Danmarks Tekniske Universitet

Bachelor Thesis

Visualization and evaluation of nature-inspired metaheuristics for optimization problems

s224758
Silas Thule Mackrill

13. June 2025

Summary

This thesis presents the design and development of a web-based application for visualizing and interacting with nature-inspired optimization metaheuristics. The primary goal of the application is to provide an intuitive and extensible platform for exploring how different algorithms solve optimization problems over time. Users can select between problem types—such as bit string optimization (OneMax and Leading Ones) and the Traveling Salesman Problem (TSP) and observe how various algorithms improve their solutions through different display features and performance graphs.

The system is built with scalability and modularity in mind, allowing for easy integration of additional algorithms, problems, and experiment types. The backend handles computation and data aggregation, while the front end focuses on usability and dynamic presentation. To demonstrate the application's capabilities, a series of experiments were conducted comparing the (1+1) Evolutionary Algorithm, Simulated Annealing, and Max-Min Ant System. These evaluations support the utility of the tool in analyzing algorithm behavior and highlight how performance can vary depending on problem structure and parameter settings. The results of these experiments were analyzed and compared to those of already established theory. Overall, the application acts as a visualization and evaluation tool for metaheuristics applied to optimization problems.

Contents

1 Introduction

Nature-inspired metaheuristics can be strong solutions to problems where exact solutions are computationally infeasible. These are optimization problems where instead of finding the best solution, you iteratively improve a solution until it becomes good enough. This thesis investigates three algorithms: (1+1) Evolutionary Algorithm, Simulated Annealing, and Max-Min Ant System and applies them to two classes of optimization problems: bit string-based challenges (OneMax and Leading Ones) and the Traveling Salesman Problem (TSP). The Objective of this project is to create an application to visualize and evaluate these algorithms against the specified problems. A web application was developed that enables users to visualize algorithm behavior and performance interactively. A design goal of this application was to provide an easily scalable and extendable framework, so more algorithms and problems could be added later.
The project consists of three main parts: the theoretical foundation of the algorithms and problems, the design and implementation of the web-based system, and a set of experiments aimed at comparing algorithm performance across various conditions. Users can configure and run experiments directly from the front end, then explore the results through dynamic visualizations and graphs. Different experiment modes are available to either step through algorithm iterations, compare final solutions, or evaluate performance across multiple problem sizes. The interface was designed to be intuitive and accessible, while the backend was built with modularity in mind to support future expansion. Overall, the application bridges the gap between abstract algorithm theory and practical experimentation, making it easier to explore how optimization algorithms behave in real-time scenarios.

2 Theory

Starting with a real-world example of the problem: Imagine that you have 10 letters which you want to hand deliver to 10 friends who live in different cities. You want to visit each person in one trip and return home, but you want to do so in a time/fuel efficient way by taking the shortest overall route. The perfect solution can be found by checking all possible routes and seeing which one is the shortest. The problem is that the amount of solutions scale by n! where n is the number of cities. The problem is known as NP-hard meaning it is at least as hard as the hardest NP-problems which are problems without deterministic solutions in polynomial time. So if we can't find the best solution, what we can try is to find a good enough solution.

2.1 Graph Theory

A basic understanding of graph theory is necessary to understand The Traveling Salesman Problem and it's solutions[1]. To simplify the problem of finding a route between cities, we abstract the problem using graphs. A graph consists of vertices and edges, where the vertices are our cities and edges are the roads that connect them [1].

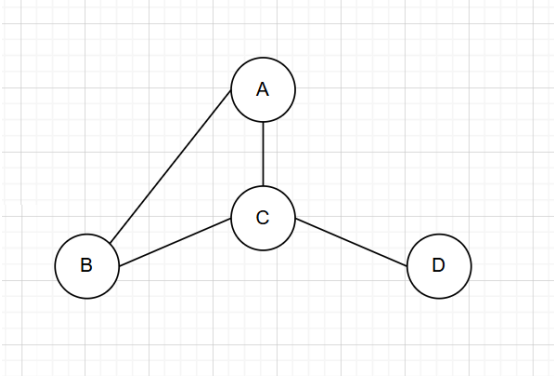


Figure 1: Graph

The edges can then have weights which could correspond to the distance between two cities in the real world. To find a good solution to our problem we first need to define what a solution to the problem even is. We will do this by first defining a walk and then work our way to what a solution to the salesman problem would look like.

- A walk is a non-empty alternating sequence of vertices and edges $v_0e_0v_1e_1...e_{k-1}v_k$ of a graph G where k is the length of the walk and e_i is an edge connecting v_i and v_{i+1} in G.
- A trail is a little more restricting than a walk. All trails are walks, but only walks without repeated edges are classified as trails.
- A path is a trail without any repeated vertices. This is important as we don't want a solution that visits the same place twice as that would be a waste.
- A cycle is a path except that it has to finish in the same vertex as it started. This is also part of our delivery problem, as we want to return home when we are done.
- A Hamiltonian cycle is a cycle where every vertex is visited exactly once. A Hamiltonian cycle can then be used to solve our problem as we need to every single vertex and return.

[1]Reinhard Diestel: Graph Theory, 6th ed., vol. 173 (Graduate Texts in Mathematics), 6th Edition, 2025, URL: https://doi.org/10.1007/978-3-662-70107-2.

Now we know the solution to the traveling salesman problem has to look like a Hamiltonian cycle. The deeper issue of TSP is; what if there are more than one Hamiltonian cycle in our graph as this is very likely in a real world example. That question is the basis of this thesis. We not only need to find a solution but the best solution. As mentioned in the beginning of this section, finding the best solution might not always be feasible, so instead we need a way to find good solutions in a practical amount of time.

2.2 Nature Inspired Algorithms and Bit Strings

There are multiple algorithms that can be applied to TSP that can iteratively improve a solution, thus improving the travel cost / distance. This paper will focus on algorithms that are based on observations in nature. These algorithms are so-called metaheuristic algorithms, which means that they themselves are high level procedures that select lower level procedures to find good enough solutions to optimization problems. The optimization algorithms which will be investigated in report are: (1 + 1) Evolutionary Algorithm, Simulated Annealing and Max-Min Ant System which is a variation of Ant Colony Optimization algorithms. Before applying these algorithms to the TSP, they will also be analyzed in a different and simpler search space; namely bit strings.

In this search space we are exploring 2 problems: OneMax and Leading Ones. In OneMax the goal of the algorithm is to start with a random string of bits, e.g. 00101100, then iteratively flip bits in the string and compare the new solution to the previous to reach the maximum amount of ones: 11111111. Leading Ones is a little different; here the fitness of a solution is based on how many consecutive ones there are, starting from the left side. The fitness functions F_{Max} and F_{Lead} return the fitness of a given solution: $F_{Max}(11001101) = 5$ and $F_{Lead}(11001101) = 2$. This means F_{Lead} will evaluate 11000000 higher than 10111111 even though the second solution is clearly closer to being perfect.
In this search space the first 2 algorithms we are looking at are very similar: (1+1) EA and Simulated Annealing.

2.2.1 (1 + 1) Evolutionary Algorithm

(1+1) EA is inspired by the way living things evolve in the real world. For every new generation there is a chance that each part of the living being mutates, e.g. size, strength, color... In the same way (1+1) EA can mutate any part of the solution with a certain probability. This means that 0 or multiple mutations per generation are also possible[2].

Algorithm 1: (1+1) Evolutionary Algorithm

1. Choose $s \in [0, 1]^n$ uniformly random.
2. Copy s as s' and for each bit b' of s', $b' = 1 - b$ with probability $1/n$ else $b' = b$.
3. If $F(s') > F(s)$ replace s with s'.
4. If $F(s) < n$ repeat from step 2.

[2]Frank Neumann/Carsten Witt: Bioinspired Computation in Combinatorial Optimization: Algorithms and Their Computational Complexity, Berlin, Heidelberg 2010.

2.2.2 Simulated Annealing / RLS

Simulated annealing is very comparable to (1+1) EA. This algorithm always selects 1 bit and flips it each iteration but additionally has the property to escape local optima when solving optimization problems by having a parameter called temperature, which can be seen as the algorithm's willingness to accept a worse mutation in hopes of avoiding getting stuck in a local optima. Higher temperatures mean higher probabilities of acceptance. The temperature property isn't very useful in the bit string search space, as both the OneMax and Leading Ones problems only have 1 local optima, which is therefore also the global optima. This means that accepting a solution with a lower fitness doesn't provide a benefit to the algorithm. So in this search space, Simulated Annealing will just be replaced with Randomized Local Search, since it behaves the same as SA with a temperature of 0.

Algorithm 2: Randomized Local Search

1. Choose $s \in [0,1]^n$ uniformly random.
2. Choose $i \in [1,2,...,n]$ uniformly random.
3. Copy s as s' where $s'_i = 1 - s_i$
4. If $F(s') > F(s)$ replace s with s'.
5. If $F(s) < n$ repeat from step 2.

2.2.3 Max-Min Ant System

The third and final algorithm discussed in this thesis is the Max-Min Ant System algorithm (MMAS)[3]. As the name suggests, the algorithm is inspired by the movement of ants. In nature, ants leave trailing pheromones wherever they travel. An ant that finds food will then return to its colony, thus leaving more pheromones on the same route. Other ants will follow stronger pheromone trails; further strengthening them, while worse trails will slowly evaporate. This idea can be modified and applied to the bit string search space. To pick construct a solution, the algorithm requires pheromones P that represent the probabilities of the solution containing a 0 or 1 for each bit. Where $P(i,x)$ is the probability that the i'th bit will be x and $x \in [0,1]$. The pheromone values are between $[1/n, 1-1/n]$ to avoid the algorithm getting stuck if the pheromone of a 0 becomes too strong and are initially set to 0.5.

Algorithm 3.1: MMAS Construct

1. $i = 1$
2. $s_i = x$ with probability $P(i,x)/(P(i,0) + P(i,1))$
3. $i = i + 1$, repeat step 2 if $i <= n$

Then we need to update the pheromones between each generation. As previously mentioned, the first step is to evaporate the current pheromones based on evaporation factor $\rho = 1/\log n$. Secondly applying more pheromone based on the previous generation. This creates a new pheromone table P' that is used in the next generation.

[3]Frank Neumann/Dirk Sudholt/Carsten Witt: Analysis of different MMAS ACO algorithms on unimodal functions and plateaus, in: Swarm Intelligence 2008.

Algorithm 3.2: MMAS Update Pheromones

1. $P_{evap}(i,x) = MAX\{P(i,x) \cdot (1-\rho), 1/n\}$
2. $P'(i,x) = \begin{cases} MIN\{P_{evap}(i,x) + \rho, 1 - 1/n\} & \text{if } s_i = x \\ P_{evap}(i,x) & \text{otherwise} \end{cases}$

MMAS then just consists of running these 2 algorithms each generation.

Algorithm 3: Max-Min Ant System

1. Choose $s = Construct()$
2. Choose $s' = Construct()$
3. Update Pheromones
4. If $F(s') > F(s)$ replace s with s'.
5. If $F(s) < n$ repeat from step 2.

2.3 Solving the Traveling Salesman Problem

The algorithms, which are the focus of this thesis, are meta heuristics. This means they are algorithms that use other heuristics to complete accomplish their goal. The most commonly used heuristics for TSP is the 2-opt and 3-opt algorithms. 2-opt is quicker and simpler, but is more limited in it's ability to improve a solution where as 3-opt is slower and more complex, but can give better results[4].

Algorithm 4: 2-Opt Heuristic

1. Choose an initial Hamiltonian cycle s.
2. Select two edges (i,j) and (k,l) such that $i \neq k$ and $j \neq l$.
3. Remove edges (i,j) and (k,l).
4. Reconnect the nodes by reversing the order of nodes between i and k.

Algorithm 5: 3-Opt Heuristic

1. Choose an initial Hamiltonian cycle s.
2. Select three edges (i,j), (k,l), and (m,n) such that they are distinct.
3. Remove these three edges, creating three segments.
4. Generate all possible ways to reconnect the segments (including reversing segments).
5. Choose the best reconnection that minimizes the total distance.

There are multiple ways of applying 3-opt, but picking the best possible recombination is the simplest. This does come with an extra concern though; as meta heuristics which use 3-opt will be able to evaluate more solutions pr iteration, which can skew results.

[4]Dirk Cattrysse Luc Muyldermans Patrick Beullens/Dirk Van Oudheusden: Exploring Variants of 2-Opt and 3-Opt for the General Routing Problem, in: INFOR: Information Systems and Operational Research 37.3 (1999), URL: https://www.jstor.org/stable/25146934.

2.3.1 (1 + 1) Evolutionary Algorithm

A very simple evolutionary algorithm, where one parent creates one offspring each iteration. Here the parent is the previous best solution and the offspring is the new candidate solution. If the offspring is at least as good as the parent, it becomes the parent for the next iteration. This algorithm uses both 2- and 3-opt. 3-opt increases the local search depth and thereby helps the algorithm escape some local optima. This variant is proved to have an upper bound of expected runtime of $O(n^4)$ as seen in this paper[5].

Algorithm 6: (1+1) EA with 2-Opt and 3-Opt

1. Choose an initial Hamiltonian cycle s.
2. Repeat until termination condition is met:
   (a) Generate a new candidate Hamiltonian cycle s' by applying either:
      - 2-Opt with probability 0.5
      - 3-Opt with probability 0.5
   (b) If $F(s') \leq F(s)$, replace s with s'.

Other than the chosen variant of the algorithm, another solution could be to run multiple 2-opts after each other pr iteration without evaluating the result after the last mutation. This could also increase search depth, potentially escaping local optima.

2.3.2 Simulated Annealing

Simulated Annealing is a metaheuristic that uses 2-opt to create new iterations. To avoid local optima it uses its temperature property. This implementation uses an initial temperature of $T(0) = m^3$ where $m = 20n$. (Values: Klaus Meer[6]). The probability of accepting a worse solution is $p_{loss} = e^{\Delta/T}$. The temperature is then reduced each iteration when it hopefully has gotten closer to the global optimum, so it can climb towards it: $T(i) = T(i-1) \cdot \alpha$ where we have defined alpha as $\alpha = 1 - c/T(0)$. $0 < c < T(0)$. c is the cooling rate: a higher value will result in a quicker drop in temperature.

Algorithm 7: Simulated Annealing with 2-Opt

1. Choose an initial Hamiltonian cycle s.
2. Set $m \leftarrow 20 \cdot n$, where n is the number of nodes.
3. Set initial temperature $T \leftarrow m^2$.
4. Set cooling factor $\alpha \leftarrow 1 - c/T$.
5. Repeat until termination condition is met:
   (a) Generate a new candidate s' by applying 2-Opt to s.
   (b) Let $\Delta \leftarrow F(s') - F(s)$.
   (c) If $\Delta \leq 0$, accept s' (i.e., set $s \leftarrow s'$).

[5] Yu Shan Zhang/Zhi Feng Hao: Runtime Analysis of (1+1) Evolutionary Algorithm for a TSP Instance, in: Proceedings of the First International Conference on Swarm, Evolutionary, and Memetic Computing (SEMCCO 2010), vol. 6466 (Lecture Notes in Computer Science), Berlin, Heidelberg 2010, pp. 296–304.

[6] Klaus Meer: Simulated Annealing versus Metropolis for a TSP instance, in: Information Processing Letters 104.6 (2007), URL: https://doi.org/10.1016/j.ipl.2007.06.016.

(d) Else, accept s' with probability $e^{-\Delta/T}$.

(e) Update temperature: $T \leftarrow \alpha \cdot T$.

2.3.3 Max-Min Ant System (MMAS)

Max-Min Ant System is a variant of Ant Colony Optimization designed to improve convergence and avoid stagnation. It constructs solutions based on pheromone trails and heuristic information, and updates pheromones only based on the best solution found so far. This implementation applies boundaries to the pheromone values, so they remain in the interval $[\tau_{\min} = 1/n^2, \tau_{\max} = 1 - 1/n]$. This algorithm is very computationally heavy, so even though it may beat the others in experiments, the runtime can be significantly longer, as it has to update all pheromone trails each iteration. As this variant uses ordinary edge insertion it has an expected runtime of $O(n^6 + (1/\rho)\log n)$ for $0 < \rho \leq 1$ as seen in this paper[7].

Each ant constructs a solution by moving from node i to node j with probability:

$$p_{i,j} = \frac{\tau[i,j]^\alpha \cdot \eta[i,j]^\beta}{\sum_{k \in \text{unvisited}} \tau[i,k]^\alpha \cdot \eta[i,k]^\beta}$$

where:

- $\tau[i,j]$ is the pheromone level on edge (i,j),
- $\eta[i,j] = 1/d(i,j)$ is the heuristic desirability (inverse distance),
- α and β control the influence of pheromone and heuristic information, respectively.

After constructing a solution s, the pheromone matrix is updated in two steps:

1. Evaporation:

$$\tau[i,j] \leftarrow \max\left(\tau[i,j] \cdot (1-\rho),\ \tau_{\min}\right)$$

2. Deposit (only for edges in the best solution s):

$$\tau[i,j] \leftarrow \min\left(\tau[i,j] + \frac{\rho}{F(s)},\ \tau_{\max}\right)$$

where:

- ρ is the evaporation rate,
- $F(s)$ is the total cost (length) of the solution s, as previously mentioned.
- $\tau_{\min}$ and $\tau_{\max}$ are bounds to maintain pheromone diversity.

With these ideas the algorithm can better prevent early convergence, resulting in local optima (suboptimal solutions) and avoid becoming stagnant with extreme pheromone values.

Algorithm 8: MMAS

1. Initialize pheromone matrix $\tau[i,j] = 1/n$ for all i,j.
2. Set $\tau_{\min} = 1/n^2$ and $\tau_{\max} = 1 - 1/n$.

[7] Timo Kötzing et al.: Theoretical analysis of two ACO approaches for the traveling salesman problem, in: Swarm Intelligence 6.1 (2012), URL: https://link.springer.com/article/10.1007/s11721-011-0059-7.

3. Precompute heuristic values $\eta[i,j] = 1/d(i,j)$.
4. Repeat until termination condition is met:
   (a) Construct a solution s':
      i. Start from a random node.
      ii. At each step, choose the next node j from the unvisited set with probability proportional to $\tau[i,j]^\alpha \cdot \eta[i,j]^\beta$.
   (b) Evaluate $F(s')$.
   (c) If $F(s') < F(s)$, update best solution $s = s'$.
   (d) Update pheromones:
      i. Evaporate:
$$\tau[i,j] = \max\left(\tau[i,j] \cdot (1-\rho),\ \tau_{\min}\right)$$
      ii. Deposit: for each edge (i,j) in s,
$$\tau[i,j] = \min\left(\tau[i,j] + \rho/F(s),\ \tau_{\max}\right)$$

3 Design

The goal of this project is to create an application that can evaluate and visualize different algorithms when solving optimization problems like OneMax, Leading Ones and TSP. This section will explain the process of going from a goal to an executable idea, diving in to the tools used in that process and the overall architecture of the project.

3.1 Architecture

It was decided early on that the project would be developed as a web application, since it is an area there has not been a lot of focus on during the 3 year bachelor, even though it is a mayor part of the software engineering industry. The general idea behind this choice was that it would make it easier to put the application on a server and host the application there in the future. This would also make it easier for others to use the software, since the calculations would be handled by the server, so anyone could use it.
The specific technologies chosen for the application were React med Node.js for the front end and ASP.NET Core Web API for the backend. React and Node.js were chosen as they are industry standard technologies. Node.js provides a lot of modules that are easily integrated into the application, which enables a very developer-friendly workflow. .NET is a well developed framework with the specific template chosen being suited for creating an API backend for web applications. It comes with some very powerful prebuilt feature:

- API endpoint Controllers
- Swagger testing
- Automatic routing
- CORS Policies

The controllers handle the endpoints, so all the developer has to do is define a C# function in the controller and then the framework handles setting up routing so an external application can access the function through the endpoint. Swagger is also very powerful; it provides an access point to the endpoints with a simple UI, so testing of the backend becomes very streamlined. It also makes it much easier when working on the communication between front- and back end.

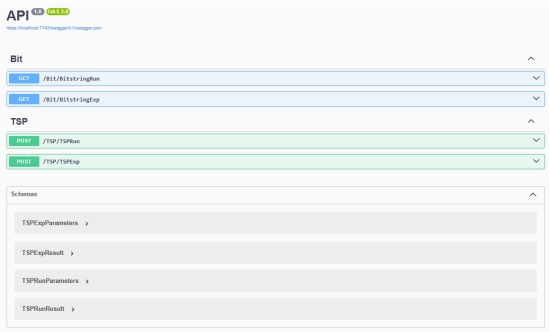


Figure 2: Swagger screenshot

3.2 Requirements

Before commencing work on either part of the application, it was necessary to plan what features the application required. Firstly it needed to be able to analyze problems in 2 different search

spaces, namely bit strings and TSP/permutations, so this is an obvious way of splitting the application up further. For each search space the user should be able to run several types of experiments:

1. Step by step:
    - Allows the user to slowly step through each iteration to see how the algorithms improve the solutions
    - Providing visuals of how the solutions change through iterations
    - Plotting the fitness through iterations
2. Singular experiments with added details:
    - Allows the user quickly compare the finished solutions of the algorithms
    - Providing visuals of how the solutions ended up
    - Plotting the fitness through iterations
3. Larger multi-experiments:
    - Allows the user to compare the performances of the algorithms across different problem sizes
    - Plotting the average fitness of the solutions at different problem sizes

Additionally, the point of the application is to be able to compare the algorithms, so the experiments should be able to run with multiple algorithms at once and display all the results together via the graphs for simple and quicker analysis. Another core requirement for the project was that it would be scalable and modular.

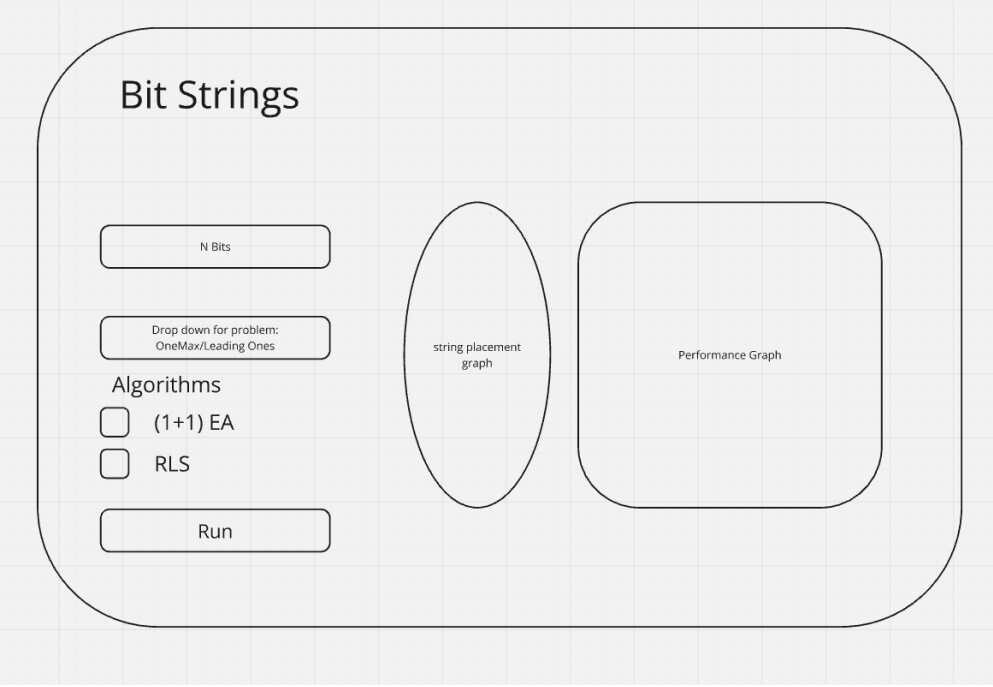


Figure 3: Initial UI Sketch of bit string page

3.3 Front End

Once the requirements were clear it was time to sketch the UI [Figure: 3]. This was now simpler as it was known what features needed to be displayed. The user can select the experiment type (This was not added to the sketch), problem size and what algorithms that should be used. When selecting the 3'rd experiment type 2 extra fields appear:

- Exp Steps, which controls how many problem sizes the experiment should include, so if $problemsize = 20$ and $ExpSteps = 4$, then 4 experiments will be run with problem size 5, 10, 15 and 20.
- Exp Count, which controls how many times the experiment at each step is run to get an average value at each problem size, that is more accurate as the algorithms are non-deterministic.

The front end would be split into 2 pages: bit strings and TSP, where appropriate parameters and graphs would be displayed.

3.3.1 Bit Strings

The sketch shows the bit string page; how the parameters were placed on the left side and the graphs would be filling up the rest of the window. On the bit string page there is an additional drop-down menu, so the user can select what problem to experiment with: OneMax or Leading Ones. The elliptic graph plots a list of bit strings, which represent the best solution of each iteration. The Y-axis shows how the percentage of 1's in a bit string and the X-axis shows the average position of the 1's in the string. This will be explained further in the implementation section, but the graph is insightful, as the user can get a better understanding of how the solutions look each iteration, especially in the Leading Ones problem where the position of 1's is crucial. The second graph has 2 uses depending on the experiment type: The first 2 experiment types plot the fitness over the iterations while the 3rd type displays the average amount of iterations to get the perfect solution over the problem size.

3.3.2 Traveling Salesman Problem

To keep the application simple, the design of the TSP-page is very similar to the bit string page. It has the same way of selecting parameters, but instead of the elliptical graph it has a coordinate system. This graph displays the TSP solutions, so the user can see all the nodes and what solution each algorithm has generated. To allow the user the ability to upload specific TSP-problems an extra button was added by the problem size field that enables this functionality. For more thorough analysis the application also displays a table with all the data from the experiment to the user, that can be downloaded as a CSV-file. This can then be imported into other software like excel and further analyzed.

3.4 Backend

The design process for the backend started after getting a rough idea of what the front end needed. The backend had some requirements, it needed to be able to:

- Solve OneMax and Leading Ones problems for the bit string search space
- Solve TSP
- Use different algorithms on the same problems
- Return all the data required for each type of experiment
- Send this data from API end points so the front end could fetch it

The code base was separated into the bit string and TSP part of the application.

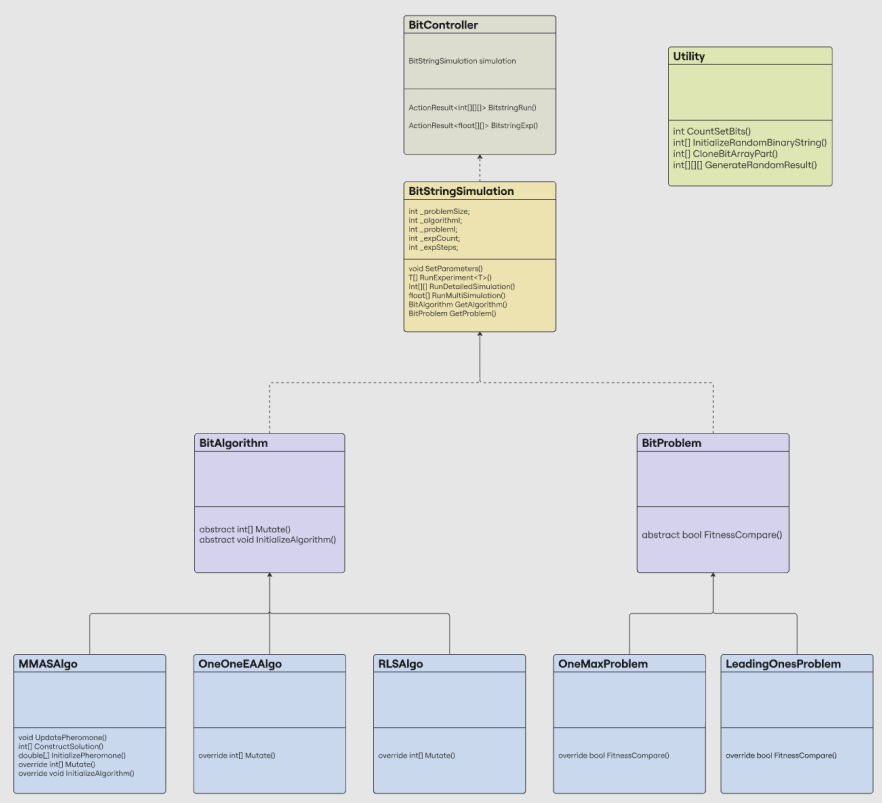


Figure 4: UML-Diagram

A UML diagram was then used to visualize how the backend could be structured. This was a good tool in early development as it helped design an efficient structure. The BitController which inherited from the .NET controller class had a reference to BitStringSimulation class. When the controller would receive a request through it's endpoints, it would call upon the Simulation class to run an experiment with the specified algorithms and problem. This was important to attain the goal of a scalable and modular application. To add more algorithms or problems, all that is necessary is to add class that inherits from BitAlgorithm and then override the abstract functions. A more detailed explanation of how the design works and how it changed over the course of the project, is found in the following section.

4 Implementation

This section will describe the implementation process including changes to the design that happened throughout development. It will mention challenges of implementation and how the project plan was used and altered.

Week	Objective
9	Planning project architecture and sketching diagrams (UML and application flow). Information gathering of the technical problem and potential solutions.
10	Setting up development framework and writing project plan.
11	Information gathering on the bit-string search space and different nature-inspired metaheuristics. Developing the backend capabilities for bit-string search space.
12	Developing the frontend to visualize and enable analysis of experiments in bit-string search space.
13	Initial development of backend functionality to handle the traveling salesman problem.
14	Initial development of frontend to handle the execution of experiments in the permutation search space.
15	Refactoring, providing documentation and writing unit tests.
16	Continued backend development: implementing multiple different algorithms and different running modes such as step-by-step, algorithm-compare-mode and maybe more.
17	Implementation. Specific area is determined closer to date, dependent on what challenges have presented themselves.
18	Implementation. Specific area is determined closer to date, dependent on what challenges have presented themselves.
19	Experimentation and logging. Analyzing data and comparing to theory.
20	Finishing touches for implementation
21	Buffer time for implementation
22	Full focus on writing the report
23	Full focus on writing the report
24	Final touches before hand-in
25	Prepare for oral form
26	Prepare for oral form

Figure 5: Schedule from project plan.

Light red: Implementation. Light blue: Report related work. Light green: Miscellaneous.

4.1 Bit string search space

Following the project plan the first part of the application to be implemented was the backend capability to handle bit string experiments. This is where the UML-diagram [4] was very helpful as there was a clear sense of what components needed to be created and how they should relate to each other.

4.1.1 API

The BitController inherits from the .NET Controller class that automatically generates routing request handling for C# functions defined within it. This controller handles 2 end points:
BitStringRun
Handles requests for the first 2 experiment types, generating a detailed result of each selected algorithms solution pr iteration. This would return a list containing the best solution at each iteration for each algorithm selected. It would therefore be sent to the front end as a 3D-array. For step by step experiments, the trick was to run the whole experiment and let the front end handle stepping through it. The end point takes 3 parameters: problem size, algorithm index and problem index:

- Problem size is the length of the bit strings.
- Algorithm index determines what algorithms have been selected. It is handled as a binary

value where each bit represents if a specific algorithm is selected, e.g. 011 would mean that algorithm 1 and 2 are selected (read from right to left), which would be (1+1)EA and RLS.

- Problem index is simpler as there can only be one problem selected at a time. 0 = Max Ones and 1 = Leading Ones.

BitStringExp
The second endpoint is for handling experiments that compare results across different problem sizes, which is for the 3rd experiment type called "performance comparison" in the application. It provides the average amount of iterations used for each algorithm at different problem sizes. It returns an array for each algorithms result for each experiment step, resulting in a 2D-array. It contains the same 3 parameters as the other endpoint but has an additional 2:

- Exp Count as explained in the design section provides precision and accuracy by running multiple experiments at each problem size. This doesn't affect the format of the returned result.
- Exp Step determines how many different problem sizes will be tested between 0 and the selected problem size. This controls the length of the 2D array.

4.1.2 Experiment Handling

When a request has been made to the controller, the BitStringSimulation class has a RunExperiment method that is called by the controller 1. The method takes another function called "simulation" with a generic return type, RunExperiment then returns a list of that return type. This is because "simulation" is a function that takes an algorithm and a problem and runs a certain type of experiment one them. For the first 2 experiment types the "simulation" function passed in is RunDetailedSimulation 2. This simulation function returns a 2D-array, which is an array of bit strings (that are represented as integer-arrays). As this function is run for each algorithm this creates a 3D-array that RunExperiment then returns to the BitController. Using these generic types further improves the back end's scalability since a new experiment with a different return type easily could be integrated as the RunExperiment function can handle any types.

4.1.3 Algorithms

The (1+1)EA and RLS are very simply and similarly implemented. They both inherit from the BitAlgorithm class, which gives them the Mutate method to override. The Mutate function is called once pr iteration; it takes in an integer-array and returns a new one (Example RLS mutate 3). (1+1) EA flips every bit with a $1/n$ probability while RLS guaranteed flips 1 random bit.
Now the MMAS on the other hand is more complicated. It also needs to use the other inherited function from the BitAlgorithm class: InitializeAlgorithm. This function is called before the first iteration. That is important as the MMAS algorithm relies on a pheromone matrix that needs to be reset between experiments. Otherwise the algorithm would be able to remember it's pheromone values in between experiments.

4.1.4 Bit string Web Page

The implementation of the bit strings page followed the original design very closely. The main difference being the table at the bottom. This feature made it much simpler to test that the front end was working as intended and that the graphs were displaying the data correctly, as well as enabling the user to download the results for more data analysis.

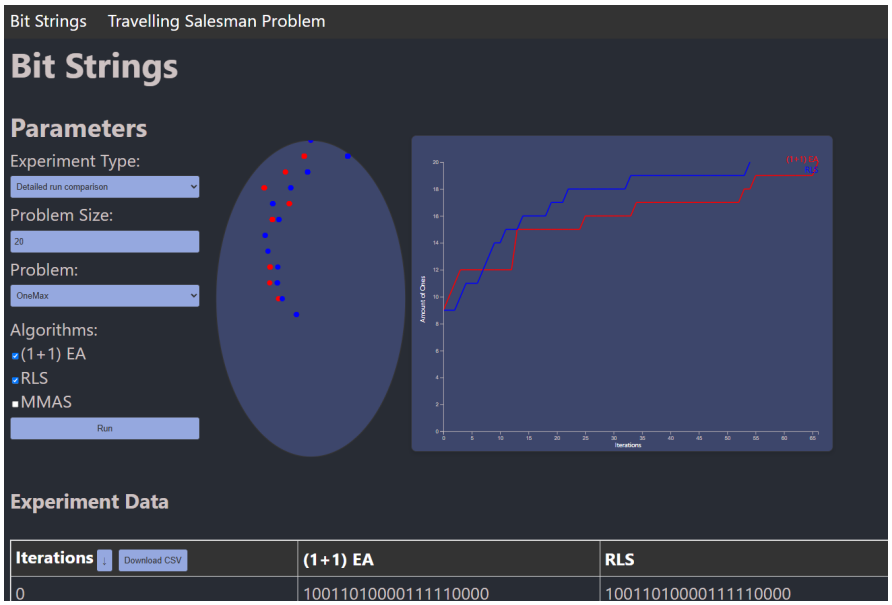


Figure 6: Screenshot of the page for bit experiments

At the bottom is a table containing the results of the experiment

The graphs were created as React components that made the front end more modular, potentially enabling different configurations of the page. The right graph would scale the content within to always take up all the space of the graph, making for a simpler plug and play UI. All that it requires are the points and label names, then it would display multiple results in the same plot. The left graph would calculate you Y-coordinates by finding the percentage of 1's in bit strings and scale that by the height of the diagram. The X-coordinate would be calculated as the average position of the 1's in the bit string and then scaled by the width of the ellipse.

4.2 Traveling Salesman Problem

When planning on how to add TSP functionality to the back end, it was clear that main structure used for the bit string simulations could be reused for the TSP simulations as well. Therefore it made sense to update the class structure of the back end by creating new classes:

- Algorithm. A new class from which BitAlgorithm and TSPAlgorithm could inherit.
- Simulation. A new class from which BitStringSimulation and TSPSimulation could inherit.

This meant that elements from BitStringSimulation could be moved to the Simulation class if they were relevant for TSPSimulation. By strengthening the structure of the back end it would improve scalability and modularity. New features could be implemented directly into the simulation class and be accessible for both search spaces. More interestingly it also meant that other search spaces could be explored by inheriting from the Simulation class.

4.2.1 API

On the TSPController the endpoints were also updated to combine all parameters into JSON objects. This made it easier to add more when some algorithms required more customization such as α and β values on MMAS. Like with bit strings, TSP also used 2 endpoints:

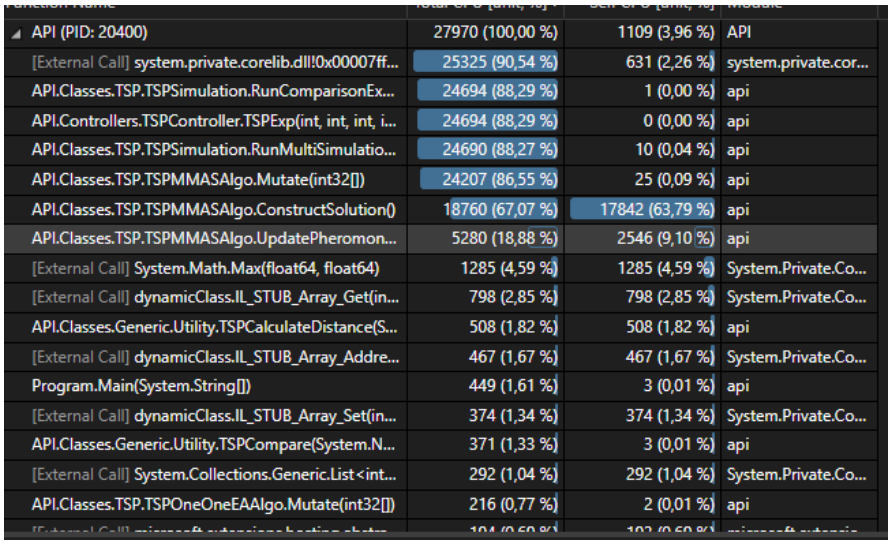
1. **TSPRun**
   This endpoint is used for both Step by Step and Detailed Comparison experiments. It returns 3 JSON objects:
   - Nodes: The coordinates for the TSP-graphs points. 2D-Array with size n by 2. The 2 coming from requiring an X and Y coordinate.
   - Solutions: An array of each algorithms best solution for each iteration, thereby being a 3D-array with size *algorithms* by *iterations* by n, since each solution is an integer-array storing the index of each node visited in order.
   - Results: The distance cost of each solution. 2D float array with size *algorithms* by *iterations*.
2. **TSPExp**
   This endpoint only returns the Results object as it runs multiple experiments with different TSP-graphs. Here the results are the average distance cost per problem size step. This makes it less relevant to plot in a coordinate system.

4.2.2 Algorithms

Implementing (1+1) EA with 2-opt and 3-opt was quite simple. As it doesn't need to remember anything between iterations it only overrides the inherited Mutate method where it calls either 2-opt 3-opt which are helper functions defined in the Utility class. This class contains various static helper functions such as: CalculateTSPDistance, RandomTSPSolution, GenerateGraph... They are defined here to reduce clutter in other classes by providing a single place to keep small useful functions.
The Simulated Annealing algorithm also uses 2-opt but uses a temperature variable that determines the probability for it to accept a worse solution 4. Here the temperature value is multiplied by $0 < \alpha < 1$ every iteration which lowers the temperature.
MMAS finds it's solutions through the pheromone matrix, but in the TSP search space, it also uses the heuristic information of the distance between the nodes. It then calculates the influence of the pheromone values and the heuristic information with α and β every iteration to construct a new solution. This is very computationally intense and makes this algorithm much slower than the rest in terms of CPU time.

4.2.3 Optimization

When experimenting with the application, MMAS was taking much longer time than the other algorithms. To investigate this issue the Visual Studio Profiler was a very powerful tool 7.
The profiler showed that MMAS's ConstructSolution method was using 64% of the total CPU time of the experiment, while testing against all algorithms. When taking a closer look at what caused the method to take so long, it was apparent that calculating the influence of pheromone values and distances was the problem. This was because the probability of picking a certain unvisited node was proportional to $\tau[i,j]^{\alpha} \cdot \eta[i,j]^{\beta}$. Using Math.Pow to calculate this was very slow.
Since the heuristic values were constant each iteration, they could be calculated once for the first iteration and then reused. This small idea reduced total CPU running time by 20%.



API (PID: 20400)	27970 (100,00 %)	1109 (3,96 %)	API
[External Call] system.private.corelib.dll!0x00007ff...	25325 (90,54 %)	631 (2,26 %)	system.private.cor...
API.Classes.TSP.TSPSimulation.RunComparisonEx...	24694 (88,29 %)	1 (0,00 %)	api
API.Controllers.TSPController.TSPExp(int, int, int, i...	24694 (88,29 %)	0 (0,00 %)	api
API.Classes.TSP.TSPSimulation.RunMultiSimulatio...	24690 (88,27 %)	10 (0,04 %)	api
API.Classes.TSP.TSPMMASAlgo.Mutate(int32[])	24207 (86,55 %)	25 (0,09 %)	api
API.Classes.TSP.TSPMMASAlgo.ConstructSolution()	18760 (67,07 %)	17842 (63,79 %)	api
API.Classes.TSP.TSPMMASAlgo.UpdatePheromon...	5280 (18,88 %)	2546 (9,10 %)	api
[External Call] System.Math.Max(float64, float64)	1285 (4,59 %)	1285 (4,59 %)	System.Private.Co...
[External Call] dynamicClass.IL_STUB_Array_Get(in...	798 (2,85 %)	798 (2,85 %)	System.Private.Co...
API.Classes.Generic.Utility.TSPCalculateDistance(S...	508 (1,82 %)	508 (1,82 %)	api
[External Call] dynamicClass.IL_STUB_Array_Addre...	467 (1,67 %)	467 (1,67 %)	System.Private.Co...
Program.Main(System.String[])	449 (1,61 %)	3 (0,01 %)	api
[External Call] dynamicClass.IL_STUB_Array_Set(in...	374 (1,34 %)	374 (1,34 %)	System.Private.Co...
API.Classes.Generic.Utility.TSPCompare(System.N...	371 (1,33 %)	3 (0,01 %)	api
[External Call] System.Collections.Generic.List<int...	292 (1,04 %)	292 (1,04 %)	System.Private.Co...
API.Classes.TSP.TSPOneOneEAAlgo.Mutate(int32[])	216 (0,77 %)	2 (0,01 %)	api

Figure 7: Screenshot of profiling tool

5 Experiments

This section will describe each experiment conducted using the developed application. It will explain the goal of each experiment, how it is conducted, including configurations for the experiments and why it is conducted in this way. Finally showcasing and briefly describing them and then analyzing them. Here they will be evaluated and compared.

5.1 EX1) OneMax Comparison

The first experiment conducted was to compare the algorithms on the One Max problem. There are multiple reasons why this experiment is interesting:

- Looking at how each algorithm compares to each other.
- Comparing this to the expected result.
- Comparing results to that of experiment 2, do the algorithms behave differently for different problems.

5.1.1 Configuration

The original experiment was configured to run with:

- $n_{max} = 100$
- $Exp_{Count} = 100$
- $Exp_{steps} = 10$

It was desirable to have a high Exp_{Count} but that meant the program would take a lot longer to run. The program would become very slow after $n = 100$. To circumvent this issue, another experiment was conducted to supplement the first one. The second experiment would have a lower accuracy but higher problem sizes. The supplementary experiment was configured to run with:

- $n_{max} = 1000$
- $Exp_{Count} = 5$
- $Exp_{steps} = 10$

5.1.2 Results

Results of the two experiments: Table: 1 and Table: 2 can be found in the Data section along with the plots generated by the program: Figure: 8 and Figure: 9. A quick glance tells us that (1+1)EA and MMAS seem to behave very similarly. RLS is clearly more performant in both experiments.

5.1.3 Evaluation

The experimental comparison of (1+1) EA, RLS, and MMAS on the OneMax problem aligns somewhat with the theoretical insights presented in Neumann and Witt's work[8].

The OneMax problem, which involves maximizing the number of ones in a bit string, can be used as a benchmark in evolutionary computation due to its simplicity and well-understood behavior. Theoretical analysis in the literature shows that:

[8]Neumann/Witt: Bioinspired Computation in Combinatorial Optimization: Algorithms and Their Computational Complexity (see n. 2).

- **(1+1) EA** is expected to be the slowest algorithm to reach an optimal solution.
- **RLS (Randomized Local Search)** is expected to outperform (1+1) EA.
- **MMAS (Max-Min Ant System)**, when configured with appropriate pheromone bounds and evaporation rate, can achieve similar performance, though its runtime is more sensitive to parameter tuning. Proven to match (1+1)EA with $\rho = 1$.

The experimental results confirm these theoretical expectations:

- The **supplementary experiment with larger problem sizes** (up to $n = 1000$) and fewer repetitions preserves the relative performance trends, though with increased variance due to the lower experiment count.
- **RLS outperforms (1+1) EA** in both sub-experiments, consistent with theory, as RLS avoids unnecessary mutations and converges more efficiently.
- **(1+1) EA and MMAS behave similarly**, which indicates improper parameter tuning as MMAS should be able to outperform (1+1)EA.

Comparing these results to those from Experiment 2 will help assess the generality of each algorithm's strengths. It will show if the behavior of the algorithms is similar across different problems.

5.2 EX2) Leading Ones Comparison

The Leading Ones experiment shares the goals of the OneMax experiment. Here the main interest being how the algorithms performed under a different fitness function, as this one provides less information about each solution to the algorithm.

5.2.1 Configuration

For this experiment it made sense once again to split it up into 2, with different configurations:

- $n_{max} = 100$
- $Exp_{Count} = 100$
- $Exp_{steps} = 10$

It was desirable to have a high Exp_{Count} but that meant the program would take a lot longer to run. The program would become very slow after $n = 100$. To circumvent this issue, another experiment was conducted to supplement the first one. The second experiment would have a lower accuracy but higher problem sizes. The supplementary experiment was configured to run with:

- $n_{max} = 1000$
- $Exp_{Count} = 5$
- $Exp_{steps} = 10$

5.2.2 Results

Results of the two experiments: Table: 3 and Table: 4 can be found in the Data section along with the plots generated by the program: Figure: 10 and Figure: 11. When looking at the first sub-experiment the results are very similar to those of EXP1, meaning when comparing the algorithms to each other MMAS and (1+1)EA are close together while RLS is significantly better. That being said the amount of iterations required has increased by a factor of 8 for

$n = 100$. The amount of iterations needed for each problem size is visibly growing as seen by a ramping graph.
Now looking at the second sub-experiment one will notice that the iteration count reaches a maximum of 150000 after $n = 400$. This is because of a limit set in the back end to keep it from running some configurations forever. The data points after $n = 400$ will therefore be discarded for the analysis.

5.2.3 Evaluation

The Leading Ones problem differs from OneMax in that it provides less information to the algorithms, making it a more challenging benchmark for evolutionary strategies. Theoretical expectations suggest that:

- **RLS** should still outperform (1+1) EA due to its more focused mutation strategy, especially in problems where flipping a single bit is more likely to yield improvement. Nevertheless, it should still be slower than with OneMax.
- **(1+1) EA** is expected to perform worse due to its higher probability of flipping multiple bits, which can hurt the solution in Leading Ones.
- **MMAS** may struggle more than in OneMax due to the lack of information.

The experimental results support these expectations:

- In the first sub-experiment, **RLS significantly outperforms both (1+1) EA and MMAS**, with the performance gap increasing with problem size.
- **(1+1) EA and MMAS perform similarly**, indicating the same issue as the OneMax experiment. This suggests that MMAS is improperly tuned as it behaves the same across different problems.
- In the second sub-experiment, all algorithms hit the iteration cap of 150000 after $n = 400$, indicating that **Leading Ones becomes a much harder optimization problem for these algorithms at large scales**.

Compared to Experiment 1, the performance degradation is much more severe, especially for (1+1) EA and MMAS. This highlights how algorithm efficiency is highly problem-dependent and that RLS is more robust across problem types. Overall the algorithms show similar tendencies in different problems when compared to each other.

5.3 EX3) Simulated Annealing - Cooling Rate

This experiment investigates the effect of the cooling rate c (in data written as CR) parameter on the performance of the Simulated Annealing algorithm when solving the Traveling Salesman Problem. The goal is to understand how different cooling schedules influence the convergence behavior and solution quality of the algorithm.

5.3.1 Configuration

The experiment was configured to run with a maximum problem size of 500, $Exp_{Count} = 100$, $Exp_{steps} = 10$, and iterations set to 10000. The cooling rate was varied across the values $c \in [100000,\ 500000,\ 1000000,\ 2000000,\ 10000000]$ to observe its impact.

Additionally, a baseline was included using a random solution for comparison. The experiment was conducted across multiple problem sizes to observe how the algorithm behaved at extreme cooling rates with different problem sizes.

5.3.2 Results

The results are summarized in Table 5. This data was plotted on a graph 12 with the addition of the random baseline.

From the table, it is evident that lower cooling rates (e.g., CR 100000) result in worse performance for larger problem sizes. However, using a higher cooling rate also raises a problem. Looking at the left hand side of the graph cooling rates above 2M don't improve upon the purely randomized solution for low problem sizes.

5.3.3 Evaluation

This experiment demonstrates the sensitivity of Simulated Annealing (SA) to its cooling rate parameter when applied to the Traveling Salesman Problem (TSP). Theoretical expectations suggest that:

- **Higher cooling rates** (i.e., faster cooling based on the formula presented in section 2.3.2) may lead to premature convergence and poor solutions as the algorithm will find a local optimum.
- **Lower cooling rates** (i.e., slower cooling) allow more exploration but may not be efficient for small problem sizes. This can result in the algorithm never converging to find a good solution.

The experimental results confirm these expectations:

- For **larger problem sizes**, higher cooling rates (e.g., CR 2M and above) yield significantly better solutions, outperforming lower cooling rates by a wide margin.
- For **smaller problem sizes**, high cooling rates perform similarly to or worse than random search, indicating that the algorithm does not converge quickly enough.
- There is a clear **trade-off between exploration and exploitation**, and no single cooling rate performs best across all problem sizes.

These results emphasize the importance of parameter tuning in SA. Unlike RLS or (1+1) EA, SA's performance is highly dependent on the cooling schedule, and improper settings can lead to suboptimal results even on relatively simple instances.

Interestingly SA is quite robust being able to find similar quality solutions across a large cooling rate interval. Noting that changing the cooling rate by a factor of 100 still produced results with no significant difference for some problem sizes.
It is worth noting that these experiments were run with a fixed iteration count; as the optimal cooling rate also depends on how many iterations the algorithm has to reduce the temperature. In another experiment it could be interesting to see how the optimal cooling rate depends on both problem size and iteration count.

5.4 EX4) Max-Min Ant System - α & β

This experiment explores how varying the β parameter in the Max-Min Ant System (MMAS) affects performance on the Traveling Salesman Problem (TSP). The β parameter controls the influence of heuristic information (the inverse of distance) during solution construction. The goal is to determine how sensitive MMAS is to changes in β and potentially finding optimal values for α and β.

5.4.1 Configuration

The experiment was configured to run with a fixed $\alpha = 1$ and varying values of $\beta \in [1, 5, 10, 15, 25, 30, 40]$. The problem size was varied from $n = 0$ to $n = 100$ in steps of 10. A small problem size was used in this experiment as MMAS tends to be much slower in CPU time. Each configuration was run with a fixed number of iterations and averaged over multiple runs to ensure consistency. The (1+1) EA was included as a baseline for comparison.

5.4.2 Results

The results are presented in Table 6. The table shows the total distance of the best solution found for each problem size and α/β-configuration. This data was plotted against (1+1)EA for reference Figure: 13

From the results, we observe that:

- MMAS significantly outperforms (1+1) EA across all problem sizes and β values.
- Lower β values (e.g., $\beta = 1$) result in worse performance compared to higher values.
- Performance improves rapidly as β increases from 1 to 10, after which the gains become more marginal.

5.4.3 Evaluation

This experiment highlights the sensitivity of MMAS to the relation between the α and β parameter, which control the influence of pheromones and heuristic information during solution construction. Theoretical analysis of MMAS for the TSP[9] shows that:

- **Low β values** (e.g., $\beta = 1$) reduce the influence of heuristic information, making the algorithm rely more on pheromone trails. This can lead to more exploratory behavior but may slow convergence.
- **High β values** increase the influence of heuristic information, biasing the search toward shorter edges. This can accelerate convergence to good solutions but may also increase the risk of premature convergence to suboptimal tours.

The experimental results align well with these theoretical expectations:

- **MMAS significantly outperforms (1+1) EA** across all tested problem sizes and β values, confirming the advantage of combining pheromone memory with heuristic guidance.
- **Performance improves rapidly as β increases from 1 to 10**, indicating that heuristic information is essential for effective search in TSP.
- **Beyond $\beta = 10$**, the performance gains plateau, suggesting diminishing returns. This is consistent with theoretical findings that overly strong heuristic bias can reduce diversity and trap the algorithm in local optima.

These findings reinforce the theoretical result that MMAS with a well-balanced use of heuristic information (moderate β) improves the algorithms ability to good solutions quickly. However, excessive reliance on heuristic information may be detrimental, especially in instances where the optimal tour is not easily inferred from local edge weights.
Interesting to note, that the algorithm uses the large influence of the heuristic information to improve the solution by large amounts in the first couple of iterations. This means that the algorithm out-competes the other algorithms in low iteration count experiments, as the other algorithms don't have time to quickly converge on a solution.

[9] Kötzing et al.: Theoretical analysis of two ACO approaches for the traveling salesman problem (see n. 7).

5.5 EX5) TSP Comparison Fixed Iterations

This experiment compares the performance of three algorithms—(1+1) EA, Simulated Annealing, and MMAS—on the Traveling Salesman Problem (TSP) with a fixed amount of iterations. The goal is to evaluate how effectively each algorithm minimizes distance when given the same computational effort.

5.5.1 Configuration

Each algorithm was run for a fixed number of 10000 iterations. The problem size was varied from $n = 0$ to $n = 500$ with $Exp_{steps} = 10$ but only $Exp_{Count} = 1$. Ideally there should be more executions per problem size step, but that was not feasible with how long the experiments already took. SA used a cooling rate of 1000000 and MMAS used $\alpha = 1$ and $\beta = 10$ based of of results from the previous experiments.

5.5.2 Results

The results, summarized in Table 7 and plotted in Figure:14, show a clear difference in performance. MMAS consistently outperforms both (1+1) EA and Simulated Annealing across all problem sizes. Simulated Annealing performs better than (1+1) EA, especially at smaller problem sizes, but the gap between it and MMAS remains substantial. The (1+1) EA algorithm shows the steepest increase in distance as problem size grows.

5.5.3 Evaluation

This experiment provides a direct comparison of three algorithms—(1+1) EA, Simulated Annealing (SA), and MMAS on the TSP under equal amount of iterations permitted. Theoretical expectations suggest:

- **MMAS** should perform best due to its use of both pheromone trails and heuristic information.
- **Simulated Annealing** can perform well if the cooling schedule is appropriate.
- **(1+1) EA** is expected to perform worst due to its lack of memory and guidance mechanisms.

The experimental results confirm these expectations:

- **MMAS consistently achieves the lowest total distances across all problem sizes**, demonstrating its superior ability to exploit problem structure. Relying more on heuristic information proves useful for these problem sizes when only having 10000 iterations to find a solution.
- **Simulated Annealing outperforms (1+1) EA**, showing that even a basic cooling schedule provides better exploration than pure mutation.
- **(1+1) EA shows the steepest increase in distance** as problem size grows, indicating poor scalability for TSP.

These results reinforce the importance of algorithm design in combinatorial optimization. MMAS, with its hybrid use of memory and heuristic guidance, proves to be the most effective under fixed iteration constraints. MMAS is also more computationally intense so it can be argued that this isn't a 100% fair comparison. In a real world scenario SA could also be the better choice as it iterates far quicker. Another experiment could be conducted where the algorithms CPU time was compared to gain a more practical comparison.

6 Challenges

The challenges of the project mainly lied in the back end implementation of the application. The algorithms were inherently based on randomness, meaning testing was more difficult, since it was not possible to predict an algorithm's result when testing. To solve this issue, when conducting manual functional tests the VS Debugger was used. This tool allowed easy examination of the application state during execution. Specifically the MMAS algorithm was very troublesome. While implementing it, it would perform very poorly and issues were hard to diagnose. It was difficult to distinguish between implementation mistakes and improper tuning.
MMAS was also significantly more complex both in implementation but also execution, as each iteration of MMAS was much more computationally intense when compared to (1+1) EA and SA. When experimenting with the algorithms it would take much longer in terms of CPU time for the MMAS to execute. This meant that MMAS was bottle-necking the capacity for experiments. So when comparing the algorithms' performances based on a fixed amount of iterations, MMAS had a glaring advantage, as it was doing many more calculations per iteration. It could have been interesting to see performances based on CPU time limit. Such an experiment would reveal the more efficient algorithms. Even with this experiment setup MMAS might still have an advantage if the CPU time limit is quite low. That is because MMAS has access to the heuristic information (distance between nodes). With a higher β compared to α, the algorithm would improve it's solution dramatically in the first couple iterations. Therefore it is quite difficult to compare the algorithms in a fair way.

7 Conclusion

It has been successfully demonstrated how nature-inspired metaheuristics can be applied to different optimization problems. This was done by creating an interactive web-application to visualize and evaluate these algorithms. When comparing the different algorithms across problem types it revealed that the algorithms behaved similarly relative to each other. SA clearly outperformed (1+1) EA in all experiments, though MMAS was able to outperform SA in TSP when given heuristic information that the other algorithms didn't have. The application itself provides tools for visualizing and understanding the iterative behavior of these algorithms. Several challenges were encountered during development, particularly around performance optimization and parameter tuning, which were addressed through refactoring and profiling. Future work could involve expanding the set of algorithms, adding support for other problem types, and enabling runtime comparisons based on actual CPU time. Overall, the project meets its objectives in evaluating, comparing, and visualizing the behavior of nature-inspired optimization metaheuristics.

Data

Problem Size	(1+1) EA	RLS	MMAS
0	0	0	0
10	41.66	21.5	44.66
20	130.91	57.04	133.04
30	225.53	100.77	229.7
40	350.94	146.74	330.95
50	472.41	195.1	454.27
60	560.43	229.27	558.92
70	684.61	278.29	716.02
80	801.96	347.74	845.83
90	910.91	377.19	919.88
100	1083.64	440.68	1115.94

Table 1: Performance comparison of algorithms across problem sizes ONEMAX

At each problem size the algorithms are run 100 times on different initial bit strings to gain a better average.

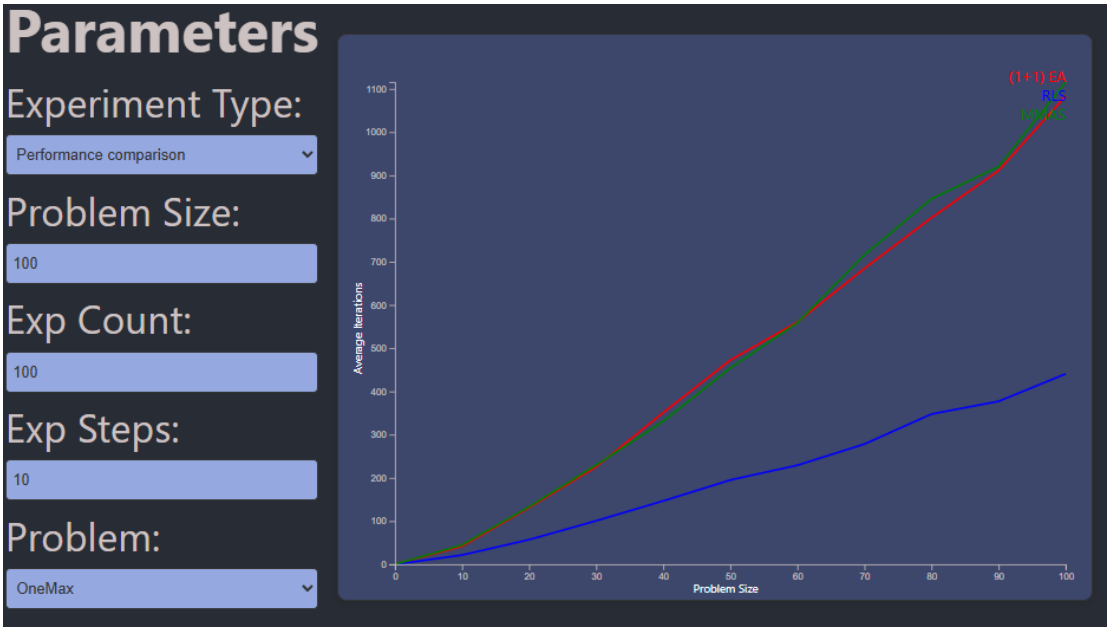


Figure 8: Graph based of 1 and the chosen parameters

Problem Size	(1+1) EA	RLS	MMAS
0	0	0	0
100	1293.6	327	933.6
200	2432.8	1032.8	2575.8
300	4084.8	1694.4	3726.8
400	5305.4	2080.6	5486
500	8951.2	3226.4	6735.8
600	8900.4	4340.4	10599.4
700	10453.6	4739.4	11193
800	11285.6	5001	14531.8
900	19192.2	5807.6	17353.2
1000	15353	6480.4	16426.4

Table 2: Performance comparison of algorithms across larger problem sizes ONEMAX

Lower accuracy but higher values

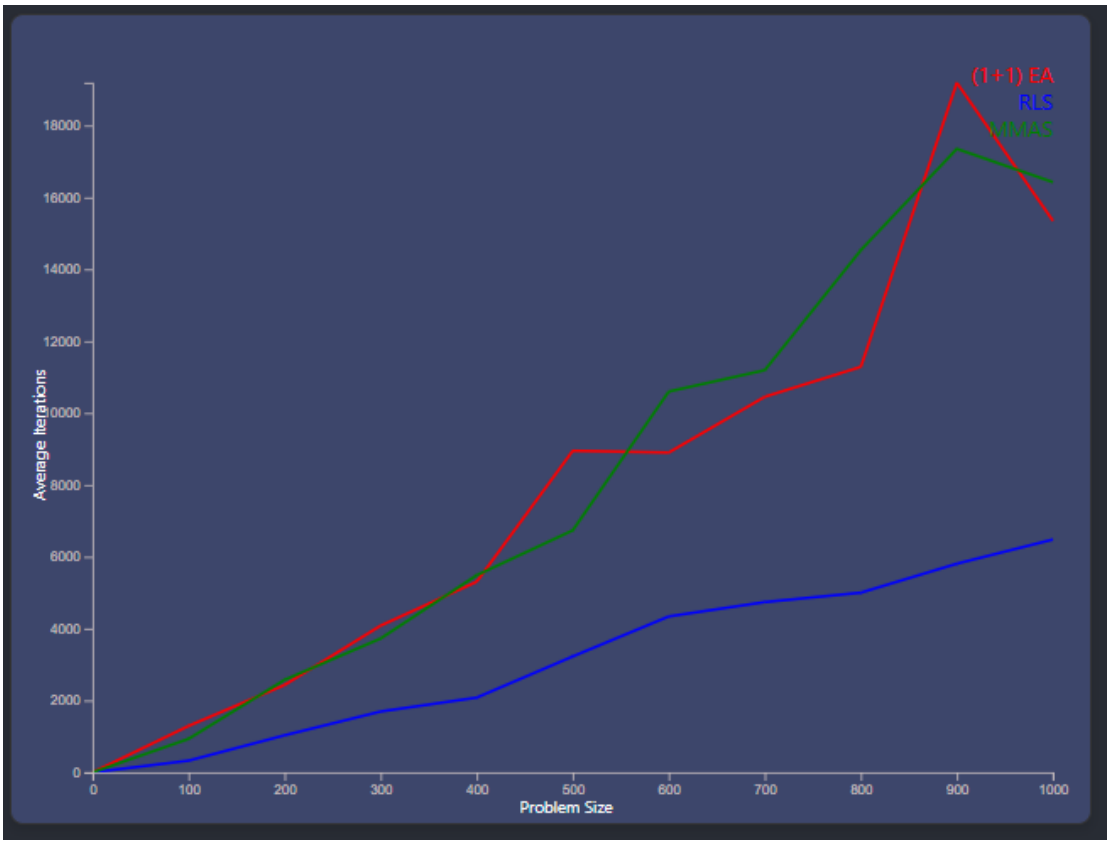


Figure 9: Graph based of 2

Problem Size	(1+1) EA	RLS	MMAS
0	0	0	0
10	83.42	48.19	82.05
20	339.48	200.44	311.92
30	725.55	426.94	737.65
40	1419.27	822.62	1397.62
50	2245.57	1202.29	2136.22
60	3051.44	1715.56	2970.33
70	4347.55	2467.72	4185.24
80	5560.35	3172.61	5381.94
90	6751.94	3998.19	6808.87
100	8308.45	5031.9	8422.71

Table 3: Performance comparison of algorithms for Leading Ones

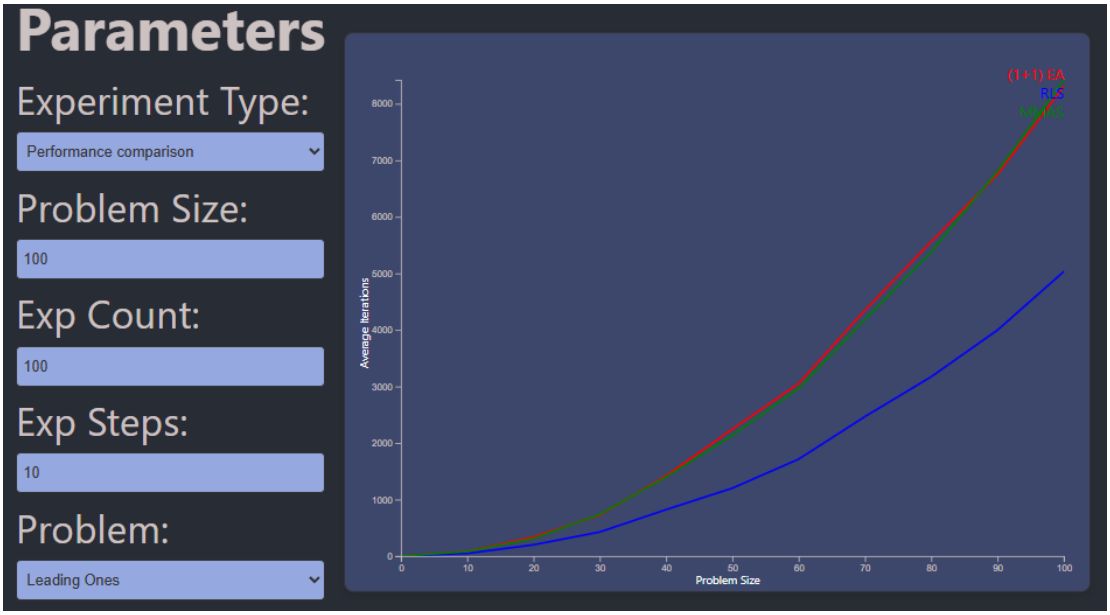


Figure 10: Graph based of 3 and the chosen parameters

Problem Size	(1+1) EA	RLS	MMAS
0	0	0	0
100	8640.2	5075	7662.6
200	35430.8	18755	34990.8
300	79522.8	45805.6	74423.6
400	141651.6	76758.6	144347.4
500	150000	128760.6	150000
600	150000	150000	150000
700	150000	150000	150000
800	150000	150000	150000
900	150000	150000	150000
1000	150000	150000	150000

Table 4: Performance comparison of algorithms for Leading Ones with larger values. Limit of 150000 iterations reached.

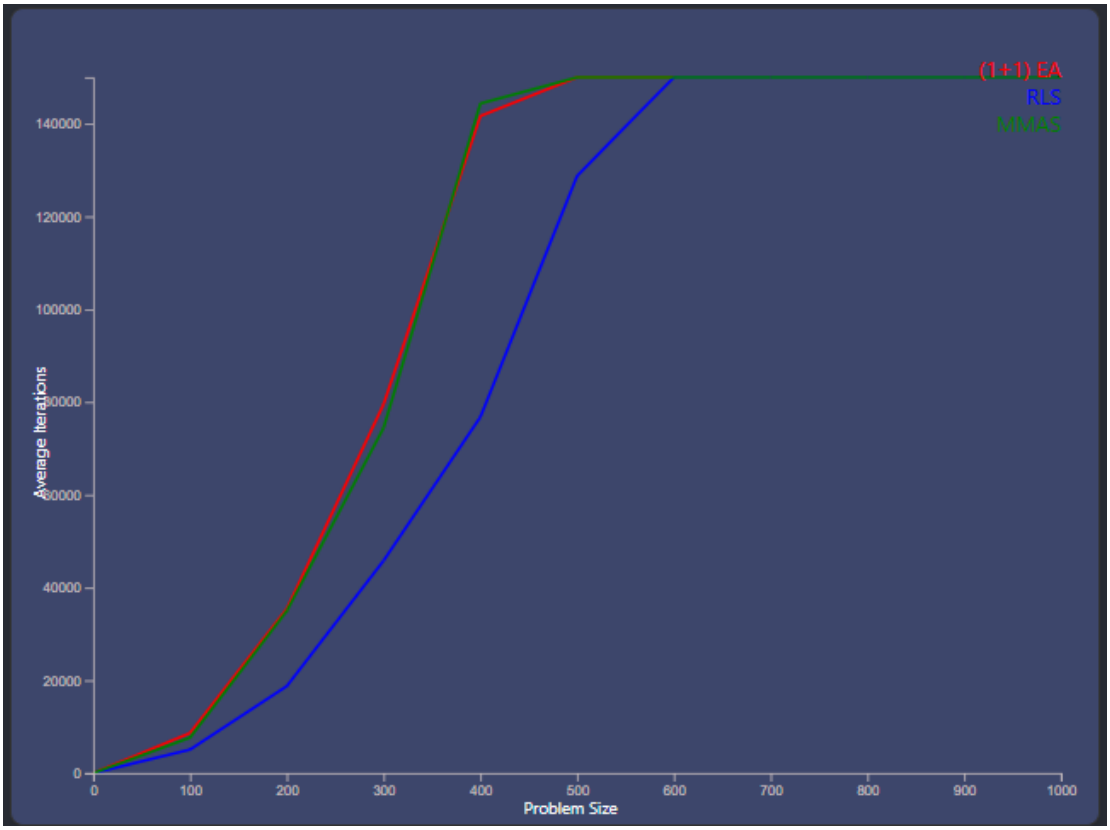


Figure 11: Graph based of 4

Problem Size	CR 100000	CR 500000	CR 1000000	CR 2000000	CR 10000000
0	0	0	0	0	0
50	3025.3447	3047.4004	3065.239	12960.911	13094.787
100	4786.571	4775.754	4778.877	4756.8613	26376.08
150	7403.213	7176.45	7180.3213	7186.936	34723.914
200	11040.679	10358.167	10259.7705	10232.731	10152.312
250	15705.825	13993.004	13846.107	13668.536	13709.535
300	22448.664	18153.217	17658.908	17619.494	17424.035
350	32975.336	22809.96	22252.033	21856.357	21650.78
400	54168.96	28026.912	26808.977	26472.957	26133.875
450	106068.68	33892.19	32297.283	31401.904	31027.43
500	129422.27	40682.58	37991.254	36718.215	35841.336

Table 5: Performance comparison across different cooling rates for SA with 10000 iteration limit.

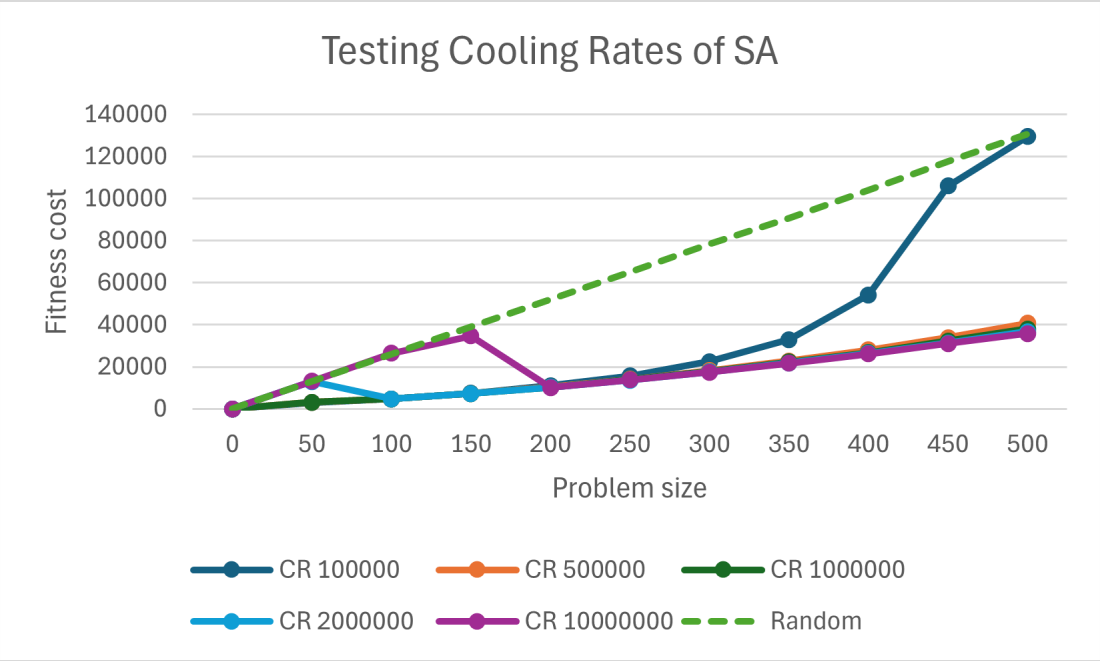


Figure 12: Graph based of 5, plotted against pure random solutions

n	(1+1) EA	MMAS a1 b1	a1 b5	a1 b10	a1 b15	a1 b25	a1 b30	a1 b40
0	0	0	0	0	0	0	0	0
10	2328.278	1495.4172	1416.0884	1438.1564	1471.1234	1447.6719	1441.8352	1444.683
20	4725.192	2803.1829	1945.112	1952.886	1964.3	1964.6628	1952.1998	2000.7634
30	7003.2754	4232.412	2382.344	2362.2441	2346.9124	2344.9712	2364.045	2392.421
40	9203.957	5702.5244	2808.5706	2719.7097	2672.9485	2716.2498	2690.74	2746.6152
50	11540.008	7263.4917	3247.6782	3014.7627	2997.5981	3016.3716	3019.021	3008.9229
60	13748.39	8580.613	3583.6262	3283.4844	3295.2124	3326.73	3298.434	3345.4497
70	16330.996	10173.481	3897.6191	3574.584	3539.8843	3552.125	3542.9668	3585.3904
80	18726.328	11791.415	4201.55	3779.75	3809.4048	3805.7078	3810.009	3803.79
90	20794.537	13215.171	4439.5796	4041.613	4003.1853	3989.5884	4039.56	4052.0618
100	23166.148	14732.869	4650.925	4282.0605	4225.366	4196.248	4269.974	4287.3

Table 6: Performance MMAS with varying β values. (1+1)EA for reference

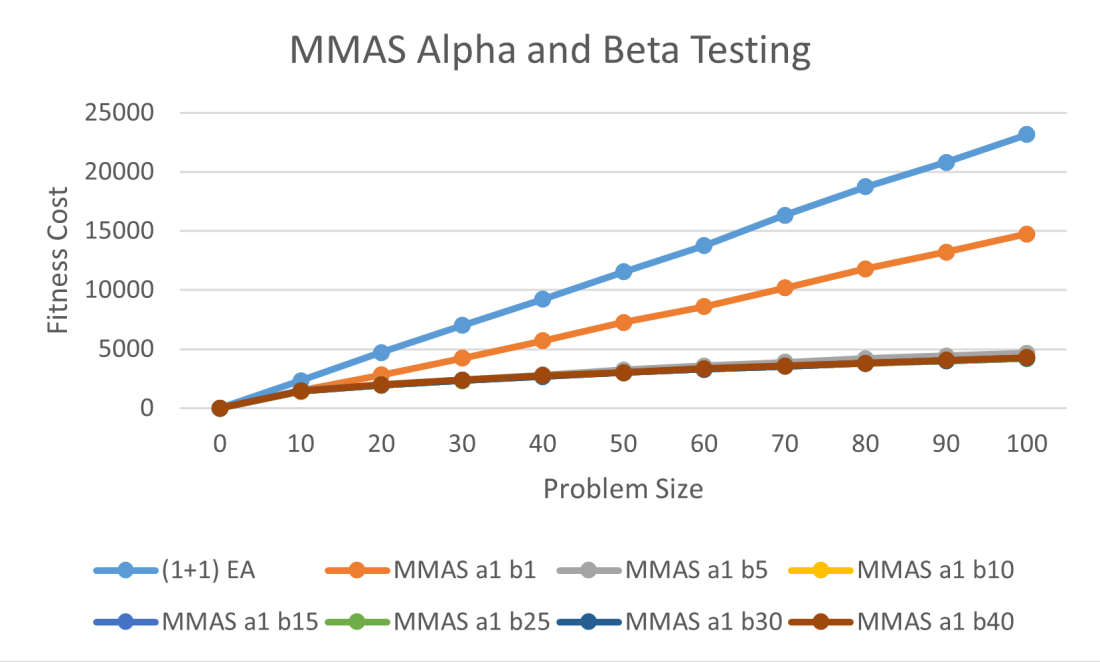


Figure 13: Graph based of 6, plotted against (1+1)EA for reference

Problem Size	(1+1) EA	Sim Anneal	MMAS
0	0	0	0
50	10926.214	3199.1362	2715.869
100	21023.293	5027.9385	4101.42
150	34031.97	7281.5034	5162.4463
200	47701.168	9723.545	5945.54
250	56432.867	14304.248	6760.7236
300	69666.52	18118.83	6852.1235
350	83438.586	22247.512	8096.647
400	90885.98	28032.059	8656.938
450	100645.69	32467.818	9165.531
500	113195.09	38077.42	9715.736

Table 7: TSP Average distance comparison of (1+1) EA, Simulated Annealing, and MMAS with 10000 iterations and max problem size 500

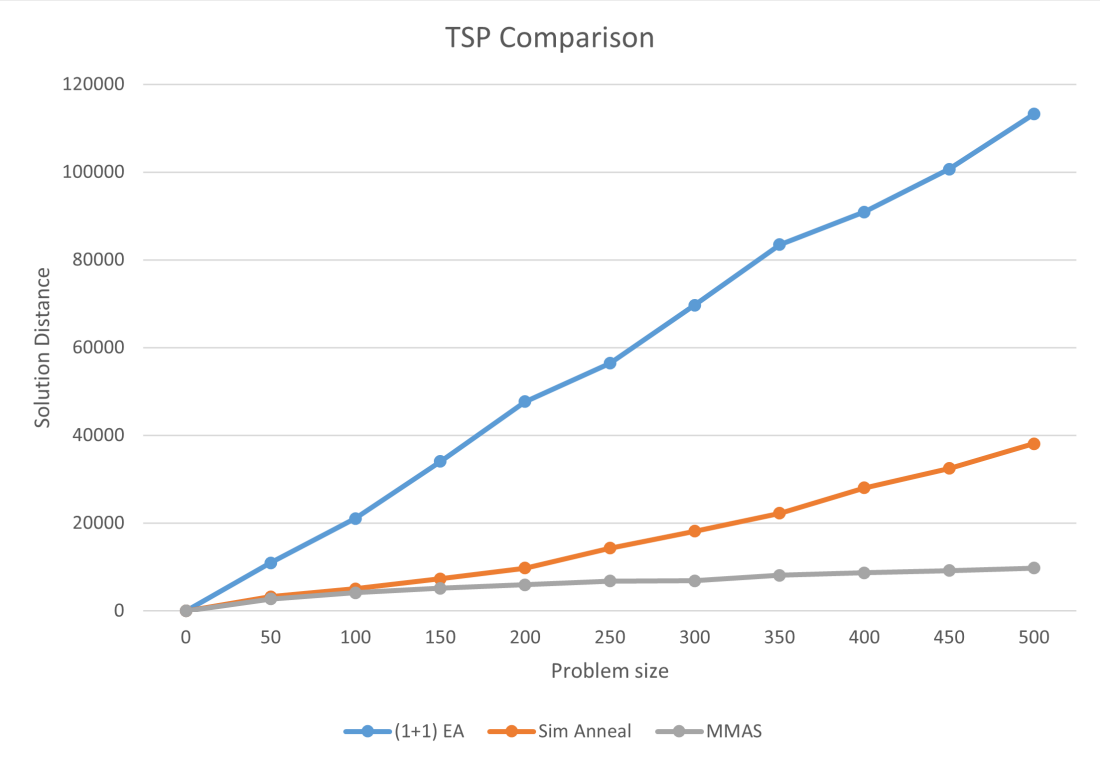


Figure 14: Graph based of 7

Code Snippets

Listing 1: RunExperiment method in BitStringSimulation

```
public T[] RunExperiment<T>(Func<int[], BitAlgorithm, BitProblem, ...
    T> simulation)
{
    int bitstring = algorithmI;
    T[] result = new T[Utility.CountSetBits((ulong)algorithmI)];

    int currentAlgo = 0;
    BitProblem selectedProblem = GetProblem(problemI);

    int[] startValue = ...
        Utility.InitializeRandomBinaryString(problemSize);
    for (int i = 0; i < ALGORITHM_COUNT; i++)
    {
        if ((algorithmI & 1 << i) != 0)
        {
            result[currentAlgo] = simulation(startValue, ...
                GetAlgorithm(i, selectedProblem), selectedProblem);
            currentAlgo++;
        }
    }
    Debug.WriteLine($"Result:\n{Utility.DisplayAnyList(result)}");
    return result;
}
```

Listing 2: RunDetailedSimulation method in BitStringSimulation

```
public int[][] RunDetailedSimulation(int[] startValue, ...
    BitAlgorithm algorithm, BitProblem problem)
{
    List<int[]> result = new List<int[]>();
    result.Add(startValue);
    algorithm.InitializeAlgorithm(startValue.Length);
    for (int i = 0; i < MAX_ITERATIONS; i++)
    {
        int[] bestRes = result[result.Count - 1];
        int[] mutatedRes = algorithm.Mutate(bestRes);
        if (problem.FitnessCompare(bestRes, mutatedRes))
        {
            result.Add(mutatedRes);
        }
        else
        {
            result.Add(bestRes);
        }
        //Debug.WriteLine($"Iteration {i}: ...
            {Utility.CountSetBits(result[result.Count - 1])}");
        if (Utility.CountSetBits(result[result.Count - 1]) == ...
            problemSize)
        {
            Debug.WriteLine($"Solution found after {i} iterations");
            break;
```

```
        }
    }

    return result.ToArray();
}
```

Listing 3: Mutate for bit RLS

```
public override int[] Mutate(int[] original)
{
    int[] mutated = new int[original.Length];
    int flippedIndex = random.Next(original.Length);
    for (int i = 0; i < original.Length; i++)
    {
        mutated[i] = original[i];
        if (i == flippedIndex) mutated[i] = 1 - mutated[i];

    }

    return mutated;
}
```

Listing 4: Mutate for tsp Simulated Annealing

```
public override int[] Mutate(int[] original)
{
    int[] mutated = Utility.TSPOpt2(original);
    if ( Utility.TSPCompare(nodes, mutated, original)) // If ...
        mutated is better, take that
    {
        temperature *= alpha;
        return mutated;
    }
    else
    {   // If mutated is worse, take it with a probability
        float distDiff = Utility.TSPCalculateDistance(nodes, ...
            original) - Utility.TSPCalculateDistance(nodes, mutated);
        float lossProb = MathF.Exp( distDiff / (float)temperature);
        temperature *= alpha;
        return random.NextDouble() < lossProb ? mutated : original;
    }
}
```

References

Diestel, Reinhard: Graph Theory, 6th ed., vol. 173 (Graduate Texts in Mathematics), 6th Edition, 2025, URL: https://doi.org/10.1007/978-3-662-70107-2.

Kötzing, Timo et al.: Theoretical analysis of two ACO approaches for the traveling salesman problem, in: Swarm Intelligence 6.1 (2012), URL: https://link.springer.com/article/10.1007/s11721-011-0059-7.

Luc Muyldermans Patrick Beullens, Dirk Cattrysse and Dirk Van Oudheusden: Exploring Variants of 2-Opt and 3-Opt for the General Routing Problem, in: INFOR: Information Systems and Operational Research 37.3 (1999), URL: https://www.jstor.org/stable/25146934.

Meer, Klaus: Simulated Annealing versus Metropolis for a TSP instance, in: Information Processing Letters 104.6 (2007), URL: https://doi.org/10.1016/j.ipl.2007.06.016.

Neumann, Frank, Dirk Sudholt, and Carsten Witt: Analysis of different MMAS ACO algorithms on unimodal functions and plateaus, in: Swarm Intelligence 2008.

Neumann, Frank and Carsten Witt: Bioinspired Computation in Combinatorial Optimization: Algorithms and Their Computational Complexity, Berlin, Heidelberg 2010.

Zhang, Yu Shan and Zhi Feng Hao: Runtime Analysis of (1+1) Evolutionary Algorithm for a TSP Instance, in: Proceedings of the First International Conference on Swarm, Evolutionary, and Memetic Computing (SEMCCO 2010), vol. 6466 (Lecture Notes in Computer Science), Berlin, Heidelberg 2010, pp. 296–304.