

determined the influence of the heuristic relative to the pheromone trail values. We expect that this would help a great deal in guiding the search, especially in the beginning where there are no previous pheromone trails that influence the ant's decisions on where to go. With $\beta = 0$, the ants will in the beginning make pure random choices of where to go because all edges have the same probability of being chosen no matter how high a weight they have. Because the pheromone deposited is relative to the utility of the tour and only the iteration-best solution gets pheromone, the better solutions should get more pheromone after a while though, which will help the ants avoid the bad ones. This is clearly seen by the very bad fitness in the beginning in figure 44, but after some time this is much lower, although it never gets anywhere close to the solutions generated by the algorithm with optimal settings in figure 43.

Now look at the effect of changing the evaporation factor, which is set to a value of 0.5, which is significantly higher than what is used in the optimal parameter settings. As expected this makes the fitness of the solution generated decrease much faster. We also see that solution quality converges to a fitness value worse than for the optimal parameters. This may be explained by the initial explorative search phase being too short to find the best area of solutions in the search space, because the pheromone wil freeze very quickly due to the high evaporation rate. If a very low number of function evaluations are allotted, however, a high evaporation rate may be desirable, since it very quickly finds a decent solution and is only overtaken by the optimal parameter settings after around 14000 function evaluations.

Next we look at the change of the number of ant agents by using only a single ant compared to the usual 52. Because there is now only one ant to search through the search space, we would also expect a quicker convergence for this parameter setting. Even though it is still only one tour that gets pheromone each iteration when pheromone is deposited as in the case of many ants, fewer tours are constructed in each iteration among which the best is found. This in turn, means that it is likely that the tours constructed in the first few iterations are reused over and the edges of which will get a large amount of pheromone. Thus, the search is much less explorative. Nevertheless, the algorithm with these parameters does achieve really good results and comes close to the optimum after 20000 function evaluations. Something that is not evident in figure 43, however, is that the CPU time was significantly higher when only one ant was used, because the number of iterations will be much higher, due to a lower number of function evaluations per iteration.

Because MMAS is an extension of AS with the four changes mentioned in section 2.5, most notably the restriction of pheromone to an interval and only allowing either the best-so-far solution or the iteration-best solution to be the subject of pheromone deposit, it was easy during the implementation to implement AS and then extend this functionality rather than only implement MMAS. This

allows us now to test how MMAS fairs relative to AS, where we expect the results of MMAS to be better. The figure below shows both algorithms with the optimal settings run on the berlin52 TSP instance. We also test MMAS with the best-so-far solution when depositing pheromone. The parameter settings for AS also be based on table 2 found in section 2.5.

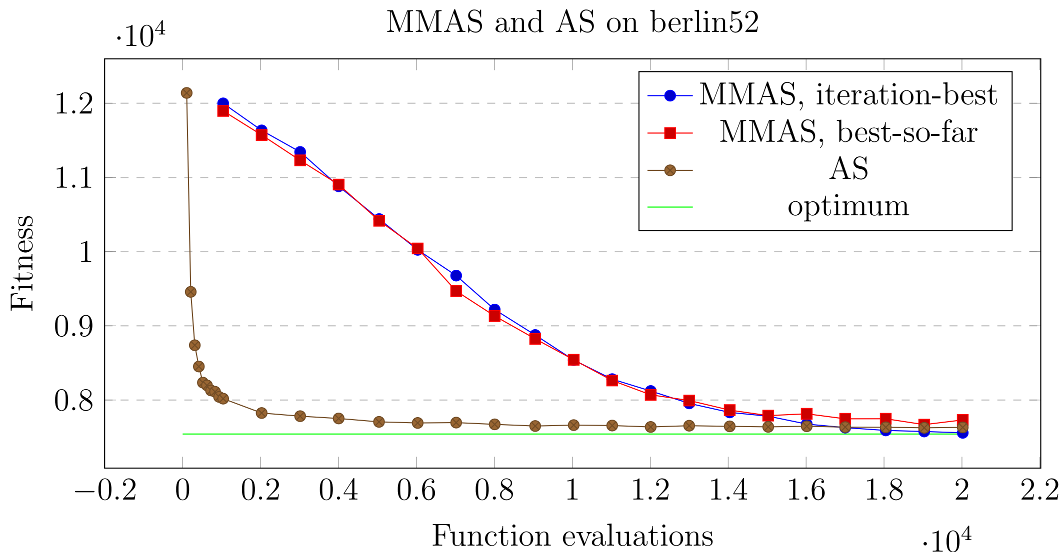


Figure 45: Averaged over 50 runs

We observe that AS converges much faster than MMAS and after the first 40000 function evaluations the solution quality it achieves is much better than that of MMAS. After 20000 function evaluations, however, MMAS with iteration-best solution being the subject of pheromone deposit will generate better solutions and actually manages to find the optimum in almost all runs after 20000 function evaluations used. Comparing AS with MMAS using the best-so-far solution when depositing pheromone we observe that AS actually manages to achieve better solutions for all data points. It therefore seems important to use the iteration-best solution when depositing pheromone on this TSP instance with a low amount of cities.

We now try to run AS and MMSA on the bier127 TSP instance with more cities and see how they compare when n is higher. The results of this are shown in figure 46.

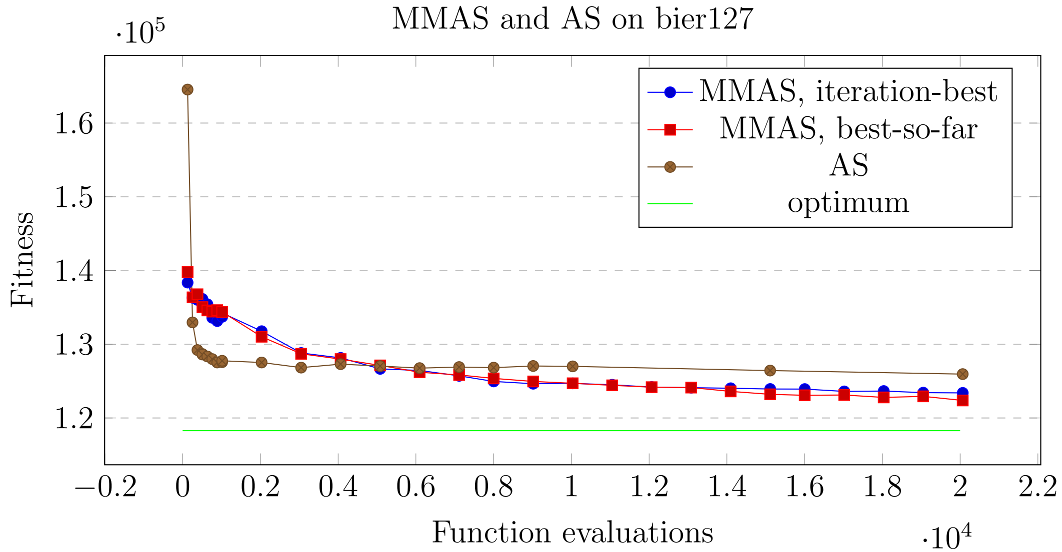


Figure 46: Averaged over 25 runs

We see the same convergence pattern as before with AS converging much faster than MMAS. On this TSP instance both MMAS with the iteration-best solution and the best-so-far solution used when depositing pheromone achieves much better solutions than AS though. As expected the performance when using the best-so-far solution is much better relative to using the iteration-best solution and in this test it looks like MMAS with the best-so-far solution is the best among the two, which is different than for the berlin52 instance before. We therefore conjecture that for TSP instances with a number of cities of 127 or higher it is better to use the best-so-far solution when depositing pheromone. Alternating between the iteration-best and best-so far solution may also be beneficial. The program does support this, but we will not make tests for this here.

6.4 Comparing the Algorithms

In this section we will study how the performance compare for the algorithms (1+1) EA, SA and MMAS. The observations and results from each of the previous individual sections for these algorithms will be used to set the parameters in a way that should allow each algorithm to have the best performance for the problem in question. We will compare the optimization time as well as the CPU time for the algorithms on the problems OneMax, LeadingOnes and TSP and discuss the results for each of these tests. In section 6.4.3 we will also analyze the difference between the (1+1) EA and SA on the functions Jump_k and f_2 and compare this with the theorems and discussion of section 2.4.3.

6.4.1 OneMax

When comparing the algorithms on OneMax we will use the (1+1) EA with standard mutation probability $p = 1/n$, simulated annealing with starting temperature 0 corresponding to RLS and MMAS with $m = 1, \rho = 1$ and $\tau_{max} = 1 - 1/n$ and $\tau_{max} = 1/n$. The figure below shows the optimization time of the different algorithms.

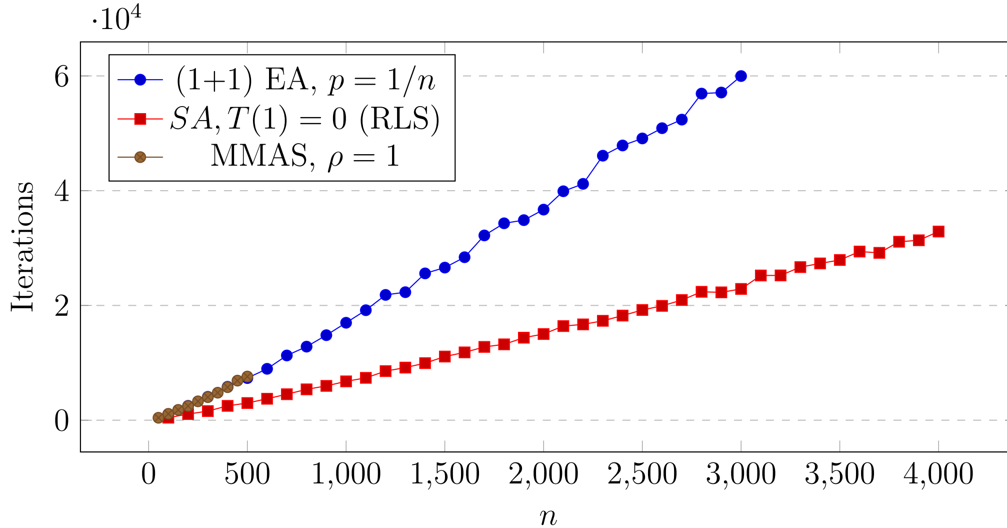


Figure 47: (1+1) EA and SA averaged over 100 runs. MMAS averaged over 50 runs.

The reason why so few data points have been generated for MMAS is because of a very high CPU time (see figure 48). Still, we are able to observe that the (1+1) EA and MMAS seem to have a very similar optimization time, which is as we expect, since $\rho = 1$ should make the pheromone reach their borders very quickly and MMAS collapse to the (1+1) EA. For $\rho = 1$ the upper bound on their optimization time is very similar, which also leads to this expectation. SA with a temperature of 0 beats both the (1+1) EA and MMAS on OneMax in optimization time as expected.

We now look at the CPU time of the algorithms seen in figure 48.

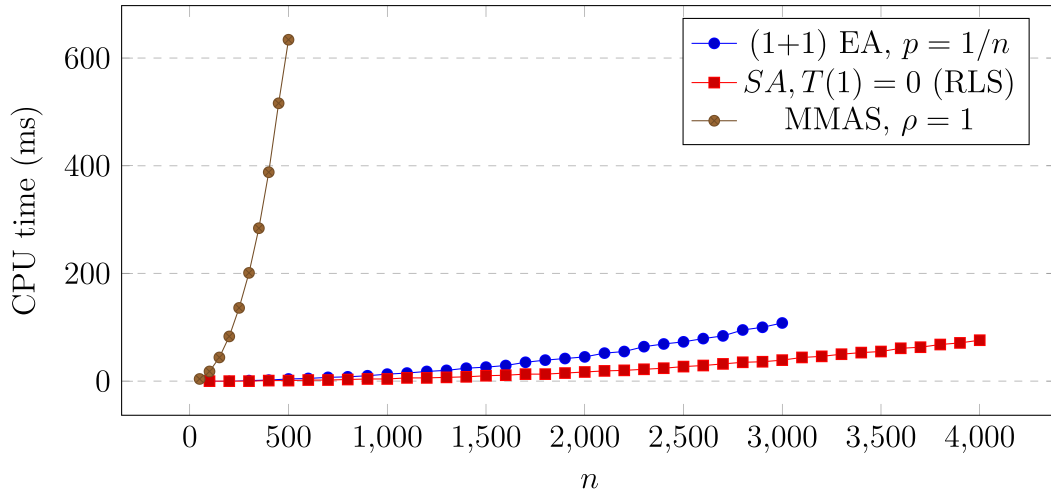


Figure 48: (1+1) EA and SA averaged over 100 runs. MMAS averaged over 50 runs.

It is clearly evident in this test that MMAS uses much more CPU time than the other algorithms, which we suspect is because MMAS in each iterations makes a lot more operations in order to update all relevant fields. For example in the evaporation step of MMAS it makes a multiplication operation for all edges, which for pseudo-boolean functions means that this is of order $\Theta(n)$. On the contrary, the optimized (1+1) EA on average only makes one mutation in each iteration and SA is guaranteed one make one mutation per iteration, so the running time with respect to the number of operations each iteration for these is $O(1)$. Recalling the huge difference in CPU time between the non-optimized and optimized implementation of the (1+1) EA in section 5.4.1, which had a running time of $O(n)$ and $O(1)$, respectively, it is not surprising to see the same big difference in CPU time between MMAS and the (1+1) EA and SA here as well.

6.4.2 LeadingOnes

We now run the three algorithms with the same settings as for OneMax except the mutation probability for the (1+1) EA is set to the optimum value of $p = 1.5936/n$.

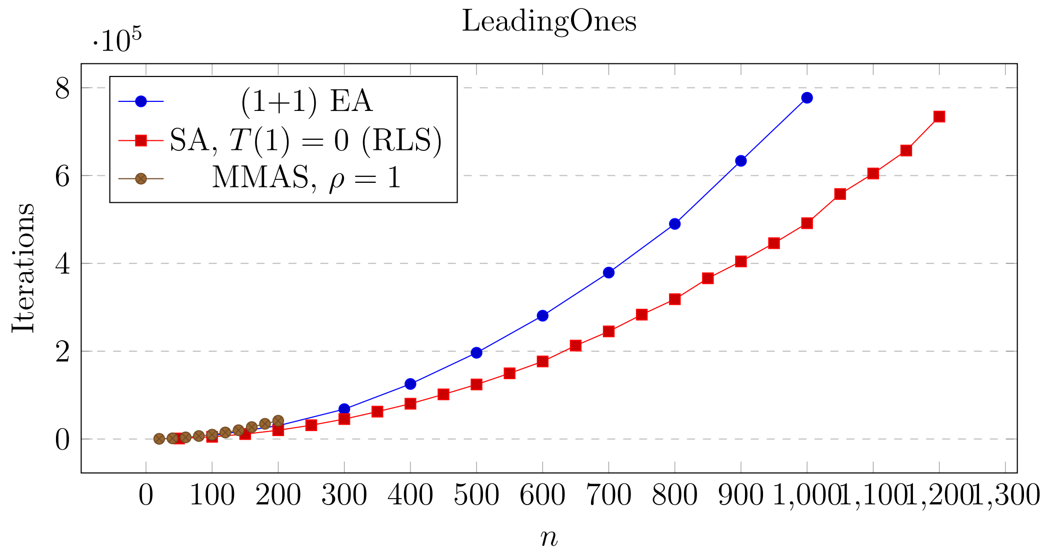


Figure 49: Averaged over 50 runs.

Again the CPU time is limiting the number of data points we generate for MMAS. The results are almost the same as for OneMax with SA with $T(1) = 0$ giving the lowest optimization time and MMAS achieving an optimization time close to that of the (1+1) EA.

The figure below shows the CPU time it takes for the algorithms to find the optimum.

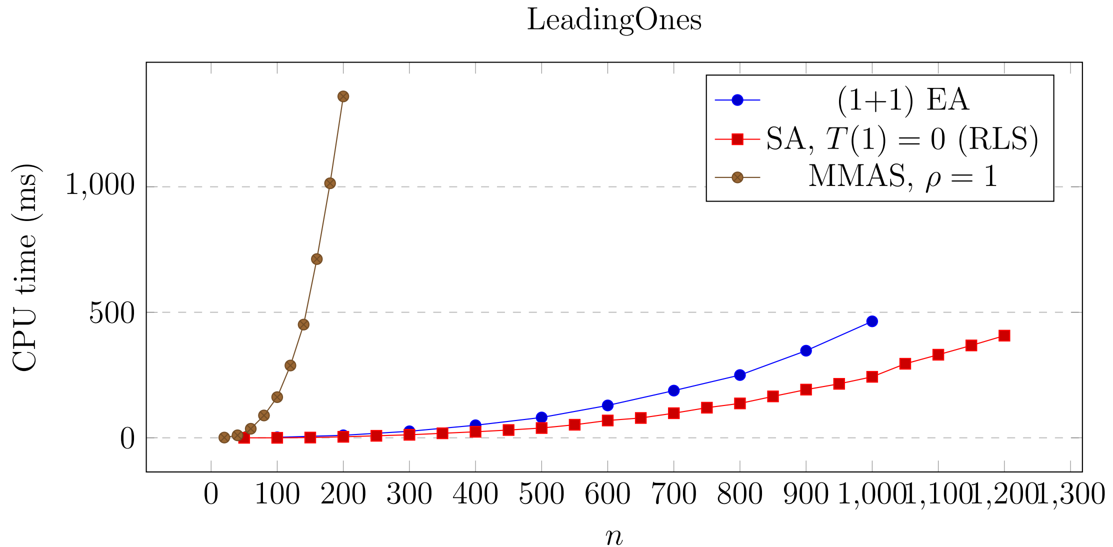


Figure 50: Averaged over 50 runs.

Looking at the results for the (1+1) EA and SA on both OneMax and LeadingOnes it seems like the optimization time and CPU time are correlated as expected. Because the (1+1) EA and SA have a very fast running time the time it takes to evaluate the fitness function is relatively longer for these than for MMAS and for this reason it is more clear that the the increase in optimization time and CPU time is asymptotically very similar.

6.4.3 (1+1) EA and SA on Jump$_k$ and f_2

We now try to examine the difference in optimization time between the (1+1) EA and SA on the pseudo-boolean functions Jump$_k$ and f_2. The goal of these tests will be to show that the difference in their way of handling obstacles can make them very different in performance. The figure below shows the optimization time of the (1+1) EA and SA on Jump$_2$.

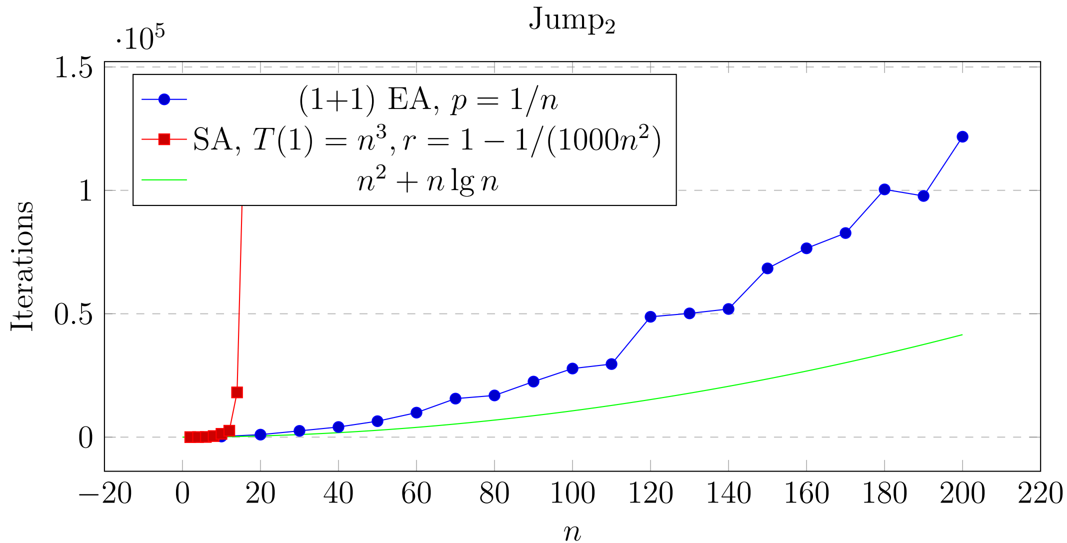


Figure 51: (1+1) EA averaged over 50 runs. SA averaged over 10 runs.

We immediately notice the difference in how the optimization increases with larger values of n for the two algorithms. When n gets higher than 10 the optimization time for SA increases exponentially, whereas the (1+1) EA follows a polynomially looking curvature as expected. Since Theorem 2.10 states that SA uses exponential time to optimize the function f_1, which is like Jump$_2$ we would also expect this behavior in this test. A cooling schedule has been chosen that features both a high starting temperature and slow cooling. The fact that it initially does well for the very low values of n and then the optimization time suddenly increases

so much suggests that it is not because of a bad cooling schedule that SA is not able to compete with the (1+1) EA in optimization time.

Although it looks like the optimization time for (1+1) EA increases polynomially the empirical data does not seem to allow us to confirm the tight bound of Theorem 2.4 since the expected line and the data points do not line up too well at first glance. Like Jansen and Wegener in their paper [11], we therefore try to plot the optimization time quotient, that is the empirical optimization time divided by the expected. This is shown in the plot below.

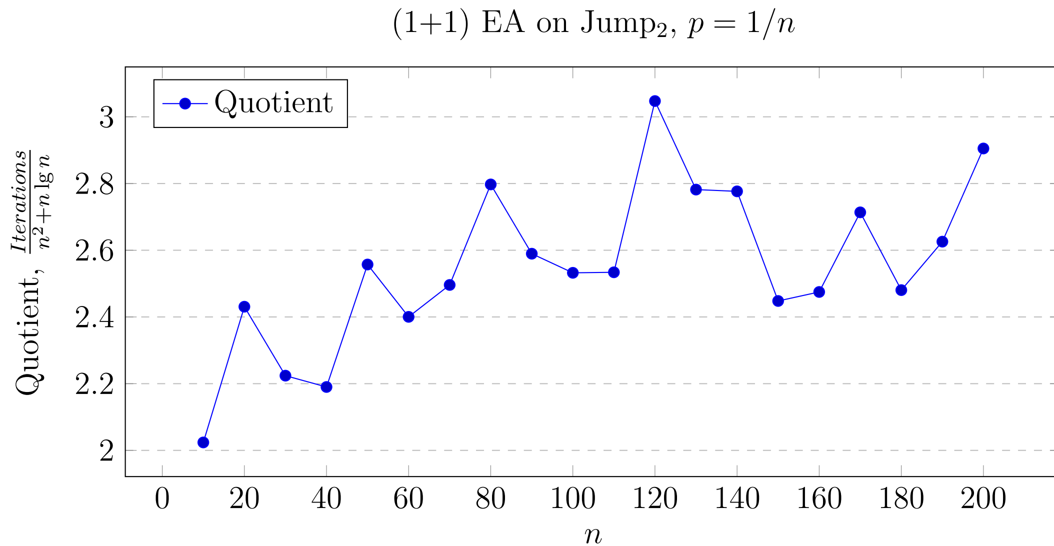


Figure 52: Averaged over 200 runs.

Albeit not a very clear pattern, it does look like the quotient does not get much higher than three and that it after $n = 100$ does not increase for larger n. The result is very similar to what Jansen and Wegener get, so in conclusion the test does seem to suggest the correctness of the the upper bound.

We now try testing the algorithms on f_2. Theorem 2.11 gives us the exact cooling schedule of MA to make it able to optimize the function in polynomial time in n. Note that we will use the metropolis algorithm for this test, since this is used in the theorem. We could alternatively have mimicked this by choosing a cooling schedule with very slow cooling for SA and would expect to get similar results. Running with this cooling schedule and the (1+1) EA with standard mutation probability we get the results shown below:

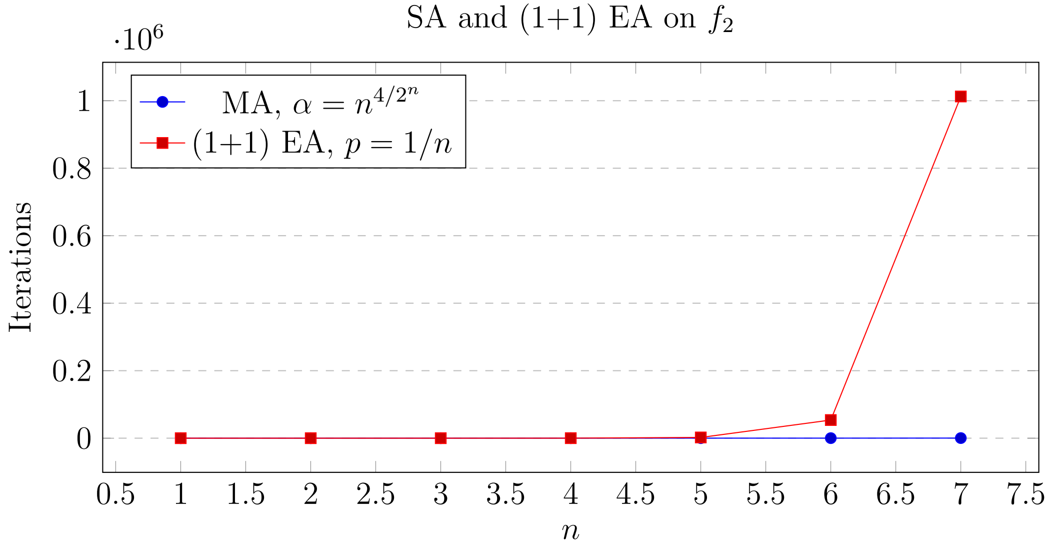


Figure 53: MA averaged over 50 runs. (1+1) EA averaged over 10 runs.

We clearly see that the optimization time of the (1+1) EA increases with what looks like an exponential rate, whereas MA with the chosen cooling schedule is able to find the optimim relatively easily. We cannot from this plot confirm that MA does this in cubic time, since we can hardly tell the difference in optimization times apart between data points for the different values of n, but it is evidently not as big of an increase as the (1+1) EA and does not look exponential, so we conjecture that the bound of Theorem 2.11 is true. The primary purpose of the test was also only to show that for the obstacle in f_2 MA would fair much better than the (1+1) EA.

In conclusion, although the (1+1) EA and SA have a very similar optimization time on OneMax and LeadingOnes (with an appropriate cooling schedule for SA), the algorithms each have their strengths and weaknesses when it comes to the two different kinds of pseudo-boolean obstacles seen here. The (1+1) EA is good at overcoming gaps with big drops in fitness, since it can jump over them, while SA with the right cooling schedule is good at overcoming wide gaps as long as they are not too deep. It would be interesting to test MMAS on these problems as well, where we would expect it behave similarly to the (1+1) EA, but this will not be done here.

6.4.4 TSP

We start with analyzing fitness of the solutions the algorithms achieve on the berlin52 TSP instance after 50000 function evaluations. For SA we use the cooling schedule $T(1) = n, r = 1 - 1/10n^2$ and for MMAS we use the optimal parameters in section

2.5.2, the iteration-best solution when depositing pheromone, re-computation of τ_{min} and τ_{max} when better solutions are found and if no improvement in fitness has happened for 1000 consecutive iterations, then the pheromone trails are initialized again for all edges.

When making a comparison between the (1+1) EA and SA with MMAS we will keep in mind that MMAS utilizes some heuristic information about the problem by the ants basing their decision of where to go (in `asDecisionRule` shown in section 5.1) while the (1+1) EA and SA do not. For this reason we expect MMAS to get the best results.

Apart from comparing the algorithms with each other we will also compare their performance with respect to the theoretical approximation of Christofides' algorithm, namely 3/2 times the optimum fitness of 7542. In practice Christofides' algorithm will most likely achieve a solution with fitness much lower than 3/2 times the optimum, but there is no theoretical guarantee for this, so the value is still an interesting bound to compare with. Another fitness bound we will compare with is that of a purely greedy search: We will be running an algorithm implemented using the nearest neighbor heuristic with nearest neighbor depth equal to n, meaning for each city it will preprocess all the cities in sorted order with respect to that city; after this it will start at an arbitrary city and choose the closest unvisited city until the tour has been completed. Because the algorithm runs so fast it will be allowed to try and start in all cities and choose the best tour found among these as its final solution. The solution it achieves will be very interesting to compare the algorithms with, since a greedy algorithm is very simple and easy to implement, which was also one of the advantages of metaheuristics. Figure 54 shows the (1+1) EA, simulated annealing and MMAS with optimal settings run on the berlin52 TSP instance as well as the other said fitness values.

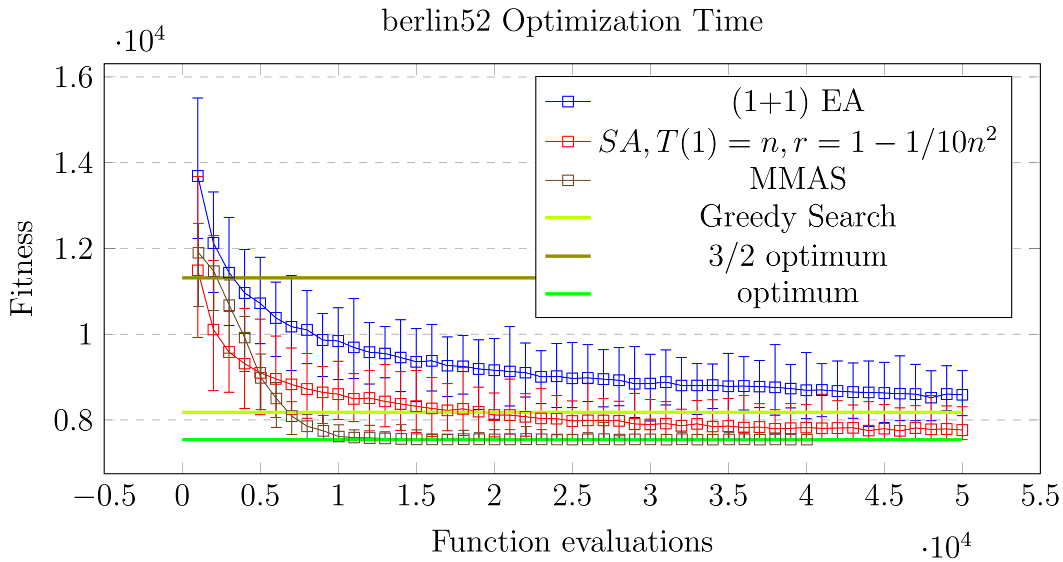


Figure 54: MMAS averaged over 50 runs. (1+1) EA and SA averaged over 100 runs.

As expected MMAS has the best optimization time among the algorithms and is able to in almost all of the 50 runs find the optimum after about 20000 function evaluations.

We notice that after 20000 function evaluations SA is actually also able to beat the greedy search solution and as also noted earlier does find the optimum in some cases as seen by the minimum whisker being on line with the optimum. We will examine the number of times it finds the optimum further down below. Just by looking at this figure it is fair to say that SA with the chosen cooling schedule is doing very well. Especially considering that it has no knowledge of the problem whatsoever apart from using the 2-OPT mutation operator, which is a good mutation operator for the TSP. The greedy search achieves a solution with fitness 8181 while SA after 50000 function evaluations achieves an average fitness of 7766. These values are 8.5 % and 3.0 % from the optimum, respectively, so the test shows that SA is quite a bit better than the greedy search on average.

The (1+1) EA does not perform as well as neither MMAS nor SA. Nevertheless, the results are still remarkable considering its simplicity. It is the simplest among all the algorithms and does not have a lot of parameters that need to carefully be set in order to achieve good results. The parameter $\lambda = 1$ in the Poisson distribution (corresponding to the mutation probability) could be adjusted, but the results on the pseudo-boolean functions showed that the standard mutation probability $p = 1/n$ was a good choice in general, so this is not a parameter one needs to worry about too much for the most part. It is not able to beat the greedy

search algorithm after 50000 iterations corresponding to 50000 function evaluations, but it does come quite close with an average of 8628, which is about 14.4 % from the optimum. Greedy algorithms often give pretty good approximations, so this is not a big set back for the algorithm. If we compare it to the 3/2 OPT bound of Christofides' algorithm on the other hand it is well below this value and does so on average after only 3500 iterations.

We now compare the algorithms on berlin52 with respect to their CPU time. See figure 55.

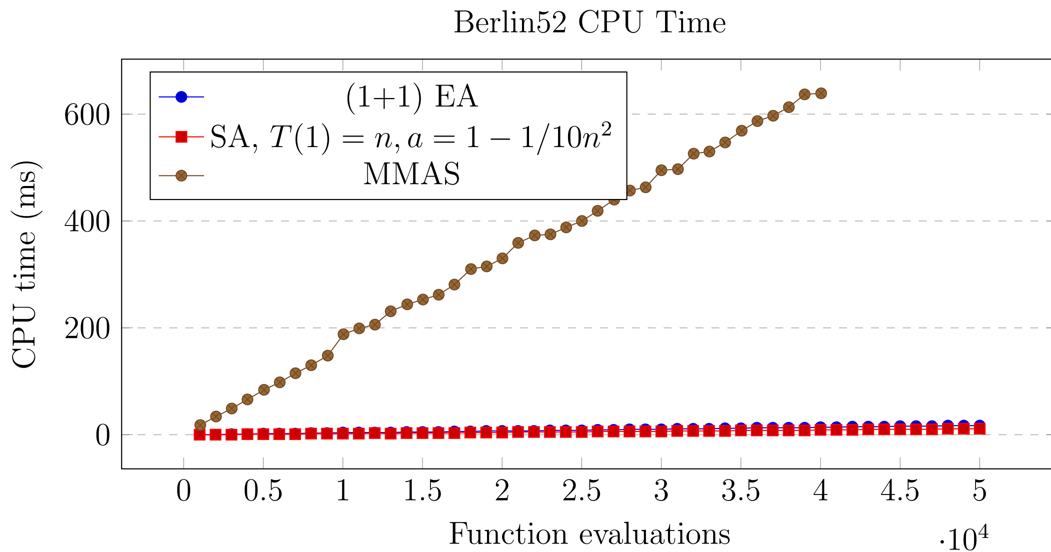
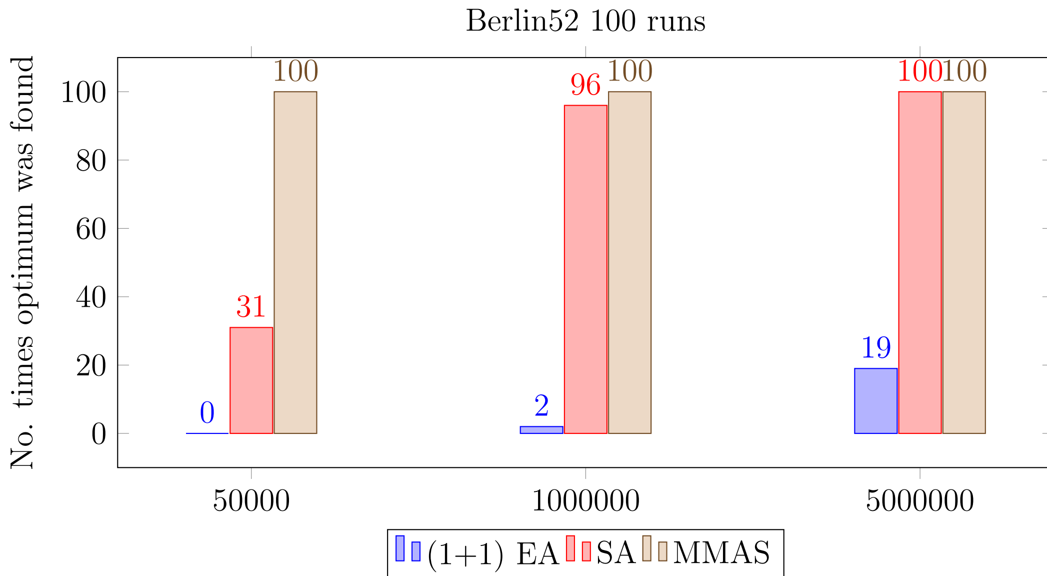


Figure 55: MMAS averaged over 50 runs.

We see that despite MMAS having the fastest optimization time and also being the best to optimize the problem overall after 50000 function evaluations, this is at the expense of higher CPU time compared to the other two algorithms, which are really fast because of the low number of operations per iteration. These results show that in practice the CPU time is very much still relevant when comparing the algorithms. For this reason we end this section with a comparison with a CPU time bound instead of a number of function evaluations.

First, however, we will try to test the algorithms when they are allowed to use a very high number of function evaluations and see how many times they are able to find the optimum. The bar chart below shows how many times the optimum was found for the given number of function evaluations allotted.



The data points for the (1+1) EA and SA are based on the results from the previous figures and tables. For the (1+1) EA the data is from figure 28 and table 4. For SA the data is from figure 35 and table 5. We are able to generate the data for MMAS despite the very high CPU time, because it is able to find the optimum in all the runs we did; the algorithm was allowed to terminate as soon as the optimum was found so in practice it used far fewer function evaluations than it had available. If that would not have been the case we would have to resort to set a CPU time limit instead, since it would take way too long for MMAS to do 1,000,000 function evaluations.

We see that with a very high number of function evaluations SA is able to find the optimum as effectively as MMAS, since it found the optimum in all cases after 5,000,000 function evaluations and almost all runs after 1,000,000 function evaluations. The (1+1) EA is not nearly as effective, but after 5,000,000 function evaluations is still comparable with finding the optimum 19 % of the time.

We end this section with a test of the performance of the algorithms with respect to the CPU time rather than the number of function evaluations. Because the CPU time for MMAS is much higher than for the (1+1) EA and SA it is very interesting to compare the algorithms when we allow each algorithm to run for a certain amount of milliseconds rather than function evaluations and see how good a solution they will be able to find in that time.

We will make a test on the TSP instances berlin52 and bier127 and set the time bound to 750 ms. The reason for choosing a somewhat low time bound is to make it practical to run multiple tests that we will calculate the average, max and min fitness for. Unlike the parameters for the (1+1) EA and MMAS, the cooling schedule for SA is dependent on how many iterations it will run for; since we now

use the CPU time as a bound instead of the number of iterations we cannot readily calculate the best value of the constant c in the cooling factor. However, since each iteration of SA on average should take the same amount of milliseconds for a problem with fixed dimension size n, running the algorithm for 750 ms we can read the amount of iterations it will perform within that time frame; knowing the number of iterations it will approximately perform when it will run out of time we can calculate the constant c like we have done before to get a good cooling schedule. Doing this we get that the number of iterations performed after 750 ms is about 1,890,000. For berlin52 we found in our previous experiments that starting with $T(1) = n$ was better than $T(1) = n^3$, so we will use this starting temperature in test on berlin52. The constant we should use will then be $c = 177$. Using this cooling schedule and the optimal settings for the (1+1) EA and MMAS we get the following result:

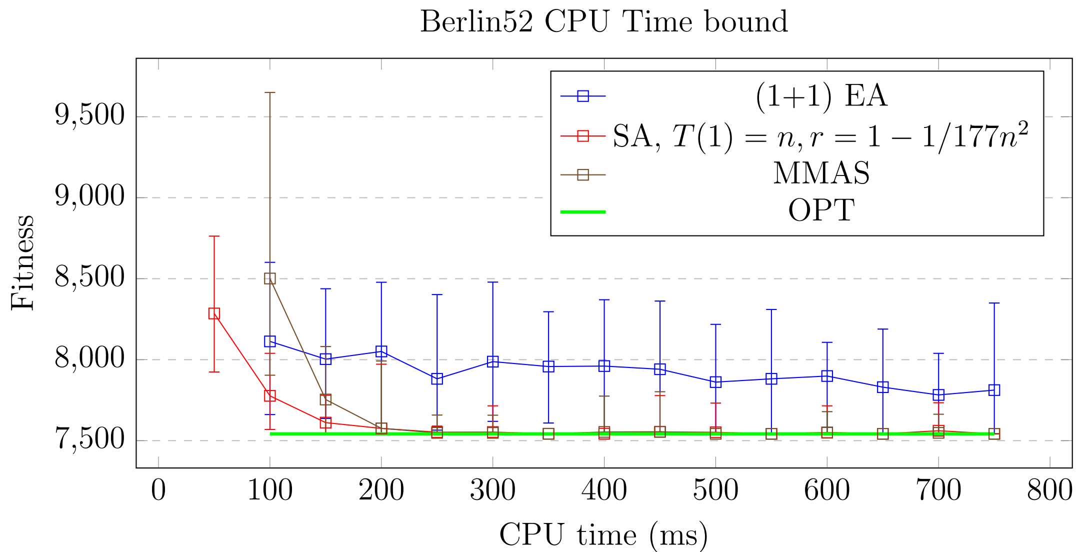


Figure 56: Averaged over 20 runs.

We see that both MMAS and SA are able to find the optimum in all cases seen by the last data point for 750 ms being on the same line as the optimum without any maximum whiskers. The (1+1) EA is as expected not nearly as effective, though we observe that the minimum whisker for the data point at 750 ms is touching the optimum line meaning it found the optimum at least once in the 20 runs it did with this amount of CPU time allotted. Despite MMAS having a much higher CPU time than SA, because its optimization time is so low it still manages to find the optimum very fast. After about 250 ms it is able to find the optimum in almost all 20 runs seen by the average fitness being so close to the optimum. Because we

set the cooling of SA optimal for ending after 750 ms we can not conclude that MMAS is faster at optimizing berlin52 than SA with respect to CPU time. If we want to test this more accurately we would have to use 250 ms as the CPU time bound instead and recalculate a new cooling schedule for this value and see if SA is able to get a better solution after this amount of time compared to MMAS. We will not do that here though and remark that this approach will start to make the test a bit unfair for MMAS where we do not modify the parameters based on how long it will be run for: Despite the fact that the parameters being used for MMAS should be optimal our analysis in section 6.3.3 showed that changing e.g. ρ and m can make the algorithm find good solutions more quickly.

We now run the algorithms on bier127 instead. Like we did for berlin52 we choose an arbitrary value for c in the cooling schedule for SA, run the algorithm for 750 ms, read the average amount of iterations performed at this time and use this to calculate a better value for c. A difference from before though is that we will use a starting temperature of $T(1) = n^3$, which was found to give good results on this TSP instance (refer to table 6). The results are shown in figure 6.4.4 below.

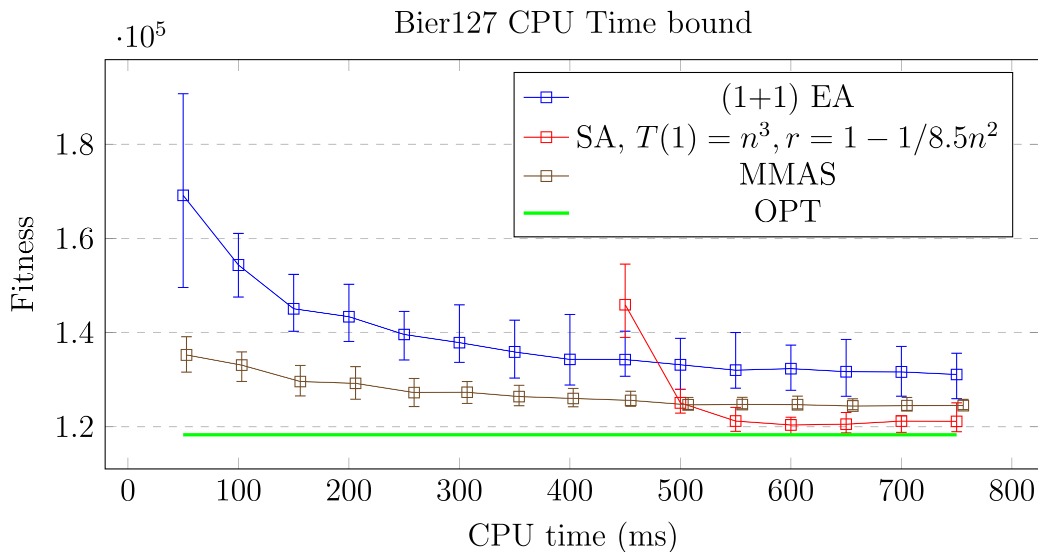


Figure 57: Averaged over 20 runs.

Note that the data points for SA for the CPU time being between 50 ms and 400 ms are not shown since the average fitness is so high that it makes it difficult to see which algorithm is best at 750 ms.

Looking at the plot we see that it is actually SA that is the best performing algorithm among the three in this test. Considering how well MMAS has performed in the other tests we have done on TSP this is a very impressive result for SA.

Furthermore, we also see that for the data point at 600 ms the minimum fitness is equal to the optimum meaning the algorithm has managed to find the optimum in at least one of the 20 runs in only 600 ms. These results seem to indicate that SA is the best performing algorithm with respect to CPU time, though we remark that the testing done here is not extensive. We have not tested how well the algorithms perform when the algorithms run for a higher amount of time and for larger TSP instances, where we might get different results.

Nevertheless, the fact that SA performs so well in both this test and the test on berlin52 shows that even without evidently utilizing problem-specific information like the distance between cities as a heuristic it is possible to achieve very good solutions through the use of a clever strategy for guiding an underlying local search, which is one of the key properties defining a metaheuristic. Admittedly, the 2-OPT mutation operator used in SA works very well for the TSP in that it for example can remove crossings of the edges in the tour, which very often is a good tactic for optimizing the TSP. Having said that, the 2-OPT mutation operator is not an unusual mutation operator in local search for other problems as well, so in conclusion the performance of SA is still very impressive all things considered.

It is not surprising that the (1+1) on TSP is not able to outperform SA and MMAS on TSP despite it like SA also making use of the 2-OPT mutation operator. It is very impressive, however, how well it is actually performing nonetheless considering how simple it is. Based on the simple idea of an evolutionary algorithm with only a single search point in the population, which is mutated over an over, and only replaced if better solutions are found, the (1+1) EA is able to find a solution with an average fitness of 131100 (seen from the plot) after just 750 ms, which is 10.8 % from the optimum of 118282. This is not a bad result for such a low amount of time and could probably be improved even further if it is run for longer.

7 Future Work

Throughout the work with the project many ideas came up relating to more content that could be added, more visualization options, further optimizations and other aspects that could be analysed. It has been necessary to prioritize, however, in order to properly delimit the project and to meet the deadline. Below are listed some ideas for future work concerning varous parts of the project.

7.1 Graphics and Visualization

The graphical user interface in its current state is relatively basic in a many ways and could certainly be changed and extended. As discussed earlier in section 3 it

has not been a high priority of the project to make a very nice looking graphical user interface; the focus was instead on it being intuitive, functional and responsive and that it allowed for the user to choose between the most important options and settings of the program in a straightforward fashion. Still, an effort has been made to customize the layout by styling the buttons and some of the text. Because the program uses the JavaFX stylesheet functionality it is quick to make changes to the layout across the entire application, so it would not take a long time to change the current layout in the future if need be.

One part of the graphical user interface that could be improved in particular would be to add more feedback to user about the current status of the application and if the user tries to something invalid. An example of this is that the program does not state which stopping criteria was met among the ones enabled when the algorithm finishes a run; the user is only told that finished indicated by the *Done* status flag in table in the right-most column. Another thing is that the choice of selection operator and crossover operator has no effect when choosing e.g. the (1+1) EA, so these options ought not to be available when the (1+1) EA is currently selected. Sanitizing the input in the input text fields in setting the stopping criteria and in the batch creation system would be something the program could benefit from as well. These are all minor details though that have not been deemed critical for the user to find the user interface intuitive still. The actions that have been carried out to achieve this are more important and include the simple navigation system, buttons and checkboxes with names that tell what they do and the styling of the buttons that make the them slightly grayed out if clicking them has no effect.

Concerning the visualization options, there are a lot more could be added. As discussed in section 2.6, the FrEAK framework features a lot more visualization options than in the application part of this project, which one could take inspiration from. As mentioned in section 3 and 4 attention has been payed from the start of the project to ensure that the program is modular, flexible and extensible. For this reason it would not be difficult to extend the currently available visualization options or include more alternative ways to visualize the solution construction. It entails some thought that must go into to altering the design and layout slightly to best incorporate this though.

Currently the program only features the boolean hypercube for bit strings and the graph for tsp instances. These visualization tools cover most of the needs together with the table where the current search point can be seen. There is still room for adding more however and extend these in a lot of ways.

7.2 Content

The program features all the elements part of the minimum requirements to the application part of the problem statement and project description, but more

problems, algorithms and visualization options have been added. There are still a lot more interesting problems and algorithms that could be added still though. A particularly interesting feature that would have been interesting to implement and analyze the performance of relative to the algorithms in the project is the generalized partition crossover operator (GPX), which is mentioned briefly in section 2.3. If this operator were to have been implemented it would have been easy to use it as part of the GenericEA algorithm by just choosing this operator as the crossover operator to use in step three in the process of creating a schedule. The implementation is not simple, however, if it is to be done efficiently and the choice was therefore made to invest time in optimizing the other algorithms instead. As seen in figure 27 the operators that have been implemented achieve fair results on the berlin52 TSP instance. Since experiments in the literature show that GPX is one of the most effective crossover operators for the TSP, we may suspect that using the generalized partition crossover would make the GenericEA able to compete much more with the well performing algorithms SA and MMAS.[8]

It would also be interesting to use mutations operators similar to 2-OPT, which all the algorithms that do mutations e.g. the (1+1) EA and SA currently use on the TSP. For example another variant is the 3-OPT mutation operator, which chooses three edges instead of to and reveres the orientations of part of the tour before putting these back again [31]. Since the 2-OPT mutation operator has been well performing when used with e.g. SA in this project it could be interesting to see if using the 3-OPT mutation operator instead or together with the 2-OPT mutation operator could improve the performance even further. It could potentially allow the (1+1) EA and SA have an easier time avoiding local optima.

7.3 Analysis

It quickly became apparent how many aspects, working principles and properties that could be analyzed in this project. An effort has been made to analyze the most prominent figures of the different metaheuristics like fitness as a function of iterations/function evaluations and CPU time as well as some different parameter settings to see what worked well. Concerning parameters there are many more to tweak and only a limited number of combinations have been examined here, so it would be interesting to do more extensive analysis for other parameters combinations. For MMAS for example it would be interesting to analyze the influence and effect of the nearest neighbor depth in the `asDecisionRule` (see section 5.1). How does this affect the solution quality and CPU time of the algorithm on large TSP instances? An effort has been made to test the algorithms with different parameters in a systematic way, but the approach could still be more structured. It would not be far-fetched to imagine that the algorithms coupled with some machine learning strategies for finding the optimal parameters could

lead to very good results. In fact this is an actual field of research that has lead to the term hyperheuristic and metacondition that aims to use clever ways of choosing between a number of heuristics to find which one is the most effective for a given problem.[29]

The visualization tool could have been used more to aid the analysis of the performance of the algorithms on the TSP in particular. By studying how the solutions constructed look it may give some valuable insight what is impeding the algorithm the most in finding the optimum. It may be that an algorithm is particularly susceptible to get stuck at a certain local optima for a given TSP instance and that tweaking the parameters slightly with this in mind can better be avoided.

Instead of only analyzing the performance of the algorithms with respect to their optimization time and the fitness of the solutions they generate with a certain amount of function evaluations or CPU time allotted, other performance metrics could be interesting to look at as well. This could include how much memory is used and things like cache misses etcetera. Although this has not been documented MMAS uses much more memory than the (1+1) EA and SA, which can make the space consumption of the algorithms a factor for very large TSP instances. The choice was made to not delve into such a performance analysis though and focus on the more important performance metrics mentioned before.

Figure 44 showed that MMAS was inefficient on the berlin52 TSP instance with the parameter $\beta = 0$. Dorigo and Stützle acknowledge this fact in their book as well, but show that the search may yet be effective if local search is added. Local search is an optional operation that can take place in between the construction of the tours by the ant agents and the update of the pheromone (see Algorithm 5). The local search could for example be made using a series of 2-OPT mutations like the (1+1) EA and SA uses. Thus, it is possible to make MMAS work on identical terms as the (1+1) EA and SA and it would be very interesting to test how the algorithms would compare in this case. Since SA showed great performance on the TSP instances tried it was not prioritized to make such a test though, since allowing MMAS to utilize some heuristic information about the problem still made for an interesting comparison of the algorithms (refer to section 6.4.4).

7.4 Optimizations

Despite the optimizations described in section 5.4 the program could still be optimized in various ways. In particular, the work behind the scenes of the graphical user interface is not as efficient as it could be. For example, resetting or switching to the next batch when the graphical user interface is used (this does not apply to the program when it is used without the GUI) interrupts the thread controlling the `AlgorithmRunner` instance, creates a new thread and instantiate

all the necessary objects anew. Creating new thread objects is somewhat expensive resource wise, so it would be more efficient to devise a way to reuse the thread that has already been created. Since this does not affect the performance of the algorithms though it has not been a high priority to optimize this part of the program.

Some subroutines in the algorithms still have room for improvement as well; in particular for MMAS. Because the ant agents in MMAS construct a tour in an iteration independently, it is not difficult to add multithreading to this part of the application and run it concurrently. Specifically, one could create a `ThreadPool` and an `ExecutorService` and define the `Task`s to be an ant agent constructing a solution. The threads should synchronize at a *barrier* after all ants have completed their tour. The ant system will then handle the evaporation and pheromone deposit as usual. The benefits of multithreading varies with the system the program is run on; primarily in the number of CPU cores and threads. For the system used for the benchmarking in this project, the i5 CPU features 4 cores and 4 threads, so this could potentially make for a significant improvement in the running time of MMAS, which as the tests show was its biggest weakness compared to the (1+1) EA and SA.

8 Conclusion

A framework for visualizing and evaluating the working principles of nature-inspired metaheuristics has successfully been designed and implemented. Using a graphical user interface, the user of the program is able to create a schedule by choosing among a number of combinatorial optimization problems and algorithms and specify custom parameters for these. The schedule can then be run and the solution construction visualized with either a boolean hypercube or a graph representation depending on the search space. The program features several different pseudo-boolean functions in the search space of bit strings, which include OneMax, LeadingOnes, BinVal, Trap, Jump_k and Needle as well as the traveling salesperson problem (TSP) in the search space of permutations. The algorithms implemented as part of the framework are most notably the (1+1) EA, simulated annealing and Max-Min Ant System. Key figures from running the algorithms are extracted and used to evaluate them through an extensive set of experiments in order to study their performance and working principles.

The tests show that all the algorithms achieve good results in practice on OneMax, LeadingOnes and the TSP that also support the theoretical bounds on the expected optimization time from the literature. The (1+1) EA is the simplest of the algorithms examined, yet very versatile and easily applied to the different problems without the need to carefully tweak its parameters. Simulated annealing on the other hand is very well performing on both OneMax, LeadingOnes and on the travelings salesperson problem, but requires a cooling schedule that is highly dependent on the problem at hand and how long it will be run for in order to be effective. By using the right set of parameters Max-Min Ant System is also performing incredible well, especially for the TSP, though it unlike the (1+1) EA and simulated annealing utilizes some heuristic information by knowing the distance between each pair of cities a priori. The Max-Min Ant System algorithm has the lowest optimization time on the berlin52 TSP instance among three said algorithms, but is significantly worse with respect to CPU time, which is impeding its ability to solve pseudo-boolean functions like OneMax and LeadingOnes for high dimensions.

It has become evident during the work with this project that nature-inspired metaheuristics for the solution of combinatorial optimization problems is an exciting field of research with remarkable tools and strategies to tackle computationally hard problems and many different aspects of both theoretical and pragmatic nature that could be analyzed further. In particular, it would be very interesting to optimize the running time of the Max-Min Ant System algorithm by making it run concurrently using threads. It would also be interesting to compare the performance on the traveling salesperson problem with the (1+1) EA and simulated annealing when it is not allowed to utilize heuristic information, but instead use local search in between the solution construction and pheromone deposit steps.

Emil Lundt Larsen (s153214)
11/6-17

9 References

[1] Anne Auger and Benjamin Doerr. *Theory of randomized search heuristics: Foundations and recent developments*, volume 1. World Scientific, 2011.

[2] Süntje Böttcher, Benjamin Doerr, and Frank Neumann. Optimal fixed and adaptive mutation rates for the leadingones problem. In *International Conference on Parallel Problem Solving from Nature*, pages 1–10. Springer, 2010.

[3] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[4] Duc-Cuong Dang, Tobias Friedrich, Timo Kötzing, Martin S Krejca, Per Kristian Lehre, Pietro S Oliveto, Dirk Sudholt, and Andrew M Sutton. Escaping local optima using crossover with emergent or reinforced diversity. *arXiv preprint arXiv:1608.03123*, 2016.

[5] Benjamin Doerr and Carola Doerr. The impact of random initialization on the runtime of randomized search heuristics. *Algorithmica*, 75(3):529–553, 2016.

[6] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT Press, 2004.

[7] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1-2):51–81, 2002.

[8] Paul Fischer and Carsten Witt. Lecture notes from course 02249 computationally hard problems version 2.68. DTU Compute, 2017.

[9] Ruprecht Karls Universität Heidelberg. Tsplib. `https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`. Accessed: June 2018.

[10] Thomas Jansen. *Analyzing evolutionary algorithms: The computer science perspective*. Springer Science & Business Media, 2013.

[11] Thomas Jansen and Ingo Wegener. On the local performance of simulated annealing and the (1+ 1) evolutionary algorithm. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 469–476. ACM, 2006.

[12] Thomas Jansen and Ingo Wegener. A comparison of simulated annealing with a simple evolutionary algorithm on pseudo-boolean functions of unitation. *Theoretical Computer Science*, 386(1-2):73–93, 2007.

[13] Thomas Jansen, Ingo Wegener, and Project Group 427. Freak – the free evolutionary algorithm kit. University of Dortmund, 2003.

[14] Thomas Jansen and Christine Zarges. Analysis of evolutionary algorithms: From computational complexity analysis to algorithm engineering. In *Proceedings of the 11th workshop proceedings on Foundations of genetic algorithms*, pages 1–14. ACM, 2011.

[15] Alexander Dahl Juhl. Visualising and evaluating the working principles of nature-inspired optimisation metaheuristics. DTU Compute, 2017.

[16] Timo Kötzing, Frank Neumann, Heiko Röglin, and Carsten Witt. Theoretical analysis of two aco approaches for the traveling salesman problem. *Swarm Intelligence*, 6(1):1–21, 2012.

[17] Timo Kötzing, Frank Neumann, Dirk Sudholt, and Markus Wagner. Simple max-min ant systems and the optimization of linear pseudo-boolean functions. In *Proceedings of the 11th workshop proceedings on Foundations of genetic algorithms*, pages 209–218. ACM, 2011.

[18] P. L'Ecuyer. SSJ: Stochastic simulation in Java, software library. DIRO, Université de Montréal, 2016. `http://simul.iro.umontreal.ca/ssj/`.

[19] P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: A framework for stochastic simulation in Java. In E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 234–242. IEEE Press, 2002. Available at `http://simul.iro.umontreal.ca/ssj/indexe.html`.

[20] Klaus Meer. Simulated annealing versus metropolis for a tsp instance. *Information Processing Letters*, 104(6):216–219, 2007.

[21] Frank Neumann, Dirk Sudholt, and Carsten Witt. Analysis of different mmas aco algorithms on unimodal functions and plateaus. *Swarm Intelligence*, 3(1):35–68, 2009.

[22] Frank Neumann and Carsten Witt. Bioinspired computation in combinatorial optimization: Algorithms and their computational complexity. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 567–590. ACM, 2013.

[23] Oracle. Javafx. http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html. Accessed: June 2018.

[24] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[25] Jens Scharnow, Karsten Tinnefeld, and Ingo Wegener. The analysis of evolutionary algorithms on sorting and shortest paths problems. *Journal of Mathematical Modelling and Algorithms*, 3(4):349–366, 2005.

[26] Ian Sommerville. Software engineering 9th edition. *Person Education Ltd*, 2011.

[27] Chaoxu Tong. Lecture notes - orie 6300 mathematical programming i. https://people.orie.cornell.edu/dpw/orie6300/Recitations/Rec12.pdf, 2014. Accessed: June 2018.

[28] Ingo Wegener. Simulated annealing beats metropolis in combinatorial optimization. In *International Colloquium on Automata, Languages, and Programming*, pages 589–601. Springer, 2005.

[29] Wikipedia. Hyper-heuristic. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: June 2018.

[30] Carsten Witt. Tight bounds on the optimization time of a randomized search heuristic on linear functions. *Combinatorics, Probability and Computing*, 22(2):294–318, 2013.

[31] Yuren Zhou. Runtime analysis of an ant colony optimization algorithm for tsp instances. *IEEE Transactions on Evolutionary Computation*, 13(5):1083–1092, 2009.

10 Appendix

10.1 Miscellaneous

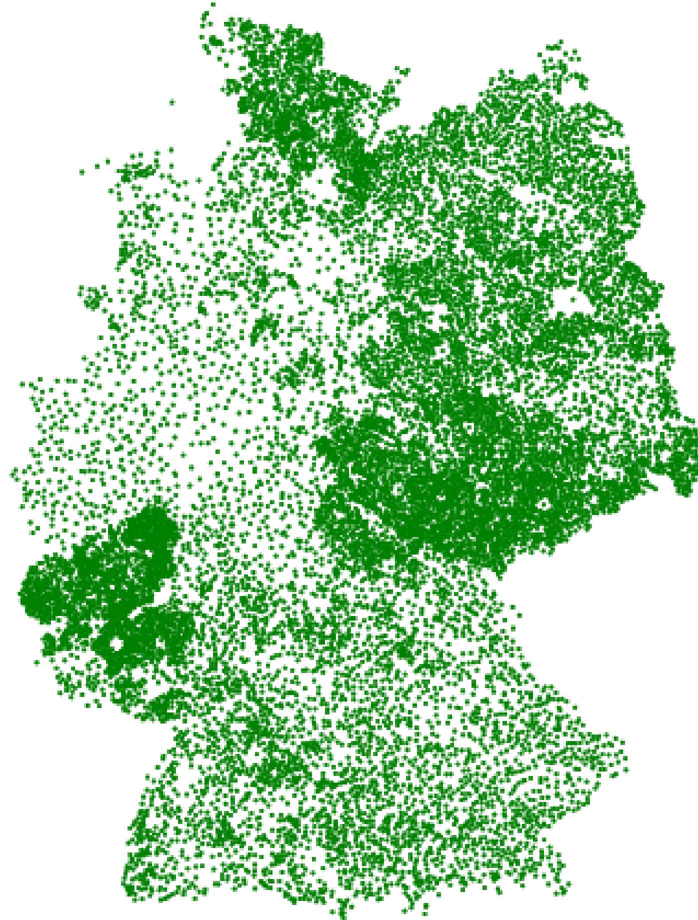


Figure 58: The d15112 TSP instance with 15112 cities in Germany from TSPLIB [9]. The cities are drawn using the visualization framework, but the radius of the dots has been decreased to be able to better render the instance. If the instance is used in the program the dots will have higher radius and the positions of the cities will not be as easily differentiated.

10.2 Program User Guide

This section provides a brief guide explaining how the framework of the project is used and details what options and settings are available. To better illustrate the usage of the program we will give a concrete example of a use case, where the user creates a schedule for simulated annealing on LeadingOnes.

Initially the search space should be selected with the options being either bit strings or permutations. The user selects an option by clicking on it in the table and then pressing the *next button*. Just before clicking the next button it should look like this:

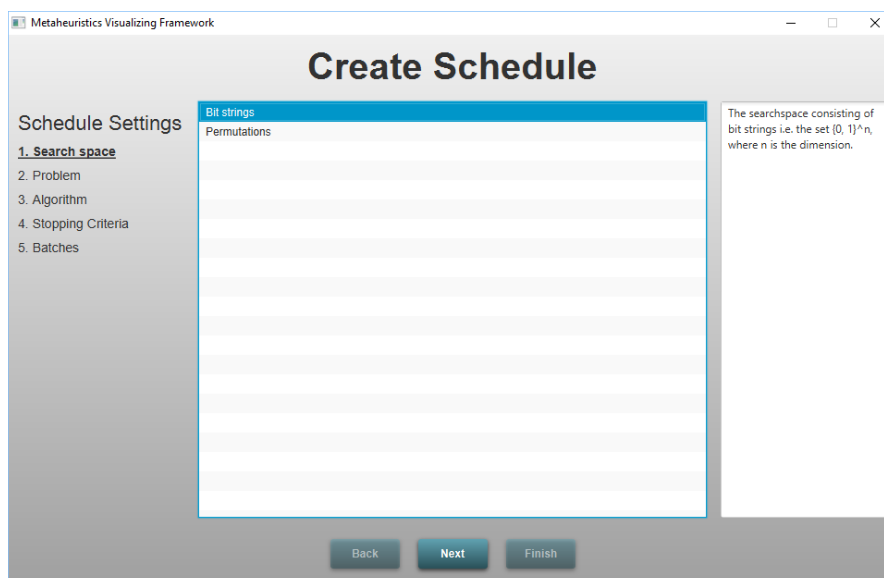


Figure 59: Step 1

Having chosen the bit string search space all the available pseudo-boolean functions (problems) are shown in the table in the center. These include OneMax, LeadingOnes, BinVal, Trap, Jump_k, Needle and f_2 (refer to section 2.2.1). Note that currently Jump_k is equivalent to Jump_2, since $k = 2$ internally in the program and is not adjustable as of this moment; the value can be changed within the program if need be. Had the user chosen the permutation option in the previous step instead the available problems would only just be the traveling salesperson problem (TSP). The user selects the second option, which is LeadingOnes, and presses the next button (see figure 60). At the third schedule creation step the user is presented with all the algorithms available. The options for the algorithms are: (1+1) EA, RLS Generic EA, simulated annealing, Ant System, Max-Min Ant System. All of these algorithms are available regardless of the user having chosen a bit string problem or the TSP. For the TSP, the user is also able to choose the

greedy search algorithm. A table with selection operators, crossover operators and mutation operators are shown below. The options are for selection, uniform and tournament selection, for crossover the user can choose between OX and PMX and for mutation the user can choose swap and bit flip or 2-OPT depending on whether bit strings or permutations were chosen. Note as already mentioned in section 4.2 that the (1+1) EA, RLS and SA always use the *bit flip* mutations operator for bit strings and *2-OPT* for TSP. Continuing our example of a use case, the user chooses the simulated annealing option. At this point the screen will look like figure 61.

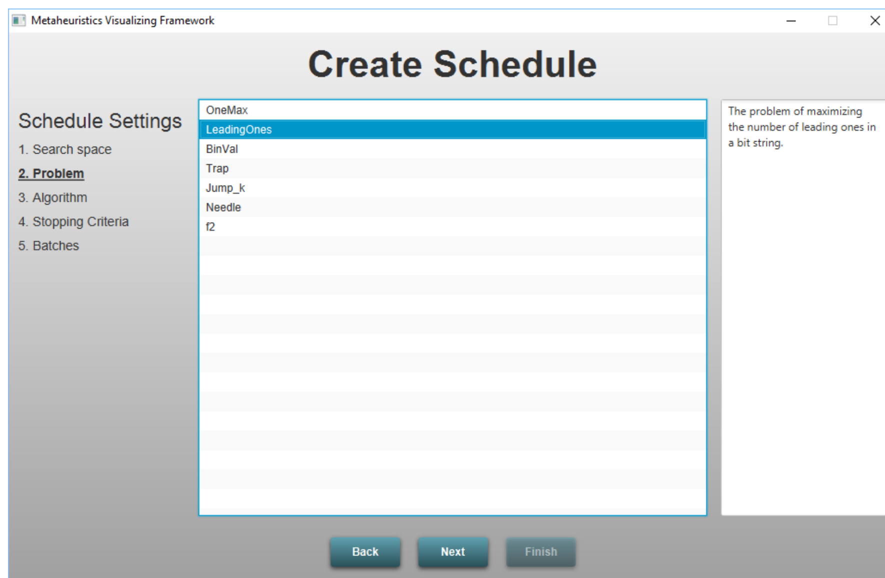


Figure 60: Step 2

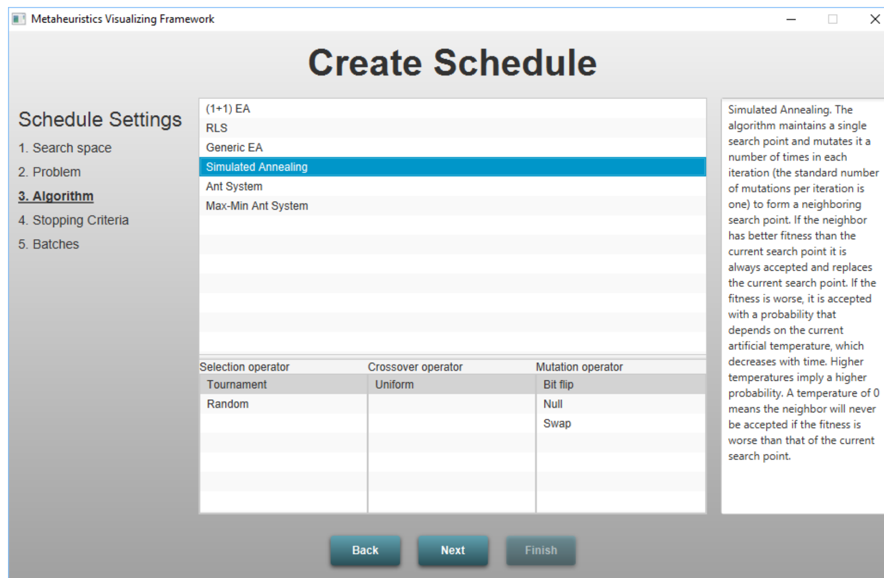


Figure 61: Step 3

At the forth creation step the stopping criteria can be set. These include bounds for the fitness, iterations, fitness function evaluations and stagnation; if one of these options is chosen the algorithm will finish the current run as soon as the number inputted is reached or exceeded during the course of running it. The stagnation bound means that the algorithm stops if the number of consecutive iterations without improvement in fitness of the best solution found so far is exceeded. In order to customize the stopping criteria, the user checks the checkboxes next to the stopping criterion he or she wishes to be active and then fills in the appropriate value in the input text fields next to them, which is done by simply typing in the desired value. Only integers may be written in these text fields; other types of input e.g. a string with literals will cause an exception, though the value may still afterwards be changed to a valid integer value and the program can proceed to function normally. In our example, the user wants to set a limit on the number of iterations the algorithm is run, so (s)he clicks on the checkbox named *iteration bound* and then types in a value of 1000 in the text field right next to it. The situation is shown in figure 62. The user then presses next to get to the final step.

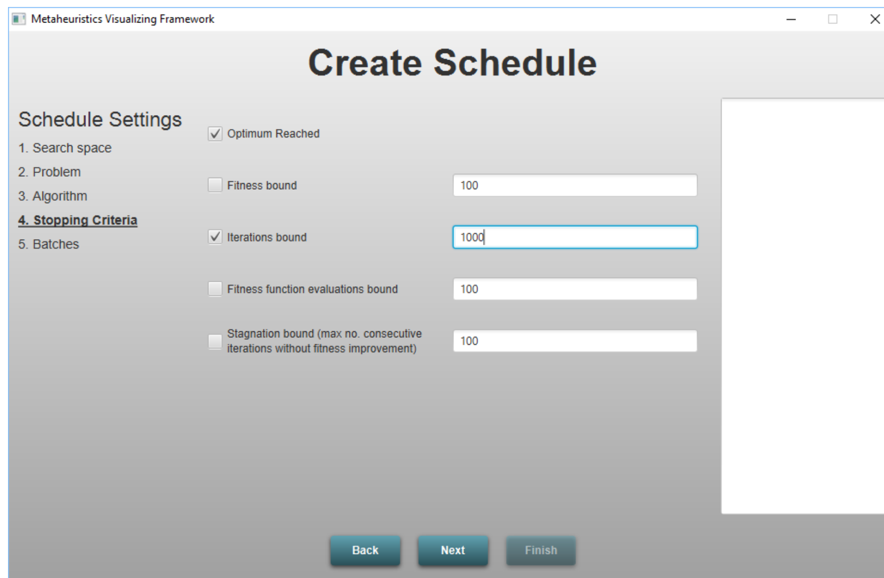


Figure 62: Step 4

At the last step the user can set up the batches of runs. Initially a single batch is shown in the table in the middle of the screen with some default values (see figure 63). In order to change an existing batch the user clicks the table cell (s)he wishes to change and then types in the new value and hits enter to complete the change. If the user does not hit enter after typing in the new value the edit will not take effect. By pressing anywhere on a row marking an existing batch this batch will be selected; the user may then delete this batch by clicking the delete button or duplicate it by clicking the copy button. The new button creates a new batch with the default values filled in like the one there was present initially. Below the three buttons are two checkboxes and also a drop down menu if the TSP was chosen earlier on as the problem, which allows the user to specify the TSP instance to use. If the checkbox labeled "use optimal algorithm settings" is ticked the parameters for all batches will be based on some special values set internally within the program and will ignore the values the user has set in the batches; the number of runs and dimension will still be kept though; it is only for the algorithm settings. The checkbox "Run in console mode" does as it says and will print a summary of the results to the terminal when a batch completes. Furthermore, a file will be created if it does not exist at where the program is located where some parts of the summary will be stored.

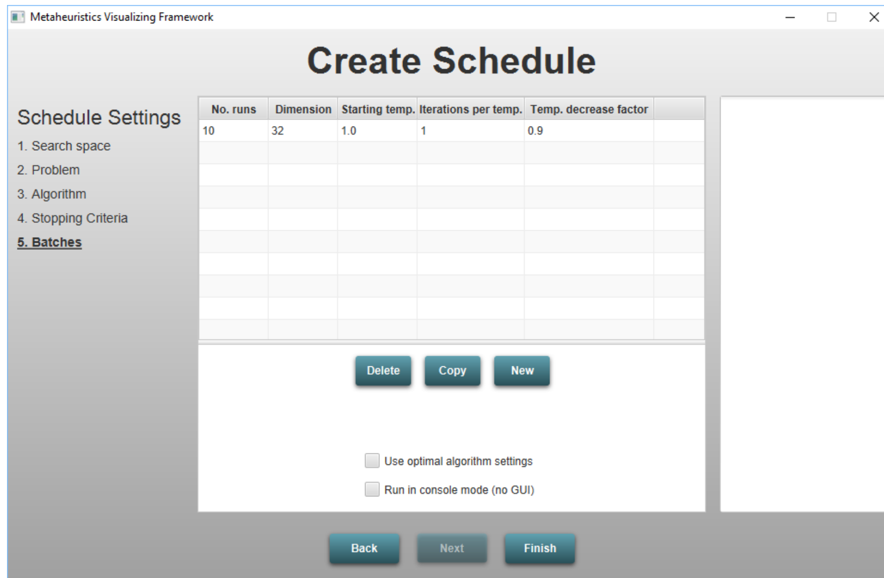


Figure 63: Step 5. Screen as it is initially shown.

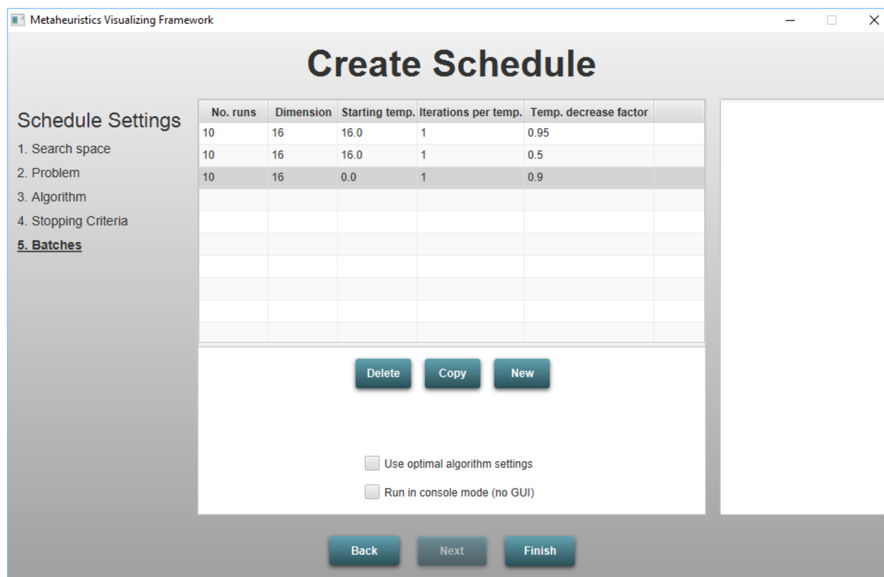


Figure 64: Step 5. Screen as it is shown after the user has finished setting up three custom batches.

In our example we assume the user has the intention of testing the effect of starting with different temperatures and temperature decreasing factors. To test this, the user creates three batches and changes the values to end up with the

configuarition in figure 63. The user then presses finish, which makes the program switch to the run scene, which will initially look like what is shown in figure 65.

Figure 65: Run Scene as it is initially shown.

Starting from the left with the navigation buttons in the bottom of the screen we have the *back to menu* button. This button will exit the run scene and go back to the first page of create schedule scene again. If the program is running when the button is pressed it will terminate, so the user need not worry about the program using resources and computation power in the background unnecessarily. To the right of this we have the main control buttons. In the example these are all initially click-able. The reset batch will set the current batch back to its initial state, meaning that all the statistics gathered so far will be discarded and the run number will be reset to 1. The *pause* and *start* buttons pause and start the application, respectively. The *next run* button allows the user to skip the current run and advance the run number counter by 1; this may be useful if the user is not interested in letting the current run finish for whatever reason. Since the current run will not finish as a result of clicking this button, the statistics will not take that run into account. The *next batch* button goes to the next batch, where the user has to then press the start button again in order to start the application.

Continuing right along the bottom of the screen we have a slider for controlling the speed. Initially the slider is set in the middle, which corresponds to running one iteration of the algorithm every $0.5 \cdot 0.5$ seconds (plus the time it takes to execute the iteration). The granularity of the slider increments is set to 0.05 with the min value being 0.01 and the max being 0.99. Setting the slider in the left

most configuration means the application will do an iteration every $0.5 \cdot (1 - 0.01)$ seconds while the right most configuration means it will update every $0.5 \cdot (1 - 0.99)$ seconds excluding the time it takes for the iteration to finish.

At the bottom right of the screen all the auxiliary settings are placed. The first three checkboxes will always be present no matter what problem or algorithm has been chosen. The first checkbox *stop at the end of each run* will make the program stop when a run is finished and wait for the user to press the *next run* button before continuing. The *expand table* checkbox makes the `run table` expand upwards to show five more rows at a time. This is only useful for the GenericEA, which usually contains more than one search point in the population. The scroll bars have been disabled in the table, but the user can still scroll up and down using the mouse wheel if (s)he wishes to view the other search points that are not in the first five rows. The *stepwise mode* checkbox toggles the step wise mode on and off. In stepwise mode the program will only run one iteration at a time before waiting for further user input; before the program continues the user must click the next iteration button after which the program will run a single iteration before the user has to press the next iteration button again. To the right of the three checkboxes are a number of checkboxes that depend on the chosen problem and algorithm. Below is a list of all the checkboxes that can show up here along with a description of them:

- *Show optimal tour*: For some TSP instances it is possible to show the optimal tour on top of the usual visualization of the algorithm running. The lines in the optimal tour are shown with thick green lines. It is important to stress that it is only for the TSP instances berlin52 and tsp225 that this option is available. The reason for this is that the TSPLIB website only had files with an optimal tour for these two instances.

- *Show city numbers*: For TSP instances the city numbers can be shown above the corresponding city. Note that the city numbers will be 0-indexed even though the cities are 1-indexed in the files.

- *Display temperature*: For simulated annealing this option displays the current temperature in the top left corner of the canvas.

- *Display pheromone trails*: For AS and MMAS on TSP this option displays the pheromone trails along the edges. An example of this was shown in figure 4.2.3. The pheromone is drawn with orange lines, where the opacity is calculated based on the pheromone value relative to the highest amount of pheromone deposited so far.

- *Display iteration-best sol.*: For AS and MMAS on TSP turning thos option on will make the iteration-best solution be rendered with red lines on top

of the best-so-far solution which is always rendered with the standard blue color.

In the left-hand side of the screen the user can see the information and statistics of the current batch. The information will always include the problem, algorithm and which operators are used; if the operators are not important for the selected algorithm it will say "NA" next to them. Below the general settings are listed the *batch number*, *run number* and *dimension* of the problem. For the *batch number* and *run number* the first number next to them is the current run number; the number after the slash-symbol is the number of batches or runs in total, which the user set during the batch creation step when creating the schedule. The number of batches will stay the same, but the total number of runs will update when a new batch is initiated. The dimension will for pseudo-boolean functions be the number of bits in the bit strings and for TSP it will be the number of cities in the TSP instance used.

Below this are the statistics of the current batch. These statistics will always be shown regardless of which algorithm and problem have been chosen. The statistics will update each time a run fully completes, that is, when one of the stopping criteria the user has is met. If the user skips to the next run using the next run button the statistics will not update.

In the middle of the screen are a canvas and the *run table*. The table has five columns that starting from the left display the current iteration, number of fitness function evaluations, fitness of the current search point, search point value and status. The status will attain the values, `Ready`, `Running` and `Done` depending on what the current status of running the algorithm is. `Ready` means the current batch has been initialized and the algorithm is ready to run. `Running` means the batch is currently being processed; for this reason it will say running regardless of the program currently being paused or not as long as the start button has been clicked at least once for the current batch. `Done` means the batch has finished the last run.