Visualizing and evaluating the working principles of nature-inspired optimization metaheuristics

Adam Ømosegård Bischoff
s174295

Kongens Lyngby 2020

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

One of the great questions in computer science is how to efficiently find solutions to hard problems. One way of doing so is through metaheuristics that are inspired by how nature solves similar problems. An example of this is how evolution has generated organisms better at surviving, or how efficiently ants find paths between food and their anthill. This project will visualize the workings of the metaheuristics *(1+1) Evolutionary Algorithm*, *Random Local Search*, *Simulated Annealing* and *Min-Max Ant System*, and evaluate their performance. These metaheuristics solve the problems *OneMax* and *LeadingOnes* on the bit string search space, and the *Traveling Salesman Problem* on the permutation search space. In addition, three heuristics are implemented that can be used for creating the initial tour for the Traveling Salesman Problem, which is then iterated on further by the metaheuristic. The heuristics implemented and evaluated are *Nearest Neighbor*, *Cheapest Insertion* and *Random Insertion*. This is achieved by creating a flexible, modular and extensible framework with which new problems, metaheuristics and visualizations can easily be added. The fastest of the four metaheuristic is found to be Random Local Search. This is due to its sheer simplicity, which comes at the cost of finding comparatively worse solutions for hard optimization problems. Simulated Annealing is found to generate the best solutions for hard problems, and does so without increasing the running time excessively. (1+1) Evolutionary Algorithm can quickly find a *good* solution which can be improved indefinitely, though with diminishing gains. Min-Max Ant System is seen to produce good results, but its long running time often makes it unfit for use. An interesting observation is that Random Local Search achieves significantly better results when first initialized with the Nearest Neighbor heuristic, while Cheapest insertion decreases the quality of the tour found compared to a random initialization. This is in spite of the fact that Nearest neighbor in itself produces worse tours than the two other heuristics.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring a B.Sc. in Software Technology.

Lyngby, 28-June-2020

Adam Ømosegård Bischoff
s174295

Acknowledgements

I would like to thank my supervisor Carsten Witt for help and guidance throughout the project, in spite of the accompanying difficulties from the university being locked down due to the COVID-19 outbreak.

Contents

Chapter 1

Introduction

In computer science, there are several problems where an *efficient*[1] algorithm is yet to be found. An example of this is the *Travelling Salesman Problem (TSP)*. In this problem you are given a complete weighted graph of n vertices. A solution to this problem is any Hamiltonian cycle, meaning a cycle visiting every vertex in the graph exactly once. A solution is considered good if it has a low total weight. The only known way of finding the optimal solution is comparing every possible permutation of vertices and returning the smallest one. There are $n!$ permutations. This can be sped up significantly using dynamic programming, but the asymptotic running time is still exponential $O(2^n n^2)$ [HK61].

Since exact algorithms are exponential (and thus often infeasible) for problems like TSP, we will consider a set of algorithms called *metaheuristics*. These algorithms sacrifice accuracy for speed, while the hope is that the returned solution is *good enough*. This means these algorithms will have no guarantee of finding the optimal solution, but finds it much faster. These algorithms are characterized as being non-problem specific strategies that efficiently guide an exploration of the search space [BR03]. This means that they iterate over one or more solutions, usually improving them for each step (decided by a score calculated by the given problem). In this project, I will primarily work with a set of metaheuristics inspired by nature, e.g. evolution or how ant colonies find

[1] In this context, an *efficient* algorithm is one with asymptotic polynomial running time complexity

paths between food and their anthill. These metaheuristics will be evaluated using a set of problems, and the efficiency of each will be evaluated.

Chapter 2

Theory

This chapter describes the theory used throughout the project. This consists of central components used throughout the project, and theory needed to understand this.

2.1 Graphs

A graph consists of a set of points, called *vertices*, which can be connected by *edges* [BM76, 1st chapter]. Graphs are useful for describing many real-world situations. An example might be representing people as vertices and connecting those that are friends by edges.

Mathematically, a graph G is an ordered triple

$$G = (V(G), E(G), \psi_G) \tag{2.1}$$

consisting of a set of vertices $V(G)$, a set of edges $E(G)$ and an *incidence function* ψ_G, associating each edge with its attached vertices [BM76, 1st chapter]. Take for example the graph in figure 2.1. As labeled on the graph, $V(G)$ and $E(G)$ are defined as:

$$V(G) = \{v1, v2, v3, v4, v5\} \tag{2.2}$$

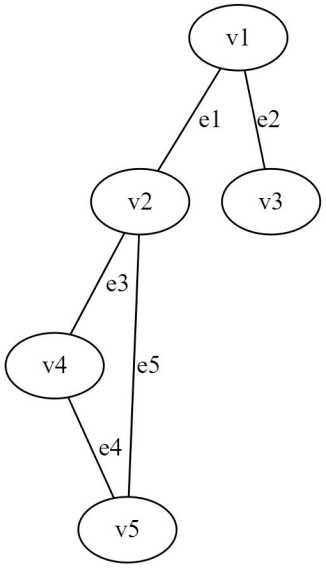


Figure 2.1: Example of a graph

$$E(G) = \{e1, e2, e3, e4, e5\} \tag{2.3}$$

And the incidence function ψ_G is defined as:

$$\psi_G(e1) = v1v2, \psi_G(e2) = v1v3, \psi_G(e3) = v2v4, \psi_G(e4) = v4v5, \psi_G(e5) = v2v5 \tag{2.4}$$

Edges may either be directed (only going from one vertex to another) or undirected. A directed edge is usually drawn as an arrow.

A simple graph is a graph where no edge starts and ends in the same vertex (loop), and no two edges join the same two vertices (multiple edge). [BM76, 1st chapter]

A complete graph is a simple graph in which all pairs of distinct vertices are connected by edges. [BM76, 1st chapter]

A weighted graph is a graph where each edge e is associated a real number $w(e)$ called its *weight*. In the example of a graph representing friendships, each edge might be given a weight referring to the strength of their friendship. [BM76, 1st chapter]

A walk in a graph G is a non-empty sequence $W = v_0e_1v_1e_2v_2 \dots e_nv_n$ of

alternately vertices and edges. [BM76, 1st chapter]

A trail (or edge sequence) is the sequence of edges in a walk, in which every edge is distinct. [BM76, 1st chapter]

A path is the sequence of vertices in a walk, in which every vertex is distinct. [BM76, 1st chapter]

A connected graph is a graph where for any set of vertices (u, v), there exists a path containing both u and v. [BM76, 1st chapter]

A cycle is a trail that begins and ends in the same vertex. [BM76, 1st chapter]

A Hamilton cycle is a cycle containing every vertex in the graph. [BM76, 4th chapter]

2.2 Search spaces

A search space is the set of possible states/solutions for a given problem. How good a state is is determined by a fitness score given by the optimization problem (section 2.3). This state is then iteratively changed and (hopefully) improved by the metaheuristic (section 2.4), until it is eventually stopped. I will now explain the two search spaces used in this project.

2.2.1 Bit strings

One of the two search spaces explored in this project is bit strings. Simply put, a bit string is a sequence of n bits, each bit being either 1 or 0. A sequence of bits is often times referred to as a *string* of length n. A string s of lengths n is thus an element in the set

$$s \in \{0, 1\}^n \tag{2.5}$$

For instance, having a search space of bit strings with $n = 5$, two possible strings are “00101” and “11111”.

Since any combination of n bits are in the search space, the amount of strings S grows exponentially with regards to n, more precisely

$$S = 2^n \tag{2.6}$$

Having an exponentially growing search space makes an exhaustive search, where every possible solution is considered, practically infeasible for larger values of n.

A bit string is often modified by *flipping* one or more bits. Flipping a bit b means that if b is 1, we change it to 0, and if b is 0, we change it to 1. In other words

$$b \leftarrow 1 - b. \tag{2.7}$$

2.2.2 Permutations

A permutation is a unique sequence of elements in a set. An example might be a deck of playing cards, and how they are ordered in the deck. Here, we are not interested in *which* cards are in the deck, but rather *how* the cards are ordered. A perfectly ordered sequence of cards (Ace of spades, 2 of spaces, ..., Queen of clubs, King of clubs) is a permutation of the deck. However, if only one card is moved to a different location in the deck, it becomes a new permutation.

Assume we have a set with n unique elements, and we want to know how many permutations are possible. There are n possible elements to pick as the first in the sequence. Since the same element cannot be picked again, there are $n - 1$ elements to consider as the second one to add. This continues until all elements have been placed. This makes the amount of permutations $n!$:

$$(n) \cdot (n-1) \cdot (n-2) \cdot (\dots) \cdot 1 = n! \tag{2.8}$$

Thus, in the example of the deck of playing cards, there are $52! \approx 8.07 \cdot 10^{67}$ permutations. Similarly to bit strings, this growth rate makes an exhaustive search practically infeasible, even with a relatively small amount of elements.

2.2.2.1 Modifying a permutation

There are several methods one can use to modify a permutation. For two elements in the permutation i and j, three simple operations are: [STW05]

- `exchange(i,j)`: Swaps the position of i and j.
- `jump(i,j)`: Moves i to the position of j, shifting all elements between them by 1 in the appropriate direction.
- `reverse(i,j)`: If i appears before j, reverses the ordering of all elements between i and j.

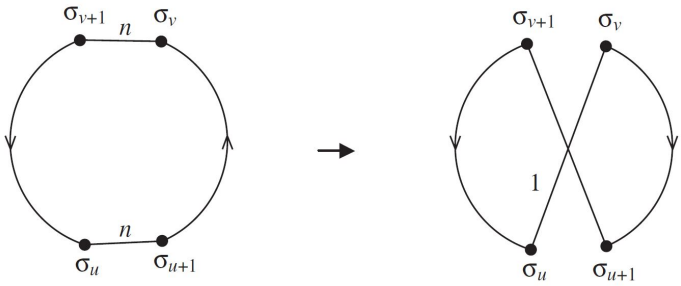


Figure 2.2: A visual representation of the 2-OPT move performed on the edges (σ_u, σ_{u+1}) and (σ_v, σ_{v+1}). [Zho09]

In this project, however, I will be using the operation called the *2-OPT* move, as this usually provides better results for the permutation problem explored. 2-OPT primarily differs from the operations mentioned above in that it considers the permutation as a graph where the edges are modified, as opposed to the vertices.

To best explain 2-OPT, we imagine the complete graph G, each element being a unique vertex. A permutation is thus the edge sequence of a Hamilton cycle in G, i.e. an edge $e = (u, v) \in G$ means u comes directly before v in the permutation[1]. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ be a permutation. This gives us the edges $E = \{(\sigma_1, \sigma_2), (\sigma_2, \sigma_3), \ldots, (\sigma_n, \sigma_1)\}$.

A *2-OPT move* is the process in which two edges connect crosswise. If we perform a 2-OPT move by crossing the edges (σ_u, σ_{u+1}) and (σ_v, σ_{v+1}) in the permutation σ, the resulting permutation is: [Zho09]

$$\sigma' = (\sigma_1, \ldots, \sigma_u, \sigma_v, \sigma_{v-1}, \ldots, \sigma_{u+1}, \sigma_{v+1}, \ldots, \sigma_n) \tag{2.9}$$

Informally, we reverse the elements in the permutation from σ_{u+1} to σ_v. If considering the graph visually, we connect the edges (σ_u, σ_{u+1}) and (σ_v, σ_{v+1}) crosswise, and reverse the direction of all edges in between. This is illustrated on figure 2.2. There are different definitions of which edges can be selected for a 2-OPT move. If the the edges chosen are the same, the resulting permutation would be reversed, and if $\sigma_{u+1} = \sigma_v$, the permutation will not change at all. One could choose to exclude these, but for simplicity, all edges selected for 2-OPT moves in this project are valid.

[1]Note that the last element in the permutation σ_n also has an edge leading to the first σ_1. I.e. the edge sequence contains the edge (σ_n, σ_0).

2.3 Optimization problems

An optimization problem, later just referred to as a *problem*, is specific to a search space. It evaluates how good a given state is by assigning it a fitness score.

2.3.1 OneMax

OneMax is an optimization problem in the bit string search space. The score is simply the number of 1's in the string. For a string x, this is mathematically defined as: [Sud10]

$$\textsc{OneMax}(x) = \sum_{i=1}^{n} x_i \tag{2.10}$$

2.3.2 LeadingOnes

LeadingOnes is another optimization problem in the bit string search space. Here, the score is the number of consecutive 1's in the string, reading from left to right. For a string x, this is mathematically defined as: [BDN10]

$$\textsc{LeadingOnes}(x) = \sum_{i=1}^{n} \prod_{j=1}^{i} x_j \tag{2.11}$$

2.3.3 Symmetric TSP

The "traveling salesman problem" (or TSP for short) is an optimization problem in the permutations search space. The name comes from the idea of a traveling salesman who wants to visit every city in a set of cities, starting and ending in the same city, while minimizing the distance he needs to travel while doing so. A point in this search space is thus a tour, being a permutation of cities. A city cannot be visited twice unless it is the starting city, and the starting city must also be the ending city. [BM76]

This problem is *symmetric* as opposed to *asymmetric*, in that the distance from city a to b is equal to the distance from b to a. It is implied that any further mention of TSP refers to "symmetric TSP". The distance between two cities is

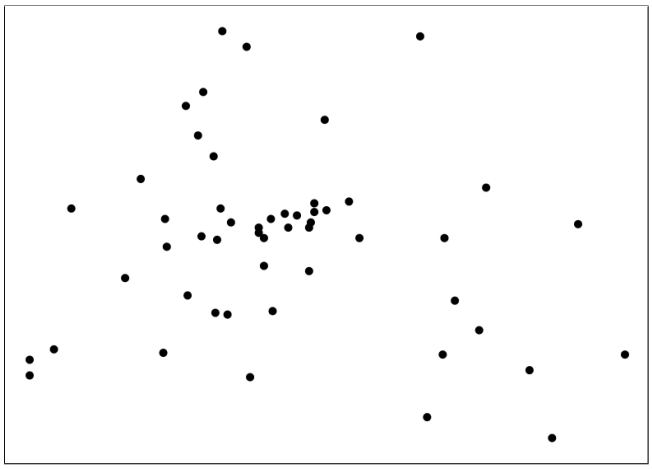


Figure 2.3: berlin52.tsp from TSPLIB

calculated as the euclidean distance between them. The score of a tour is thus the sum of distances in it.

TSP is often represented as a *weighed, complete graph*, with each city being a vertex, and the weight of each edge being the *distance* between its cities. A point in this space is then any Hamilton path, often called a *tour*, and the aim is to find a minimum-weight Hamilton Path. [BM76, 4^{th} chapter]

TSP is interesting, as we have yet to find an efficient way of returning the optimal tour. The best known method is exhaustively searching the entire search space, but using dynamic programming. This improves the running time from $O(n!)$ to $O(2^n n^2)$, which is still exponential, but significantly better [HK61]. Using metaheuristics, this running time is expected to be at most polynomial in time complexity. However, this will come at the price of accuracy in that the tour is thus not guaranteed to be optimal.

The TSP instances worked with in this project all come from the TSPLIB library [Rei97]. There are multiple types of instances. The type worked with in this project is `EUC_2D`. Each instance is a separate file, containing the placement of each city in a two-dimensional plane. An example of an instances from TSPLIB is `berlin52.tsp`, which is shown in figure 2.3, each city being a dot (vertex). Note that there are no edges on the figure, even though the instance is a complete graph. These is intentionally left out as they would clutter the visualization. We will later see that edges drawn on the graph represent a tour through the instance.

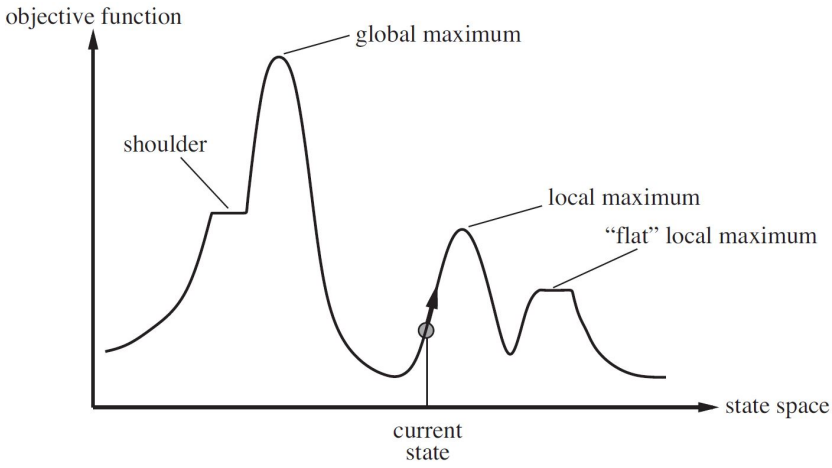


Figure 2.4: A state-space landscape identifying key locations [RN09, section 4.1]

2.3.4 Global and local optima

Consider a simple search strategy (metaheuristic) which simply moves to a neighbor if its fitness score is better. A neighbor is a state which can be reached with a single move by the search strategy. To understand global and local optima, we consider a *state-space landscape*. This landscape is a 2-dimensional representation of a search space in which the x-axis represents different states in the space, and the y-axis represents the fitness score of that given state. Two states are placed next to each other if they are neighbors[2]. On figure 2.4, this landscape is illustrated. We now consider each *peak* in this landscape. If a peak has the highest fitness score of all states, it is the global maximum. If some other state has a higher fitness score, the peak is a local maximum. [RN09, section 4.1]

On figure 2.4, two special locations are identified where all neighboring states are equally good, namely the *shoulder* and *"flat" local maximum*, wherein only the latter is an actual optimum.

When a maximizing optimization function is used, global and local optima are respectively the global and local maxima. Conversely, if the optimization function if minimizing, the optima are the global and local minima.

[2]Note that a state-space landscape only rarely is capable of depicting an actual search space, and is thus just a model used for the sake of intuition. It is simply impossible to graphically show a search space where a state can have more than three neighbours, as this would require more than three dimensions to show. For this reason, a state-space landscape displays each state as if they have just two neighbors.

Many optimization problems contain several local optima. This is essentially why the choice of metaheuristic is important, as they differ in their strategy to escape local optima. Their efficiency in doing so is ultimately discussed in chapter 5.

2.4 Metaheuristics

A metaheuristic is characterized as a non-problem specific strategy that efficiently guides an exploration of the search space [BR03]. Thus, most metaheuristics search blindly through the search space, and is only guided by the given optimization problem.

A metaheuristic usually starts with a random point in the search space. It modifies it, and if this new *candidate solution* is as least as good as the previous, it is adopted as the new one. This process is respectively called *variation* and *selection*, and the process as a whole can be referred to as *variate-select*.

I will in this section explain the metaheuristics worked on in the project, most of which are inspired by nature. Nature has had many millions of years to develop efficient methods for complex problems, which makes it an obvious target for inspiration when we wish to develop efficient algorithms for solving similar problems ourselves.

Note that while the same metaheuristic might work on the same principle for different search spaces, their implementations are usually different. For this reason, the more detailed explanation of the metaheuristic is given for each search space independently.

When discussing the running time of a metaheuristic, there are two key measures, namely actual running time and the number of iterations. To best discuss the theory opposed to the implementation of a metaheuristic, we measure the running time in the number of iteration it needs instead of the actual running time. The reason for this will be discussed in chapter 5.

2.4.1 (1+1) Evolutionary algorithm

Evolutionary algorithms (EA) are inspired by the reproduction of living species in nature. It consists of an initially randomized population of solutions in the search space. In each iterations, a set of *parents* are selected from the popula-

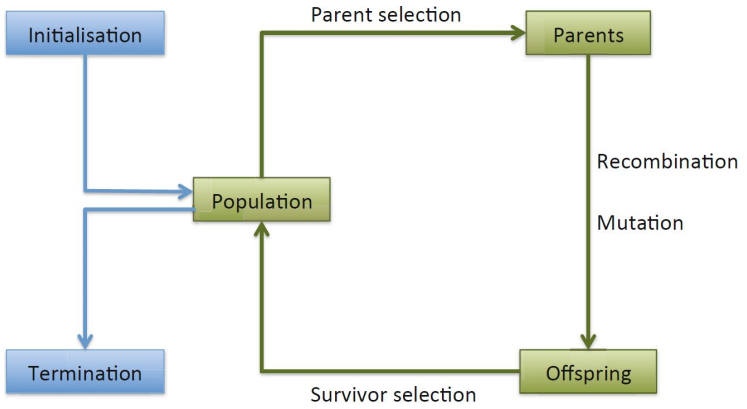


Figure 2.5: General flow of an evolutionary algorithm. [ES15]

tion. These parents are recombined and their offspring are mutated (i.e. sexual reproduction). Finally, a set of the population survive to the next iteration [ES15]. Figure 2.5 shows this process as a flowchart.

Evolutionary algorithms have many variants, each different in how the steps in the flow are performed. The algorithm I will use in this project is the simple variant, where the population is restricted to a single member and we do not recombine. This variant is called the *(1+1) Evolutionary Algorithm*, or (1+1)EA for short. In each iteration, the member produces a single child, which replaces its parent if the fitness score is at least as good [DJW02].

At each iteration, (1+1)EA has the possibility of mutating every single element at once, and can thus, in theory, reach any state from any other state in a single iteration [DJW02]. This effectively means there are no neighbors, and thus no local optima for this metaheuristic, and that the algorithm, in theory, will always find the global optimum given enough time. However, the chance of escaping decreases significantly as more changes are needed at the same time. In practice, this means the algorithm still get stuck in local optima, as the probability of escaping reaches very close to 0.

2.4.1.1 Bit string

For bit strings the algorithm first chooses a randomly generated bit string s in the search space. It then produces a candidate solution by flipping each bit in s independently with probability p (where $0 < p < 1$). If this candidate solution is as least as good as the old, it is replaced. The algorithm continues generating

new candidate solutions, until some stop condition is met (e.g. optimal fitness score) [WIT13]. The algorithm is explained with pseudocode in algorithm 1, where $f(s)$ describes the fitness score of s using the given problem. To ensure the same number of expected mutations for all problem sizes, the probability of flipping a bit is set to $\frac{k}{n}$, where k is a self chosen constant and n is the number of bits in the string. Thus, k becomes the expected number of iterations, and is usually chosen to be $k = 1$.

Algorithm 1 (1+1)EA_b

```
1: Choose s ∈ {0,1}^n uniform at random
2: loop
3:    s' ← s
4:    for all bits b ∈ s' do
5:       Flip b with probability p
6:    end for
7:    if f(s') ≥ f(s) then
8:       s ← s'
9:    end if
10: end loop
```

Given enough time, (1+1)EA is guaranteed to find the optimal solution for both OneMax and LeadingOnes, however, there is no *guaranteed* upper bound on the optimization time for either. Luckily, the *expected* optimization times can be proven. When the probability of flipping a bit is $\frac{1}{n}$, the expected optimization time on OneMax is proven to be

$$en \ln n + c_1 n + \frac{1}{2} e \ln n + c_2 + O(n^{-1} \ln n) \tag{2.12}$$

Where $c_1 \approx -1.892541788\ldots$ and $c_2 \approx 0.59789875\ldots$ [HPR+18][3]. Furthermore, $\frac{1}{n}$ is found to be the optimal mutation rate for OneMax. [WIT13].

On LeadingOnes, the expected optimization time, with $p = \frac{1}{n}$, is proven to be approximately [BDN10]

$$0.85914n^2 \tag{2.13}$$

Interestingly, the optimal mutation rate is around $p = \frac{1.5936}{n}$, giving the expected optimization time of $0.77201n^2$ for sufficiently large n. [BDN10]

[3]Note that the article forgets a *minus* before c_1 in equation 2.

2.4.1.2 Permuation

Unlike bit strings, where the state of each bit can be changed, permutations are modified by the order of elements that are visited. This means we cannot modify a element by itself, but the order in which they are in. For this I will use the 2-OPT move. This means there is an exponential growth in the possible operations w.r.t. the number of elements in the permutation, thus making it very impractical to iteratively choose each possibility with a set probability. Inspired by Scharnow, Tinnefeld and Wegener [STW05], I use a poisson distribution to calculate the number of mutations at each step. This poisson distribution is selected with the expected number of moves k being $\lambda = k$. This can be used as the number of changes in an iteration is asymptotically poisson distributed. [STW05]

The algorithm is explained using pseudocode in algorithm 2. The goal is to minimize the fitness of the tour $f(\xi)$.

Algorithm 2 (1+1)EA$_{\mathbf{P}}$

```
Choose ξ as random permutation of cities {1, 2, ..., n}
loop
    ξ' ← ξ
    Choose v as poisson distribution with λ = 1
    while v > 0 do
        Mutate ξ' on two uniform random edges using 2-OPT
        v ← v − 1
    end while
    if f(ξ') ≥ f(ξ) then
        ξ ← ξ'
    end if
end loop
```

2.4.2 Randomized Local Search

Randomized local search (abbreviated RLS) is the simplest of the four meta-heuristics worked with in this project. It is initialized with a random state s, and at each iteration creates a candidate solution s', which is a randomly chosen neighbor of s. If s' is at least as good as s, s' replaces s [NW10, section 3.3]. Determining which neighbors a given state has is defined specifically for each search space, although it is often the set of states possible with a single modification. The algorithm will simply *climb* the state-space landscape until it has

reached a peak. In this peak, the current state will have a higher fitness than all neighbors, making it impossible for the algorithm to accept any of these, and it can thus never reach the global optimum if the peak is a local maximum.

This metaheuristic is interesting precisely because it has no sophisticated method for escaping local optima. This means it can act as a *baseline* for the other metaheuristics used and how much speed they sacrifice for the potentially gained fitness.

2.4.2.1 Bit strings

For bit strings, the neighbors of a string s is the strings that can be reached from s with any single bit flipped. Pseudocode for RLS on bit strings is shown in algorithm 3. [NW10, section 4.1]

Algorithm 3 RLS_b

```
Choose s ∈ {0,1}^n uniform at random
loop
    s' ← s
    Flip one uniformly chosen random bit in s'
    if f(s') ≥ f(s) then
        s ← s'
    end if
end loop
```

In [DD16], Doerr and Doerr argues that the expected run time of RLS on OneMax for a random initial bit string is $n\ \ln(n/2) + \gamma n \pm o(1)$. However, following their own method, I believe this is missing the constant $+1/2$, which I will argue for below.

Given a string of n 0-bits, RLS on OneMax is expected to need nH_n iterations to find the optimum, where H_n is the n'th harmonic number defined as: [DD16]

$$H_n = \ln(n) + \gamma + \frac{1}{2n} \pm \Theta(n^{-2}) \tag{2.14}$$

Where $\gamma \approx 0.5772156649\ldots$ is the Euler-Mascheroni constant. Giving the expected number of iterations:

$$nH_n = n\ \ln(n) + \gamma n + \frac{1}{2} \pm o(1). \tag{2.15}$$

Since the initial bit string is expected to have half of its bits *completed* (i.e. set to 1), the expected number of iterations for a bit string with exactly $n/2$ 1-bits is: [DD16]

$$nH_{n/2} = n \ln(n/2) + n\gamma + 1 \pm o(1) \tag{2.16}$$

It is, however, not guaranteed that *exactly* half of the bits are initially 1, when initialized randomly. And since the expected iteration count from equation 2.16 is not linear, it will not change symmetrically if the initial fitness score is either higher or lower than the average $n/2$. Therefore the *expected* number of iterations for RLS to find the optimal state on OneMax is $1/2 \pm o(1)$ less than the average number of iterations needed, if the beginning state has half the bits set correctly (i.e. equation 2.16) [DD16]. This gives the expected number of iterations for RLS on OneMax with an initial string of length n:

$$nH_{n/2} - 1/2 \pm o(1) = n \ln(n/2) + \gamma n + 1/2 \pm o(1) \tag{2.17}$$

The expected number of iterations of RLS on LeadingOnes is proven to be [DD16]

$$(1 + o(1))n^2/2 \tag{2.18}$$

The intuition for this expectation can be explained as follows. At any state, there is only a single bit that can improve the fitness score by being flipped, namely the left-most 0-bit. Flipping any bit to the left of it would result in a lower fitness score, and any bit to its right would not change the fitness score. The expected number of iterations before this bit is flipped is thus n, since all bits have an equal probability of being chosen. Because the string is picked at random, we expect only half of the bits to initially be 0, and would thus require the process explained above. Since we expect to perform the process $n/2$ times, and it is expected to take n iterations, this gives a total expectation of $n^2/2$ iterations. The small value $o(1)$ is a combination of the expected fitness score of the initial string and the number of so-called *free riders*, which are *free* fitness scores due to bits already, by chance, being 1. [DD16]

2.4.2.2 Permutation

For TSP-instances, the neighbors of a tour t are the states that can be reached with a single 2-OPT operation on t. The pseudocode for RLS on TSP can be found in algorithm 4.

Algorithm 4 $\mathrm{RLS_P}$

```
Choose ξ as random permutation of cities {1, 2, ..., n}
loop
    ξ' ← ξ
    Mutate ξ' on two uniform random edges using 2-OPT
    if f(ξ') ≥ f(ξ) then
        ξ ← ξ'
    end if
end loop
```

2.4.3 Simulated Annealing

In each iteration, simulated annealing (abbreviated SA) explores one neighbouring point in the search space, which is chosen similarly to RLS. If this point has a better fitness value, it is adopted. If not, it is accepted with some probability depending on a temperature variable. A high temperature means a higher probability of adopting a worse candidate. After each iteration, this temperature is decreased, depending on some *cooling scheme* [KGV83]. Note that when the temperature reaches 0, the algorithm simply turns into a *Random Local Search*.

Let ΔE be the change in fitness from one state to another, on a maximization problem. If $\Delta E \geq 0$, the new state is accepted. If $\Delta E \leq 0$, the probability $P(\Delta E)$ of accepting the new state with a given temperature T is: [KGV83]

$$P(\Delta E) = e^{\frac{-\Delta E}{T}} \tag{2.19}$$

The probability of accepting a worse state thus depends on the given temperature, and how much the fitness value changes. Note that without the context of the problem being solved, a given change in fitness ΔE does not tell anything of *how much* the state has been improved. E.g. two instances A and B of TSP with the same layout, but A being scaled to have 100 times more distance between the cities than B. An increase in the fitness of B will thus be *worth* 100 times more than it would in A. This illustrates the importance of a well-designed cooling schedule, which ideally is tailored to fit any given instance. The choice of cooling schedule is discussed further in section 2.4.3.1.

Because SA is able to accept worse states, it is able to escape local optima by taking another path through the search space. It will, however, often require the algorithm to accept multiple worse states in a row, which is why the temperature and its cooling scheme is so important. As the temperature cools down, the algorithm will hone in on an optima, which is hopefully better than that of a simple RLS.

This metaheuristic is inspired by a process called annealing, where a metal is heated up beyond its crystallization point, where small structures with internal stress are *relaxed*, making the metal reach a lower energy state, increasing the metal's ductility [KGV83]. In simulated annealing, we think of the energy state as the fitness value, and the relaxation as finding optima.

2.4.3.1 Cooling schemes

How quickly the temperature cools, is determined by the cooling scheme. Two widely used schemes $T(t)$ are the exponential [NA98]

$$T(t) = T_0 \cdot \alpha^t \tag{2.20}$$

And the linear schedule

$$T(t) = T_0 - \eta t \tag{2.21}$$

Where t is the iteration count, T_0 is the initial temperature, α is a constant factor ($0 < \alpha < 1$), and η is a positive constant.

The cooling scheme used in this project is the exponential (2.20).

Finding the ideal temperature and cooling rate for permutation instances is hard. The initial temperature should be high enough that the search gets near the optimum, regardless of the beginning state, but low enough to not excessively waste time by randomly moving around the search space. The cooling rate should be high enough to give the search time enough to hone in on the global maximum, but low enough to finish within a reasonable time. The more elements in the permutation, the higher these values should be. For problems where the relative fitness changes between iterations are large, the temperature should be large as well.

In [Mee07], Meer explores a cooling schedule for SA on TSP using a graph of n vertices and $m = 20n$ edges. For this he uses the initial temperature $T(1) = m^3$ and the exponential cooling scheme, where $\alpha = (1 - \frac{1}{cm^2})$ for $c > 0$. In the TSP-instances explored in this project, all cities are connected to each other, i.e. a complete graph. The number of edges in a complete graph of n vertices is $\frac{n(n-1)}{2} = O(n^2)$. If we were to use the same cooling schedule as Meer, this would give us an initial temperature of $O(n^2)^3 = O(n^6)$. To best work from the cooling schedule thought out by Meer, I therefore use an initial temperature which asymptotically scales the same as his, when increasing the number of *vertices*. To allow the user to still modify this, a factor k is multiplied to this. This gives the initial temperature $T(1) = kn^3$ for instances of permutations. The value of α is similarly changed from Meer's to asymptotically match it, by

growing with the number of vertices. I.e. I will use $\alpha = 1 - \frac{1}{cn^2}$ for permutation. The initial temperature does not scale with how *large* fitness changes are in the permutation. Though, one could do this by measuring the largest distance between two cities if the chosen problem is TSP.

Since both problems on bit strings worked with in this project have no local optima, and a back-move is thus never wanted, the best initial temperature is always 0, consequently making the cooling rate unimportant. There are, however, many problems *with* local optima, so a general cooling schedule for bit strings should be added as well. As discussed earlier, the number of states in a permutation of n elements is $n!$ while the number of bit strings of size n is n^2. Since the bit string search space grows much slower than permutations, its cooling schedule should too. Like permutations, the initial temperature and cooling rate can be modified respectively by the factor k and c. The cooling schedule for bit strings will be $T(1) = kn^2$ and $\alpha = 1 - \frac{1}{cn}$.

2.4.3.2 Bit string

Assuming an exponential cooling scheme, the pseudocode for simulated annealing on bit strings is shown in algorithm 5. [JW06]

Algorithm 5 Simulated Annealing$_\text{b}$

```
1:  Choose s ∈ {0,1}^n uniformly at random
2:  Choose t ∈ ℝ as initial temperature
3:  Choose α ∈ [0.0, 1.0] as cooling rate
4:  loop
5:      s' ← s
6:      Flip one uniformly chosen random bit in s'
7:      Choose r ∈ [0.0, 1.0] uniform at random
8:      if min(1, e^((f(s')−f(s))/t)) ≥ r then
9:          s ← s'
10:     end if
11:     t ← α · t
12: end loop
```

2.4.3.3 Permutation

The pseudocode for SA on permutations is shown in algorithm 6.

Algorithm 6 Simulated Annealing$_{\mathrm{P}}$

Choose ξ as random permutation of cities $\{1, 2, \ldots, n\}$
Choose $t \in \mathbb{R}$ as initial temperature
Choose $c \in [0.0, 1.0]$ as cooling rate
loop
  $\xi' \leftarrow \xi$
  Mutate ξ' on two uniform random edges using 2-OPT
  Choose $r \in [0.0, 1.0]$ uniform at random
  if $min(1, e^{\frac{f(\xi')-f(\xi)}{t}}) > r$ **then**
    $\xi \leftarrow \xi'$
  end if
  $t \leftarrow t \cdot c$
end loop

2.4.4 Min-Max Ant System

Ant colony optimization (ACO) is inspired by how quickly ants find short paths between food and their anthill by following trails of pheromones laid by other ants. [DS04]

The metaheuristic based on ACO used in this project is the "Min-Max Ant System" (MMAS*). This algorithm works as follows. The algorithm tries to find a solution by generating new candidate solutions each iteration. If a candidate solution is better than the best solution found so far, it replaces that [4]. Constructing a candidate solution works by imagining an ant walking on a *construction graph*, which differs between search spaces. When walking the construction graph, the probability of an edge e being chosen at a given vertex is (sometimes among other things) based on its pheromone value $\tau(e)$. After each iteration we then imagine the ant laying its own pheromones, thus updating the pheromone values of the edges in the construction graph w.r.t. the current best solution. [NSW09, KNRW12]

Let τ be the pheromone values on the construction graph, and let x be the current best solution. The process of updating the pheromone values after each iteration is shown in algorithm 7. If an edge is included in the walk, its pheromone value is increased, and decreased if not. The values of τ_{min} and τ_{max} are specific to the search space, but are often set to $\frac{1}{n}$ and $1 - \frac{1}{n}$ respectively, where n is the problem size. The constant ρ (where $1 \leq \rho \leq 0$) is called the

[4] Note that a candidate solution with an equally good fitness score is not accepted, unlike the other metaheuristics explored in this project. This is the reason for the star (*) in its name.

evaporation factor, and determines how quickly the pheromones on each edge are changed. A large value of ρ indicates a quick evaporation and vice versa. [KNRW12]

MMAS* is able to escape local optima similarly to (1+1)EA, as it in theory can reach any other state in the search space with a single iteration. Pheromone evaporation makes it possible for the ants to *forget* bad decisions made in earlier walks, in order to make an improvement. A sufficiently low evaporation factor means that if an edge was only recently included in the walk, it has a high probability of being discarded again. This builds on the hypothesis that if an edge has been in the walk for many iterations, it is likely to contribute to a good fitness, and should therefore have a low probability of being replaced. In some way, MMAS* can be compared to SA, since as the search progresses, the pheromones eventually reach either its minimum or maximum value, and SA reaches a low temperature, both of which resulting in a search that makes fewer changes. It is interesting to note that if $\rho = 1$, the metaheuristic simply turns into (1+1)EA with $p = \tau_{min}$, if $\tau_{max} = 1 - \tau_{min}$. This is because an evaporation factor of 1 makes makes it so an edge e always gets $\tau(e) = \tau_{min}$ if it was *not* walked on in the construction graph, i.e. the same probability of the edge being chosen in (1+1)EA.

Algorithm 7 Update(τ, x)

1: **for all** Edges $(u, v) \in x$ **do**
2: $\quad \tau_{(u,v)} \leftarrow min\{(1-\rho) \cdot \tau_{(u,v)} + \rho, \tau_{max}\}$
3: **end for**
4: **for all** Edges $(u, v) \notin x$ **do**
5: $\quad \tau_{(u,v)} \leftarrow max\{(1-\rho) \cdot \tau_{(u,v)}, \tau_{min}\}$
6: **end for**

2.4.4.1 Bit strings

The construction graph for bit strings is a directed graph, often called a *chain*, and is illustrated on figure 2.6. For a bit string of length n, the construction graph has $3n+1$ vertices and $4n$ edges. We imagine an ant walking though this graph starting in v_0, and the resulting trail T determines the bit string. Whether a bit x_i is 1 or 0 is determined by which edge the ant chose from vertex $v_{3(i-3)}$. If T contains the edge $(v_{3(i-3)}, v_{3(i-3)+1})$, x_i is set to 1. Otherwise x_i is set to 0 (and T contains the edge $(v_{3(i-3)}, v_{3(i-3)+2})$). After the edge has been chosen, there is only one more edge to choose, namely the one leading to v_{3i}, and the next bit x_{x+1} can be determined. The ant stops the trail when no more edges can be walked. [NSW09].

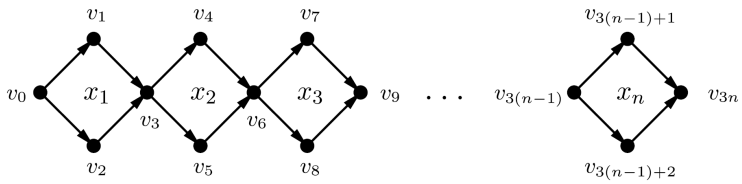


Figure 2.6: Construction graph for bit strings, often called *chain*. [NSW09]

For a construction graph $C = (E, V)$ and the pheromone values $\tau : E \to \mathbb{R}$, then Construct(C, τ) is the algorithm determining a random walk in C defined by algorithm 8. [NSW09]

Algorithm 8 Construct(C, τ)

1: Choose v as starting vertex in C.
2: **while** v has successors in C **do**
3: Choose u as successor of v with probability $\tau_{(v,u)}$
4: $v \leftarrow u$
5: **end while**
6: Return the path taken through C.

The pseudocode for MMAS* on bit strings is shown in algorithm 9. The values for τ_{min} and τ_{max} are respectively $\frac{1}{n}$ and $1 - \frac{1}{n}$, where n is the number of bits in the string. [NSW09]

Algorithm 9 MMAS*$_{\mathrm{B}}$

1: Choose $\tau_{(u,v)} \leftarrow \frac{1}{2}$ for all edges $(u, v) \in E$.
2: Choose x using Contruct(C,τ).
3: Update pheromones τ from x.
4: **loop**
5: Choose x' using Construct(C,τ).
6: **if** $f(x') > f(x)$ **then**
7: $x \leftarrow x'$
8: **end if**
9: Update pheromones τ from x.
10: **end loop**

The expected optimization time of MMAS* on LeadingOnes is proven to be bounded from above by: [NSW09]

$$O(n^2 + (n \log n)/\rho) \tag{2.22}$$

And if $\rho = 1/poly(n)$, from below by: [NSW09]

$$\Omega(n^2 + \frac{n/\rho}{log(2/\rho)}) \tag{2.23}$$

The expected optimization time of MMAS* on OneMax is proven to be: [KNSW11]

$$O(n \log n + n/\rho) \tag{2.24}$$

2.4.4.2 TSP

MMAS* for TSP uses a heuristic in its construction, which means it will not work on any other problem in the permutation search space. For TSP, the construction graph represents each city as a vertex, one of them being the designated start node. The graph is complete, and each edge has a weight corresponding to its pheromone value. Similar to MMAS* on bit strings, this algorithm starts by constructing a tour with a designated `Construct` function. At each iteration, a new tour is constructed, and if better than the current tour, it is replaced. After each iteration, the pheromone values are updated w.r.t. the current tour. This process is shown in algorithm 10. [KNRW12]

Algorithm 10 MMAS*$_{\text{TSP}}$

Choose $\tau(e) \leftarrow \frac{1}{|V|}$ for all $e \in E$.
Choose x using Contruct(τ).
Update pheromones τ from x.
loop
  Choose x' using Construct (τ).
  if $f(x') > f(x)$ **then**
    $x \leftarrow x'$
  end if
  Update pheromones τ from x.
end loop

Inspired by [Zho09], I set τ_{min} and τ_{max} to $1/n^2$ and $1-1/n$ respectively for TSP instances. Since the number of edges in the construction graph grows quickly with the number of cities, $O(n^2)$, there should be a lower tolerance for choosing assumed bad edges, i.e. edges that have not been used for many iterations. Setting the minimum pheromone value matchingly low helps the algorithm not pick as many bad edges.

The `Construct` function is different to that of bit strings in a couple of ways. We imagine an ant traversing the construction graph from the start node. The ant then visits the next city $z \in N$ randomly with a probability based on the pheromone value of the edge leading to that city $\tau(z)$ and a heuristic value, being the distance to it $\eta(z)$, where N is the neighborhood of the next city to add, i.e. the unvisited cities. MMAS* for tsp also has two additional self-chosen variables, namely α and β, respectively being the *pheromone power* and the *heuristic power*. These are used to control the construction of the tour, and how much the ant favors the pheromone values compared to the euclidean distance to it. Algorithm 11 shows the construction of the tour. The algorithm iteratively adds edges to a tour, resulting in a Hamilton path through the construction graph. At each iteration, each unvisited city z is given a *score* equal to $\tau(z)^\alpha \cdot \eta(z)^\beta$. The variable R computes the sum of scores of all neighbouring cities. The probability of a city being chosen next in the path thus becomes its score divided by R. [KNRW12]

Algorithm 11 Construct(τ)

1: **for** $k = 0$ **to** $n - 2$ **do**
2: Choose $R \leftarrow \sum_{y \in N(e_1,\ldots,e_k)} \tau(y)^\alpha \cdot \eta(y)^\beta$
3: Choose one neighbor $z \in N(e_1, \ldots, e_k)$ at random, each with the probability $\frac{\tau(z)^\alpha \cdot \eta(z)^\beta}{R}$.
4: Add the edge leading to z to the tour.
5: **end for**
6: Choose e_n as the edge completing the tour.
7: **return** the tour $(e_1, \ldots, e_n)$

2.4.4.3 Choosing ρ

To use the lower bound of LeadingOnes shown in equation 2.23, the evaporation factor must be of some form $\rho = 1/poly(n)$. A simple way of achieving this is to choose the polynomial $k \cdot n$, where k is a self chosen constant. This gives $\rho = \frac{1}{kn}$. This allows the evaporation to scale with the problem size, since a larger search space gives more choices in the construction graph, and the algorithm thus should be less confident in the path chosen.

2.5 State initialization

If nothing else is mentioned, all searches are initialized with a random state in the search space. For bit strings, this is simply done by selecting each bit with probability 1/2. For permutations, this is done by shuffling the permutation, using the *Fisher-Yates shuffle*, resulting in a uniformly random permutation. This algorithm was first described by Ronald Fisher and Frank Yates [FY38], and later improved in [Dur64]. This shuffle is described in algorithm 12.

Algorithm 12 Fisher-Yates shuffle

1: Let A be a permutation
2: **for** i from $n-1$ to 1 **do**
3: Select j as random integer, such that $0 \leq j \leq i$
4: Swap A_i and A_j
5: **end for**

These initializations all aim to randomize the state as much as possible. However, one could imagine that if the initial state had a better fitness score, the search would terminate faster, and possibly with a better score. Since the optimum for both problems in the bit string search space are easily determined, approximating the initial state is of little interest. However, since the optimum for TSP-instances are hard to determine, the initial state can be approximated, and the metaheuristic could thus finalize the search.

A total of three TSP-initializers are explored in this project, all of which are similar in concept. They all work by greedily adding new cities to its tour until all cities have been visited, each with their own heuristic.

In [RSI77], Rosenkrantz, Stearns and Philip II describe the *Nearest neighbor algorithm* for approximating a TSP-instance in $O(n^2)$ time. They explain the algorithm to work as shown in algorithm 13. They show that this approximation returns a tour of length no worse than $\frac{1}{2}\lceil \ln(n) \rceil + \frac{1}{2}$ times the optimum.

Algorithm 13 Nearest neighbor

1: Let T be an initially empty permutation.
2: Choose the initial city as any city c, and add it to T.
3: **while** T does not contain all cities **do**
4: Let c be the unvisited city, closest to the last city in in T.
5: Insert c into T as the last element.
6: **end while**
7: **return** T.

The same article [RSI77] also describes two *insertion methods*, one of which is *cheapest insertion*, which produces tours no longer than twice the optimal length. Cheapest insertion is shown in algorithm 14. The function $d(i, j)$ computes the euclidean distance from i to j. This algorithm iteratively inserts the unvisited city that would result in the shortest resulting tour, i.e. the cheapest insertion. Cheapest insertion can be implemented to run in $O(n^2 \log(n))$ time. [RSI77]

Algorithm 14 Cheapest insertion

1: Choose the city c uniformly at random.
2: Find the city g minimizing $d(c, g)$, i.e. the city closest to c.
3: Create the sub-tour T containing cities c and g.
4: **while** T does not contain all cities **do**
5: Find the connected cities $(i, j) \in T$ and $r \notin T$ resulting in the cheapest insertion, i.e. minimizing $d(i, r) + d(r, j) - d(i, j)$.
6: Insert r between i and j in T
7: **end while**
8: **return** T.

The third and final initialization algorithm is inspired by a lecture in the course CITS3001, which I attended at the University of Western Australia [Fre19, Slide n. 31]. It works similarly to *cheapest insertion*, however, the unvisited city selected for insertion is picked at random. This city is then inserted into the tour in the position that would create the shortest resulting tour. The algorithm is shown in algorithm 15.

Algorithm 15 Random insertion

1: Choose the cities c and d uniformly at random.
2: Create the sub-tour T containing the cities c and d.
3: **while** T does not contain all cities **do**
4: Let $r \notin T$ be a city chosen uniformly at random.
5: Find the connected cities $(i, j) \in T$ resulting in the cheapest insertion, i.e. minimizing $d(i, r) + d(r, j) - d(i, j)$.
6: Insert r between i and j in T.
7: **end while**
8: **return** T.

Chapter 3

Design

In this chapter, I will discuss the overall goals and ideas of the program, and how they will be fulfilled. First, this is done by discussing the structure, i.e. how different functions and classes in the program should be coupled together, to ensure the goal of building a framework which can be developed further with relative ease. Next I discuss how these different components are combined to run a simulation, after which I discuss the flow of using the program, and how elements are displayed to the user in the form of visualizations and logging.

3.1 Program structure

The program is made using Java 10.0.1. Java is an object-oriented programming language, so the design should follow its philosophies. The following bullets highlight some of the functionality of the program that are each encapsulated as their own objects. If objects share a lot of functionality, they are made to inherit it from the same abstract class, which also allows for polymorphism.

- **Search spaces**
  Have the primary function of storing a state.

- **Problems**
  Implement the function that determines the fitness score for a given search space.
- **Metaheuristics**
  Variates on the current state, and determines if it is adopted.
- **Stopping criteria**
  Determines when the search should be terminated.
- **Visualizations**
  Contain different ways of getting visual insight in the current search.

The program is designed based on the idea that for any search space, a set of problems and metaheuristics are tied to that. I.e. a problem should work on all instances of the associated search space, and all metaheuristics in the same search space should be able to use that problem. Because two search spaces are worked with in this project, problems and metaheuristics are separated between these. However, both TSP and MMAS* for permutations are exceptions to these. TSP requires the permutation to be elements of a special type, since it must contain coordinates to calculate distances, and the MMAS* metaheuristic for permutations combines its random walk with a heuristic that assumes the problem is TSP. Thus, problems of type TSP require the user to use a permutation of cities, and MMAS* for permutations only works for TSP.

The program is designed to be extensible, such that one easily can extend the program with additional search spaces, problems, metaheuristics or visualizations. All problems and metaheuristics that work on the same search space are implemented from their own abstract class to eliminate as much redundant code as possible. E.g. all four metaheuristics working in the bit string search space inherit the same abstract class, since they all need identical methods such as the constructor, returning the currently best solution etc., while still being able to customize the methods that are unique to each of them (e.g. variate-select). A class diagram explaining the metaheuristic class and the heuristics inheriting it is shown on figure 3.1. The problems are designed in a similar way, where they each inherit an abstract class for each search space.

3.2 Running a simulation

A central class handles the interaction between the chosen metaheuristic, problem, stopping criteria and visualizations. It ensures the problem is passed on

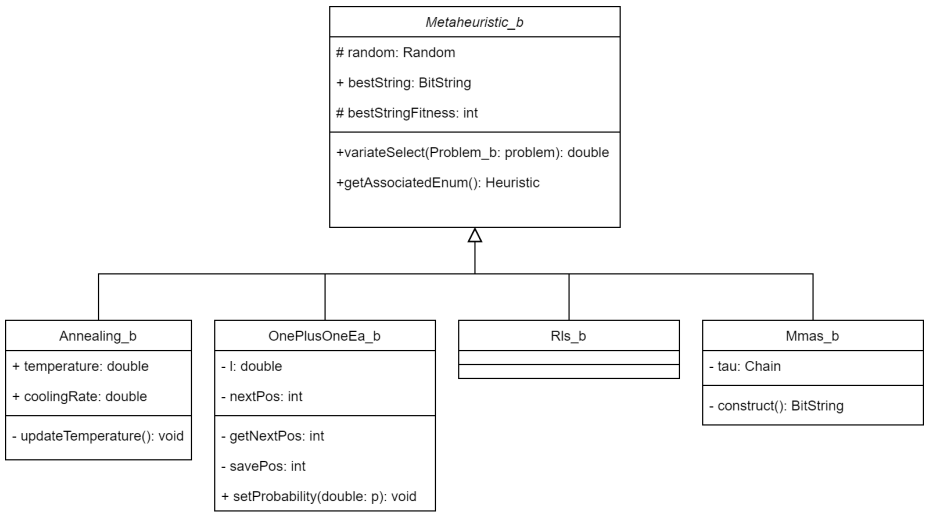


Figure 3.1: Class diagram of metaheuristics on the bit string search space

to the metaheuristic, and continuously lets the metaheuristic search until the stopping criteria is met, while updating the selected views on the way. If the user chooses to start the simulation manually, as opposed to using a template file, they will get the option of choosing the speed of the simulation as *updates per second*. This ensures the goal of being able to visualize the workings of a metaheuristic, as it would otherwise complete too fast for the graphics to "keep up" and for the user to see what is happening. Each time the search accepts a new state, the program waits for an amount of time calculated from the selected value, assuming no time was spent finding it.

The users chooses the run configurations either via a template file or the user interface. Each option group has an *enum*, e.g. all problems have an associated constant in the the *Problem enum*. This will be the method of storing the selected options in the program. Each constant will have a set of values attached to it. For instance, all constants have a name for printing/displaying, problems will have a boolean of whether it is a maximization problem or not, visualizations contains an array of problems they can display the state of etc. Visualizations are tied to the problems instead of the search spaces, since for instance, drawing a TSP-tour can not be done for all problems regarding permutations, though it is the only one used in this project.

3.2.1 State initialization

For TSP, it is possible to initialize the beginning state using one of three greedy heuristics: nearest, cheapest and random, as explained in section 2.5. If the user chooses neither, the initial state is simply a random permutation. Bit strings are always initialized randomly.

3.2.2 Stopping criteria

There are currently four stopping criteria used in the project. These are *target fitness*, *stagnation*, *time limit* and *iteration limit*. The target fitness criteria continues the search until a given fitness has been reached and is aimed at problems where the optimum fitness is known, and a search is guaranteed to find it given enough time. This is primarily useful for problems with no local optima, as some metaheuristics like RLS could otherwise get stuck indefinitely. The stagnation criteria continues the search until the fitness score has not changed for a set number of iterations and is, on the other hand, useful for problems *with* local optima, paired with metaheuristics that can get stuck in these. The time limit criteria simply continues the search until a set amount of time has passed, and can be useful for searches where the search can be improved from local optima, but the chance of it happening reduces with time, e.g. (1+1)EA. The final criteria is the iteration limit, which stops the search after a set number of iterations. This is useful when comparing how well two metaheuristics perform on a set time frame, without relying on the implementation specifics of each like *time limit*.

3.3 Program flow

Though the user experience is of little importance in this project, the overall flow has been considered. When using the program, you have two options; either you select the desired configuration (search space, metaheuristic etc.) manually and run it once, or you use a premade template file. Entering the options manually is useful when you wish to experiment with different options or to visualize an algorithm. The template files are useful for running a set of configurations multiple times in a row for testing purposes. A flow chart showing how the program is interacted with, is shown in figure 3.2. Appendix C shows a guide on how the program is used.

The program is designed such that everything can be done from the same screen.

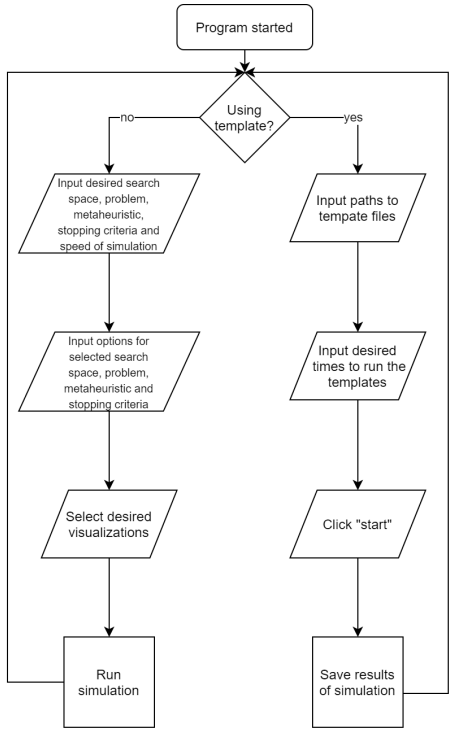


Figure 3.2: Flow chart showing operation of the program

When a simulation starts, this start screen stays open, making it easy to run the simulation again, when it has ended.

3.4 Visualizations

One of the goals of the project is to visualize how a metaheuristic works. To do this, it is important to have a way of *seeing* what a state in the search space looks like, to understand how it is modified by the metaheuristic.

3.4.1 TSP tour

How to best visualize a permutation depends on the underlying problem. For instance, in a sorting problem, an element can be seen as a column with a size corresponding to its value as seen on websites like `VisuAlgo.net` [Hal]. However, in this project, only TSP is being used, and thus a visualization has been made specifically for it. Because visualizations are thus not *search space specific*, but *problem specific*, the constants in the enum for problems have an associated array of *problems* instead of a search space, as one might naively think.

The visualization of a TSP-problem displays the permutation as a graph, where each vertex is a city, connected by an edge if they are visited in succession in the given tour. Each city is placed in a 2d-plane, such that they represent the coordinates given for the instance. It is important to note the distinction between conceptualizing the *instance* as a graph and the *permutation*, as the instance is a complete graph of every city whereas the permutation is a Hamilton cycle through it. The permutation graph is, in fact, a subgraph of the instance graph.

3.4.2 Boolean hypercube (Onion graph)

The toolkit FrEAK [JW04], uses a visualization of bit strings called *the boolean hypercube* (or *onion graph* due to its appearance), which I was inspired to use for this project as well. An example of the boolean hypercube is shown on figure 3.3.

The given string is shown with a dot moving around in the search space each time a new candidate is found, where the placement is determined by the content

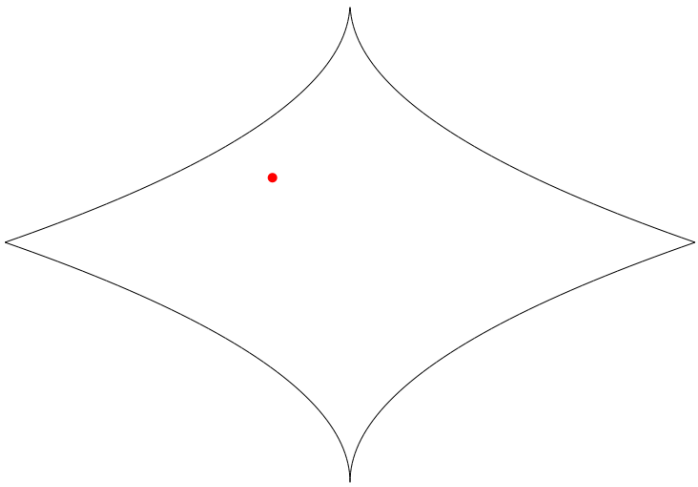


Figure 3.3: Visualization of a bit string using a boolean hypercube. The string is the red dot. From the figure it can be gathered that the string has more 1's than 0's, and the 1's are primarily distributed to the left.

of the bit string. The vertical placement of a bit string is determined by the number of 1's and 0's in it, such that bit strings with many 1's appear higher up and vice versa. The horizontal placement is determined by the distribution of the bit string's 1's and 0's, such that a bit string with its 1's primarily on the left side of the string, appears further to the left and vice versa. A bit string with equally many 1's and 0's, distributed symmetrically around the center bit, will thus be placed exactly in the middle.

The hypercube is drawn such that it is widest when there are equally many 1's and 0's. This is due to the fact that the number of unique strings are highest when this is the case. This amount grows exponentially until reaching the middle; more exactly it is n *choose* $\frac{n}{2}$ or $\binom{n}{n/2}$. Ideally the shell of the hypercube would thus be drawn using an exponential function, however, this would make it much wider, sacrificing readability. The function used to draw the shell is simply $f(x) = x^2$, as this was deemed an appropriate approximation.

For a given bit string, we can independently determine its vertical and horizontal score. Using these scores, the placement is then determined by how these values compare to the minimum- and maximum values.

For a given bit string $w \in \{0,1\}^n$, the vertical score is simply the number of 1's in w. The minimum possible number of 1's in a word of size n is always 0, and

the maximum is always 1. Therefore the vertical placement of w is simply the number of 1's in w divided by n.

The horizontal score of $w = (x_1, x_2, \ldots, x_n)$ is the sum of all 1's in w times their index in the string. More formally:

$$\sum_{i=1}^{n} i \cdot x_i \tag{3.1}$$

This results in 1-bits with a higher index contributing more to the sum than 1-bits of low index. To find the minimum value, we utilize the fact that a bit string with y 1's of minimum horizontal score, will always have all 1's positioned to the left in the string, i.e. being the string of y 1's followed by $n - y$ 0's. Likewise the bit string of maximum horizontal value will have all 1's positioned to the right in the string. The horizontal position of w is thus determined by where the horizontal score of w is in the range between the minimum and maximum scores.

It is important to note that two distinct bit strings *can* appear in the same position in the boolean hypercube. This is, for instance, the case with the two words "10001" and "01010", where both have a vertical value of 2, and a horizontal value of 6. This visualization can thus not be used to determine the exact bit string being shown, but is a tool to get an idea of some general properties of the bit string in question.

3.4.3 Other visualizations

Two other visualizations are used in the project, namely a *fitness-iteration chart* and a *key figures* window. These do not help visualize the actual state, but give various insights into the current simulation. The fitness-iterations chart is a chart, plotting the fitness scores in comparison with the iteration number of it. The key figures window displays various information about the current search, e.g. the current fitness score, for how long the search has gone, the temperature while running SA etc. If the user chooses to run the simulation at a reduced speed, the time spent waiting will be deducted from the time shown.

3.5 Logging

The program is made to make evaluating the performance of metaheuristics as easy as possible. The easiest way to test the performance of a specific configu-

ration is to make it a template and run it multiple times. A template file has the file type `*.tem` and contains text. Each line in the template file consist of a key-value pair. For instance the pair `(SEARCH SPACE: PERMUTATION)` means that the search space used in the simulation will be the permutation space. The template file ends with the line `END_OF_DEFINITIONS`. All key-value pairs can be seen in appendix A.

When a simulation that has been started from a template finishes, the result is logged in a **.csv*-file, making it easy to extract the results to other programs such as Microsoft Excel. Each simulation is logged in a row in the file, displaying the options used for starting it, and information such as the number of iterations used, the running time and the fitness score reached.

Chapter 4

Implementation

In this chapter, I will discuss how key concepts are implemented in the program. I will go through selected methods used to run a simulation, from handling the chosen run-configuration to performing the search. This also includes how the visualizations are created, and how the results are logged to a file if a template file is used.

4.1 Running a simulation

For each search space, a central class handles running the actual simulation, once the configuration has been chosen. These classes must be separate, since the fundamental objects they work on are different, i.e. simulating on bit strings uses the abstract class `Metaheuristic_b.java`, while the permutations search space uses `Metaheuristic_p.java`. Because these two simulation-classes are similarly implemented, I will in my explanation focus on simulating bit strings. The constructor (shown in code snippet 1) receives only a `RunOptions`-object, which contains everything necessary to know which options have been chosen. Next, it calculates the delay between each update, initializes the metaheuristic, problem and stopping criteria, and finally the selected visualizations.

```
1  public RunBitstring(RunOptions options) {
2      this.options = options;
3
4      long tmpNanoseconds = Conversions.updatesPerSecondToSleepDelayNano(options.
           ↪ getUpdatesPerSecond());
5      this.updateDelayNano = (int) (tmpNanoseconds % 1000000);
6      this.updateDelayMilli = (int) (tmpNanoseconds / 1000000);
7
8      initializeHeuristic();
9      initializeProblem();
10     initializeStoppingCriteria();
11
12     enableChosenViews();
13
14     if (showKeyFigures) initializeKeyFigures();
15     if (showFitnessChart) initializeFitnessChart();
16     if (showBooleanHypercube) initializeBooleanHypercube();
17 }
```

Code snippet 1: The constructor for the class running the simulation on bit strings.

Running the simulation is intensive on the processor, so to prevent the user interface from freezing, it is run on a separate thread. This is done by calling the method `startBitstring()`, which then creates a thread running only the simulation. This method is shown in code snippet 2.

```
1  public void startBitstring() {
2      Runnable task = this::runBitstring;
3
4      Thread backgroundThread = new Thread(task);
5      backgroundThread.setDaemon(true);
6      backgroundThread.start();
7  }
```

Code snippet 2: Starting the simulation on a separate thread.

Running the simulation consists of a while loop, running until the stopping criteria determines so. This loop can be seen in code snippet 3. An iteration of the search happens each time the loop begins. Each time, the heuristic variates on its candidate solution using the given problem. If the heuristic selects a new candidate, various variables and views are updated where appropriate. Finally, the thread sleeps for some time, depending on the speed determined by the user. The thread only sleeps if the fitness is changed, since the `Thread.sleep()` method is not very exact at very small sleep intervals. A rough test showed that the thread would sleep an average of 1.6 ms, when it was in fact supposed to sleep for just 0.01 ms. With the long delay, the program would be too slow to use

properly for problems like LeadingOnes, where the candidate solution is often times modified with no improvement to the fitness score. The displayed running time of the search takes into account the time spent sleeping by measuring the time before and after, which is added to a variable, keeping track of the total amount of time slept. The value of this variable is then subtracted whenever the running time is being displayed.

```
1  while (stoppingCriteria.shouldContinue()) {
2      heuristic.VariateSelect(problem);
3      if (heuristic.didUpdate()) {
4          score = heuristic.getBestStrFitness();
5          if (showBooleanHypercube) {
6              booleanHypercube.redraw(heuristic.getBitString());
7          }
8          candidatesUsed++;
9          loadChartDataPointIfShould(lastScore, heuristic.getBestStrFitness());
10
11         if (score != lastScore) {
12             stoppingCriteria.updateFitness(score);
13
14             tmpSleepTimeNanoStart = System.nanoTime();
15             try {
16                 Thread.sleep(updateDelayMilli, updateDelayNano);
17             } catch (InterruptedException e) {e.printStackTrace();}
18             totalSleepTimeNano += System.nanoTime() - tmpSleepTimeNanoStart;
19         }
20         lastScore = heuristic.getBestStrFitness();
21     }
22     iterationCount++;
23     stoppingCriteria.increaseIterationCount();
24 }
```

Code snippet 3: The simulation loop. Continues until the stopping criteria determines so.

4.1.1 Metaheuristics

All metaheuristics inherit an abstract class corresponding to its search space. In code snippet 4, most of this mother class for the bit string search space is shown. This class contains methods that are useful to all inheriting classes, as well as abstract methods that are needed, but where the implementation differs. The most useful method of these, is `variateSelect`, whose job is to create a new candidate solution and select or discard it. Once this method has executed, the field `hasUpdated` signals whether the state was changed, and if so, `bestStrFitness` updates to the new fitness score. The mother class for permutations is implemented similarly.

```
1  public abstract class Metaheuristic_b {
2      protected Random rand = new Random();
3      protected BitString bestStr;
4      protected double bestStrFitness;
5      protected boolean hasUpdated = false;
6
7      public Metaheuristic_b(int length) {
8          bestStr = new BitString(length);
9      }
10     public abstract void variateSelect(Problem_b problem);
11
12     public double getBestStrFitness() {
13         return bestStrFitness;
14     }
15
16     public boolean didUpdate() {
17         return hasUpdated;
18     }
19 }
```

Code snippet 4: Abstract mother class of all metaheuristics on bit strings. Some methods like *getters*, as well as methods for logging/displaying, have been omitted.

A naive implementation of (1+1)EA for bit strings could be running through the entire string in a `for`-loop and flipping each bit with a probability. Though this would work, it would result in a linear running time complexity of $O(n)$. To improve this, Jansen and Zarges [JZ11] describe an algorithm for doing that, with a running time asymptotic to the expected number of mutations in the string k, i.e $O(k)$. This algorithm works by "jumping" from one end of the string to the other, *landing* an average of k times, flipping the bit as it is visited.

4.1.1.1 Search spaces

Each search space has an associated object for storing its states. Bit strings store the string as an array of *booleans*, where `true` corresponds to a 1, and `false` to a 0. A metaheuristic can flip the bit by calling `flip` with the index in the string to flip.

Permutations are stored as an array of *Objects*, and the order in which they appear determine the permutation. A metaheuristic can perform a random 2-OPT operation by calling the method `makeRandom2Opt`. This generates the two random edges to recombine crosswise. Say we have the permutation as an array of objects P and we wish to perform 2-OPT on the two edges starting in α

and β. This is done by reversing P *from* $\alpha + 1$ *to* β, moving left-to-right. It is important to note that if β appears earlier than α in P, the array should still be reversed as if the end of the array continued at the beginning. For this reason, the method `getIndexAfterOverflow` has been created, which simply returns the corresponding index inside the bounds of the array, despite the integer passed being either negative or larger than the array size.

When creating a candidate solution, most metaheuristics modify the current solution by copying the old. It is important to make this copy *deep* enough, in order to not accidentally modifying the current solution as well. An example could be if we copy a bit string, but this new copy merely copied the pointer to the array, and a modification to this thus would result in a modification in the original as well. Therefore, bit strings have a deep copy method, avoiding this problem. For permutations, two methods are implemented; a shallow and a deep copy. The shallow copy still copies the array itself deeply, but the objects it contains are not. Thus, if one was to modify a city's location in one candidate, it would change for them all. For TSP, this is not a problem, as the cities themselves are not modified, but the case might be different for other. Therefore a deep copy has been implemented as well. However, this method turned out to result in much slower simulations, and is therefore not currently in use.

4.1.2 Problems

The implementation of a problem is quite simple. Like metaheuristics, all problems in a given search space inherit the same abstract class. However, only three methods are used: A `toString`-method to display it nicely when written, `getAssociatedEnum` which simply returns the *enum* corresponding to the implemented problem, and the fitness function `fitness`. The fitness function receives a search point and returns its fitness score.

4.1.3 Stopping criteria

All stopping criteria inherit the abstract class `StoppingCriteria.java`. This abstract class is shown in code snippet 5. After each iteration, the method `increaseIterationCount` is called. If an iteration results in a new fitness score, the methods `updateFitness` is called. Some problems have a lot of updates, where the fitness score is not improved. This is the case for LeadingOnes, where the bits after the first 0 are frequently flipped without fitness improvement, or symmetric-TSP if the 2-OPT operation causes the tour to simply reverse direction. Because of this, the `updateFitness` method ignores the up-

date if the fitness has not changed. This was especially important when using the *stagnation* stopping criteria, as the search could last forever when using a metaheuristic performing 2-OPT on TSP. The only method required to be implemented specifically for a stopping criteria is `shouldContinue`. This method is called once each iteration, and when it returns *false*, the search immediately stops.

```
1  public abstract class StoppingCriteria {
2      final boolean maximizeFitness;
3      protected double currentFitness;
4      protected int iterationCount = 0;
5      protected int iterationsSinceNewCandidate = 0;
6
7      public StoppingCriteria(boolean maximizeFitness) {
8          this.maximizeFitness = maximizeFitness;
9      }
10
11     public void increaseIterationCount() {
12         iterationCount++;
13         iterationsSinceNewCandidate++;
14     }
15
16     public void updateFitness(double fitness) {
17         if (currentFitness == fitness) {
18             return; //Ignore if fitness has not changed
19         }
20         currentFitness = fitness;
21         iterationsSinceNewCandidate = 0;
22     }
23
24     public abstract boolean shouldContinue();
25 }
```

Code snippet 5: Abstract mother class of all stopping criteria. Some unimportant methods have been omitted.

4.2 Graphics

This section will cover how the various graphics in the program have been made. Most of the screens presented throughout the program, consist of basic JavaFX components. An example of this is the start screen of the program, located in `SelectionsMenu.java`. All screens are located in a separate class, which have the following methods:

- A constructor that builds the scene.

- `getScene()`, returning the scene built in the constructor.

- `setContinueButtonAction(EventHandler<ActionEvent> e)`, used by the caller to determine what happens when the continue button is pressed.

- Relevant *getters*, to extract the information entered in the screen.

The caller thus creates the screen as an object, and uses the scene on the desired stage.

The two screens that are an exception to this are the visualization of a TSP-tour, and the boolean hypercube. Both of these use the external library GraphStream[1], whose main purpose is to handling dynamic graphs. GraphStream uses an event-based enginge, which allows it to react to changes dynamically, instead of redrawing the entire graph for each change [PDGO08]. This makes it ideal for this project, since changes in the candidate solution are usually small modifications (e.g. when a bit string moves).

When initialized, the boolean hypercube draws the *shell*. The shell is made up of invisible vertices connected by visible edges. The placement of each vertex is determined by an x- and y-coordinate. The y-coordinates of the graph ranges from -1 to 1, and the x-coordinate is determined by the *drawing function* symmetrically around x=0. Code snippet 6 shows the right side of the the shell being drawn. On each side of the graph, the side is drawn using a constant number of vertices, currently set to 75. The y-coordinate is initially set to -1.0, and at each iterations increased by an amount such that the final vertex is positioned on y=1.0. How much is incremented is shown on line 2. On line 5 the given vertex is added to the graph, given a unique *id*. This id is used later, when the vertices are connected. Line 6 places the vertex in the correct position. The y-coordinate is simply the variable `currentY`, while the x-coordinate is calculated using another function. Finally, line 7 makes the vertex invisible by setting it *alpha-value* to 0. The left side is drawn similarly, but with a negative x-coordinate when being positioned. The string is drawn as the red vertex. When the string is updated, it simply deletes the old one and draws it again.

[1] Website: `http://graphstream-project.org/`

```
1 double currentY = -1.0;
2 final double shellIncrement = 2 / (double) (Constants.
      ↪ POINTS_TO_DRAW_ONION_SHELL - 1);
3
4 for (int i = 0; i < Constants.POINTS_TO_DRAW_ONION_SHELL; i++) {
5     shellRight[i] = graph.addNode("r" + i);
6     shellRight[i].setAttribute("xy", onionShellFunction(currentY), currentY);
7     shellRight[i].addAttribute("ui.style", "fill-color: rgba(0,0,0,0);");
8     currentY += shellIncrement;
9 }
```

Code snippet 6: Drawing the right side of the boolean hypercube.

Drawing a tour in TSP-instance is also done using GraphStream, in the class `TspTour.java`. When drawing the graph, the city permutation is given, and each vertex is placed in the coordinates corresponding to the city. When a permutation is modified, one might think it would be smart to simply modify the graph only from what has changed (e.g. only modifying the two edges recombined from a 2-OPT). However, the class deciding when to update the view does not know in which way the graph has been changed, only that it has. Switching this, would confuse the design of the program, since the metaheuristic would then become responsible for updating it, and different methods would need to be used in the visualizer for each of them. The method responsible for drawing the tour is shown in algorithm 7. Since this visualization only works for TSP instances, and not just any permutation, the elements first get casted to an array of cities on line 2.

```
1 public void redraw(Permutation tour) {
2     City[] cities = (City[]) tour.getElements();
3     graph.clear();
4     nodes = new Node[cities.length];
5
6     addNodes(cities);
7     addEdges(cities);
8 }
```

Code snippet 7: Drawing a TSP tour from a permutation.

4.3 Logging

When a simulation has run using a template file, key information about it is stored in a log file. Logging a file is done using an object from the class

Instance name	Problem	Heuristic	Stop condition	Machine name	Date	Fitness	Time (ms)	Iterations	Candidates
n = 2048	OneMax	Random Local Search	Target fitness (2048.0 fitness)	Lenovo-PC	13/06/2020 18:15:33	2048.0	136	15367	1047
n = 2048	OneMax	Random Local Search	Target fitness (2048.0 fitness)	Lenovo-PC	13/06/2020 18:15:35	2048.0	95	13073	998
n = 2048	OneMax	Random Local Search	Target fitness (2048.0 fitness)	Lenovo-PC	13/06/2020 18:15:36	2048.0	79	14922	1029
n = 2048	OneMax	Random Local Search	Target fitness (2048.0 fitness)	Lenovo-PC	13/06/2020 18:15:38	2048.0	100	21336	998
n = 2048	OneMax	Random Local Search	Target fitness (2048.0 fitness)	Lenovo-PC	13/06/2020 18:15:40	2048.0	66	12305	990
n = 2048	OneMax	Random Local Search	Target fitness (2048.0 fitness)	Lenovo-PC	13/06/2020 18:15:42	2048.0	61	16304	1031
n = 2048	OneMax	Random Local Search	Target fitness (2048.0 fitness)	Lenovo-PC	13/06/2020 18:15:44	2048.0	76	11944	1026
n = 2048	OneMax	Random Local Search	Target fitness (2048.0 fitness)	Lenovo-PC	13/06/2020 18:15:46	2048.0	64	22373	1022

Table 4.1: Result log of 8 simulations. The Excel functions calculating averages has been omitted.

`Logger.java`. As shown in code snippet 8, the object is constructed, various fields are set, after which the method `log(String templatePath)` logs the file on disk.

```
1  Logger logger = new Logger();
2  logger.setInstanceName("n = " + options.getBitStringLength());
3  logger.setProblemName(problem.toString());
4  logger.setHeuristicName(heuristic.toString());
5  logger.setStopConditionName(stoppingCriteria.toString());
6  logger.setFitness(score);
7  logger.setRunningTime(System.currentTimeMillis() − timeStart − (totalSleepTimeNano /
       ↪ 1000000));
8  logger.setIterations(iterationCount);
9  logger.setCandidates(candidatesUsed);
10 logger.setMetaheuristicVariables(heuristic.getVariousLogs());
11
12 logger.log(options.getTemplatePath());
```

Code snippet 8: Using `Logger` to create a log of the finished simulation.

The log is written to a file located in the same folder as the template used. The filename is identical to the template file, but with the suffix "`_result`" and the file type *csv*. If the file does not already exist, it is created, initialized with a header explaining the fields, as well as some simple functions to calculate averages in Microsoft Excel. An example of the resulting log is shown in table 4.1

If the user chooses to run the program using template files, they input the paths into the corresponding text field, each separated with a line break. Each template is then run in order the selected number of times. Once the user presses the button "Use selected templates", each line is parsed into a `RunOptions`-object, and they are collected into an array. All these run configurations are then simulated, in order, the number of times selected. To run a search space, the `RunOptions` is simply updated, and is simulated as usual. When a simulation is started from a template, a new thread is spawned, which monitors when the simulation has ended. When finished, a new configuration is loaded and run.

This happens continuously until all desired templates have been simulated.

Chapter 5

Evaluation

To evaluate the performance of the implemented metaheuristics, a number of configurations have been tested using the program. The results of these tests are discussed in this chapter.

When comparing different metaheuristics, we look at different values depending on the search space. For searches where we are guaranteed to find the optimum fitness, we mainly compare the number of iterations needed to reach it. However, for a problem like TSP, where we are not guaranteed to reach optimum, we primarily compare how good the reached fitness is, and how many iterations it takes to get there. The actual running time (in seconds) is only discussed briefly, and not in much detail. comparing using the iteration count allows us to compare the metaheuristics more theoretically, as the specifics in my implementation could otherwise lead to misleading results. Instead of measuring the iteration count, one might instead use the number of fitness evaluations, as these in many configurations have an asymptotically higher, or as high, running time than the rest of the iteration. In this project, all metaheuristics uses the fitness evaluation at most once, and thus these two values are asymptotically the same. Evaluating both OneMax and Symmetric-TSP requires searching through the whole state, and LeadingOnes will on average search through half of them, i.e. they all have an asymptotic running time of $O(n)$. On bit strings, both SA, RLS and (1+1)EA variates on the state in $O(1)$ time, and MMAS* in $O(n)$. For permutations SA, RLS and (1+1)EA modify the state in $O(n)$ time,

since this is the time complexity of a 2-OPT move, and updating all edges in the construction graph for MMAS* takes $O(n^2)$ time. At each iteration, the fitness function is called once on the new candidate solution, except for (1+1)EA where this is skipped if no change are made. This will unfairly make it seem slower compared to the other metaheuristics when comparing them.

This chapter is divided into sections for each metaheuristic, however, since RLS is a special case of SA, it will be included in that section.

All simulations have been performed on an Intel Core i7 (8^{th} Gen) 8565U / 1.8 GHz with 8 MB of cache. It has 4 cores, making it 8 logical cores with hyper-threading. The memory is 16 GB of LPDDR3 SDRAM at 2133 MHz. Other PC specification are of little importance in these simulation results.

In this chapter, the metaheuristics on TSP are only evaluated on the instance berlin52. To prevent my findings to only be relevant for berlin52, I have done all the same simulations on the instance bier127, seen in appendix B.

5.1 (1+1)EA

5.1.1 OneMax

The chart in figure 5.1 shows the average number of iterations when running (1+1)EA on OneMax at different problem sizes and at different mutation probabilities. We first see that the actual iteration count of (1+1)EA on OneMax closely resembles the expectation shown in equation 2.12, when the mutation probability is $p = \frac{1}{n}$. The red dotted line shows the theoretical iteration count, and the green the measured when $p = \frac{1}{n}$. As discussed earlier, $p = \frac{1}{n}$ is the optimum mutation rate for OneMax. On the chart below is shown that this $p = \frac{1}{n}$ is indeed better than, for instance, $p = \frac{0.8}{n}$. It was also tested whether it would be better than $p = \frac{1.2}{n}$, and it was. However, the improvement was so little that it was not visible when shown in the chart, and is thus omitted.

5.1.2 LeadingOnes

The theoretical and measured iteration counts of LeadingOnes are compared in the chart on figure 5.2, with $p = \frac{1}{n}$. Again we see that these align very well. It was also discussed that the optimal mutation rate for LeadingOnes is $p = \frac{1.5936}{n}$, which is indeed tested to be better than $p = \frac{1}{n}$, as shown in the chart.

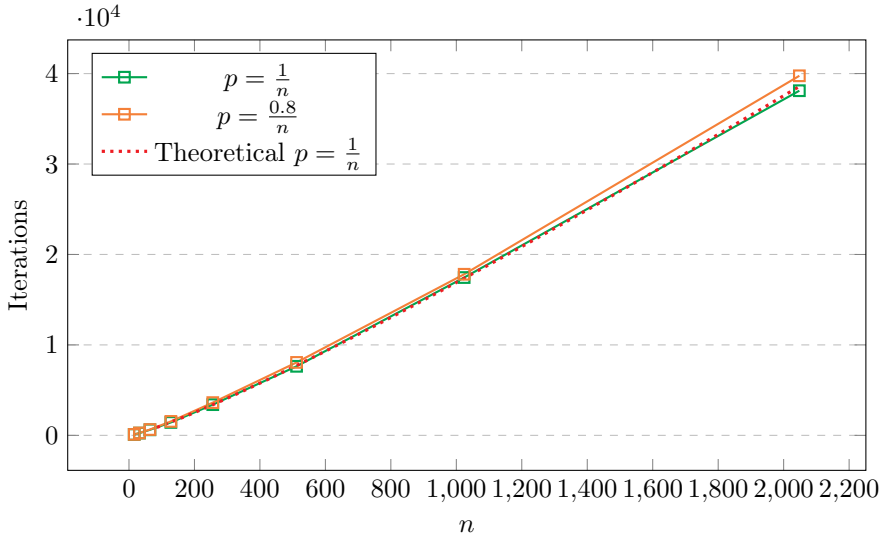


Figure 5.1: Iteration count of (1+1)EA on OneMax. Sample size 300.

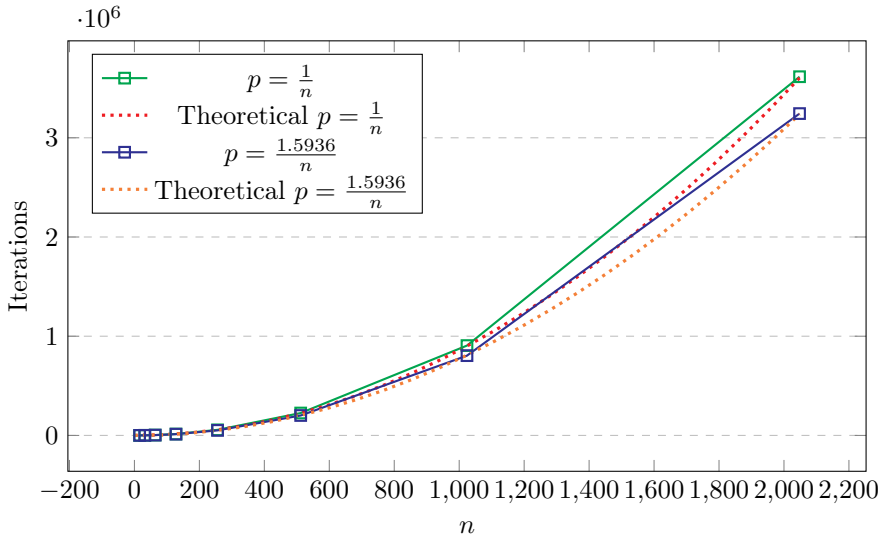


Figure 5.2: Iteration count of (1+1)EA on LeadingOnes. Sample size is 300.

5.1.3 TSP

The chart in figure 5.3 compares the fitness of (1+1)EA on berlin52 using a range of iteration limits, when $\lambda = 1$, i.e. we expect one 2-OPT move every iteration, and $\lambda = 2$. As discussed earlier, (1+1)EA has the entire search space as its neighbor, and would, given unlimited time, eventually reach optimum. However, this becomes increasingly unlikely the more moves would be needed in the same iteration. As shown in the chart, both searches quickly reaches a relatively good fitness at around 2000 iterations, and each iteration from there have diminishing improvements. When $\lambda = 1$ the average fitness was measured to be 8242 after 50,000 iterations. Though not shown in the chart, running the search for 20 times as many iterations (i.e. 1,000,000), this average was improved to be 8039. As expected, the search with $\lambda = 2$ leads to a search that starts off slower, but is better at improving the fitness once the search reaches a point where multiple changes need to be made at once. With this configuration, a search of 50,000 iterations is measured to have an average fitness of 8208.108. A search of 1,000,000 iterations is measured to an average fitness of 8009.012. Simulating berlin52 with $\lambda = 10$ over 1,000,000 iterations, resulted in an average fitness of 8878.706, with the lowest fitness achieved being 8057 in a total of 500 runs. This is a clear indication that a search will become ineffective with too much variation in each iteration. It is expected that this effect becomes less drastic with instances of more cities. Note that the search in some cases *increases* in fitness with more iterations, e.g. for $\lambda = 1$ at iteration 60000 to 70000. I have not found an explanation to this, other than variance. Altso note that the chart does not display from $y = 0$, making the results look further from optimum than they actually are.

5.2 Simulated Annealing and RLS

5.2.1 OneMax

The chart on figure 5.4 compares the iteration count for RLS and SA to reach optimum fitness. We see that the theoretical iteration count for RLS closely resembles that measured with this program. As expected, the RLS outperforms SA, as there are no local optima in the problem.

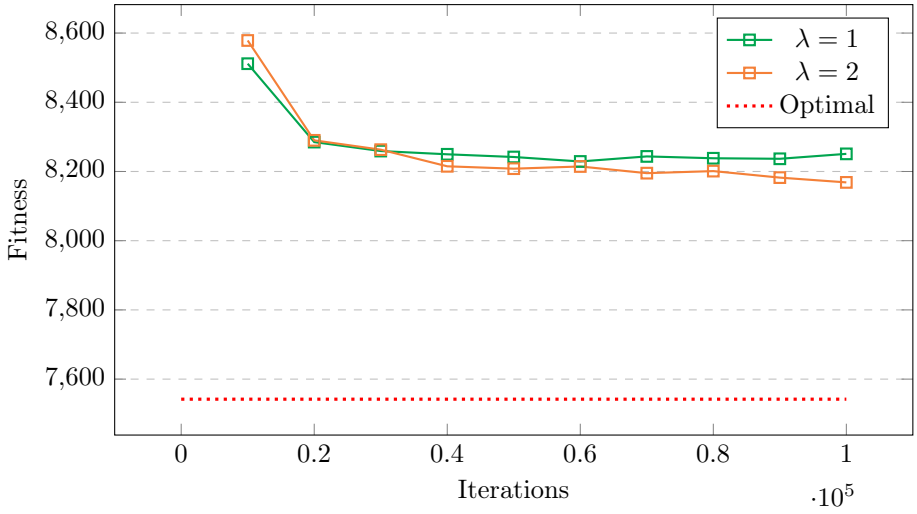


Figure 5.3: Fitness of (1+1)EA on berlin52. Sample size is 500.

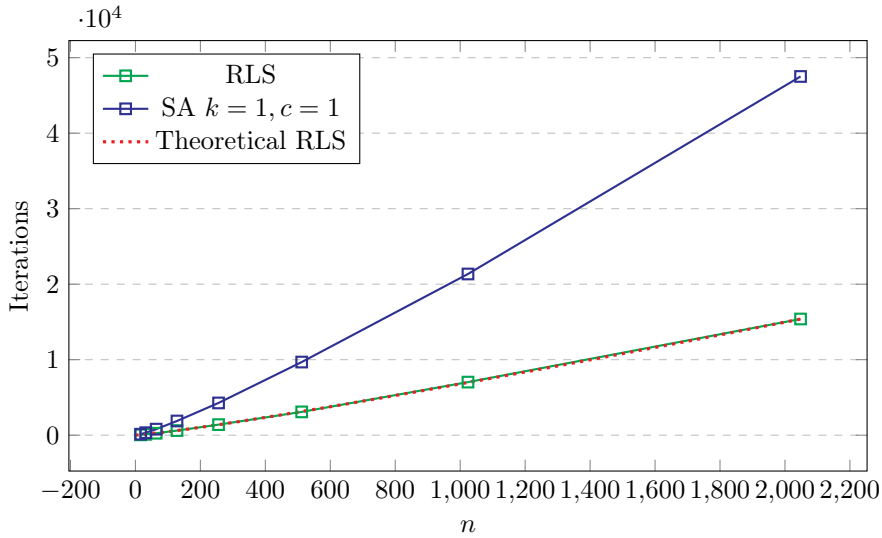


Figure 5.4: Iteration count of SA/RLS on OneMax. Sample size is 300.

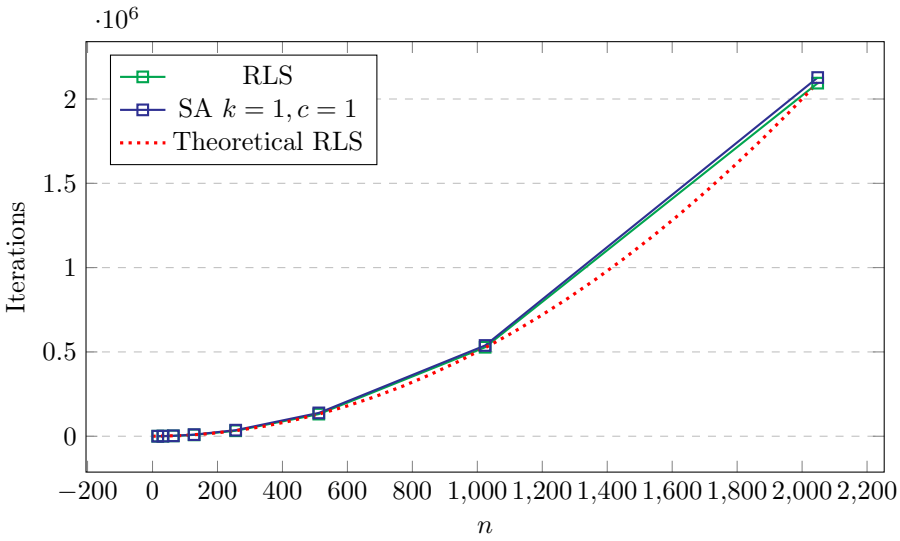


Figure 5.5: Iteration count of SA/RLS on LeadingOnes. Sample size is 300.

5.2.2 LeadingOnes

The chart on figure 5.5 also shows a close correlation between the theoretical iteration count and that measured empirically w.r.t. RLS on LeadingOnes. Again RLS outperforms SA, but in this case only slightly compared to OneMax. This is likely due to this search needing around 100 times more iterations to reach optimum, giving it more iterations to cool down, and thus turning into an RLS.

5.2.3 TSP

The chart on figure 5.6 shows the result of running SA with different configurations at different iteration limits. Using RLS on TSP, the search quickly reaches a stable fitness that in the case of berlin52 is $\approx 10\%$ higher than optimum. However, as this metaheuristic has no method of escaping local optima, it quickly gets stuck, making further iterations in vain. The chart also shows SA run with the default configurations of $k = 1$ and $c = 1$. The result is $\approx 7\%$ higher than optimum, using twice as many iterations as RLS. The run of $k = \frac{1}{52}$ is used for later discussion.

In the chart on figure 5.7, the temperature is set to cool 10 times slower than in figure 5.6, as $c = 10$. This also results in the simulation needing 10 times more

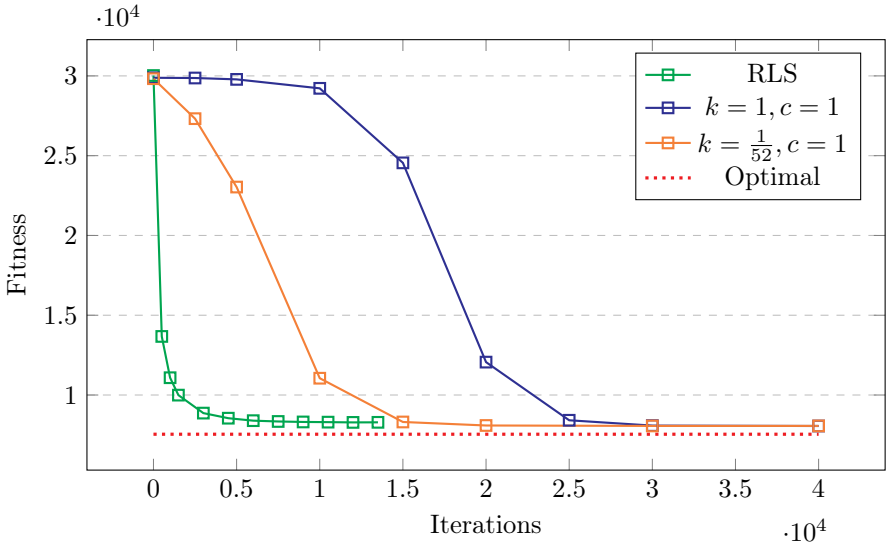


Figure 5.6: Fitness of SA/RLS on berlin52. Sample size is 500.

iterations to settle on a state. The resulting state when $k = 1$ is only $\approx 1.8\%$ higher than optimum. It is interesting to compare figure 5.6 and 5.7 in terms of their progression through the search. If one looks closely, the fitness values from iteration 20,000 / 200,000 and further look similar, but the latter has a slightly better fitness throughout it all.

When looking at figure 5.6, the point at which the simulation seemingly stops moving randomly around in the search space is around iteration 10,000. Under the assumption that the search before this point is not improving the final solution, one could simply start the search with the temperature at that point. Since there are 52 cities in th instance, this is calculated to be $T(10,000) = 52^3 \cdot (1 - \frac{1}{52^2})^{10,000} \approx 3480.12$. The temperature after this point is shown in figure 5.8. Interestingly, this temperature closely resembles the initial temperature if $T(1) = n^2$ instead of $T(1) = n^3$. To change the initial temperature, one can modify k to be $k = \frac{1}{n}$, which is plotted on figure 5.6 and 5.7. When $c = 10$, it is empirically tested that when $T(1) = n^3$, the search results only has $\approx 0.1\%$ better fitness score after 40,000 iterations compared to $T(1) = n^2$ on berlin52. However, this results in a search that is around 10,000 iterations slower at settling on a fitness. This suggests that the initial temperature is perhaps better set at $T(1) = n^2$, though this has not been tested extensively enough to say for certain.

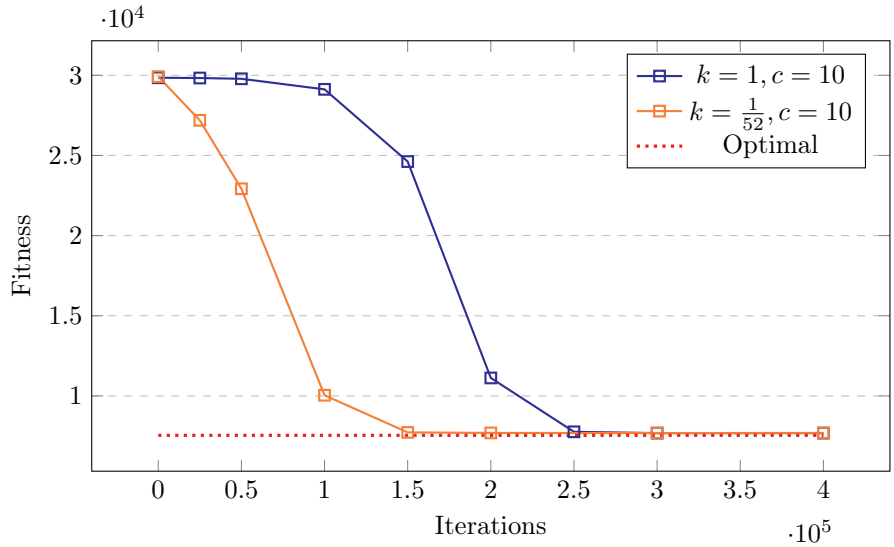


Figure 5.7: Fitness of SA on berlin52, with $k = 1$ and $c = 10$. Sample size is 500.

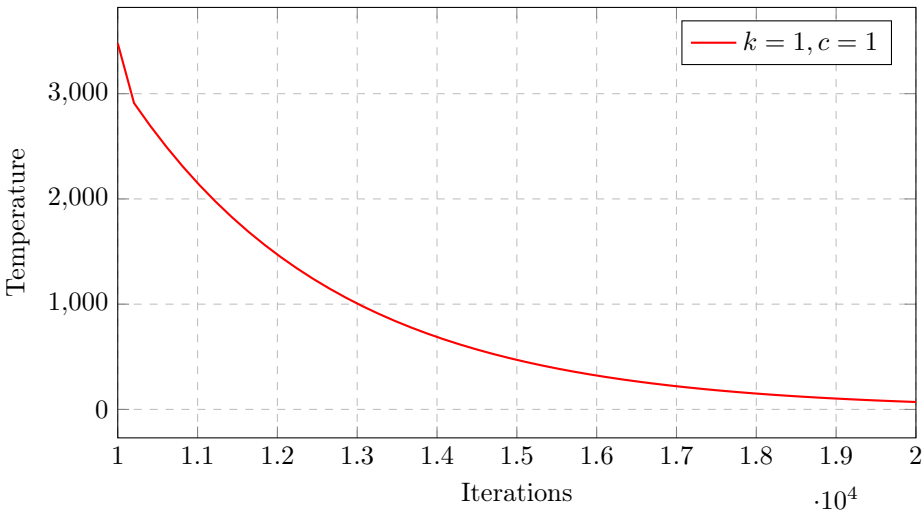


Figure 5.8: Temperature cooling of SA on berlin52.

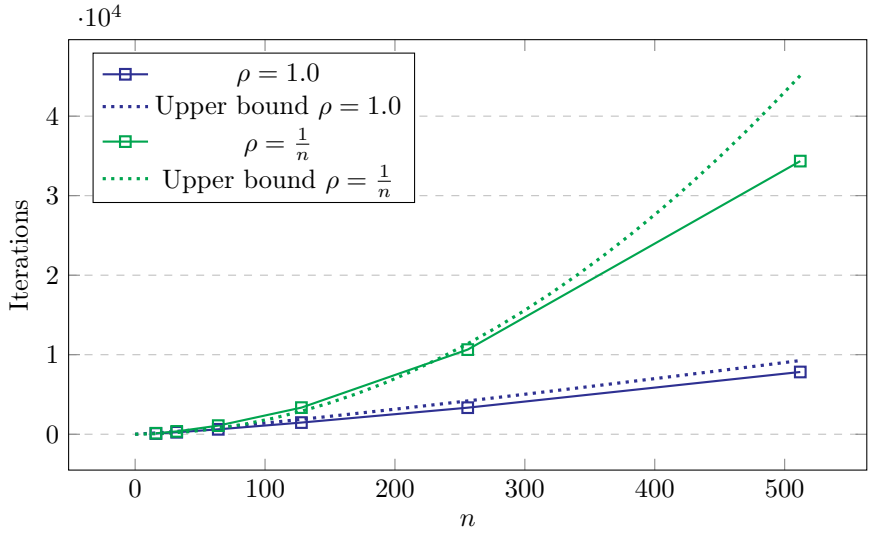


Figure 5.9: MMAS on OneMax. Sample size is 200.

5.3 MMAS*

5.3.1 OneMax

The chart on figure 5.9 shows the iteration count of MMAS* on OneMax to reach optimum compared to length of the string. As expected it seems that a higher evaporation factor results in a faster search, as there are no local optima in the problem. Note that because the upper bounds are asymptotic, a self-chosen constant has been chosen for each of them. Thus, these are not used as a prediction for the iteration count, but are included to show that they do indeed bound the algorithm from above.

5.3.2 LeadingOnes

The chart on figure 5.10 similarly shows the iteration count of MMAS* on LeadingOnes. Again we see that a high evaporation factor decreases the iteration count. We also see that both configurations seem to be correctly bounded both from above and below by our expectations. For both OneMax and LeadingOnes, it seems these bounds are tighter when $\rho = 1$, which makes it equivalent to (1+1)EA. This might be due to the fact that the bounds are derived from

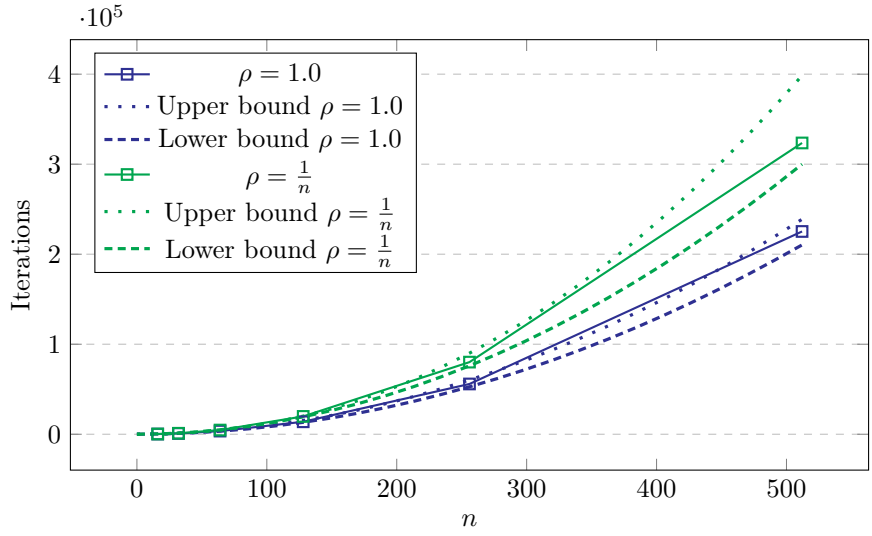


Figure 5.10: MMAS* on LeadingOnes. Sample size is 200.

(1+1)EA, which has been studied more extensively than MMAS*.

5.3.3 TSP

The chart on figure 5.11 shows the fitness value obtained by simulating MMAS* at various iteration counts and different values of ρ, α and β. We first notice that changing ρ from $\frac{1}{n}$ to $\frac{1}{2n}$ does not change much, and thus the two lines lie on top of each other. We then see that the fitness gets quite bad when the pheromone power α is set to 2, and β is only at 1. This configuration discourages the ant too much from exploring edges outside of the current state, and thus changes very little. Conversely, the configuration with a higher heuristic power performs better than all the others, at all times during the search.

5.4 Comparing the metaheuristics

This section will summarize the strengths and weaknesses of each metaheuristic explored in this project.

There are two key factors we look at, when we compare the performance of

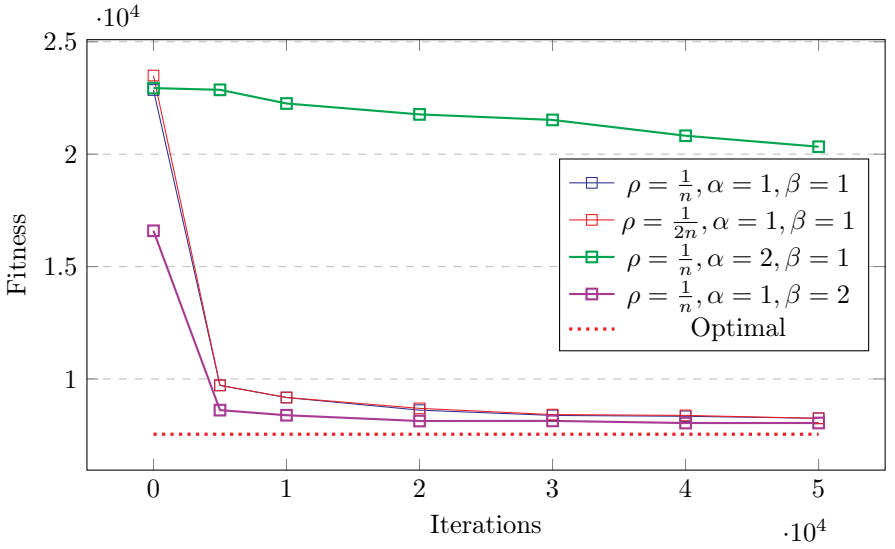


Figure 5.11: MMAS* on berlin52. Sample size is 50

metaheuristics: speed and accuracy. Throughout this whole chapter, the only speed measure used has been the number of iterations. This has allowed us to see the metaheuristic as a *black box*, which is independent of how the programmer has implemented it and the platform it is run on. In practice, however, the actual running time can have big impacts regarding the usefulness of the metaheuristic. Simply from the asymptotic running time, it is expected that (1+1)EA, RLS and SA have a significantly better running time than MMAS*.

The chart in 5.12 shows the accumulated time used at different iteration counts on TSP in berlin52. It is clear that MMAS* by far is the slowest of the four algorithms. SA is measured to be the second-slowest, but at 20000 iteration it is still ≈ 165 times faster than MMAS*.

For bit strings, the same thing is seemingly apparent, as shown with OneMax in the chart on figure 5.13. LeadingOnes also showed the running time of MMAS* to be significantly higher, though this is not shown.

Compared to RLS, (1+1)EA with $p = \frac{1}{n}$ is ≈ 2.5 times slower on OneMax, and ≈ 1.55 times slower on LeadingOnes. SA with $k = 1, c = 1$ is ≈ 3 times slower on OneMax, and nearly identical on LeadingOnes. In terms of iteration count, MMAS* with $\rho = 1$ performs identically to (1+1)EA, and varying the pheromone value does seem to improve significantly on OneMax and LeadingOnes. Comparing the running time of TSP is not as easy, as they each reach

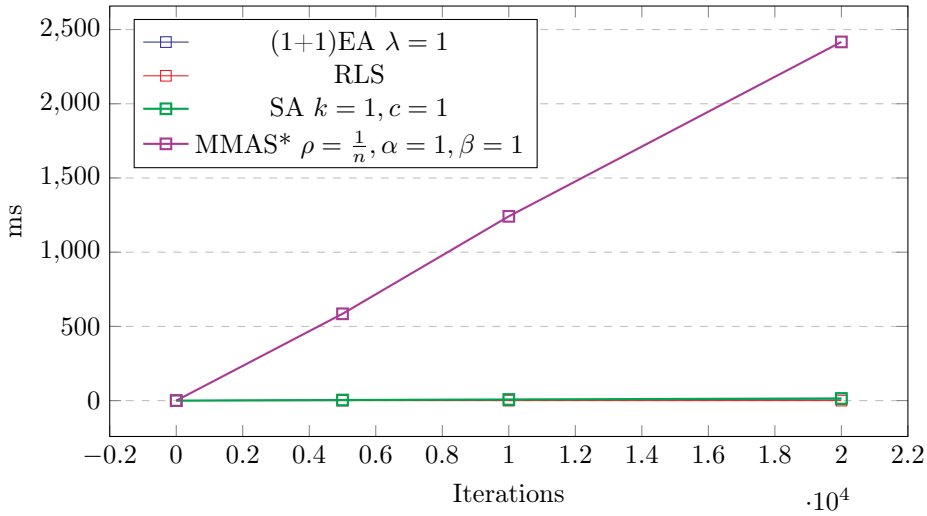


Figure 5.12: Running time of metaheuristics at different iteration counts on berlin52. Data taken from previous sections.

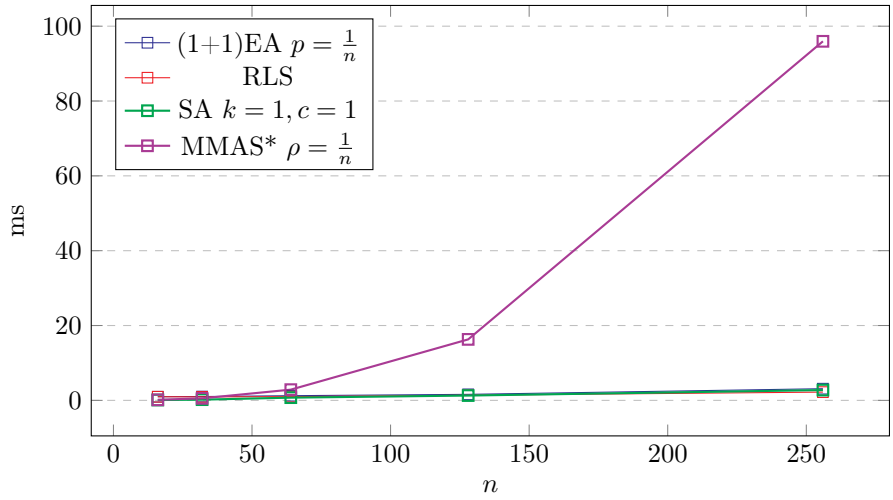


Figure 5.13: Running time of metaheuristics at different problem sizes of OneMax. Data taken from previous sections.

different fitness values. In addition, (1+1)EA and MMAS* always have the possibility of escaping local optima, making it unfair to compare the speed at which they reach this. Though generally, all metaheuristics except for SA quickly reach a state from where they only make slight modifications. (1+1)EA is about three-four times slower at reaching this than RLS. MMAS* reaches it quickly due to its built in heuristic, but at $\rho = \frac{1}{n}, \alpha = 1, \beta = 1$, it still does this slower than RLS. However, this can be sped drastically the higher the heuristic value is set, and at sufficiently high values simply turns into the *nearest neighbor* initialization heuristic.

Regarding the performance, all metaheuristics naturally perform equally good on bit strings, as they always reach the optimum. However, on TSP some metaheuristics seemingly produce better tours than RLS. The configuration tested so far that has yielded the best fitness scores was SA with $k = 1, c = 10$. After 500,000 iterations, this configuration returned the optimal score almost every second simulation, and on average a fitness score of just 7677. The configuration that resulted in the second-best fitness scores was MMAS* with $\rho = \frac{1}{n}, \alpha = 1, \beta = 2$, which only gave an optimal score in $\approx \frac{1}{50}$ simulations, but on average gave a score of 8042. With many iterations, (1+1)EA can achieve an acceptable average fitness, but has failed to get an average fitness below 8000, even after 1,000,000 iterations and $\lambda = 2$.

5.5 Pre-initialization of TSP

The program gives the option of initializing a TSP-tour using one of three heuristics: *nearest neighbor* (NN), *cheapest insertion* (CI) and *random insertion* (RI). When discussing the quality of the resulting tour, there are two major things to consider; the fitness value and how well it can be improved. In the case for TSP, the fitness value is considered good if it is low, and can be tested by simply observing the average fitness value after running it a number of times. This was tested on berlin52, and the result of this is shown in figure 5.14. Compared to the default, random initialization, all heuristics find a fitness value lower than a third. We observe that NN produces the worst fitness value of the three heuristics, and RI the best. It is interesting to note that RI outperforms CI, as one might intuitively think a tour would be worse by adding random cities, opposed to only adding best possible option. It is not surprising that NN performs the worst, as it only considers the possibility of adding a city next to *one* other, instead of the *whole* tour built so far.

How well the state can be improved is somewhat harder to test. What we would like to examine is if the heuristic causes the search to end up near a “bad” or

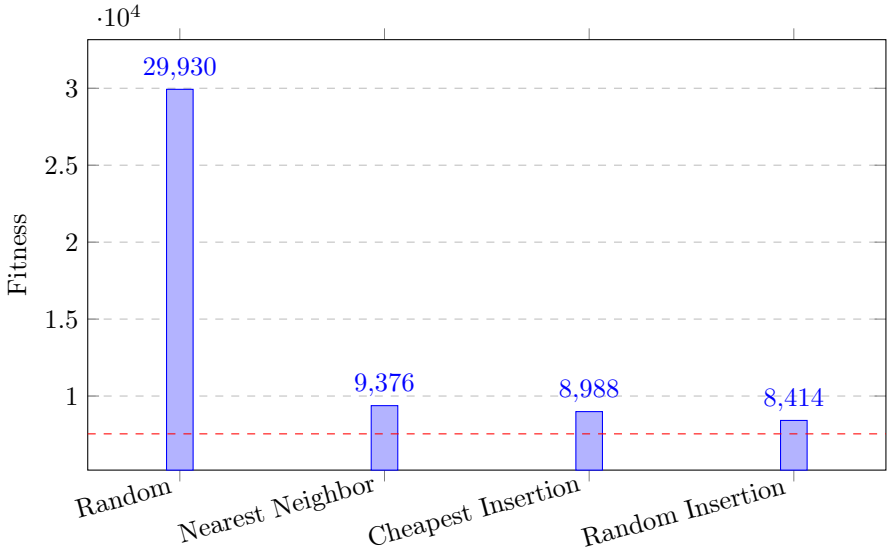


Figure 5.14: Fitness of different initialization methods on berlin52. The red line shows the optimum fitness. Sample size is 1000.

"good" optima. To test this, I have chosen to run RLS for 15,000 iterations on berlin52, after being initialized with each of the heuristics. I chose 15,000 iterations, since the search seems to have found an optimum at this point, when starting on a random permutation (see the chart on figure 5.6). The results are shown in figure 5.15. Interestingly, the three heuristics seem to produce very varying results. NN is the only initialization heuristic that clearly seems to find an optima better than than starting with a random permutation. Initializing with CI seems to lead the search into a high optima. RI seems to improve the optima found slightly, but not significantly. There is no clear correlation between the initial fitness value and the resulting fitness after running the metaheuristic. I have yet to find a good explanation as to why NN outperforms all other methods of initialization, and as suggested in appendix B, it is not just the case for this TSP-instance. It might be that the initialization algorithm should avoid ending too close to a local optima, so the metaheuristic has a better chance at *avoiding* it. This could explain why CI performs so poorly, as it minimizes the total tour length, and thus always keeps the search close to an optimum. Because RI still has a random element there might still be room for improvement by RLS. However, it seems counter-intuitive that RI should end further from an optimum, as it performed better on figure 5.14.

Finally, I would like to compare the fitness obtained when initializing the tour using NN and finalizing it with the four metaheuristics explored in this project.

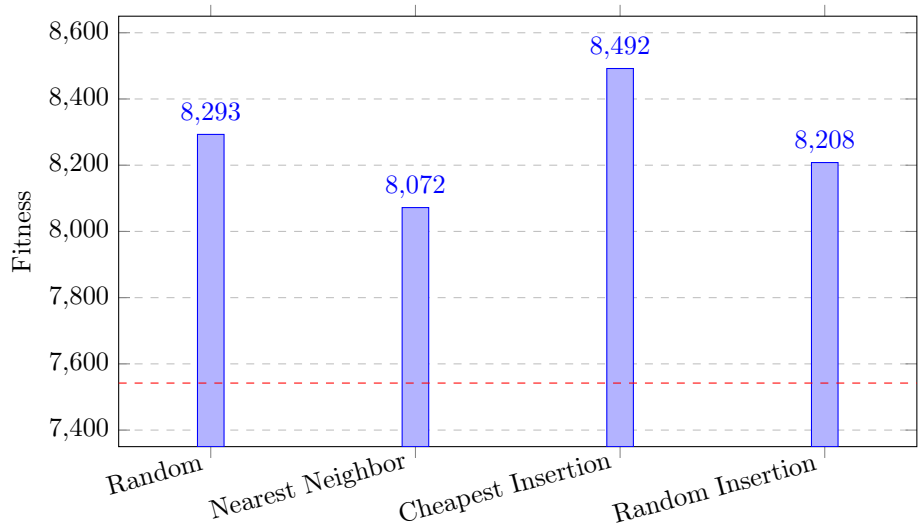


Figure 5.15: Fitness after 15,000 iterations using RLS on berlin52. Each column shows which method was used for initialization. The red line shows the optimum fitness. Sample size is 1000.

The results are shown in figure 5.16. Each simulation was performed using 50,000 iterations, to ensure they all at least came close to an optimum, especially that SA reaches a sufficiently low temperature. (1+1)EA uses $\lambda = 1$, SA has $k = 1, c = 1$ and MMAS* has $\rho = \frac{1}{n}, \alpha = 1, \beta = 1$. The first thing we see, is that all four metaheuristics reach similar fitness values, and that the difference between them might simply be from variance. Though the evidence is not strong, it is at least suggested that MMAS* outperforms the others. When MMAS* begins the search, all pheromone values are equal, meaning it gets to *explore* nearby states, while not straying too far off since it only accepts better candidates. When MMAS* reaches a state similar to that given by NN, it usually has most of the pheromone values set to either τ_{min} or τ_{max}. This may suggest that the MMAS* in general could benefit from *resetting* its pheromone values once it stagnates. Especially since NN is very similar to the heuristic used by MMAS* itself.

Compared to random initialization, none of the four metaheuristics seem to perform worse when initialized using NN. The metaheuristic that seems to get the most benefit from NN is RLS, closely followed by (1+1)EA. Since (1+1)EA with a high mutation rate is better at improving an already good state, a combination of this with NN could lead to good results. There seems to be no benefit

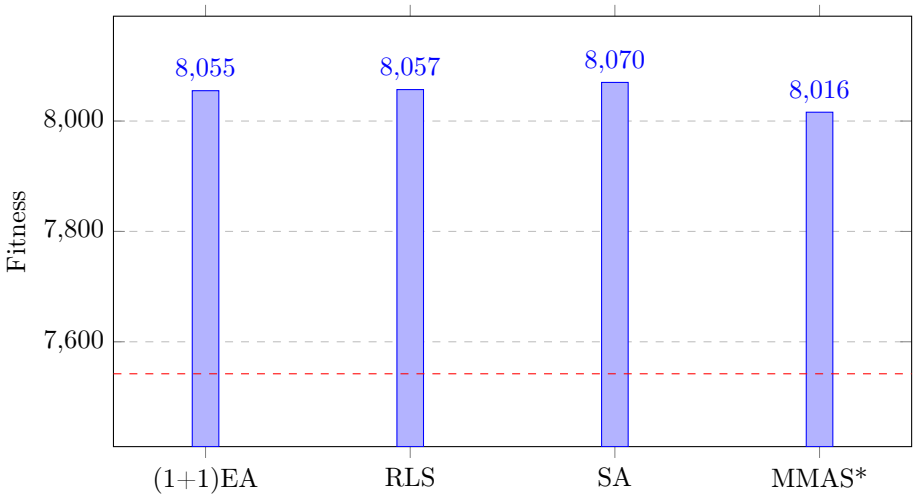


Figure 5.16: Fitness after simulating 50,000 iterations on berlin52, after being initialized with NN. Sample size is 400.

of initializing SA with NN. This is because SA starts with a temperature so high that it *undoes* the initialization, scrambling it to a state similar to that gained from random initialization. It could be interesting to find a method for calculating a temperature where it would only make small, local changes, and see if this would lead to better fitness values after being initialized with NN.

Chapter 6

Conclusion

The project has succeeded in creating a framework that can both visualize and evaluate the working principles of nature-inspired optimization metaheuristics. This is done by creating a Java framework that allows for easily setting up a simulation that can visualized, or tested multiple times using a template file that logs the results to a file on the computer. The implemented optimization problems are found to closely resemble that proven in theory, and the consequences of modifying the variables in each metaheuristic have been explored. Furthermore, the impact of initializing the search with a relatively fast approximation algorithm has been discussed, in particular the combination of initialization with *Nearest Neighbor* and the very simple *Random Local Search* metaheuristic. *Simulated Annealing* is found to be a very efficient metaheuristic, if the initial temperature and cooling scheme are appropriately adjusted. *(1+1)Evolutionary Algorithm* is found to provide acceptable results in relatively good time, and *Min-Max Ant System* is found to provide relatively good solutions, but does so at the cost of a long running time.

6.1 Future Outlook

This section will summarize areas that would benefit the most from additional work. A large issue in the project is the lack of automated testing such as unit

testing. This has in part been due to the random nature of the simulations, making it hard to predict an outcome of a test. However, this could have been avoided by using *seeds* for random number generation, or the ability to initialize the search with a specific state, though this would require a lot of work to ensure that each iteration in the test was performing as intended.

In terms of the program itself, the user interface could be improved. Though this was never a focus of mine, it could benefit from looking nicer, but also being easier to handle in code. Each window for editing options in the configuration is created from its own independent class. This introduces quite a lot of boilerplate code, that could be avoided by using an abstract class. Additionally, the *flow* in the code could use more classes, increasing the cohesion of each class. This would significantly increase the readability of the code.

It would be interesting to go into further detail on why NN seems to create states that are much better to improve compared to CI and RI.

There is currently no way of actually seeing the solution that was found in the search, apart from guessing what a string looks like from the boolean hypercube, or studying the visualization of a TSP tour. This would be nice to include in the user interface and the log created from a template.

Appendix A

Template keywords

Here are all the currently implemented keywords, that a template may contain. The value of this keyword can either be one of a set of predefined words, or just any value of a given data type. This is respectively shown either by the keyword's sub points, or by the value written in *italics*. The order in which the template file is written does not matter.

- `SEARCH_SPACE`
  - `BIT_STRING`
  - `PERMUTATION`
- `PROBLEM`
  - `LEADING_ONES`
  - `ONE_MAX`
  - `SYMMETRIC_TSP`
- `HEURISTIC`
  - `BIT_(1+1)EA`
  - `BIT_RANDOM_LOCAL_SEARCH`

  - BIT_SIMULATED_ANNEALING
  - BIT_MIN_MAX_ANT_SYSTEM
  - TSP_(1+1)EA
  - TSP_RANDOM_LOCAL_SEARCH
  - TSP_SIMULATED_ANNEALING
  - TSP_MIN_MAX_ANT_SYSTEM
- STOP_CONDITION
  - STAGNATION
  - TARGET
  - TIME_LIMIT
  - ITERATION_LIMIT
- TSP_INITIALIZATION
  - NEAREST_NEIGHBOR
  - CHEAPEST_INSERTION
  - RANDOM_INSERTION
- STOP_STAGNATION_VALUE: *int*
- STOP_TARGET_VALUE: *int*
- STOP_TIME_VALUE: *int*
- STOP_ITERATION_LIMIT_VALUE: *int*
- BIT_STRING_LENGTH: *int*
- TSP_FILE_PATH: *String*
- (1+1)EA_FLIP_CHANCE_PER_N: *int*
- SA_INIT_TEMP: *double*
- SA_COOLING_RATE: *double*
- MMAS_RHO: *double*
- MMAS_ALPHA: *double*
- MMAS_BETA: *double*

Appendix B

Simulations on bier127

All simulations on TSP in chapter 5 use the instance berlin52. In this appendix, all the same simulations are done on the instance bier127, to validate the findings are not just specific to berlin52.

(1+1)EA

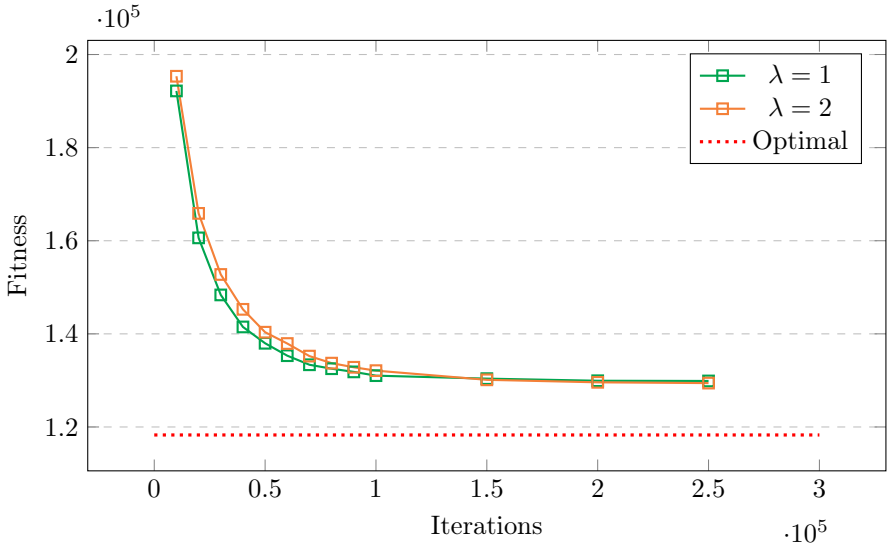


Figure B.1: Sample size is 500.

RLS/SA

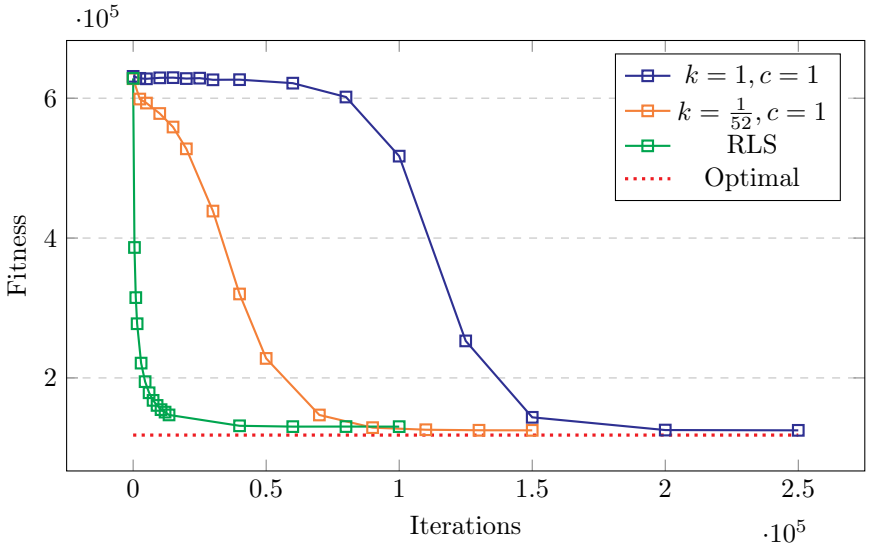


Figure B.2: Sample size is 500.

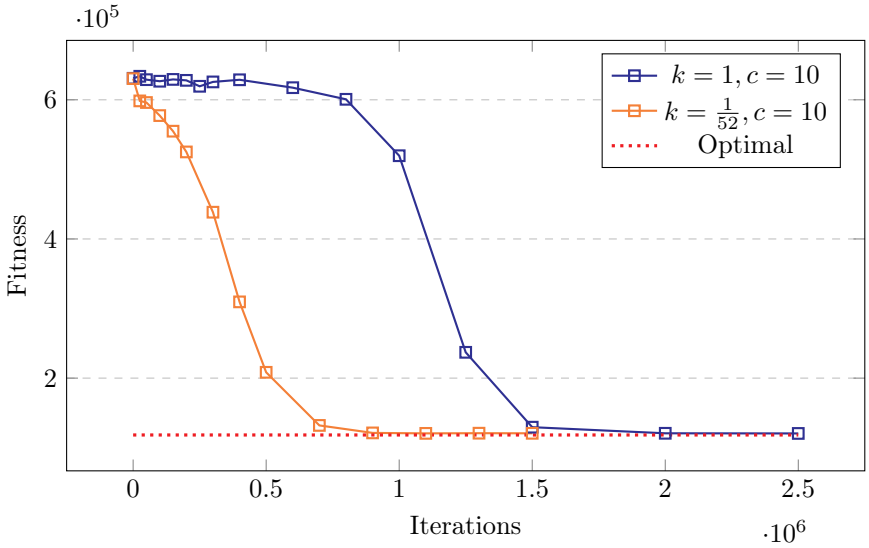


Figure B.3: Sample size is 50.

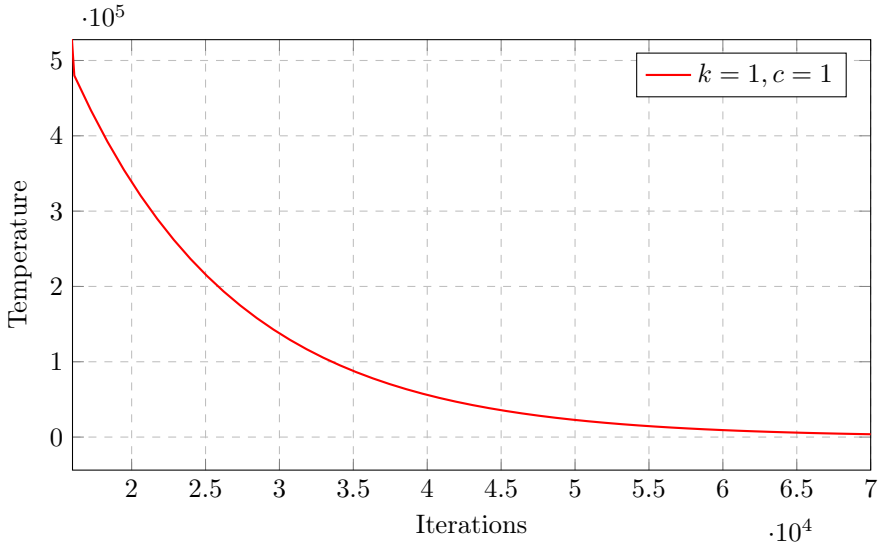


Figure B.4: Temperature cooling of SA on bier127.

MMAS*

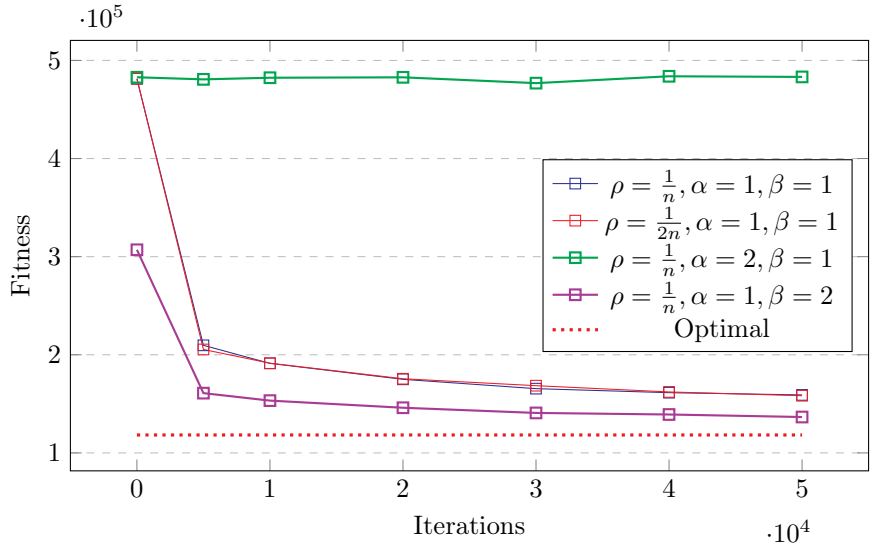


Figure B.5: Sample size is 50.

Pre-initialization of TSP

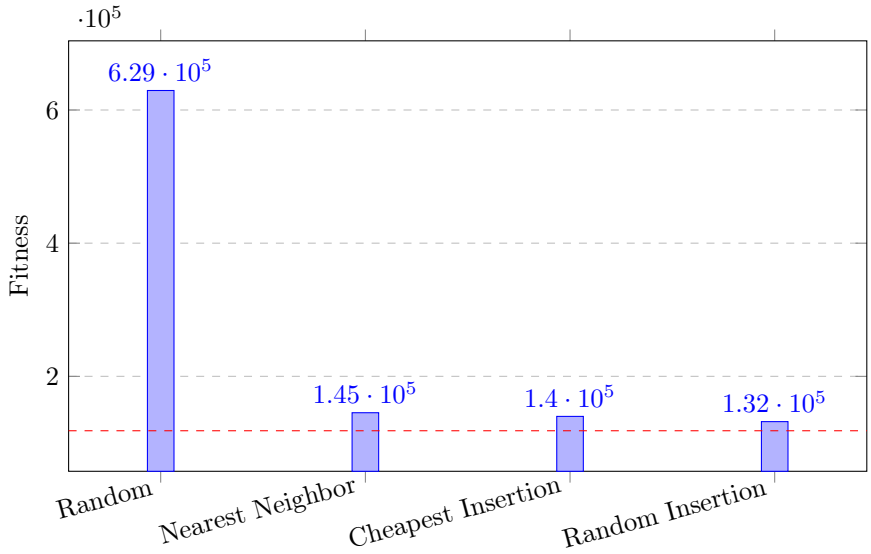


Figure B.6: Fitness of different initialization methods on bier127. The red lineshows the optimum fitness. Test was performed using 1000 samples.

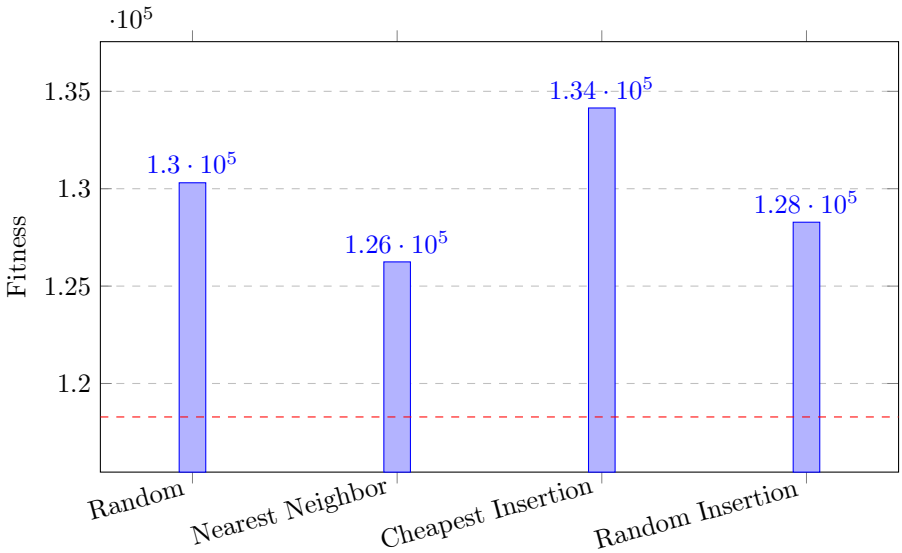


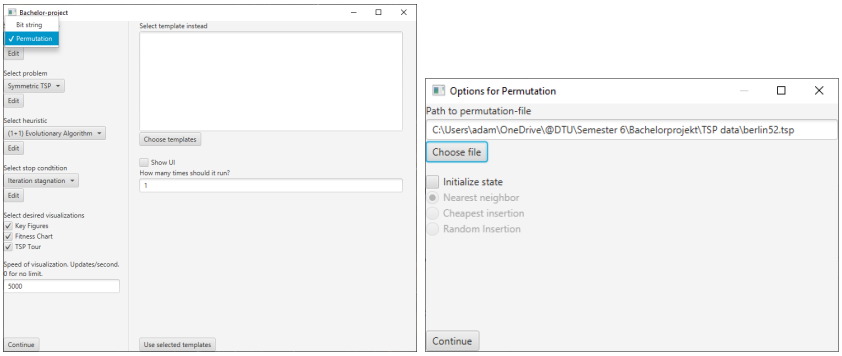
Figure B.7: Fitness after 100,000 iterations using RLS on berlin52. Each column shows which method was used for initialization. The red line shows the optimum fitness. Test was performed using 1000 samples.

Appendix C

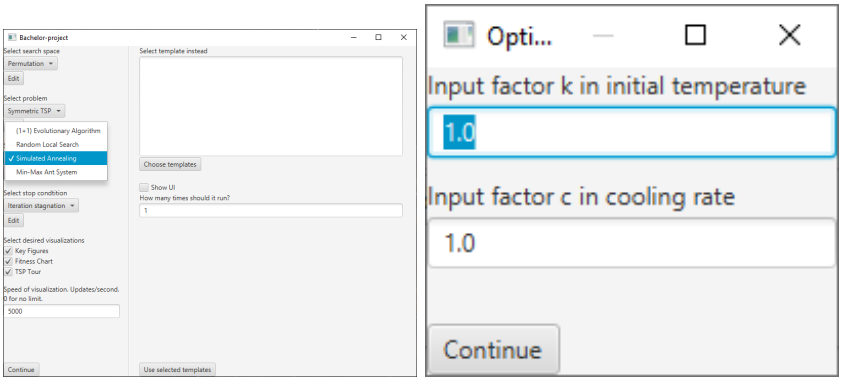
User guide

C.1 Setting configurations with the UI

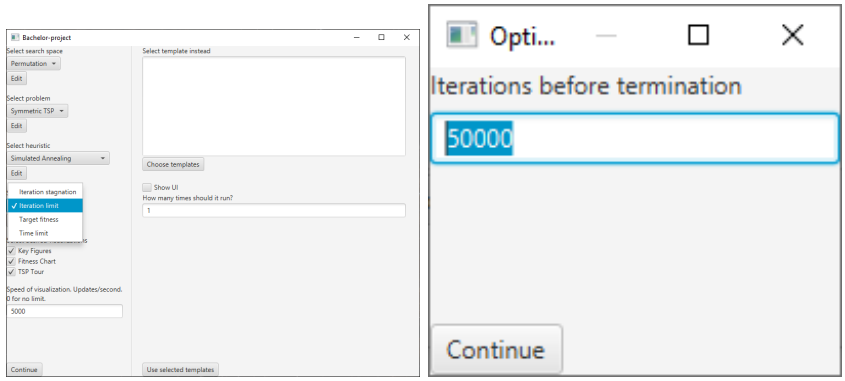
To run a simulation without a template file, all configurations are selected from the main screen. This example shows how to run a simulation in the *permutation* search space, on the problem *TSP* using the metaheuristic *simulated annealing* and the stopping criteria *iteration limit*. Firstly, we select the permutation search space and click the "edit"-button below. Here we select the path to the file collected from TSPLIB [Rei97]. We want the initial permutation to be random, so we do not select the associated checkbox.



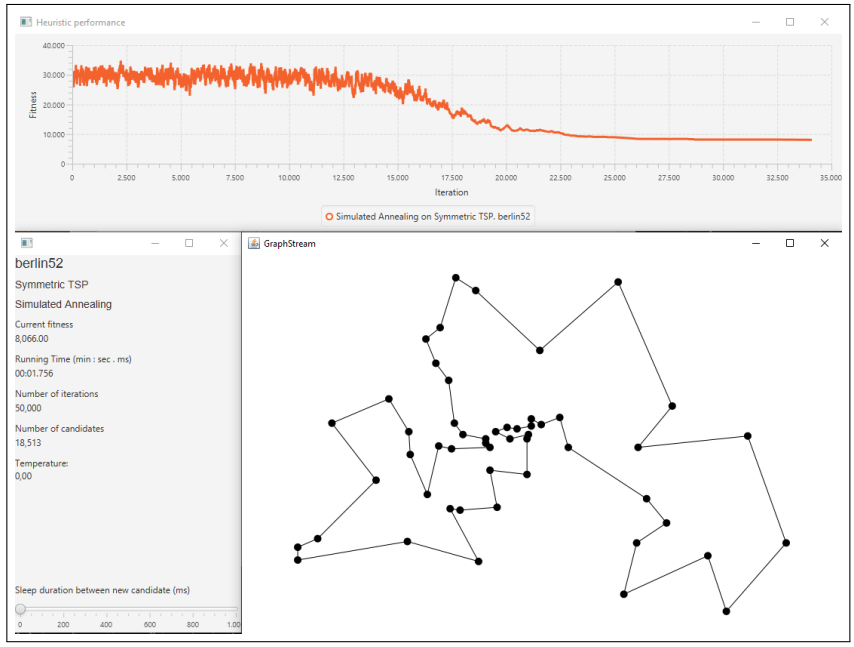
Since the TSP-problem is already selected leave the problem choice box as is. Next, we select simulated annealing as our metaheuristic. We can edit the initial temperature and cooling rate by pressing "edit".



Next we choose the stopping criteria to be *iteration limit*. We edit the iteration count to be 50,000.

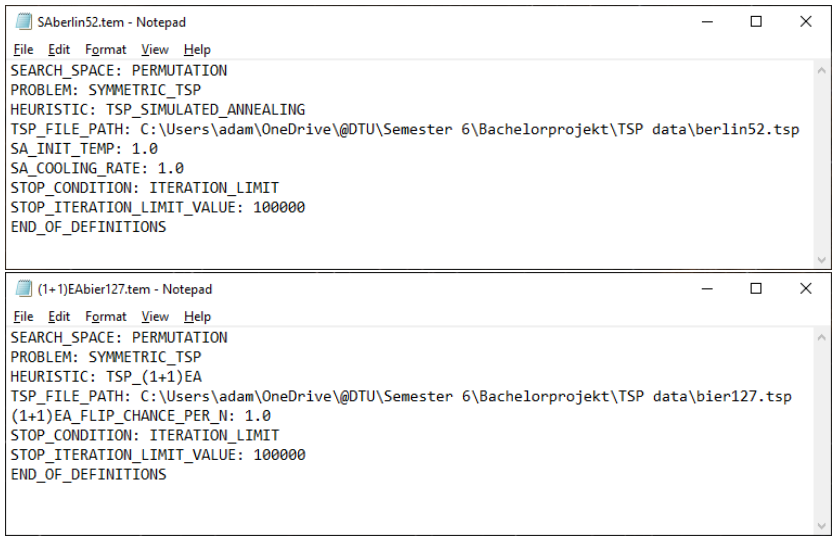


Lastly we select the visualizations we want to appear and the simulation speed. Now the configuration has finished, and we press "continue". The selected visualization will appear, and the simulation begins. The start screen is still open, and is ready to run other simulations if wanted.

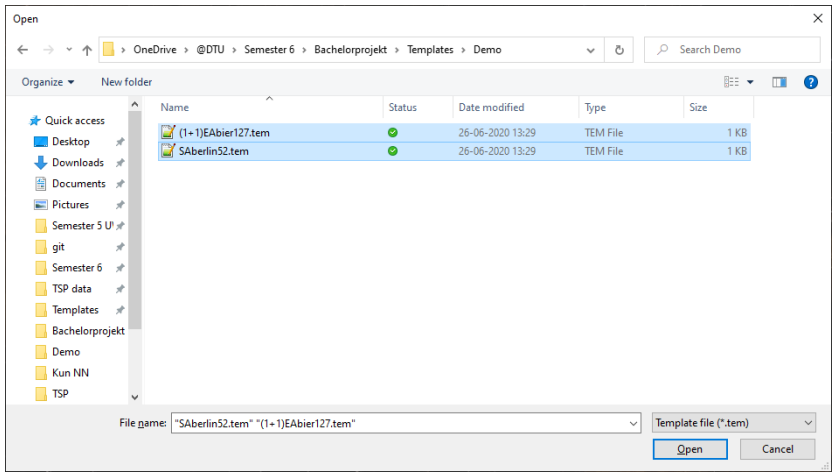


C.2 Using a template file

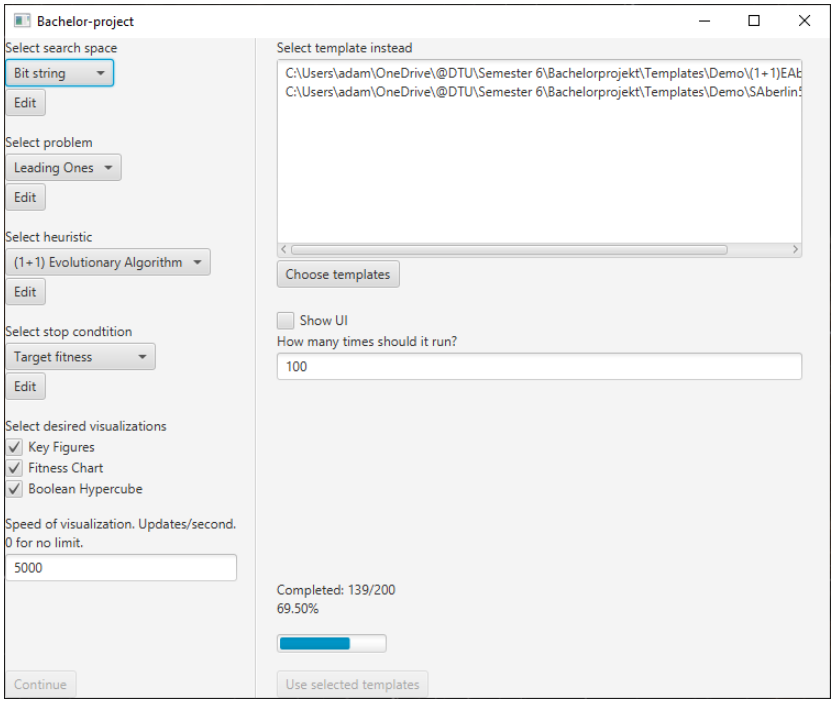
Say we would like to run simulations from two template files 100 times. We first create the template files using the key words shown in appendix A.



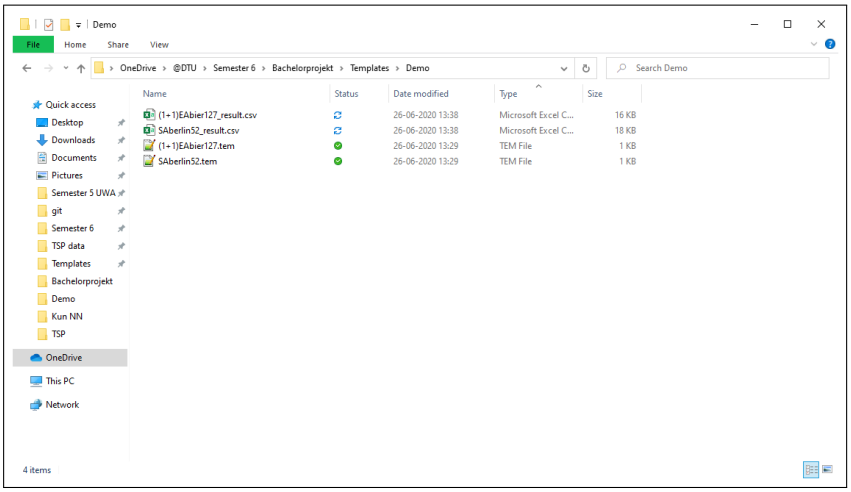
Next we press the button “Choose templates” in the program, and select the two templates we created.



We then write “100” in the text field asking for the number of simulations, and press the button “Use selected templates”. The simulations are now running.



When the progress bar is filled, the simulations are complete. The result of the simulations are now stored in the same folder created the templates.


Demo
File
Home
Share
View
OneDrive › @DTU › Semester 6 › Bachelorprojekt › Templates › Demo
Search Demo
Quick access
Desktop
Downloads
Documents
Pictures
Semester 5 UWA
git
Semester 6
TSP data
Templates
Bachelorprojekt
Demo
Kun NN
TSP
OneDrive
This PC
Network
Name
Status
Date modified
Type
Size
(1+1)EAbier127_result.csv
26-06-2020 13:38
Microsoft Excel C...
16 KB
SAberlin52_result.csv
26-06-2020 13:38
Microsoft Excel C...
18 KB
(1+1)EAbier127.tem
26-06-2020 13:29
TEM File
1 KB
SAberlin52.tem
26-06-2020 13:29
TEM File
1 KB
4 items

Bibliography

[BDN10] Süntje Böttcher, Benjamin Doerr, and Frank Neumann. Optimal fixed and adaptive mutation rates for the leadingones problem. In Robert Schaefer, Carlos Cotta, Joanna Kołodziej, and Günter Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, pages 1–10, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier, New York, 1976.

[BR03] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, September 2003.

[DD16] Benjamin Doerr and Carola Doerr. The impact of random initialization on the runtime of randomized search heuristics. *Algorithmica*, 75(3):529–553, 2016.

[DJW02] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1):51 – 81, 2002.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, USA, 2004.

[Dur64] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, July 1964.

[ES15] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd edition, 2015.

[Fre19] Tim French. Optimisation. University lecture CITS3001, `https://teaching.csse.uwa.edu.au/units/CITS3001/lectures/`, August 2019. Accessed: June 2020.

[FY38] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural aad medical research.* Oliver and Boyd, 1938.

[Hal] Steven et al. Halim. Visualgo. `https://visualgo.net/bn/sorting`. Accessed: June 2020.

[HK61] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. In *Proceedings of the 1961 16th ACM National Meeting*, ACM '61, page 71.201–71.204, New York, NY, USA, 1961. Association for Computing Machinery.

[HPR+18] Hsien-Kuei Hwang, Alois Panholzer, Nicolas Rolin, Tsung-Hsi Tsai, and Wei-Mei Chen. Probabilistic analysis of the (1+1)-evolutionary algorithm. *Evolutionary Computation*, 26(2):299–345, 2018. PMID: 28632396.

[JW04] Thomas Jansen and Ingo Wegener. Freak – free evolutionary algorithm kit. `https://sourceforge.net/projects/freak427/files/FrEAK/0.2/`, Feb 2004. Accessed: June 2020.

[JW06] Thomas Jansen and Ingo Wegener. On the local performance of simulated annealing and the (1+1) evolutionary algorithm, 2006.

[JZ11] 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*, FOGA '11, page 1–14, New York, NY, USA, 2011. Association for Computing Machinery.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KNRW12] 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.

[KNSW11] 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*, FOGA '11, page 209–218, New York, NY, USA, 2011. Association for Computing Machinery.

[Mee07] Klaus Meer. Simulated annealing versus metropolis for a tsp instance. *Information Processing Letters*, 104(6):216 – 219, 2007.

[NA98] Yaghout Nourani and Bjarne Andresen. A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, 31(41):8373–8385, oct 1998.

[NSW09] Frank Neumann, Dirk Sudholt, and Carsten Witt. Analysis of different mmas ACO algorithms on unimodal functions and plateaus. *Swarm Intelligence, v.3, 35-68 (2009)*, 3, 03 2009.

[NW10] Frank Neumann and Carsten Witt. *Bioinspired Computation in Combinatorial Optimization: Algorithms and Their Computational Complexity*. Natural Computing Series. Springer, 2010.

[PDGO08] Yoann Pigné, Antoine Dutot, Frédéric Guinand, and Damien Olivier. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. *CoRR*, abs/0803.2093, 2008.

[Rei97] Gerhard Reinalt. Tsplib. `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html`, Feb 1997. Accessed: May 2020.

[RN09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[RSI77] Daniel Rosenkrantz, Richard Stearns, and Philip II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6:563–581, 09 1977.

[STW05] Jens Scharnow, Karsten Tinnefeld, and Ingo Wegener. The analysis of evolutionary algorithms on sorting and shortest paths problems. *JMMA. Journal of Mathematical Modelling and Algorithms [electronic only]*, 4, 1 2005.

[Sud10] Dirk Sudholt. General lower bounds for the running time of evolutionary algorithms. In Robert Schaefer, Carlos Cotta, Joanna Kołodziej, and Günter Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, pages 124–133, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[WIT13] 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.

[Zho09] Y. Zhou. Runtime analysis of an ant colony optimization algorithm for tsp instances. *IEEE Transactions on Evolutionary Computation*, 13(5):1083–1092, 2009.