Technical University of Denmark

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

Bachelor Thesis

Name:
Mhd Shady Ghabour

Study number:
s224740

Supervisor:
Carsten Witt

June 6, 2025

Abstract

This research introduces a software framework designed to visualize and evaluate nature-inspired metaheuristic algorithms on combinatorial optimization problems. The framework features a user-friendly graphical interface that allows users to set up and run experiments across different problem types, including pseudo-boolean functions (e.g., OneMax and LeadingOnes) and various instances of the Traveling Salesperson Problem (TSP). It includes a range of algorithms such as Evolutionary Algorithms (e.g., $(1+1)$ EA, and $(\mu+\lambda)$ EA), Simulated Annealing, and Ant Colony Optimization (ACO) variants. Performance data is collected to evaluate how efficiently each algorithm performs under different parameter settings. Extensive experiments were carried out, and the results generally matched theoretical expectations regarding the optimization time. For pseudo-boolean problems, the $(\mu+\lambda)$ EA showed the fastest convergence, followed by randomized local search and simulated annealing. For TSP, ACO algorithms proved to be the most effective in finding high-quality solutions. These findings underline the strengths of each family of nature-inspired algorithms: the simplicity of Evolutionary Algorithms, the potential of Simulated Annealing, and the strong performance of ACO on complex problems.

Acknowledgments

We acknowledge the use of artificial intelligence models, including ChatGPT and Google Gemini, for improving the clarity and tone of the text in this report.

Contents

1 Introduction **5**

2 Background **6**
2.1 Optimization Problems . . . 6
2.1.1 Combinatorial Optimization . . . 6
2.2 Search Spaces . . . 7
2.2.1 Bitstrings . . . 7
OneMax . . . 8
LeadingOnes . . . 8
2.2.2 Permutations . . . 9
The 2-opt Technique . . . 10
The 3-opt Technique . . . 10
Fitness Calculation regarding TSP . . . 11
2.3 Evolutionary algorithms (EA) . . . 12
2.3.1 (1+1) EA . . . 12
OneMax solved by (1+1) EA: . . . 13
LeadingOnes solved by (1+1) EA: . . . 13
2.3.2 $(\mu+\lambda)$ EA . . . 14
2.3.3 (1+1) EA on TSP . . . 14
(1+1) EA solving TSP using 2-opt and 3-opt . . . 15
(1+1) EA solving TSP using Poisson-distributed Mutation (2-opt) . . . 15
2.4 Randomized Local Search (RLS) . . . 16
2.4.1 OneMax solved by RLS . . . 17
2.4.2 LeadingOnes solved by RLS . . . 17
2.5 Simulated Annealing (SA) . . . 17
2.5.1 Cooling Schedule . . . 18
2.5.2 Simulated Annealing on TSP . . . 18
2.6 Ant Colony Optimization (ACO) . . . 20
2.6.1 Construction Graph Solving Pseudo-Boolean Functions . . . 21
2.6.2 MMAS for Binary Problems . . . 21
2.6.3 1-Ant for Binary Problems . . . 22
2.6.4 TSP . . . 23
Tour Construction . . . 24
Pheromone Update . . . 24
Parameter Settings for Ant System and MMAS on TSP . . . 24
Ant System for TSP . . . 25
MAX-MIN Ant System for TSP . . . 25
2.7 Related Work . . . 25

3 Problem Analysis **27**
3.1 Implemented Scope of Algorithms and Problems . . . 27
3.2 Framework Functionalities and Visualizations . . . 27
3.3 Challenges . . . 27

4 Design **28**
4.1 Program Structure . . . 28
4.2 Graphics . . . 29
4.2.1 User Interface (UI) . . . 29
4.2.2 Program Flow . . . 31
4.2.3 Display . . . 35
Graph . . . 35
Bitstrings . . . 36
Boolean Hypercube . . . 36
TSP Visualization . . . 38
Pheromone Trails . . . 39
4.3 Testing . . . 40

5 Implementation **41**
5.1 Wizard-Based Schedule Creation Architecture . . . 41
5.1.1 MainInteraction and AppState Integration . . . 41
5.1.2 MainInteraction and InfoTextProvider Integration . . . 42
5.1.3 RadioButtonListCell for Stopping Conditions . . . 43
5.1.4 Dynamic UI Adaptation Mechanism . . . 44
5.2 ScheduleInteraction and its supporting classes . . . 44
5.2.1 Class Integrations . . . 45
5.2.2 AlgorithmFactory . . . 45
5.2.3 ProblemFactory . . . 46
5.2.4 Visualization Classes . . . 46
5.2.5 SearchSpace Classes . . . 47
5.2.6 StoppingCondition Classes . . . 47
5.3 Algorithm Implementations . . . 48
5.3.1 Mechanisms to escape oscillation . . . 49
5.4 Graphics . . . 50

6 Evaluation and Results **51**
6.1 Evolutionary Algorithms and Randomized Local Search . . . 51
6.1.1 OneMax . . . 51
6.1.2 LeadingOnes . . . 52
6.1.3 TSP . . . 54
(1+1) EA solving TSP using 2-opt and 3-opt . . . 54
(1+1) EA solving TSP using Poisson-distributed Mutation (2-opt) . . . 55
6.2 $(\mu+\lambda)$ EA . . . 57
6.2.1 OneMax . . . 57
6.2.2 LeadingOnes . . . 59
6.3 Simulated Annealing (SA) . . . 60
6.3.1 OneMax . . . 61
6.3.2 LeadingOnes . . . 62
6.3.3 TSP . . . 63
6.4 Ant Colony Optimization (ACO) . . . 64

6.4.1 1-Ant . . . 64
OneMax . . . 64
LeadingOnes . . . 66
6.4.2 MMAS . . . 67
OneMax . . . 67
LeadingOnes . . . 68
TSP . . . 69
6.4.3 AS . . . 72
TSP . . . 72
6.5 Comparison . . . 74
6.5.1 OneMax . . . 75
6.5.2 LeadingOnes . . . 76
6.5.3 TSP . . . 77

7 Future work **79**
7.1 Visualization . . . 79
7.2 Algorithms . . . 79
7.3 AI-Assisted Analysis . . . 79

8 Conclusion **80**

9 References **82**

10 Appendix **85**
10.1 Decisions made while development . . . 85
10.2 getOptionsForStep . . . 86
10.3 getOptimalTourLengthForTspFile . . . 87
10.4 FitnessBoundReached . . . 88
10.5 SA's mechanism to escape oscillation . . . 89
10.6 EA . . . 90
10.7 2-opt method . . . 91
10.8 3-opt method . . . 92

1 Introduction

Optimization is everywhere in daily lives. Whether finding the fastest route to work, scheduling tasks efficiently, or businesses trying to maximize profits while minimizing costs. Optimization involves finding the best possible solutions to complex problems. As these problems grow larger and become more complicated, finding optimal solutions becomes increasingly difficult. Consider planning a route to visit several locations, with just 10 locations, there are over 3 million possible routes. [1] For 20 locations, the number exceeds the atoms in the universe. Clearly, checking every possible solution isn't practical for many real-world problems.

This is where metaheuristic algorithms come in. These algorithms take inspiration from nature's own problem solving processes such as; evolution, physical annealing, or ant colonies finding food, to efficiently search for good solutions without checking every possibility. They provide practical approaches to solving complex problems across many fields including logistics, engineering, and computer science.

This project creates a framework for visualizing, and comparing three important families of nature-inspired algorithms. Evolutionary Algorithms, inspired by natural selection and evolution, maintain a population of solutions that evolve over time through selection, mutation, and recombination. The framework includes both the simple (1+1) EA, which works with just one solution, and the more advanced $(\mu + \lambda)$ EA, which maintains multiple solutions simultaneously. Random Local Search serves as a straightforward approach that makes small, random changes to a solution and keeps the changes only if they improve the solution. Despite its simplicity, RLS provides an important baseline for comparison with more complex approaches.

The framework also implements Simulated Annealing, which draws inspiration from the metallurgical process of heating and controlled cooling of materials. The algorithm starts with high "temperature" that allows accepting worse solutions to escape local optima, then gradually "cools down" to fine-tune the final solution. Additionally, the project includes Ant Colony Optimization, modeled after how ant colonies find efficient paths to food sources through pheromone trails.

To test these algorithms, the framework applies them to classic optimization problems in two different types of search spaces. Binary problems (OneMax and LeadingOnes) involve finding the optimal sequence of 0s and 1s, while permutation problems (Traveling Salesman Problem) involve finding the optimal ordering of elements. These problems, though relatively simple to understand, provide significant insights into algorithm behavior and performance characteristics.

What makes this framework useful is its visualization capabilities. Users can watch algorithms in action, seeing how solutions evolve over time, comparing performance between different algorithms, and understanding how parameter settings affect results. Through interactive displays, users gain insights into how these nature-inspired algorithms navigate complex solution spaces. The framework provides real-time feedback on algorithm progress, showing metrics such as generation count, fitness values, and function evaluations as the optimization proceeds. By making these complex algorithms visible and interactive, this project bridges the gap between theoretical understanding and practical application.

2 Background

2.1 Optimization Problems

Optimization problems appear in many aspects of our daily lives. From determining the shortest path between locations in navigation systems to scheduling university courses to maximize educational opportunities while minimizing time conflicts. Optimization problems involve finding the best solution from a set of possibilities according to some specified criteria.

Optimization problems can be categorized into two fundamental types based on the nature of their variables: continuous and discrete. Continuous optimization involves variables that can assume any value within a specific range. Examples include determining the optimal temperature for a chemical reaction, or finding the ideal dimensions for a structural component. These problems typically benefit from calculus-based techniques and gradient methods. The continuous nature of the search space allows for incremental adjustments toward optimal solutions, though the presence of multiple local optima often complicates the search for global optima. [2]

Discrete optimization, in contrast, deals with variables restricted to distinct, often countable values. Examples include selecting projects for investment from a portfolio, determining the most efficient route for visiting multiple locations, or assigning tasks to workers. These problems, frequently referred to as combinatorial optimization problems, present significant computational challenges as the solution space grows exponentially with problem size. [2]

2.1.1 Combinatorial Optimization

Combinatorial optimization deals with selecting the best possible solution from a discrete, often exponentially large, set of candidates. Formally, such a problem is defined by a triple (S, f, Ω), where S denotes the complete search space of candidate solutions, f is the objective (or fitness) function that evaluates each candidate's quality, and Ω represents the set of constraints that feasible solutions must satisfy. Depending on the problem's goal, f is minimized or maximized, classifying the task as either a minimization or maximization problem.

Classic examples include the traveling salesman problem (TSP), which requires finding the shortest possible route that visits each city exactly once and returns to the starting point, and the minimum spanning tree problem, where the objective is to connect all nodes of a graph with the least total edge weight. These problems are central in both theoretical computer science and practical fields such as logistics, network design, and scheduling. [2]

Due to the combinatorial explosion in the number of possible solutions, many of these problems are NP-hard, implying that no known algorithm can solve all instances optimally in polynomial time. This complexity has driven the development of heuristic and metaheuristic approaches, which are designed to efficiently explore large search spaces and yield high-quality approximations of the optimal solution. Metaheuristics like evolutionary algorithms, simulated annealing, and ant colony optimization are popular because they incorporate strategies that balance the exploration of new regions and the exploitation of known promising areas. [3]

Moreover, these metaheuristic methods are generally stochastic in nature, meaning that they incorporate random elements in their search processes. While this randomness implies that repeated runs

might yield different outcomes, it is precisely what allows these algorithms to escape local optima and adapt to different problem landscapes. [4]

2.2 Search Spaces

Search spaces cover the full set of candidate solutions for an optimization problem. While the previous section introduced the core ideas of combinatorial optimization, here we focus specifically on how these candidates are structured and what challenges arise when searching through them.

One useful way to understand a search space is by picturing it as a landscape. In this view, every candidate solution is like a point on a map, and its quality (or fitness) is represented as its "elevation." For maximization problems, higher elevations mean better solutions; for minimization problems, the aim is to reach the lowest valleys. [5]

This simple landscape metaphor highlights several key concepts:

Global Optimum: The very best solution in the entire search space either the highest peak or the lowest valley. [5]

Local Optimum: A solution that appears optimal compared to its immediate neighbors but may not be the best overall. [5]

Plateaus: Flat regions in the landscape where many solutions have the same fitness value. Plateaus can be challenging because a lack of a clear "uphill" or "downhill" direction may cause algorithms to stall. [6]

An important element in this exploration is the idea of a neighborhood. A neighborhood defines which solutions are considered "close" to a given solution. Within this context, a solution is a local optimum if none of its neighboring solutions have a better fitness value. Although determining the exact neighborhood structure can be complex, the landscape metaphor offers an easy way to think about how optimization algorithms move from one candidate to another. [2]

2.2.1 Bitstrings

Bitstrings offer a compact representation of candidate solutions by encoding them as sequences of binary digits. Defined by a dimension n, each bitstring consists of n positions where the only permissible values are 0 and 1. This restriction yields a controlled search space of 2^n possible configurations, allowing a wide variety of problems to be modeled. In the following sections, different bitstring problems will be defined and examined, and the discussion will highlight how heuristic methods are employed to navigate the bitstring search space solving those problems.

Linear Pseudo-Boolean Function is a function $f : \{0,1\}^n \to \mathbb{R}$ is considered **linear** if it can be expressed as a weighted sum of the individual bits of the bitstring. More formally, a linear function is of the form:

$$f(x_n, x_{n-1}, \ldots, x_1) = w_n x_n + w_{n-1} x_{n-1} + \cdots + w_1 x_1 + w_0$$

where $x_i \in \{0,1\}$ for all i, and $w_0, w_1, \ldots, w_n$ are real-valued weights. Linear Pseudo-Boolean Functions are particularly significant in evolutionary computation because they have a well-defined and predictable structure, making them easier to optimize. For example, the OneMax function, where $f(x)$ counts the number of 1s in the bitstring, is a simple linear function that is often used as a benchmark for testing optimization algorithms. [7]

Unimodal functions are characterized by a single global optimum, with all points in the search space either having a higher fitness than their neighbors or being locally optimal. Formally, a function $f : \{0,1\}^n \to \mathbb{R}$ is unimodal if, for every nonoptimal point x, there exists a Hamming neighbor x' such that $f(x') > f(x)$. This condition ensures that starting from any nonoptimal point, there is always a direction in the Hamming space that leads to an improvement in the function value. If the function has only one local maximum (which is also global), it is called weakly unimodal. [7]

Unimodal functions are important because they often present the possibility of finding the global optimum by following a simple local search method, such as hill climbing or mutation-based approaches. This contrasts with more complex functions that may contain multiple local optima, potentially trapping optimization algorithms in suboptimal solutions. Despite the apparent simplicity of unimodal functions, they are not always easy to optimize. For instance, the Needle function, which consists of a nearly flat plateau with a single peak, is a challenging unimodal function that requires significant exploration to locate its global optimum. [7]

In this project, we focus on two well-known fitness functions: OneMax and LeadingOnes Both of these functions serve as standard benchmarks for evaluating optimization algorithms due to their simplicity and well-understood properties.

OneMax

The OneMax function is defined as the number of 1-bits in a bitstring. Formally, for a bitstring $x = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$, the fitness value of the bitstring is given by:

$$\text{OneMax}(x) = \sum_{i=1}^{n} x[i]$$

where $x[i]$ is the i-th bit of the bitstring x. This function is unimodal, meaning there is only one global optimum, which occurs when the bitstring consists entirely of 1s. [7]

LeadingOnes

The LeadingOnes function is slightly more complex. For a given bitstring $x = (x_1, x_2, \ldots, x_n)$, the fitness value of the bitstring is determined by counting the number of leading 1s in the bitstring, starting from the left. Formally, the fitness function is defined as:

$$\text{LeadingOnes}(x) := \sum_{i=1}^{n} \prod_{j=1}^{i} x[j]$$

The LeadingOnes function rewards bitstrings that have 1s starting from the leftmost position and penalizes any 0s that occur before all the 1s. This creates a fitness landscape that is also unimodal, where the global optimum occurs when the bitstring consists entirely of 1s. The function has the property that mutations are more likely to increase fitness if they flip a 0 to a 1 at the beginning of the string, which makes it an interesting challenge for optimization algorithms. [7]

Both of these functions are linear pseudo-Boolean functions, and their simple structures make them ideal test cases for evaluating the efficiency of various heuristic algorithms, including Evolutionary Algorithms (EAs), Simulated Annealing, and Ant Colony Optimization. These functions will be the focus of our investigation as we explore how different optimization methods navigate their search spaces to locate the global optima.

2.2.2 Permutations

A permutation is defined as an ordered arrangement of a set of distinct items. In the context of the TSP involving n cities, each unique permutation of these cities represents a specific sequence in which the salesperson can visit them . For instance, if a salesperson needs to visit cities A, B, and C, the permutation (A, B, C) signifies a tour where city A is visited first, followed by city B, then city C, and finally a return to the starting city A. Each different ordering of these cities constitutes a distinct potential route for the salesperson. [8], [9]

This idea of ordered arrangements directly relates to the concept of a search space, which includes all possible candidate solutions to an optimization problem. In the case of the TSP, the search space is the set of all possible permutations of the given cities. Each permutation within this space represents a potential tour, and the objective of solving the TSP is to identify the permutation that corresponds to the tour with the minimum total travel distance. Therefore, permutations provide the fundamental language for describing and exploring the solution landscape of the Traveling Salesman Problem.

To further clarify the role of permutations in the TSP, consider a scenario with n cities, which can be labeled as $1, 2, \ldots, n$. A permutation of these cities, denoted as $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, represents a specific tour. In this tour, the salesperson starts at some city, then visits city π_1, followed by city π_2, and so on, until reaching city π_n, after which the salesperson returns to the starting city to complete the cycle. The sequence defined by the permutation dictates the order of city visits. The total number of distinct permutations for n cities is given by the factorial of n, denoted as $n!$, which is $n \times (n-1) \times (n-2) \cdots \times 1$. This number grows extremely rapidly as the number of cities increases. For example, with just 5 cities, there are $5! = 120$ possible permutations, but for 10 cities, the number jumps to $10! = 3,628,800$. For 15 cities, the number of permutations is approximately 1.3×10^{12}, and for 20 cities, it reaches 2.4×10^{18}. [1]

It is important to consider the nuances related to the starting city and the direction of travel. In many instances of the TSP, particularly those that are symmetric (meaning the distance between city A and city B is the same as the distance between city B and city A), a tour and its reverse represent the same total travel distance. Additionally, because the tour is a cycle, the specific city chosen as the starting point does not fundamentally change the tour itself, only the order in which the cities are listed in the permutation. Due to these symmetries, for a symmetric TSP with n cities, the number of truly unique tours is often considered to be $(n-1)!/2$. This adjustment accounts for the fact that fixing a starting city reduces the number of relevant permutations by a factor of n, and reversing the order of the remaining cities yields the same tour in terms of total distance. Understanding these symmetries is crucial for both theoretical analysis and the development of efficient algorithms for the TSP. [8], [10]

Local search algorithms represent a class of optimization methods that iteratively attempt to improve a candidate solution by making small, localized changes to it. Starting from an initial feasible solution, these algorithms explore the "neighborhood" of the current solution, where a neighbor is defined as a solution that can be reached from the current one by a simple modification. In the context of the TSP, where solutions are represented by permutations of cities, these modifications typically involve transforming one permutation (tour) into another by altering the order in which cities are visited. The effectiveness of a local search algorithm heavily depends on the definition of the neighborhood. A well-designed neighborhood structure allows the algorithm to explore promising regions of the search space efficiently. For the TSP, two well-known local search techniques that

operate by transforming permutations are the 2-opt and 3-opt algorithms. [11]

The 2-opt Technique

The 2-opt algorithm is a widely recognized and frequently used local search technique for the TSP due to its simplicity and effectiveness. The fundamental idea behind 2-opt is to identify and eliminate crossings in a tour, as such crossings typically indicate a suboptimal path. The algorithm operates by taking a tour that might have edges crossing each other and reordering it so that the edges no longer intersect, thereby reducing the total travel distance. The process of a 2-opt move involves the following steps: First, two non-adjacent edges are selected from the current tour. Let's say these edges connect cities A to B and city C to D. Removing these two edges effectively splits the tour into two separate paths. The next step is to reconnect the endpoints of these paths in a different way. Instead of connecting A to B and C to D, the 2-opt move considers connecting A to C and B to D, or alternatively, A to D and B to C. The connection that results in a shorter total tour length is then chosen. [11], [12]

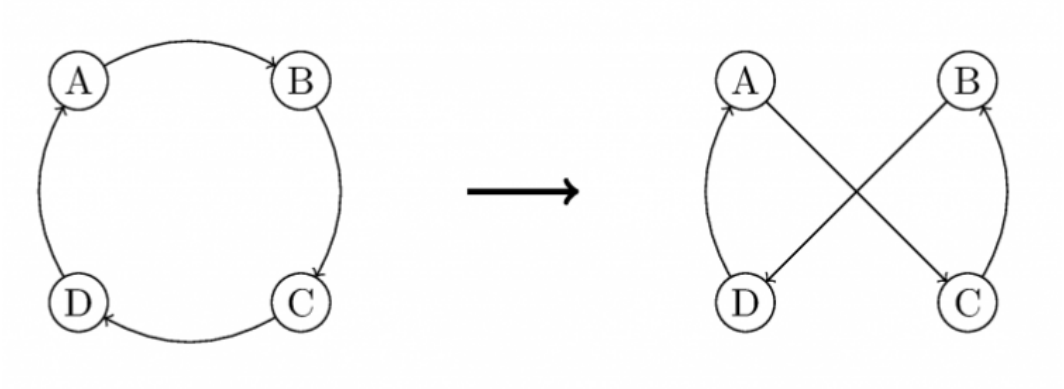

Figure 1: 2-opt Technique

The 3-opt Technique

The 3-opt algorithm is an extension of the 2-opt technique that involves more complex modifications to the tour by considering three edges instead of two. By considering rearrangements involving three edges, the 3-opt algorithm has a greater potential to escape local optima in the search space that might trap the 2-opt algorithm. A 3-opt move begins by selecting three edges in the current tour to remove. This action divides the tour into three distinct sub-paths. There are eight possible ways to reconnect these three sub-paths to form a complete tour, one of which is the original configuration (i.e., no change). The 3-opt algorithm evaluates the total length of each of the seven new possible tour configurations. If any of these new configurations result in a shorter total travel distance compared to the current tour, the shortest one replaces the current tour. It's worth noting that a 3-opt move can sometimes be achieved through a sequence of two or three 2-opt moves. While the 3-opt algorithm offers the potential to find better solutions than 2-opt by exploring a larger neighborhood of possible tours, it comes with a higher computational cost. A full exploration of the 3-opt neighborhood, which involves trying all possible combinations of three edges and their reconnections, typically has a time complexity of around $O(n^3)$ per iteration. This increased computational demand means that while 3-opt can potentially lead to higher quality solutions, it requires more computational time compared to 2-opt. The choice between using 2-opt or 3-opt often involves a trade-off between the desired solution quality and the available computational resources. [13], [14]

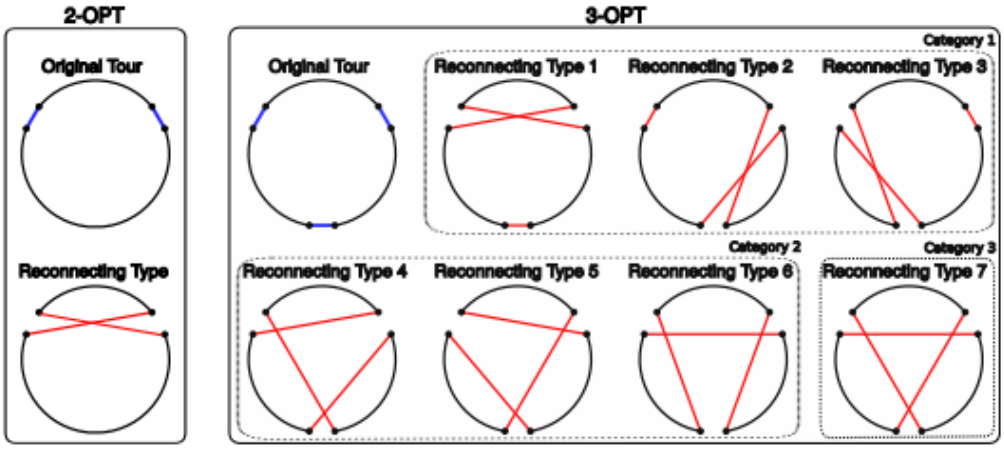

Figure 2: 2-opt and 3-opt Techniques for solving the TSP

Figure 2 shows the 2-opt and 3-opt heuristics used in the 'step-by-step improvement' strategy to solve the Traveling Salesman Problem (TSP). The blue links indicate the edges selected for removal from the current tour. Removing k links results in k segments (shown as edges in black). Once new links (shown in red) are added, a new tour is formed. In the case of the 2-opt move, there is only one reconnecting type. However, for the 3-opt move, there are seven possible reconnecting types. [14]

Fitness Calculation regarding TSP

As explained earlier, each permutation of cities represents a potential tour, and the quality of this tour is evaluated by calculating its total travel distance. This total distance serves as the "fitness" of the permutation, and the objective of solving the TSP is to find the permutation that gives the minimum total distance. The calculation of the total distance for a given permutation: $(c_1, c_2, \ldots, c_n)$ involves summing the distances between each consecutive pair of cities in the sequence. This includes the distance from the starting city to the first city in the permutation (c_1), the distances between c_1 and c_2, c_2 and c_3, and so on, up to the distance between the last city c_n and the starting city to complete the cycle.

Benchmark instances of the TSP, often sourced from libraries like TSPLIB, are frequently used to test and compare the performance of different algorithms. These instances provide sets of cities with defined distances, and for many of them, the shortest possible tour length (the optimal solution) is known. The table below presents a selection of symmetric TSP instances and their corresponding optimal tour lengths.

Name	#cities	Type	Optimal
a280	280	EUC_2D	2579
berlin52	52	EUC_2D	7542
eil101	101	EUC_2D	629
kroa100	100	EUC_2D	21282
lin318	318	EUC_2D	42029
pcb442	442	EUC_2D	50778
pr1002	1002	EUC_2D	259045
rat99	99	EUC_2D	1211
rd400	400	EUC_2D	15281
st70	70	EUC_2D	675

Table 1: Symmetric Traveling Salesman Problems from TSPLIB showing optimal tour lengths. [21]

A crucial component of this fitness calculation is the distance metric used to determine the separation between any two cities. For many TSP instances, especially those where the cities' locations are given as coordinates in a two-dimensional plane, the Euclidean distance is the most commonly used metric. The Euclidean distance between two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ is calculated using the formula:

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

This metric represents the straight-line distance between two points. Local search techniques like 2-opt and 3-opt rely on this fitness calculation to evaluate the effectiveness of the permutation transformations they perform. After a swap or reconnection, the total distance of the resulting tour (represented by the new permutation) is calculated. If this new distance is less than the distance of the previous tour, the change is accepted, and the algorithm continues its search from the improved tour. The goal is to iteratively find permutations (tours) with lower and lower total travel distances. [15]

2.3 Evolutionary algorithms (EA)

Evolutionary Algorithms (EAs) are a subset of algorithms inspired by biological processes observed in nature. These algorithms have become quite popular since the mid-1960s. EAs belong to the field of bio-inspired computation, which also includes other well-known approaches such as Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO). Evolutionary algorithms try to solve problems by evolving sets of search points, inspired by the evolution process in nature, such that satisfying results are obtained. A lot of the tasks that have been solved by EAs lie in the field of real-world applications. In real-world applications, the function to be optimized is often unknown and function values can only be obtained by experiments. [2]

2.3.1 (1+1) EA

The (1+1) Evolutionary Algorithm is a basic form of evolutionary algorithm used for solving optimization problems. It is characterized by its simplicity, as it operates on a single solution at a time, employing mutation to generate new candidate solutions. The algorithm begins with a random selection of an initial bit string. Each iteration involves creating a new bit string by independently

flipping the bits of the current string, using a mutation probability of $1/n$, where n is the length of the bit string. The fitness of this new bit string is then compared to that of the current bit string. The new string replaces the current one if it has greater or equal fitness; otherwise, the current string is retained. This process is repeated until a defined stopping criterion is met. [2]

Algorithm 1 (1+1) EA [2]

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

Algorithm 1: Illustrates how the (1+1) EA works. This algorithm iteratively improves a single solution (represented by a bit string s) by generating a mutated offspring s' and replacing the parent with the offspring if the offspring's fitness is greater than or equal to the parent's.

OneMax solved by (1+1) EA:
The (1+1) EA has been a subject of significant study in evolutionary algorithm theory. Mühlenbein (1992) provided the first theoretical result on the runtime of the (1+1) EA. Where he presented an upper bound on the expected runtime of the (1+1) EA for the OneMax function. [2]

Later Witt (2013) further improved the understanding of the (1+1) EA's runtime on oneMax, refining the bound to

$$\text{en ln n} + O(n) \approx\ 2.71\ \ n\ \ln\ n$$

which can be cosidered as the optimization time for (1+1) EA to solve the oneMax. [16]

LeadingOnes solved by (1+1) EA:
The (1+1) EA has also been analyzed for the LeadingOnes problem. Böttcher, Doerr, and Neumann showed that with a mutation probability of p, the optimization time for LeadingOnes is: [17]

$$\frac{1}{(2p)^2}\left((1-p)^{-n} + 1 - (1-p)\right)$$

For the standard mutation probability of $p = \frac{1}{n}$, this is approximately

$$0.86n^2$$

They also demonstrated that this standard mutation probability is not optimal and that a mutation probability of

$$p \approx \frac{1.59}{n}$$

yields a faster optimization time of approximately: [17]

$$0.77n^2$$

2.3.2 $(\mu + \lambda)$ EA

Evolutionary Algorithms employ various strategies to navigate complex search spaces. While the (1+1) EA operates on a single solution, many EAs utilize a population of solutions to enhance the search process. This algorithm maintains a population of μ parent individuals and generates λ offspring at each iteration. The use of a population allows for a more exploration of the search space and can provide increased robustness compared to single-solution approaches.

Algorithm 2 The $(\mu + \lambda)$ EA, maximizing a given function $f : \{0,1\}^n \to \mathbb{R}$, with population size μ, offspring population size λ and mutation rate $p = \frac{1}{n}$. [20]

Initialization:

Create a population of μ individuals by choosing $x^{(i)} \in \{0,1\}^n$, $1 \leq i \leq \mu$ uniformly at random.
Let the multiset $X^{(0)} := \{x^{(1)}, ..., x^{(\mu)}\}$ be the population at time 0. Let $t := 0$.

Optimization:

while an optimum has not been reached **do**
    $X' := X^{(t)}$;
    Mutation phase:
    for $i = 1, ..., \lambda$ **do**
        Choose $x \in X^{(t)}$ uniformly at random;
        Create x' by flipping each bit of x with probability p;
        $X' := X' \cup \{x'\}$;
    end for
    Selection phase:
    Create the multiset $X^{(t+1)}$, the population at time $t+1$, by deleting the λ individuals with lowest f-value in X';
    $t := t + 1$;
end while

The $(\mu + \lambda)$ EA proceeds iteratively, as outlined in Algorithm 2. Initially, a population of μ individuals is created by randomly generating μ bit strings of length n. These bit strings constitute the initial population, denoted as $X^{(0)}$. The algorithm then enters an iterative loop where, in each iteration t, λ offspring are produced. Each offspring is generated by randomly selecting a parent from the current population and applying a mutation operator. Typically, this mutation involves flipping each bit of the selected parent with a probability p. The resulting offspring are added to a multiset X', which contains both the parent population and the newly generated offspring. Following the generation of offspring, a selection process occurs. The next generation's population, $X^{(t+1)}$, is formed by selecting the μ individuals with the highest fitness values from the multiset X'. This selection strategy, known as $(\mu + \lambda)$ selection, ensures that the subsequent population consists of the fittest individuals from both the parents and the offspring. The algorithm continues this process of mutation and selection until a predefined termination condition is met, such as finding an optimal solution or reaching a maximum number of iterations.

2.3.3 (1+1) EA on TSP

The (1+1) EA, initially used for problems with bit strings like OneMax and LeadingOnes, can also be used for other optimization problems, such as the Traveling Salesman Problem (TSP). However,

there's a key difference: unlike OneMax and LeadingOnes where we wanted to get as many '1's as possible (a maximization problem), in the TSP, the goal is to find the shortest possible route that visits every city exactly once and returns to the starting city (a minimization problem). This means that 'fitness' has a different meaning in the TSP. A shorter tour is considered to have a higher fitness. The (1+1) EA, can be adapted in various ways to tackle the TSP. In this section, we will explore two such adaptations. The first uses a combination of 2-opt and 3-opt operators for mutation, while the second employs Poisson-distributed mutations using 2-opt.

(1+1) EA solving TSP using 2-opt and 3-opt

This algorithm describes a version of the (1+1) EA for the TSP that uses both the 2-opt and 3-opt methods to create new tours. At each step, the algorithm randomly chooses between using the 2-opt or the 3-opt method with equal probability to modify the current tour. The idea behind using both is that 2-opt is a simple and quick way to make small improvements, while 3-opt can make more significant changes that might help the algorithm find better solutions in the long run.

Algorithm 3 (1+1) EA for TSP [18]

Begin
Initialization: choose randomly a permutation of $\{1,2,\ldots,n\}$ as an initial solution ξ_0
while (termination-condition is not met) **do**
    Select 2-opt and 3-opt operator with probability $1/2$ respectively as a mutation operator
    Mutate ξ_k and get ξ'_k as an offspring
    if $f(\xi_k) > f(\xi'_k)$ **then**
        $\xi_{k+1} = \xi'_k$
    else
        $\xi_{k+1} = \xi_k$
    end if
    $k \leftarrow k+1$
end while
End

In this algorithm, the (1+1) EA uses 2-opt and 3-opt operators as mutation operators. The algorithm starts by choosing a random permutation of the cities as an initial solution. Then, it iteratively generates new solutions by applying either a 2-opt or a 3-opt operator with equal probability. The new solution replaces the current solution if it has a shorter tour length. This process continues until a termination condition is met.

(1+1) EA solving TSP using Poisson-distributed Mutation (2-opt)

Another way to adapt the (1+1) EA for the TSP is to use the 2-opt method multiple times in a row, with the number of times determined by a Poisson distribution. The Poisson distribution is a way to predict how many times something might happen in a certain period, if we know the average rate at which it happens. In this case, we use it to decide how many 2-opt changes to make to a tour in one step. The idea behind this is to introduce some randomness in how much we change the tour at each step. Sometimes, just one 2-opt change might be enough to make an improvement. Other times, we might need to make several changes to get out of a situation where the tour can't be easily improved with just one change. By using a Poisson distribution with an average of 1, the algorithm usually

makes one 2-opt change per step, but it also has a chance to make more changes in a single step.

Here's how the algorithm works:

Algorithm 4 (1+1) EA for TSP with Poisson-distributed Mutation [19]

Begin
Initialization: choose randomly a permutation of $\{1,2,\ldots,n\}$ as an initial solution π_0
while (termination condition is not met) **do**
    Sample S from a Poisson distribution with $\lambda = 1$
    Perform $S+1$ 2-opt mutations on the current solution π
    Let π' be the resulting mutated solution
    if $f(\pi') \leq f(\pi)$ **then**
        $\pi \leftarrow \pi'$
    else
        $\pi \leftarrow \pi$
    end if
end while
End

The algorithm starts by randomly creating an initial tour. Then, in each step, it does the following: It randomly picks a number S from a Poisson distribution (with an average of 1). It then performs $S+1$ 2-opt changes to the current tour. The "+ 1" ensures that at least one 2-opt change is made in each step. After making these changes, it calculates the total length of the new tour. If the new tour is shorter than or the same length as the current tour, it becomes the new current tour. Otherwise, the current tour is kept. This process continues until a stopping condition is met. This method allows the algorithm to explore the possible tours with varying levels of change at each step, which can help it find a good solution.

2.4 Randomized Local Search (RLS)

Randomized Local Search (RLS) is a simple search heuristic used for optimization problems, sharing similarities in structure with the (1+1) Evolutionary Algorithm. Both algorithms operate on a single candidate solution and employ a cycle of generating a new solution and selecting based on fitness. However, they differ in how the new solution is generated. While the (1+1) EA mutates the current solution by flipping each bit independently with a certain probability ($\frac{1}{n}$). RLS generates a new candidate solution by making a small, random change to the current solution. Specifically, it selects one bit of the current bit string uniformly at random and flips it. The fitness of this new candidate solution is then evaluated. If the new solution is at least as good as the current one (in terms of fitness), it replaces the current solution; otherwise, it is discarded. This process continues iteratively in an attempt to find better solutions.

Algorithm 5 Randomized Local Search for maximizing $f\colon \{0,1\}^n \to \mathbb{R}$. [22]

1: **Initialization:** Sample $x \in \{0,1\}^n$ uniformly at random;
2: **Optimization:** for $t = 1,2,3,\ldots$ do
  Choose $j \in [n]$ uniformly at random and set
  $y \leftarrow x \oplus e^n_j$; // mutation step: flip jth bit
  if $f(y) \geq f(x)$ **then** $x \leftarrow y$; // selection step

Algorithm 5: Illustrates the process of RLS. The algorithm maintains a single solution x and iteratively generates a new candidate solution y by flipping a single randomly chosen bit of x. The current solution x is updated to y if y has a fitness value greater than or equal to that of x.

2.4.1 OneMax solved by RLS

RLS has been analyzed for its performance on various test functions, including OneMax. For the OneMax function, RLS is known to find the optimal solution in expected time $O(n\ log\ n)$. [22] This indicates that RLS is efficient for the OneMax problem, and achieves a better expected runtime on OneMax than the standard (1+1) EA with mutation probability $\frac{1}{n}$.

2.4.2 LeadingOnes solved by RLS

The expected running time of RLS for solving the LeadingOnes problem has also been subject to theoretical analysis. It has been shown that RLS finds the optimum for LeadingOnes in expected time of $\frac{n^2}{2} + O(n^2)$. [23]

2.5 Simulated Annealing (SA)

Simulated Annealing is a powerful probabilistic metaheuristic designed to find good approximations to the global optimum of a function in a large search space. Its name and core concepts are inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to reduce defects and achieve a more stable structure. In optimization, SA mimics this process by exploring the search space while gradually reducing the "temperature" parameter, which influences the probability of accepting suboptimal solutions. Unlike greedy hill-climbing methods that only move towards better solutions, SA's key feature is its ability to occasionally accept solutions that are worse than the current one. This probabilistic acceptance of worse solutions, especially at higher temperatures, allows the algorithm to escape local optima and explore the search space more broadly. As the temperature decreases according to a defined cooling schedule, the probability of accepting worse solutions diminishes, causing the algorithm to converge towards the better regions of the search space and eventually perform a local search in a manner similar to hill climbing at very low temperatures.

Algorithm 6 Simulated Annealing for maximizing $f\colon S \to \mathbb{R}$. [24]

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

Algorithm 6: Illustrates the process of Simulated Annealing. The algorithm maintains a single solution x_t and generates a new candidate solution $f(y)$ from the neighborhood $N(x_t)$. Better solutions are always accepted, while worse solutions are accepted with a probability determined by the fitness difference between $f(y)$ and $f(x-t)$ and the current temperature $T(t)$. The temperature typically decreases over time according to a cooling schedule, influencing the balance between exploration and exploitation.

2.5.1 Cooling Schedule

Simulated Annealing's effectiveness depends on its cooling schedule, which determines how quickly the temperature parameter decreases over time. When using a geometric cooling schedule, the temperature decreases exponentially by multiplying with a cooling factor in each iteration: [25]

$$T_{t+1} = \alpha \cdot T_t$$

where T_t is the temperature at iteration t and α is the cooling factor that is typically set close to 1 (e.g., 0.9, 0.95, 0.99, etc). [25]

When the value of α is close to 1, the temperature decreases slowly at first, allowing extensive exploration of the search space, and then more rapidly as the algorithm progresses. This slow initial cooling helps avoid premature convergence to local optima, especially in rugged fitness landscapes. The cooling factor can be adjusted based on problem characteristics faster cooling (smaller α) leads to quicker convergence but potentially worse solutions, while slower cooling (larger α) increases exploration but requires more iterations.

2.5.2 Simulated Annealing on TSP

SA is well-suited for TSP due to its probabilistic approach, which allows it to navigate the complex fitness landscapes often encountered in TSP instances and avoid getting stuck in suboptimal local solutions.

In the context of TSP, a "state" in the Simulated Annealing process corresponds to a complete tour, represented by a permutation of cities. The "energy" of a state is the total length of the tour defined by that permutation. The objective of SA for TSP is to find a state (tour) that minimizes this energy (tour length).

The crucial component connecting the general SA algorithm to the TSP is the definition of the neighborhood structure and the method for generating neighboring states. For TSP, a common approach

is to define the neighborhood based on simple modifications to the current tour, such as swapping pairs of cities or reversing segments of the tour. As illustrated in the pseudocode provided (Algorithm 7), a frequent choice for the neighborhood operator in SA for TSP is the 2-opt technique. A neighbor tour is generated by applying a single random 2-opt move to the current tour, effectively removing two non-adjacent edges and reconnecting them to form a new valid tour. This 2-opt move changes the current solution locally, creating a neighboring solution whose energy (tour length) can be evaluated.

Algorithm 7 SA–2-Opt procedures for TSP [26]

1: Initialize temperature T, initial state S_0 using $S_0 = \{1,2,3,4,5\ldots,M\}$, E_0, S_{opt}, and E_{opt}.
2: **for** $i = 1$ **to** n **do**
3: $m \leftarrow 0$
4: **while** $m \leq$ number of cities or $T \geq 0.01$ **do**
5: Update state S, Calculate energy E using $E = \sum_{i=1}^{N-1} d_i + d_N$
6: Optimize the state using 2-opt algorithm
7: Calculate $\Delta E = E - E_0$, $\omega \in [0,1]$, p using $d(A,B)+d(C,D) < d(A,D)+d(B,C)$
8: **if** $\Delta E < 0$ **then**
9: $S_0 \leftarrow S, \quad E_0 \leftarrow E$
10: **if** $E < E_{\text{opt}}$ **then**
11: $S_{\text{opt}} \leftarrow S_0, \quad E_{\text{opt}} \leftarrow E_0$
12: **else**
13: **go to** step 11
14: **end if**
15: **else**
16: **if** $p > \omega$ **then**
17: **go to** step 5
18: **else**
19: **go to** step 11
20: **end if**
21: **end if**
22: $m \leftarrow m + 1$
23: **end while**
24: **if** inner loop termination criteria satisfied **then**
25: $T \leftarrow T \times r$
26: **go to** step 3
27: **end if**
28: **end for**

Algorithm 7 outlines the specific SA–2-Opt procedure used. The algorithm begins with an Initialization phase (Step 1) where an initial temperature T is set, along with an initial state S_0 (representing a tour, here initialized as $\{1,2,...,M\}$, where M is the number of cities in the tour), its corresponding energy E_0 (tour length), and variables to track the best solution found so far (S_{opt}, and E_{opt}). The algorithm then enters an iterative process structured around nested loops. The outer loop (Step 2) iterates n times, while the inner loop (Step 4) continues as long as a counter m is less than or equal to the number of cities or the temperature T is above a threshold (0.01). Within the inner loop, a

candidate new state S is generated and its energy E is calculated (Step 5). This step involves applying the 2-opt algorithm (Step 6), suggesting that a 2-opt move is used to generate the neighbor state S from the current state S_0. The change in energy ΔE between the candidate state and the current state is calculated (Step 7), along with values ω and p which are used in the acceptance decision. If the candidate state has lower energy $(\Delta E < 0)$, it is accepted as the new current state $S_0 \leftarrow S, \quad E_0 \leftarrow E$, and the best-found solution is updated if the accepted state is better than E_{opt} (Steps 8-11). If the candidate state has higher or equal energy $(\Delta E \geq 0)$, it is accepted probabilistically based on the comparison of p and ω (Steps 12-16). After the acceptance check, the counter m is incremented (Step 22). Upon termination of the inner loop based on its criteria (Step 24), the temperature T is updated according to a cooling factor $T \leftarrow T \times r$, and the algorithm returns to the start of the outer loop (Step 26). The outer loop continues for n iterations.

2.6 Ant Colony Optimization (ACO)

Ant Colony Optimization is a metaheuristic algorithm inspired by the behavior of ant colonies, specifically their method of finding the shortest paths between their nest and food sources. At its core, ACO leverages the principle of stigmergy, a form of indirect communication where individuals modify the environment to influence the behavior of others. In the biological inspiration, ants deposit pheromone on the ground while traversing paths, creating chemical trails. The concentration of pheromone on a given path attracts other ants, leading to a positive feedback loop where more frequently used or shorter paths accumulate higher pheromone levels and thus become more attractive over time. This process allows the colony to discover efficient routes. The translation of this natural phenomenon into an optimization algorithm involves artificial ants that iteratively construct solutions to different optimization problems. These artificial ants navigate a representation of the problem, making probabilistic choices based on two factors: the intensity of artificial pheromone trails associated with problem components (e.g., edges in a graph) and, typically, heuristic information relevant to the problem structure. As artificial ants complete the construction of a solution, they deposit pheromone on the components used, with the amount often correlated to the quality of the solution found. To prevent premature convergence and facilitate exploration, a pheromone evaporation mechanism is also included, gradually reducing the pheromone levels over time. This iterative process guides the search towards promising regions of the solution space, allowing the algorithm to converge on high-quality solutions. The origins of ACO can be traced back to observations and experiments on real ant colonies, such as the classic double bridge experiment where ants successfully identify the shorter path between two points by means of pheromone trails. This experiment demonstrates how the simple local actions of individual ants, combined with stigmergic communication, give rise to complex, intelligent collective behavior. [6]

ACO algorithms, such as MAX–MIN ant system (MMAS) and 1-ANT, are commonly applied to combinatorial optimization problems. However, they can also be used to solve pseudo-Boolean functions, which involve bitstring search spaces. In these problems, candidate solutions are represented as bitstrings of length n, such as in the OneMax and LeadingOnes functions. To apply ACO to these types of problems, we use a construction graph approach. The construction graph is a directed graph where the vertices represent possible states of the solution (i.e., bitstrings), and edges represent transitions between different solutions. The pheromone values on the edges are updated as the algorithm explores the search space, guiding the search towards better solutions. [27]

Building upon its application to various problem structures, including the adaptation for pseudo-

Boolean functions, Ant Colony Optimization has demonstrated effectiveness in addressing the Traveling Salesman Problem (TSP).

2.6.1 Construction Graph Solving Pseudo-Boolean Functions

Building a bitstring solution for binary optimization problems of length n is framed as a path-finding problem in a construction graph. As shown in Figure 3, the graph consists of n sequential nodes, each representing a bit position from 1 to n. At each node i (where $i = 1, 2, \ldots, n$), there are two possible edges: one corresponds to setting the i-th bit to 0, and the other to setting it to 1.

An artificial ant constructs a complete bitstring solution $(x_1, x_2, \ldots, x_n)$ by traversing this graph, making a choice (either 0 or 1) at each node i. The path taken by the ant through the n nodes forms a unique bitstring. The edges of the graph are associated with pheromone values, specifically related to the decision of setting the i-th bit to a value $b \in \{0, 1\}$. Let $\tau_{i,b}$ represent the pheromone associated with choosing the value b for bit i. An ant probabilistically selects the current bit's value based on pheromone levels and potentially heuristic information guiding the choice of edges from the current node.

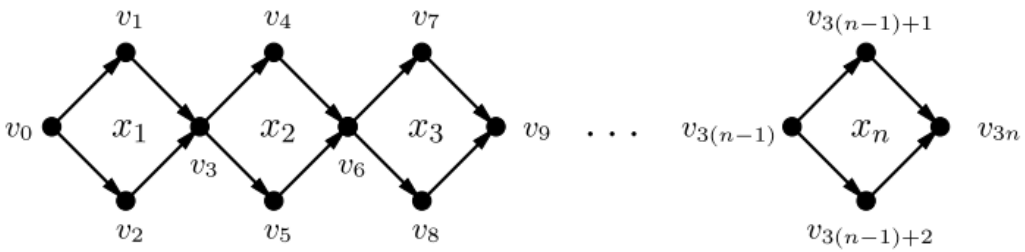

Figure 3: Construction graph for pseudo-Boolean functions. [27]

2.6.2 MMAS for Binary Problems

The MAX–MIN Ant System (MMAS) is a well-suited ACO variant for binary problems. Unlike standard Ant System where all ants might deposit pheromone, MMAS typically restricts pheromone deposition to only the best-performing ant. This strategy aims to intensify the search around the most promising solutions. Furthermore, MMAS employs minimum and maximum bounds on pheromone levels to prevent stagnation and encourage exploration.

Algorithm 8 MMAS* [27]

Set $\tau(u,v) = \frac{1}{2}$ for all $(u,v) \in E$
Create x^* using `Construct(C, τ)`
Update pheromones w.r.t. x^*
repeat
    Create x using `Construct(C, τ)`
    if $f(x) > f(x^*)$ **then**
        $x^* := x$
    end if
    Update pheromones w.r.t. x^*
until stopping criteria are met

As shown in Algorithm 8, the algorithm begins by initializing pheromone values on all edges of the construction graph, often to an equal value (Step 1). An initial best solution x^* is constructed (Step 2), and pheromones are updated based on this initial best solution (Step 3). The algorithm then enters a loop where, in each iteration, a new solution x is constructed probabilistically using the pheromone information on the construction graph (Step 5). The fitness of this newly constructed solution x is compared to the fitness of the current best solution found so far x^* (Step 6). If x is better than x^*, x^* is updated to x (Step 7). Pheromones are updated based on the current best solution x^* (Step 9). This iterative process of constructing solutions and updating pheromones based on the best solution continues until specific stopping criteria are met (Step 10).

The pheromone update mechanism in MMAS for binary problems involves both evaporation and deposition, applied to the edges of the construction graph: [27]

$$\tau'_{i,b} = \begin{cases} \min\left\{(1-\rho)\cdot\tau_{i,b}+\rho, 1-\frac{1}{n}\right\}, & \text{if } (i,b) \in P(x), \\ \max\left\{(1-\rho)\cdot\tau_{i,b}, \frac{1}{n}\right\}, & \text{otherwise.} \end{cases}$$

This formula calculates the new pheromone value $\tau'_{i,b}$ for the edge corresponding to setting bit i to value b. It applies an evaporation factor $(1-\rho)$ to the current pheromone level $\tau_{i,b}$ in both cases. If the edge (i,b) was part of the path taken to construct the current best solution x^*, a deposition amount of ρ is added. If the edge was not part of the best path, no pheromone is deposited beyond what remains after evaporation. Crucially, after the evaporation and potential deposition, the resulting pheromone value is constrained by minimum $\left(\frac{1}{n}\right)$ and maximum $\left(1-\frac{1}{n}\right)$ bounds. These bounds ensure that pheromone values do not become excessively high on any single edge, preventing the algorithm from exploiting only a very limited part of the search space, nor do they become too low, which would prematurely eliminate potentially useful exploration directions. This mechanism helps maintain diversity in the search process and improves the algorithm's ability to find the global optimum.

2.6.3 1-Ant for Binary Problems

Similar to MMAS, 1-Ant operates within the construction graph framework for bitstring problems, where artificial ants build solutions by making sequential decisions (setting bits to 0 or 1) guided by pheromone trails. A key characteristic that distinguishes 1-Ant from algorithms like standard Ant System is its focus on the contribution of a single ant per iteration to guide the search.

Algorithm 9 1-Ant [27]

1: Set $\tau_{(u,v)} = \frac{1}{2}$ for all $(u,v) \in E$
2: Create x^* using `Construct(C, τ)`
3: Update pheromones w.r.t. x^*
4: **repeat**
5: Create x using `Construct(C, τ)`
6: **if** $f(x) \geq f(x^*)$ **then**
7: $x^* := x$
8: Update pheromones w.r.t. x^*
9: **end if**
10: **until** stopping criteria are met

The structure of Algorithm 9 shares similarities with the MMAS* pseudocode. It begins with the initialization of pheromone values on all edges of the construction graph, typically to $\frac{1}{2}$ (Step 1). An initial best solution x^* is constructed (Step 2), and the pheromones are updated according to this initial best solution (Step 3). The algorithm then enters a repetitive loop. In each iteration, a single new solution x is constructed probabilistically by an artificial ant traversing the construction graph, guided by the current pheromone levels (Step 5). The fitness of this new solution x is compared to that of the current best solution x^* (Step 6). A key difference here is that the best solution x^* is updated if the fitness of the new solution x is better or equal to x^* (Step 7). If x^* is updated, the pheromone values on the graph edges are then updated based on the path taken by this new best solution (Step 8). This cycle of constructing a single solution continues until predefined stopping criteria are satisfied (Step 9).

1-Ant algorithm uses different pheromone update mechanism than the mechanism used for MMAS: [28]

$$\tau'_{i,b} = \min\left\{\frac{(1-\rho)\cdot\tau_{i,b}+\rho}{1-\rho+2n\rho}, \frac{n-1}{2n^2}\right\} \quad \text{if} \quad (i,b) \in P(x),$$

and

$$\tau'_{i,b} = \max\left\{\frac{(1-\rho)\cdot\tau_{i,b}}{1-\rho+2n\rho}, \frac{1}{2n^2}\right\} \quad \text{if} \quad (i,b) \notin P(x).$$

For edges (i,b) that are part of the path representing the current best solution x^*, the pheromone is updated by applying evaporation $(1-\rho)\tau_{i,b}$, adding a deposition amount scaled by ρ, and dividing the result by a normalization factor $(1-\rho+2n\rho)$. The final value is then capped by a maximum bound $\frac{n-1}{2n^2}$. For edges not included in the path of x^*, only evaporation and normalization are applied, and the resulting pheromone level is constrained by a minimum bound $\frac{1}{2n^2}$. These specific bounds and the normalization factor $\frac{1}{1-\rho+2n\rho}$ are characteristic of the 1-Ant algorithm and influence the exploration–exploitation balance differently compared to the MMAS* variant discussed earlier. The minimum and maximum bounds, $\frac{1}{2n^2}$ and $\frac{n-1}{2n^2}$ respectively, play a similar role to those in MMAS by preventing pheromone values from becoming too extreme, thus encouraging continued exploration across the search space.

2.6.4 TSP

To apply ACO to the TSP, the TSP is represented as a complete graph where cities correspond to nodes and the routes between cities are represented by edges. Each edge (i,j) connecting city i and city j is associated with two values: a pheromone level τ_{ij} and a heuristic value η_{ij}. The heuristic value is commonly set as the inverse of the distance between city i and city j, where $\eta_{ij} = \frac{1}{d_{ij}}$, representing the desirability of choosing a shorter edge.

Algorithm 10 ACO Metaheuristic Static [6]

1: Set parameters, initialize pheromone trails
2: **while** termination condition not met **do**
3: ConstructAntSolutions
4: ApplyLocalSearch ▷ optional
5: UpdatePheromones
6: **end while**

As shown in Algorithm 10, the process begins with initializing parameters and pheromone trails (Step 1). The algorithm then enters a loop that continues until a termination condition is met (Step 2). Within the loop, a set of artificial ants construct solutions (tours) probabilistically (Step 3). Optionally, a local search algorithm can be applied to the constructed solutions to further improve their quality (Step 4). Finally, the pheromone trails on the graph edges are updated based on the quality of the solutions found (Step 5).

Tour Construction

During the `ConstructAntSolutions` phase, each artificial ant builds a tour step-by-step. When an ant is at city i and needs to choose the next city j from the set of cities not yet visited (N_i^k for ant k), the decision is made probabilistically. The probability p_{ij}^k of ant k moving from city i to city j is calculated using the pheromone level τ_{ij} on the edge (i, j) and the heuristic information η_{ij}: [6]

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \quad \text{if } j \in N_i^k$$

Where α and β are parameters that control the relative influence of the pheromone trail and the heuristic information, respectively. A higher value of α means pheromone has a stronger influence, while a higher value of β gives more weight to the heuristic information (e.g., shorter distances).

Pheromone Update

After all ants have constructed their tours, the `UpdatePheromones` phase takes place. This involves updating the pheromone levels on the edges of the graph. The general pheromone update rule combines evaporation and deposition:

$$\tau'_{ij} = (1 - \rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k$$

where τ_{ij} is the pheromone level on edge (i, j), ρ is the pheromone evaporation rate ($0 < \rho \leq 1$), m is the number of ants, and $\Delta\tau_{ij}^k$ is the amount of pheromone deposited by ant k on edge (i, j). The value of $\Delta\tau_{ij}^k$ depends on whether ant k used edge (i, j) in its tour and the specific ACO algorithm variant. Specifically, $\Delta\tau_{ij}^k$ is defined as:

$$\Delta\tau_{ij}^k = \begin{cases} \frac{1}{C^k} & \text{if arc } (i, j) \text{ belongs to } T^k, \\ 0 & \text{otherwise,} \end{cases}$$

where C^k is the length of the tour T^k built by the k-th ant, computed as the sum of the lengths of the arcs belonging to T^k. If the arc (i, j) is not part of the tour T^k built by ant k, then $\Delta\tau_{ij}^k = 0$, which means no pheromone is deposited on that arc. In other words, pheromone is only deposited on arcs that are used by the ant in its tour.

Parameter Settings for Ant System and MMAS on TSP

The performance of ACO algorithms is highly dependent on the appropriate setting of their parameters (α, β, ρ, m, and initial pheromone τ_0). While optimal parameters can vary depending on the specific TSP instance, common values used for AS and MMAS are presented in the table below.

ACO algorithm	α	β	ρ	m	τ_0
AS [6]	1	2 to 5	0.5	n	$\frac{m}{C^{nn}}$
MMAS [29]	1	2 to 5	0.98	n	τ_{max}

Table 2: ACO algorithm parameters for AS and MMAS. [6] [29]

As shown, both AS and MMAS commonly use $\alpha = 1$, β values between 2 and 5, and set the number of ants m equal to the number of cities n. The key differences lie in the evaporation rate ρ and the pheromone initialization τ_0.

Ant System for TSP

In Ant System, pheromone trails are initialized using the formula $\tau_0 = \frac{m}{C^{nn}}$ where m is the number of ants and C^{nn} is the length of a tour generated by the nearest-neighbor heuristic. During tour construction, each ant starts from a randomly selected city and builds a complete tour by applying the probabilistic selection rule (2.6.4). The key characteristic of AS is its pheromone update mechanism, where all ants participate in modifying the environment. First, a uniform evaporation is applied to all edges: $\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$, where $\rho = 0.5$. Subsequently, each ant k deposits pheromone on every edge (i, j) in its tour T^k according to the quality of its solution: $\Delta\tau_{ij}^k = \frac{1}{C^k}$, where C^k is the length of the ant's tour. This contribution from all ants creates a "wisdom of crowds" effect, where edges that appear in multiple good tours receive more reinforcement. AS does not employ pheromone bounds, making it straightforward to implement but potentially more susceptible to search stagnation on larger problem instances.

MAX-MIN Ant System for TSP

MMAS establishes limits on pheromone values, where $\tau_{\max}$ represents the upper bound and $\tau_{\min}$ the lower bound. These bounds prevent any edge from becoming either too dominant or completely ignored. Initially, $\tau_{\max}$ is calculated as $\frac{1}{(1-\rho)\cdot f(s^{opt})}$, where $f(s^{opt})$ is the length of the best-found tour. The lower bound $\tau_{\min}$ is derived from $\tau_{\max}$ using $\tau_{\min} = \tau_{\max} \cdot \frac{1-\sqrt[n]{p_{best}}}{(avg-1)\cdot\sqrt[n]{p_{best}}}$, where p_{best} is the probability of constructing the best tour (typically 0.05) and $avg = \frac{n}{2}$ represents the average branching factor. Crucially, all pheromone trails are initialized to $\tau_{\max}$, encouraging extensive exploration in early iterations. Unlike AS, MMAS employs a highly selective pheromone update strategy where only the global best solution is allowed to deposit pheromone. After applying evaporation to all edges ($\tau_{ij} \leftarrow \rho \cdot \tau_{ij}$), only edges in the best tour receive a pheromone increase of $\Delta\tau = \frac{1}{L_{best}}$. Finally, all pheromone values are enforced to remain within the calculated bounds. This combination of bounds, selective updates, and maximum initialization enables MMAS to perform a more robust search than AS, particularly for larger TSP instances. [29]

2.7 Related Work

As we mentioned in the introduction, this project aims to develop a framework for visualizing and evaluating three families of nature-inspired optimization algorithms: Evolutionary Algorithms, Simulated Annealing, and Ant Colony Optimization. Several existing frameworks offer similar functionalities. Specifically, three projects served as significant sources of inspiration and provided valuable insights during the development of this framework.

The first is the FrEAK (Free Evolutionary Algorithm Kit), developed by group 427 in 2003. This framework provided users with extensive functionality, including diverse visualization options and

support for various problem types and algorithms. Its feature set influenced some of our features, such as Boolean hypercube visualization and the stopping criteria options implemented in our framework. [30]

Secondly, a framework developed by Emil Lundt Larsen was particularly influential in shaping the design of the "create schedule view" within our project. Furthermore, the clear explanations of theoretical concepts and practical implementations provided in his report served as a valuable inspiration when structuring our explanations and demonstrations of the algorithms. [31]

The third influential project is the bachelor thesis by Marcus Pihl, Jakob Kildegaard Hansen, and Victor Winther Saldeen. Their work offered important insights into the real-time graphical representation of optimization progress. Specifically, their approach to visualizing graph-based solutions in real time significantly influenced the interactive visualization features implemented in our framework, enhancing the user's experience. [32]

3 Problem Analysis

This section details the analysis of the problem undertaken in this project: the development of a framework for visualizing and evaluating nature-inspired optimization metaheuristics. The core challenge involves creating a software tool capable of demonstrating the working principles of selected algorithms from the families of Evolutionary Algorithms, Simulated Annealing, and Ant Colony Optimization, applied to both binary and combinatorial optimization problems.

3.1 Implemented Scope of Algorithms and Problems

The project focused on implementing and visualizing a specific set of algorithms for two primary categories of optimization problems: Linear Pseudo-Boolean Functions (binary problems) and the Traveling Salesman Problem (combinatorial problem). For binary problems, including the benchmark functions OneMax and LeadingOnes, the implemented algorithms are: the $(1+1)$ Evolutionary Algorithm, the $(\mu+\lambda)$ Evolutionary Algorithm, Randomized Local Search (RLS), Simulated Annealing, the MAX–MIN Ant System (MMAS), and the 1-Ant algorithm. For the Traveling Salesman Problem, the framework includes implementations of the $(1+1)$ EA utilizing 2-opt and 3-opt operators, a $(1+1)$ EA variant employing Poisson-distributed mutation with 2-opt, Simulated Annealing, MMAS, and the Ant System. While other abstract unimodal functions such as Trap, $Jump_k$, or Needle could have been included, the project concentrated on the implementation and visualization of OneMax, LeadingOnes, and various TSP instances.

3.2 Framework Functionalities and Visualizations

A central requirement of this project is the dynamic visualization of the optimization process to provide insight into algorithmic behavior. For binary problems, the framework incorporates three distinct visualization displays: a dynamic graph illustrating the progression towards optimal solutions, a bitstring display showing the evolution of the bitstring solution during algorithm execution, and a boolean hypercube visualization to represent the search space and the algorithm's path within it. For TSP, a visualization was developed to show the changes in the tour, illustrating city swaps and edge modifications as the algorithm progresses. For Ant Colony Optimization algorithms, a specialized display visualizes the pheromone trails on the edges, highlighting areas of high and low pheromone concentration. The framework is designed to dynamically present these visualizations alongside key metrics, extracting figures such as best-so-far fitness, running time, and the number of function evaluations during algorithm execution.

3.3 Challenges

Developing this framework presented several challenges. Initially, concerns included ensuring the proper functioning of Evolutionary Algorithms, Simulated Annealing, and Ant Colony Optimization. Furthermore, creating dynamic and interactive visual representations without causing UI lag or performance issues was a key consideration. Optimizing hyperparameters (e.g., mutation rates, cooling schedules, pheromone updates) for each algorithm presented a significant task. Designing an intuitive and informative user interface that effectively communicates algorithm behavior without overwhelming users was also crucial. Finally, managing the complexity of bit-string optimizations, and TSP instances while maintaining efficient runtime was a consistent challenge.

4 Design

4.1 Program Structure

To ensure flexibility, extensibility, and maintainability, the framework was structured with a high degree of abstraction and modularity. This was achieved through the use of object-oriented principles, including polymorphism, inheritance, and interfaces, allowing for seamless integration of diverse algorithms, problems, and search spaces.

Following a Model-View-Controller (MVC) pattern, the framework separates concerns between the data (Model), the user interface (View), and the application logic (Controller).

The core components of the framework are organized into seven primary packages:

- algorithms: This package contains the implementations of various metaheuristic algorithms, such as Evolutionary Algorithms, Simulated Annealing, and Ant Colony Optimization.
- interactions: This package manages the user interface interactions and application flow. It includes the FXML controllers that handle user input, manage the visualization components, and organize the execution of algorithms.
- other: This package contains helper classes and general functions that support the core functionality of the framework. This includes data structures, file I/O, and other components.
- problems: This package defines the optimization problems that the algorithms are designed to solve. It includes implementations of standard benchmark problems like OneMax, LeadingOnes, and the Traveling Salesman Problem (TSP).
- searchSpaces: This package defines the search spaces within which the algorithms operate. It includes implementations of the two search spaces bit strings and permutations, providing a standardized interface for generating and manipulating candidate solutions.
- stoppingConditions: This package defines the criteria used to terminate the execution of an algorithm. It includes implementations of common stopping conditions like reaching an optimal fitness, exceeding a selected fitness by the user, exceeding a maximum number of iterations, or finding no improvement for a certain period.
- visualization: This package contains the components responsible for visualizing the algorithm's progress and results. It includes classes for generating graphs, hypercube representations, and TSP representations.

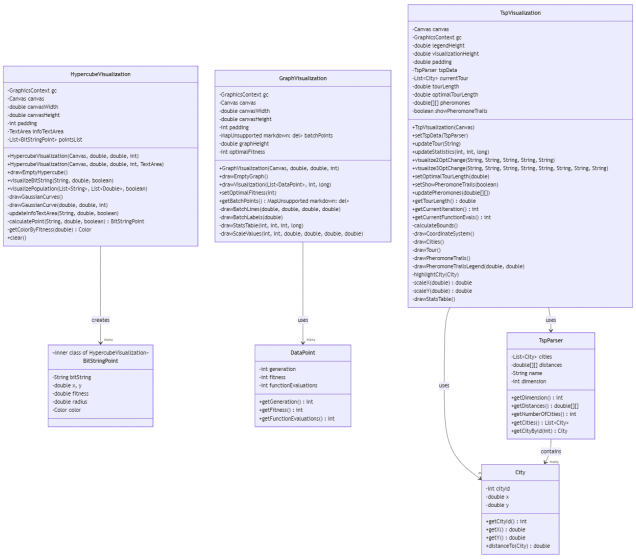

Figure 4: Class diagram of the visualization package. [33]

As shown in the class diagram, the visualization package contains three primary classes for visualizing algorithm behavior: GraphVisualization, HypercubeVisualization, and TspVisualization. GraphVisualization draws graphs for algorithm performance metrics over time. HypercubeVisualization maps binary solutions onto a 2D space using Gaussian distribution curves. TspVisualization renders TSP instances with cities and tours, supporting local search operations.

4.2 Graphics

4.2.1 User Interface (UI)

The user interface was designed with the objective of achieving a modern, intuitive, and user-friendly experience, featuring a color palette inspired by nature. The process began with initial design iterations, using Figma. In this phase, the framework developed by Emil [31] served as a significant source of inspiration, particularly influencing the design approach for creating schedules. The interface incorporates a carefully chosen palette that evokes natural elements, utilizing shades of blue to represent the sea, white symbolizing the clouds, green symbolizing trees and grass, and black symbolizing ants.

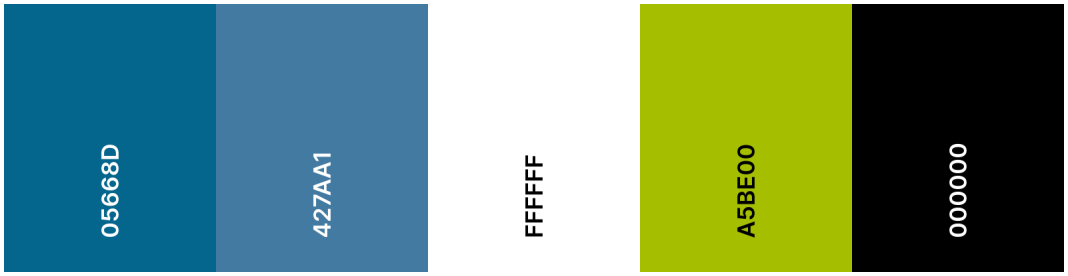

Figure 5: The color palette used in the design of the framework

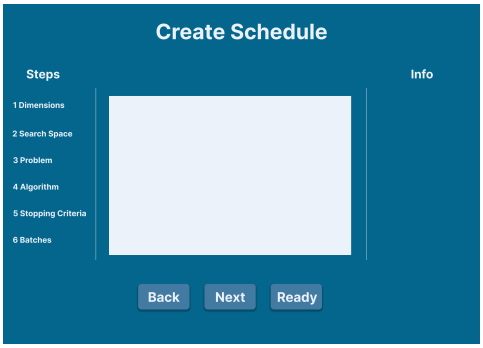

Figure 6: The initial design of the main view

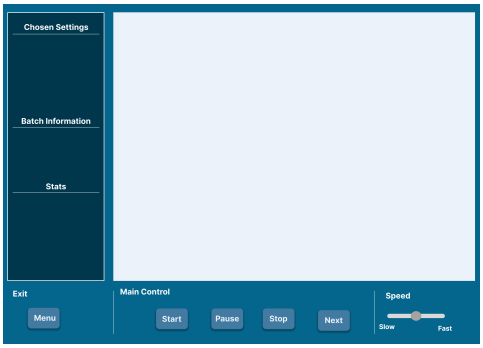

Figure 7: The initial design of the second view

The design approach for the framework's main and second views prioritized establishing a clear, linear, and intuitive user flow. The objective was to ensure that users can understand their current state within the application and anticipate the necessary subsequent actions, making the process straightforward and informative. This design philosophy was informed by the experience of using other visualization frameworks, such as the one developed by FrEAK. The need to consult a user guide to understand the workflow in that framework highlighted the importance of inherent usability in our design, aiming to create a system that is easy to use without requiring external user guide.

Consequently, the interface design incorporates features specifically intended to guide the user. For example, within the view dedicated to creating schedules, the process is broken down into explicit, numbered steps (1-6) (see figure 6), clearly indicating the sequence of actions required to complete the task and facilitating user comprehension. Furthermore, to enhance the framework's intuitiveness and informativeness, an "Info" section was included (see figure 6). This section serves the purpose of providing context-sensitive details based on user interaction. For instance, selecting a specific algorithm triggers the display of information regarding its operational principles for solving the problem and its typical initial configurations.

The second view within the framework serves to present a summary of the options selected in the main view (see figure 7), including the chosen dimension, search space, problem, algorithm, and stopping condition. Displaying these details provides the user with a clear overview of the current execution parameters. To enhance user control and interactivity during algorithm execution, controls such as Pause and Stop were implemented, allowing the user to temporarily pause or terminate an algorithm's run as needed. Upon completion of an algorithm's execution, navigation controls, such as a 'Next' button, become available, enabling the user to proceed to run subsequent configured schedules if multiple were created. Additionally, users have the capability to adjust the speed of the visualization to better observe the algorithmic process. For scenarios requiring modification or the creation of new schedules, a 'Main' button is provided to facilitate navigation back to the primary setup view.

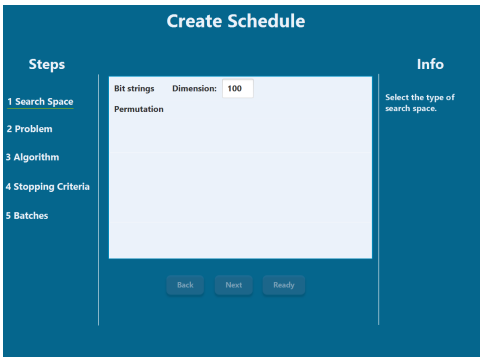

Figure 8: Final design of the main view

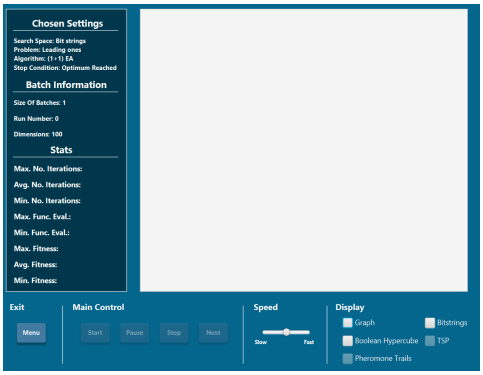

Figure 9: Final design of the second view

As implementation of the initial designs for the main and second views progressed, it became apparent that certain adjustments were necessary based on a deeper understanding of the interaction between user input and problem types, leading to design decisions. For instance, recognizing the distinction between the implemented search spaces bit strings and permutation and their differing requirements for dimensionality input, it was determined that requiring a dedicated 'dimension' step was illogical for permutation-based problems where the size is inherent to the problem instance (e.g., the number of cities in TSP instance). The design was therefore revised to link the dimension input directly to the selection of the search space type, displaying the user for a dimension input only when a bit strings search space is chosen (see figure 8).

Similarly, the initial design for the second view was found to be restrictive regarding the display of solutions, as it limited the user to viewing only one visualization output. To provide a more analytical experience, a decision was made to enhance this view by allowing users the flexibility to select and display multiple visualization options for the solution created by the algorithm (see figure 9).

4.2.2 Program Flow

With the design of the framework finalized, the program flow from the user's perspective is structured into a series of steps to guide the configuration process. The initial step (Step 1) involves selecting the search space for the optimization problem. Users are presented with two primary options to choose from: "Bit strings" or "Permutation". If the "Bit strings" option is selected, the user can choose the dimension of the search space, with a default value of 100 provided. This selection point is illustrated in Figure 8.

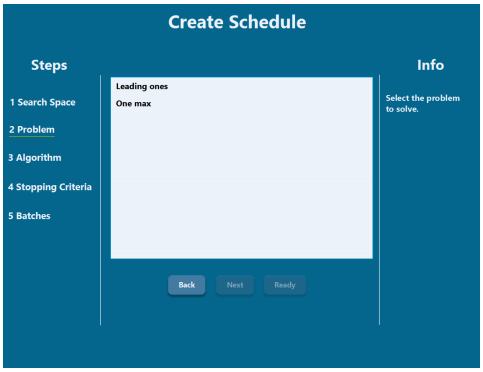

Figure 10: Step 2, when user selects "Bit strings" as search space

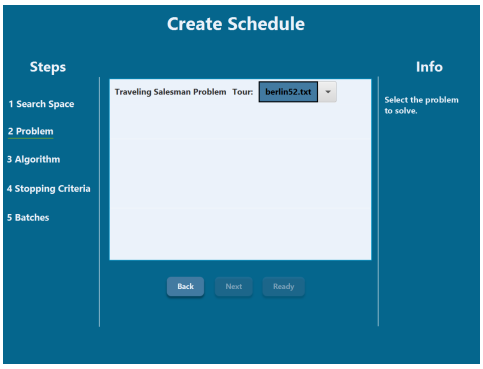

Figure 11: Step 2, when user selects "Permutation" as search space

Following the search space selection in Step 1, Step 2, involves selecting the specific optimization problem. The problems available to the user are contingent on the search space chosen. If "Bit strings" was selected in Step 1, the user can choose between "Leading ones" and "One max" as the problem for optimization (see Figure 10). However, if "Permutation" was chosen in Step 1, the 'Traveling Salesman Problem' is available for selection in Step 2 (see Figure 11). For the TSP, the interface allows the user to select among different tour instances from the TSPLib library [21], with "berlin 52" set as the default instance.

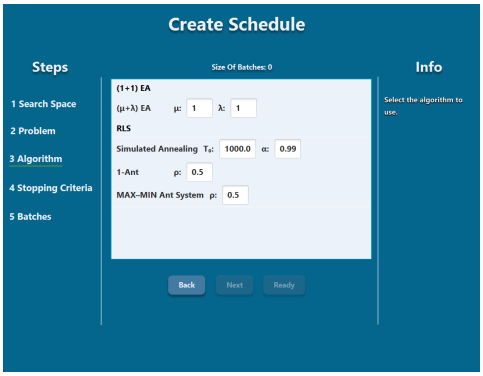

Figure 12: Step 3, algorithms available for "Bit strings" problems

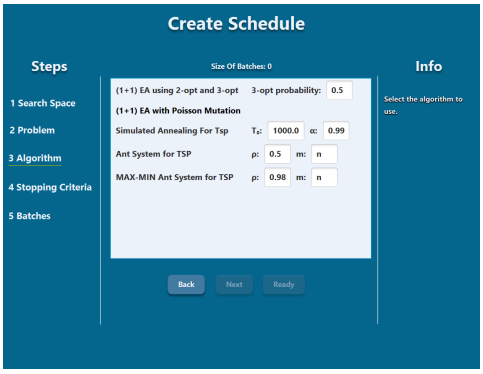

Figure 13: Step 3, Algorithms available for "Permutation" problem

Step 3 involves selecting the specific optimization algorithm to be applied to the chosen problem. As illustrated in Figures 12 and 13, the framework presents a selection of algorithms tailored to the chosen problem and search space; Figure 12 depicts the algorithms available for bitstring problems, while Figure 13 shows those for the TSP. As shown in Figure 12 and 13, the interface allows user to define specific parameter values for certain algorithms.

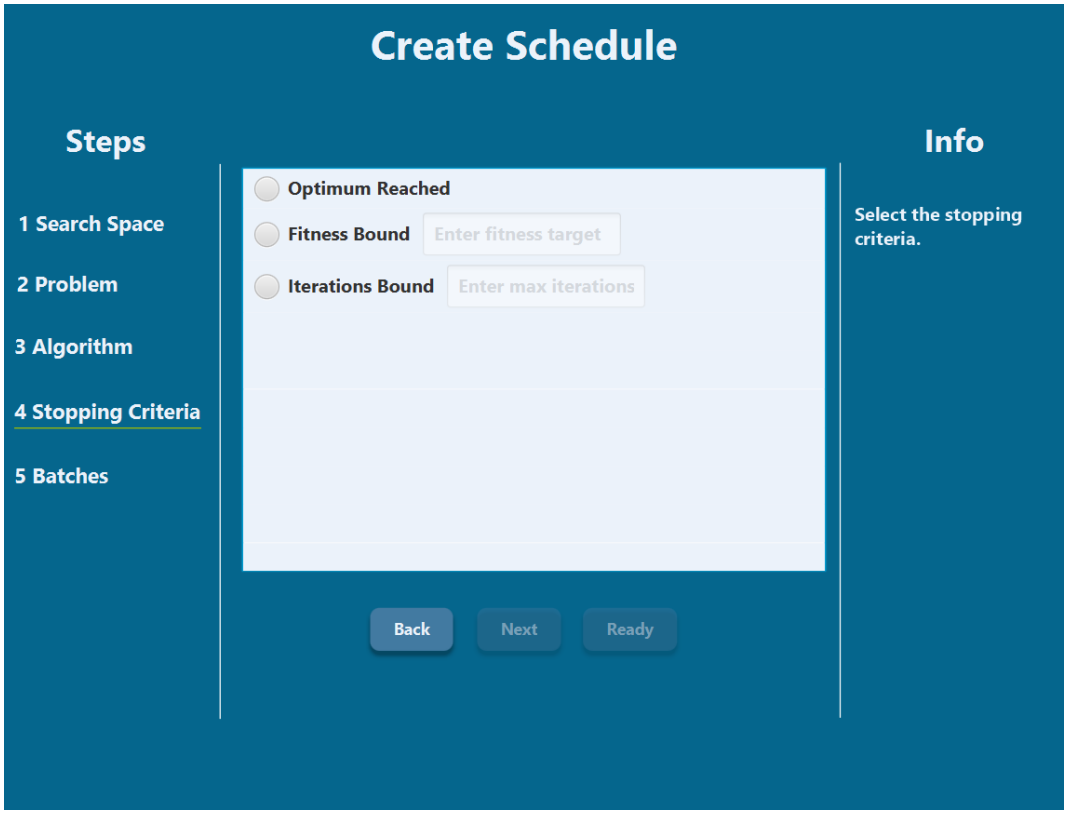

Figure 14: Step 4

Step 4 in the program flow involves selecting the stopping condition for the chosen algorithm. This step defines the criteria under which the algorithm's execution will terminate. As illustrated in Figure 14, users are presented with three primary options for stopping conditions:

- **Optimum Reached:** The algorithm will continue execution until an optimal or near-optimal solution is found.
- **Fitness Bound:** The algorithm terminates when the fitness of the best-so-far solution reaches or exceeds a specific value defined by the user.
- **Iterations Bound:** The algorithm's execution is terminated upon reaching a maximum number of iterations specified by the user.

This selection allows users to control the duration of the algorithm run based on the desired solution quality.

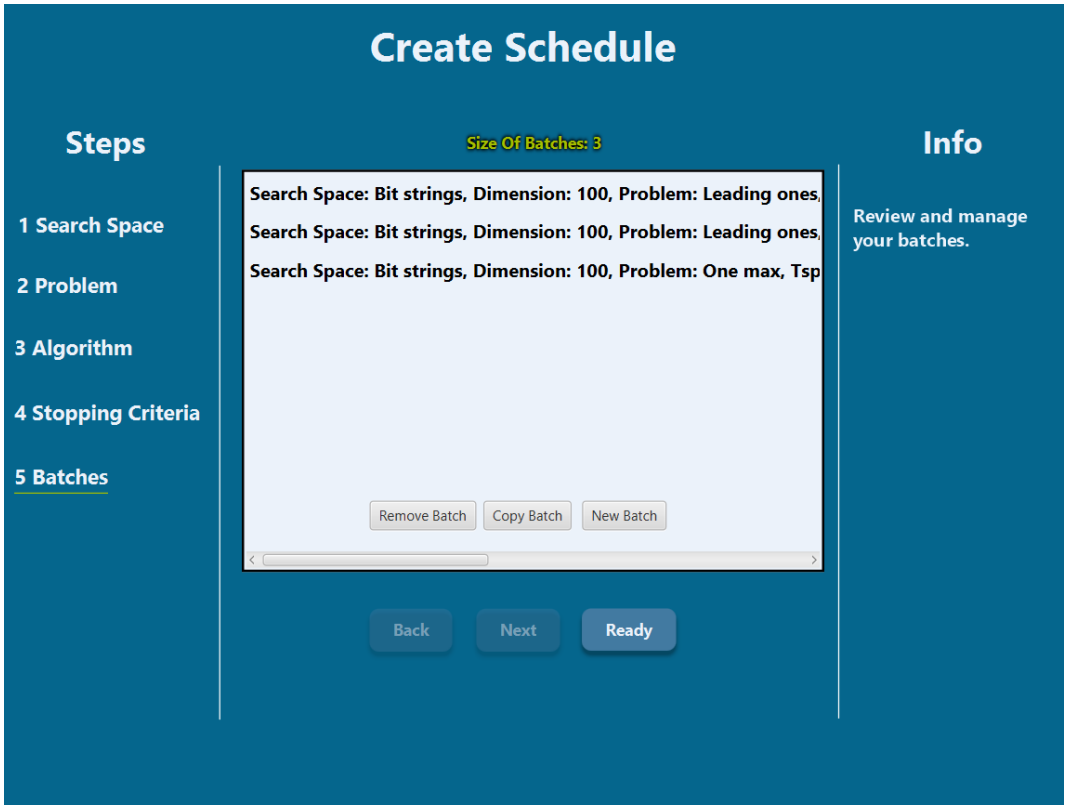

Figure 15: Step 5

Step 5 in the program flow involves managing the created batches, which represent distinct experimental configurations that the framework will execute sequentially. This stage allows users to review and modify the set of prepared configurations before initiating the algorithm runs. As illustrated in Figure 15, users are presented with a list displaying each created batch and its complete parameter set, including the search space type, dimension, selected problem, chosen algorithm, and defined stopping condition.

The interface provides several functionalities for managing these batches:

- **Remove Batch:** This function allows users to delete a selected batch configuration from the execution queue, providing flexibility to eliminate unwanted experimental setups.
- **Copy Batch:** This creates a duplicate of the most recently added batch configuration, enabling users to efficiently generate variations of similar experimental setups without needing to repeat the entire configuration process from scratch.
- **New Batch:** Selecting this option returns the user to Step 2 (Problem selection), facilitating the configuration of an additional batch. This design decision ensures that all batches created within a single session share the same search space type, which is important for maintaining consistency and accurately comparing results, particularly given the differences between maximizing problems (like those in bitstring search spaces) and minimizing problems (like TSP in the permutation search space).

This batch management system is designed to enable users to prepare multiple experimental configurations efficiently in a single session, streamlining the process for comparative analysis of algorithm

performance across different problems or parameter settings. Once satisfied with the defined batch configurations, users can click the "Ready" button to proceed to algorithm execution. At this point, the framework will process each batch sequentially and collect the corresponding performance data.

4.2.3 Display

Graph

Graph provides a real-time display of the optimization process by plotting fitness values against generations, offering users immediate insight into the overall optimization efficiency.

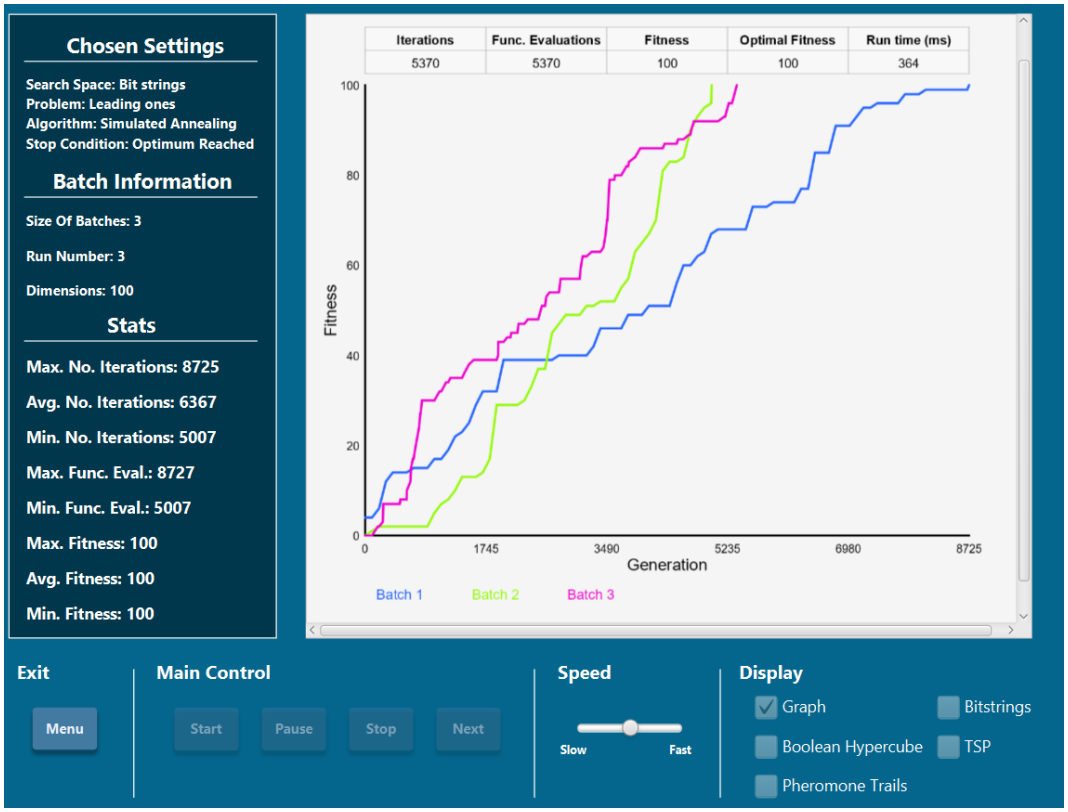

Figure 16: Graph visualization for binary problems

As illustrated in Figure 16, the visualization is structured around a coordinate system, with the x-axis representing generations and the y-axis representing fitness values. This representation allows users to observe the improvement of the solution over time. The graph incorporates automatic scaling capabilities to accommodate varying ranges of data, ensuring that both algorithms exhibiting rapid convergence over few iterations and those showing gradual improvement over many iterations are displayed with appropriate detail and clarity. A key feature implemented in the graph visualization is its support for comparing the performance of multiple batches simultaneously. Each distinct batch executed is rendered using a visually differentiated color. These colors are systematically generated, for instance, through a golden ratio-based algorithm, to ensure maximum visual distinction between different data series plotted on the graph. This color-coding system, coupled with a dedicated legend area below the graph, enables users to compare the performance curves of different algorithm configurations or problem instances. The legend displays batch identifiers and utilizes an adaptive layout system to manage space, even when numerous batches are simultaneously visualized. Finally,

the graph also incorporates a statistics table placed at the top. This table provides quantitative performance metrics for the algorithm currently being executed, including the current generation count, the number of function evaluations performed, the best fitness value achieved so far, the optimal fitness for the selected problem with the selected dimension, and the total runtime in milliseconds.

Bitstrings

Bitstrings visualization shows the initial bit strings at various generations, along with their corresponding fitness values. To enhance readability and facilitate pattern recognition, we have decided, that each displayed bit string is formatted with spaces inserted every 10 bits (see figure 17). For performance considerations, particularly in long-running simulations, the display is designed to update selectively. Instead of rendering every single iteration, updates occur at significant milestones (e.g., fixed generation intervals) or when notable changes in fitness value are detected, providing relevant progress updates without overwhelming the user.

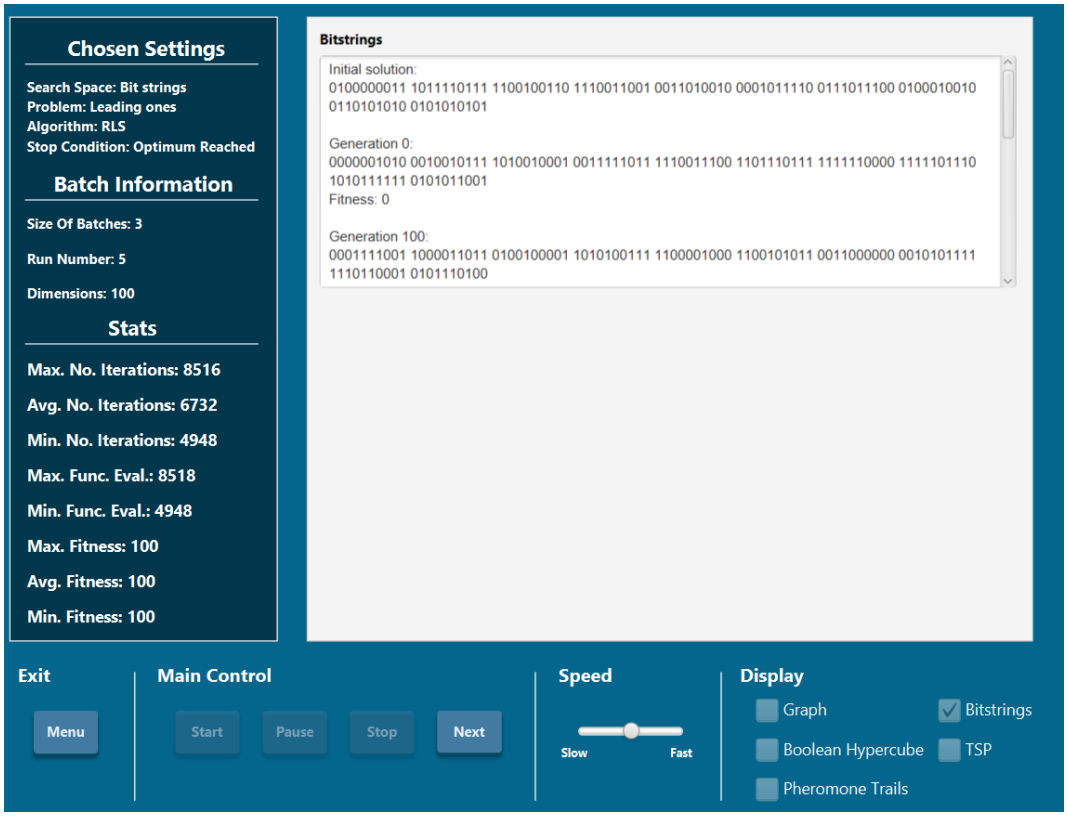

Figure 17: Bitstrings visualization for binary problems

Boolean Hypercube

The Boolean Hypercube visualization represents high-dimensional binary search spaces in a two-dimensional plane through mathematical mappings. This visualization aids in understanding the distribution of solutions and the trajectory of algorithms in binary search spaces. The visualization is illustrated in Figure 18. It maps bit strings to specific coordinates based on their structural properties. This mapping is crucial for projecting the n-dimensional binary space onto a 2D canvas. **Y-Coordinate Mapping:** The vertical position of a point is directly determined by the Hamming weight (count of '1' bits) of the corresponding bitstring. This value is normalized relative to the

bit string length n and linearly mapped to the y-axis of the visualization canvas. This results in a vertical positioning where solutions with no '1' bits appear towards the bottom of the visualization area, those with $\frac{n}{2}$ '1' bits in the middle, and solutions consisting entirely of '1' bits at the top.

X-Coordinate Mapping: The horizontal position aims to represent the distribution pattern of '1' bits within the string. For a bit string

$$x = (x_0, x_1, \ldots, x_{n-1}), \quad x_i \in \{0,1\},$$

the sum of the 0-indexed positions where '1' bits occur is calculated as

$$S = \sum_{i=0}^{n-1} i x_i$$

This sum provides a measure of how the '1' bits are distributed (e.g., clustered towards the beginning or end of the string). To position this sum within a consistent range for bit strings of the same length n and Hamming weight k, the sum S is normalized using minimum and maximum sums formulas inspired from FrEAK [30].

The minimum sum, $S_{\min}$, occurs when all k '1' bits are located at the k smallest 0-indexed positions $\{0,1,\ldots,k-1\}$, calculated as:

$$S_{\min} = \frac{k(k-1)}{2}$$

The maximum sum, $S_{\max}$, occurs when all k '1' bits are located at the k largest 0-indexed positions $\{n-k, n-k+1, \ldots, n-1\}$, calculated as:

$$S_{\max} = \frac{k(2n-k-1)}{2}$$

The normalized value derived from S, $S_{\min}$, and $S_{\max}$ is then transformed using a factor based on a Gaussian function to determine the final x-coordinate on the visualization plane.

Visual Structure: The visual structure of the Boolean Hypercube visualization is framed by curves derived from a Gaussian function. Specifically, the function used is of the form:

$$f(x) = e^{-x^2/8}.$$

This function is plotted and rotated to create curves at 90° and 270° angles, forming a symmetric, butterfly-like shape on the canvas. These resulting curves serve as visual guides, indicating the theoretical density distribution of bit strings in the projected search space based on their Hamming weight and the sum of indices of their '1' bits (see figure 18). Finally, solution quality is visually represented through a color mapping system based on fitness values. Each point is assigned a color along a spectrum from red (lower fitness) through green to blue (higher fitness). This creates an intuitive visual gradient where the optimization progress can be tracked by observing color shifts in the population. The optimal solution, when discovered, is highlighted with a distinctive red color and increased size, making it immediately identifiable within the search space.

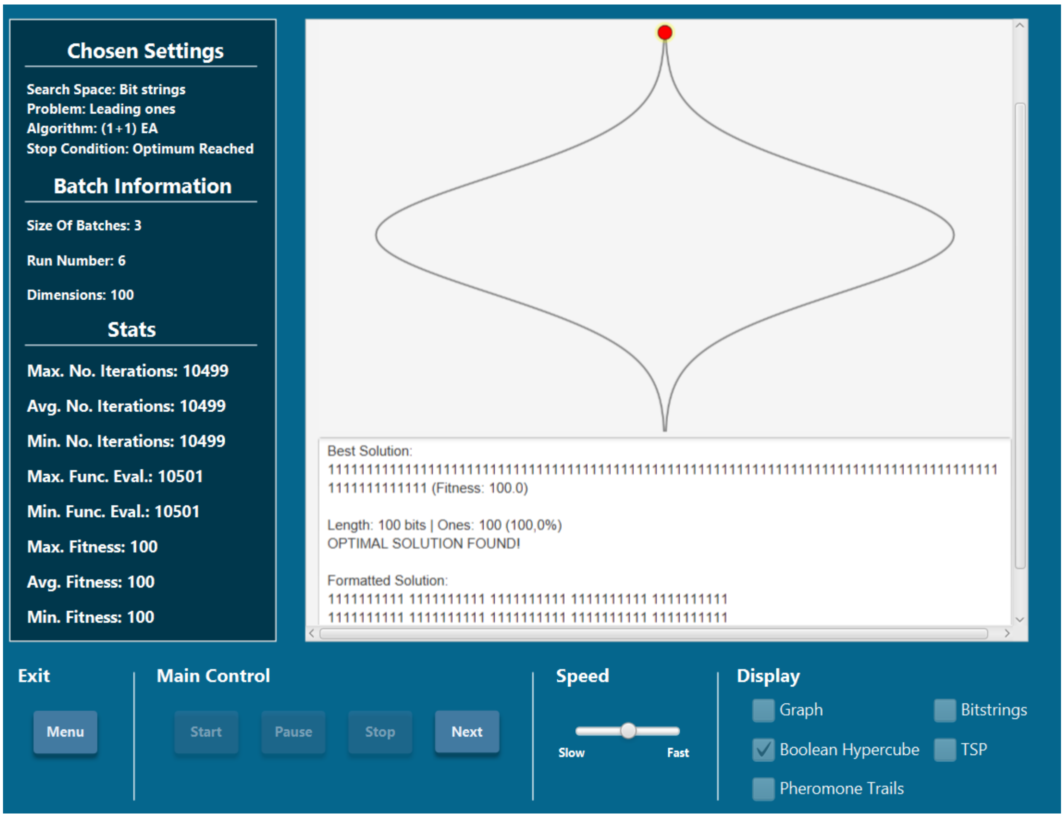

Figure 18: Boolean Hypercube visualization

TSP Visualization

The TSP visualization provides a representation of Traveling Salesman Problem instances, allowing users to observe both the geographic arrangement of cities and the quality of tour solutions. This visualization renders cities and tour paths on a two-dimensional coordinate plane, enabling real-time observation of optimization algorithms as they search for optimal tours.

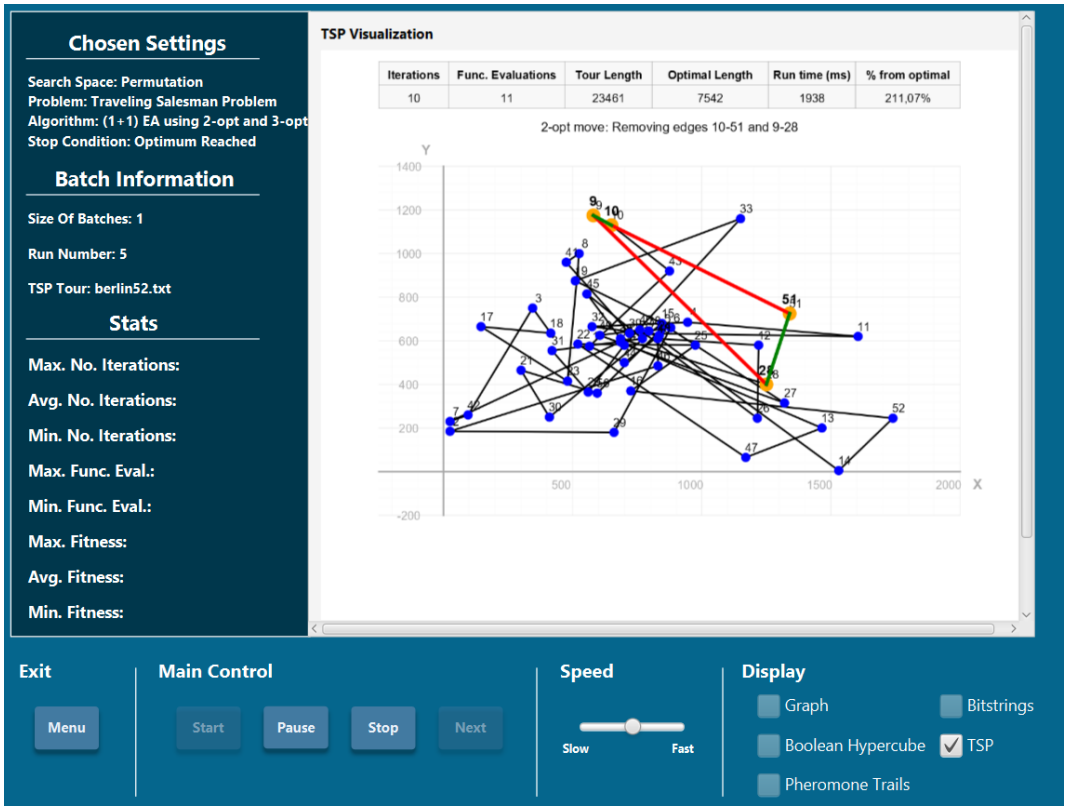

Figure 19: TSP visualization

As illustrated in Figure 19, the visualization employs an adaptive coordinate system that automatically scales to accommodate various TSP instance sizes, from small problems with dozens of cities to larger instances with hundreds of nodes. Cities are represented as circular markers with identifying labels, positioned according to their geographic coordinates. The current tour solution is depicted as a continuous path connecting all cities.

A key feature of the TSP visualization is its ability to illustrate the dynamic operation of optimization algorithms through animated transitions. When algorithms perform local search operations such as 2-opt or 3-opt edge exchanges, the visualization highlights the affected edges - removed connections are shown in red while newly added connections appear in green. This visual feedback helps users understand how incremental improvements modify the tour structure over time.

Complementing the graphical display, the visualization includes a statistical information panel. This panel presents key performance metrics for the algorithm's current run, such as the iteration count, the total number of function evaluations performed, the length of the current tour solution, the known optimal tour length for the selected instance, the total runtime in milliseconds, and the percentage deviation of the current solution's length from the optimal length.

Pheromone Trails

For ant colony optimization algorithms such as MAX-MIN Ant System (MMAS) and Ant System (AS), the visualization includes an optional pheromone trail display that renders the intensity of pheromones deposited on edges between cities. The pheromone trail visualization implements several techniques:

- **Color Gradient:** Pheromone concentrations are represented through a purple gradient, ranging from light purple for low pheromone values to dark purple for high concentrations.
- **Dynamic Line Width:** Line thickness scales proportionally with pheromone strength, varying from 0.5 to 2.5 pixels to emphasize stronger connections.
- **Legend:** A color gradient legend displays the range of the present pheromone values, showing both the minimum and maximum values with a precise numerical representation (e.g., from 0.000016 to 0.006627) as indicated in figure 20.

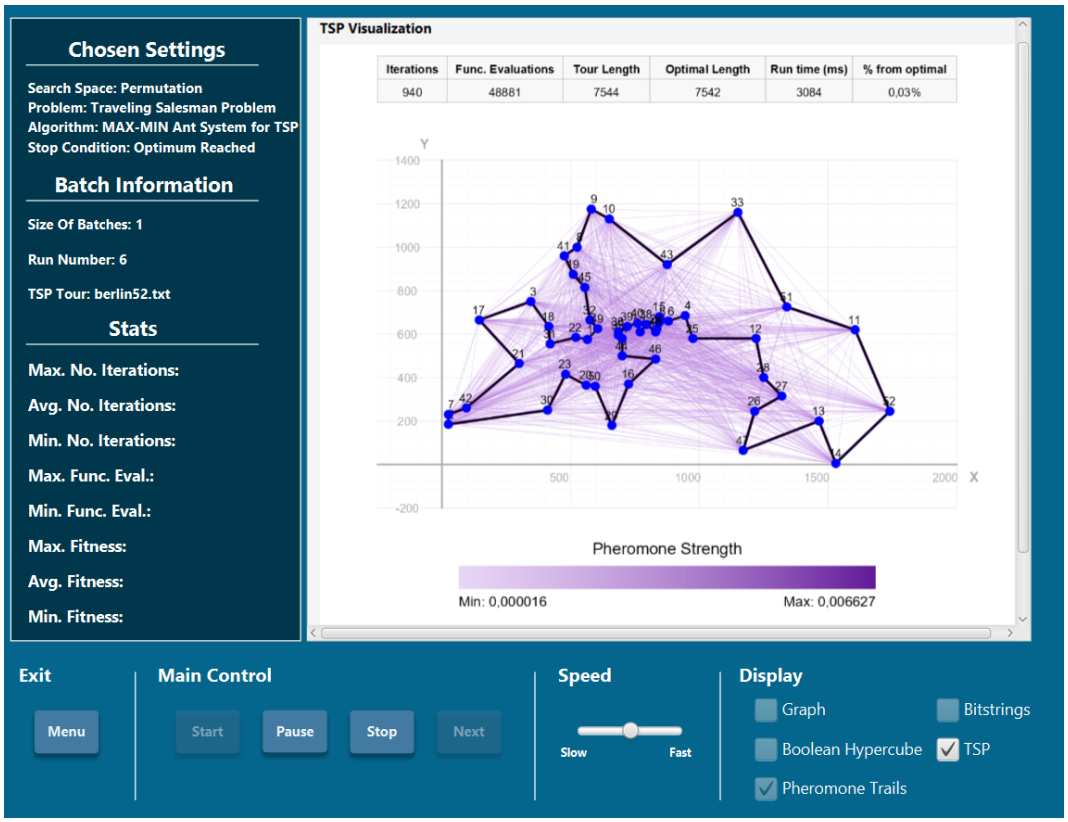

Figure 20: Pheromone Tails visualization

This visualization enables observation of how collective intelligence emerges through the reinforcement of high-quality paths. Users can track the evolution of pheromone distributions, identifying which edges gain strength over successive iterations and how the algorithm gradually concentrates search efforts on promising regions of the solution space.

4.3 Testing

While acknowledging the significant advantages typically offered by automated testing, such as enhanced efficiency and robustness in software verification, the evaluation of this framework's functionality and implemented algorithms was conducted exclusively through manual testing. The decision to rely solely on manual testing stemmed from the challenges associated with developing automated test cases for the stochastic nature of the nature-inspired metaheuristics within the project's scope. The testing process involved systematically examining each feature and implemented compo-

nent of the framework. This manual verification was integrated and conducted continuously throughout the project's development lifecycle.

5 Implementation

This section details the technical implementation of the framework, outlining the program flow from the user's perspective starting the application through configuring and executing algorithm batches and extracting key performance figures. Building upon the program flow discussed in Section (4.2.2). We will explain how the framework's classes coordinate with each other to manage user input, organize algorithm runs, update visualizations, and collect experimental data. This will be supported by relevant code snippets presented in blocks. Longer or more detailed code examples will be documented and referred to in the appendix section for review if needed.

5.1 Wizard-Based Schedule Creation Architecture

The process of creating a schedule is managed by the MainInteraction class within the interactions package. MainInteraction serves as the central controller for this wizard-style schedule creation process, coordinating interactions with several supporting classes, including AppState, and other classes such as InfoTextProvider and RadioButtonListCell from the "other" package. The implementation focuses on dynamically updating the user interface based on user selections, capturing input, structuring experimental configurations into batches, and managing these batches before execution. In the next sub sections we will be covering how MainInteraction interacting with the other classes.

5.1.1 MainInteraction and AppState Integration

The MainInteraction class directly interfaces with the AppState class to maintain consistent configuration state:

Block 1: MainInteraction.java

```
private List<List<String>> allBatches = AppState.allBatches;

// When creating a new batch
allBatches.add(new ArrayList<>(selectedOptions));
selectedOptions.clear();
```

As illustrated in block 1, all defined batches are stored in the shared collection AppState.allBatches. The MainInteraction class maintains a local reference to this collection and directly manipulates it when adding new batches or modifying existing ones.

Parameters specific algorithms, such as population sizes like $(\mu+\lambda)EA$ for Evolutionary Algorithms or evaporation rate ρ for Ant Colony Optimization, are captured from UI input fields and stored in the class AppState using the method "setAlgorithmParameter". This ensures these parameters are accessible to the algorithm execution logic. Examples include storing parameters for μ and λ belonging to the algorithm $(\mu+\lambda)EA$ and ρ for MAX-MIN Ant System (see block 2).

Block 2: MainInteraction.java

```
// For parameters of (μ+λ) EA
muField.textProperty().addListener((obs, oldVal, newVal) -> {
    if (newVal != null) {
        AppState.setAlgorithmParameter("MuValue", newVal);
    }
});

// For MAX-MIN Ant System
rhoField.textProperty().addListener((obs, oldVal, newVal) -> {
    if (newVal != null) {
        AppState.setAlgorithmParameter("RhoValue", newVal);
    }
});
```

When the user starts the creation of batches within the same session, MainInteraction queries AppState to retrieve and preserve certain settings, such as the selected search space. In other words, if the user created the first batch with a specific search space type, the other followed-created batches will have the same search space. This decision was made to ensure consistency across batches created sequentially (see block 3).

Block 3: MainInteraction.java

```
// When creating a new batch
if (preservedSearchSpace != null) {
    selectedOptions.add(preservedSearchSpace);
    AppState.chosenSettings.removeIf(s -> s.startsWith("Search Space:"));
    AppState.chosenSettings.add(preservedSearchSpace);
}
```

5.1.2 MainInteraction and InfoTextProvider Integration

The InfoTextProvider class is integrated with MainInteraction to supply contextual information about each option presented in the user interface. This allows the framework to dynamically update a dedicated information display area with relevant details as the user navigates through the configuration steps and selects options. It works by MainInteraction maintains a mapping between the string representations of UI options (e.g., "Bit strings", "Permutation", and "Travelling Salesman Problem") and corresponding information text identifiers (Key). See block 4.

Block 4

```java
// Mapping between list view options and InfoTextProvider keys
private final java.util.Map<String, String> optionToInfoKey = java.util.Map.ofEntries(
    java.util.Map.entry(k:"Bit strings", v:"BitStrings"),
    java.util.Map.entry(k:"Permutation", v:"Permutation"),
    java.util.Map.entry(k:"Traveling Salesman Problem", v:"TSP"));
    //More entries...
```

MainInteraction.java

```java
public class InfoTextProvider {
    private static final Map<String, String> infoTexts = new HashMap<>();

    static {
        // Search Spaces
        infoTexts.put(key:"BitStrings",
            "Bit strings are sequences of 0s and 1s that represent potential solutions. " +
            "Each position in the string is called a bit and can be either 0 or 1. " +
            "The number of bits is determined by the dimension value you selected earlier.");

        infoTexts.put(key:"Permutation",
        "Permutation search spaces consist of ordered arrangements of elements. " +
        "For optimization problems like TSP, a permutation represents a specific tour " +
        "through all cities. The algorithms search for the optimal ordering of elements."
        );
```

InfoTextProvider.java

As shown in block 4, when a user selects an option, the corresponding Key is used to retrieve explanatory text from InfoTextProvider, which is then displayed in the UI.

5.1.3 RadioButtonListCell for Stopping Conditions

The RadioButtonListCell class represents a custom UI component used within the optionsListView, specifically for handling the selection of stopping conditions in Step 4. This custom cell type is necessary to integrate radio button selection logic directly within the list view and to manage associated input fields for bound values. In Step 4 of the configuration process, MainInteraction configures the optionsListView to use instances of RadioButtonListCell as its cell factory. This ensures that each option in the list for this step is rendered as a radio button. See block 5:

Block 5: MainInteraction.java

```java
if (step == 4) { // Stopping condition
    // Clear the existing radio group selection before setting up new cells
    radioGroup.selectToggle(null);
    optionsListView.setCellFactory(param -> new RadioButtonListCell(radioGroup));

    // Force update of the next button state after clearing radioGroup
    updateNextButtonState();
}
```

Bound Value Management: For stopping conditions that require a boundary value (e.g., Fitness Bound, and Iterations Bound), RadioButtonListCell includes input fields. MainInteraction interacts with RadioButtonListCell to retrieve and validate these bound values before adding the stopping condition to the batch configuration. See block 6:

Block 6: MainInteraction.java

```
if (selectedText.equals("Fitness Bound")) {
    int boundValue = RadioButtonListCell.getFitnessBoundValue();
    if (boundValue <= 0) {
        errorLabel.setText("Please enter a valid fitness bound value");
        return; // Don't proceed without valid value
    }
    selectedOptions.add("Stop Condition: Fitness Bound:" + boundValue);
}
```

Logic for fitness input inside the MainInteraction

In addition to that, mainInteraction manages a ToggleGroup instance, which is passed to each RadioButtonListCell. This ensures that only one radio button within the stopping condition list can be selected. The purpose of this decision is to prevent users from creating multiple stopping conditions for a batch. See block 7:

Block 7: MainInteraction.java

```
// MainInteraction.java line 65
private ToggleGroup radioGroup = new ToggleGroup();

// In initialize method (line 130)
radioGroup.selectedToggleProperty().addListener((observable, oldToggle, newToggle) -> {
    if (newToggle != null && currentStep == 5) {
        RadioButton selectedRadioButton = (RadioButton) newToggle;
        String selectedText = selectedRadioButton.getText();
        // Process selection...
    }
});
```

5.1.4 Dynamic UI Adaptation Mechanism

A core feature of the MainInteraction class is the dynamic management of options presented to the user based on the current step and previous selections. The getOptionsForStep method is central to this, returning a list of strings that populate the optionsListView. This method incorporates logic to filter options and display them to users. This can be seen here (10.2).

This dynamic management of options, done by the method getOptionsForStep within MainInteraction (10.2), is fundamental to providing a guided and context-aware user experience during the schedule creation process, ensuring users are presented with relevant choices based on their selections.

5.2 ScheduleInteraction and its supporting classes

After configuration and selections of different options is completed through the ManinInteraction class, the ScheduleInteraction class serves as the central execution controller for the framework. Where it takes over from the MainInteraction class and becomes responsible for executing the configured batches, visualizing optimization progress, and collecting performance statistics. In other words, Unlike MainInteraction, where focus is configuration, ScheduleInteraction is dedicated to the dynamic execution of algorithms and the visualization of their performance in real-time.

5.2.1 Class Integrations

As shown below in block 8, ScheduleInteraction starts with integrating the Appstate class to retrieve and process batch configurations settings created during the previous phase.

Block 8: ScheduleInteraction.java

```
private void initialize() {
    // Load first batch settings
    if (!AppState.allBatches.isEmpty()) {
        List<String> firstBatch = AppState.allBatches.get(0);
        AppState.chosenSettings.clear();
        AppState.chosenSettings.addAll(firstBatch);

        // Get and set initial dimension
        String dimStr = firstBatch.stream()
            .filter(s -> s.startsWith("Dimension:"))
            .findFirst()
            .map(s -> s.substring("Dimension:".length()).trim())
            .orElse("N/A");
        AppState.setChosenDimension(dimStr);
    }
}
```

In addition to the Appstate class, ScheduleInteraction integrates with other classes, each responsible for a specific concern within the execution process. This class-based architecture facilitates the framework's ability to support diverse algorithms and problems while maintaining a consistent execution model. The primary integrated classes are AlgorithmFactory, ProblemFactory, Visualization classes, SearchSpace classes, and StoppingCondition classes. In the following subsections, we will explain how thses classes are integrated into SchdueleInteraction, and what their functionalities are.

5.2.2 AlgorithmFactory

AlgorithmFactory implements a instantiation mechanism that allows ScheduleInteraction to create algorithm instances without needing to understand their internal implementation details. See block 9.

Block 9: ScheduleInteraction.java

```
algorithm = AlgorithmFactory.createAlgorithm(
    algorithmChoice, finalSearchSpace, finalProblem, stoppingCondition);
```

When ScheduleInteraction creates an algorithm instance as shown in block 9, what happens is, the factory first analyzes the algorithm choice string to determine which concrete algorithm class to instantiate, for instance the string "$(1+1)$ EA" creates an instance of the OnePlusOneEA class, while "$(\mu+\lambda)$ EA" instantiates a MuPlusLambdaEA with appropriate parameters. The factory then queries AppState for algorithm-specific parameters that were captured during configuration in MainInteraction. Parameters like μ and λ values for evolutionary algorithms or evaporation rate ρ for ant colony algorithms are retrieved and applied to the newly created algorithm instance.

5.2.3 ProblemFactory

ProblemFactory handles the instantiation of optimization problem classes that define the fitness functions and evaluation logic. ScheduleInteraction determines whether the selected problem is bit string-based (like OneMax or LeadingOnes) or permutation-based (like TSP). See block 10.

Block 10: ScheduleInteraction.java

```
String problemChoice = AppState.getChosenSetting("Problem");
try {
    problem = ProblemFactory.createProblem(problemChoice);
} catch (IllegalArgumentException e) {
    problem = ProblemFactory.createProblem("one max"); // Default fallback
}
```

For bit string problems, initialization is straightforward with only dimension as a parameter. For TSP problems, however, ScheduleInteraction handles the loading of instance files with city coordinates. It extracts the filename from batch settings and passes it to the TspProblem class. ScheduleInteraction also uses a function "getOptimalTourLengthForTspFile" of known optimal tour lengths for common benchmark instances, the method can be seen here (10.3). When an optimal tour length is known, this information is provided to visualization classes to display.

5.2.4 Visualization Classes

ScheduleInteraction manages a visualization mechanism using multiple Visualization classes, each providing a different perspective on the optimization process. During initialization, it creates and configures these visualization components. See block 11.

Block 11: ScheduleInteraction.java

```
graphVisualizer = new GraphVisualization(canvas, canvasWidth, 538, padding);
hypercubeVisualizer = new HypercubeVisualization(hypercubeCanvas, canvasWidth,
                                                 canvasHeight, padding, hypercubeInfoTextArea);
tspVisualization = new TspVisualization(tspCanvas);
```

The visualization mechanism operates on an event-driven model, where algorithm events trigger appropriate visualization updates. When an algorithm emits progress information through the listener interface, ScheduleInteraction processes this information and dispatches it to the appropriate visualization classes. To maintain UI responsiveness during high-speed algorithm execution. Instead of updating visualizations for every algorithm event (which could occur thousands of times per second and slow the framework), we decided that updates are triggered periodically (every 100 generations), or on significant events like finding a new optimum.

Users can toggle different visualizations on and off during execution through a checkbox system. ScheduleInteraction also enables only those visualizations that make sense for the current problem domain for example hypercube visualization is only enabled for bit string problems, while TSP visualization is only available for permutation-based problems. This context-aware visualization system ensures that users are presented with relevant visualization options without being overwhelmed by many choices for the current problem domain.

The visualization classes are designed to be independent and reusable. For example The GraphVisualization class can plot fitness progress for any algorithm type, while specialized classes like HypercubeVisualization and TspVisualization provide domain-specific insights.

5.2.5 SearchSpace Classes

The SearchSpace classes forms the foundation of algorithm operations, defining how solutions are represented and manipulated. ScheduleInteraction creates the appropriate SearchSpace implementation based on the problem domain. See block 12.

Block 12: ScheduleInteraction.java

```
if (isTspProblem) {
    searchSpace = new Permutation(tspProblem.getTspParser().getDimension());
} else {
    searchSpace = new BitStrings(dim);
}
```

For bit string problems, the dimension is a user-configured value representing the length of the bit string. For TSP problems, the dimension is extracted from the problem instance file, representing the number of cities.

The search space type also influences how optimality is detected. For bit string problems with OneMax or LeadingOnes, the optimal solution has a fitness equal to the dimension, making optimality detection straightforward. For TSP, optimality is hard to reach, so a stopping condition like NoImprovementFound is used instead.

5.2.6 StoppingCondition Classes

The StoppingCondition classes defines when algorithm execution should terminate. ScheduleInteraction creates and configures the appropriate StoppingCondition based on user selection and problem characteristics. See block 13.

Block 13: ScheduleInteraction.java

```
if (stoppingConditionChoice.contains("Optimum Reached")) {
    chosenStoppingCondition = new OptimalFitnessReached();
}
else if (stoppingConditionChoice.contains("Fitness Bound")) {
    chosenStoppingCondition = new FitnessBoundReached(boundValue, isTspProblem);
}
else if (stoppingConditionChoice.contains("Iterations Bound")) {
    chosenStoppingCondition = new IterationsBoundReached(boundValue);
}
```

Some stopping conditions require additional parameters, such as bound values. ScheduleInteraction extracts these from the user selection string, parsing numerical values from formats like "Fitness Bound:100". If parsing fails, the framework falls back to default values stored in the RadioButtonListCell class, ensuring that execution can proceed even if user input is incomplete.

ScheduleInteraction adapts the stopping condition based on whether the problem is a maximization

problem (like OneMax and LeadingOnes) or a minimization problem (like TSP). For maximization problems, the goal is to reach fitness values above the bound, while for minimization problems, the goal is to reach values below the bound. This distinction is communicated to the FitnessBoundReached condition through a boolean parameter. This can be seen here (10.4).

ScheduleInteraction implements special handling. For example, when "Optimum Reached" is selected for TSP problems, the framework automatically substitutes a NoImprovementFound condition with a threshold (10,000 iterations) since optimality may not be reachable. This substitution happens transparently to the user, ensuring that the algorithm will eventually terminate. See block 14.

Block 14: ScheduleInteraction.java

```
// First check if this is a TSP problem with Optimum Reached (special case)
if (isTspProblem && stoppingConditionChoice != null && stoppingConditionChoice.contains(s:"Optimum Reached")) {
    // For TSP problems, use NoImprovementFound instead of Optimum Reached
    int noImprovementThreshold = 9999; // Stop after 10,000 iterations without improvement
    chosenStoppingCondition = new NoImprovementFound(noImprovementThreshold);
    System.out.println("TSP problem with Optimum Reached - using NoImprovementFound: " +
                       noImprovementThreshold + " iterations without improvement");
}
```

Lastly, during algorithm execution, ScheduleInteraction continually checks whether the stopping condition has been met by calling the condition's isMet method with the current fitness, dimension, and generation count. When the condition is satisfied, ScheduleInteraction records the final statistics and prepares for the next batch or concludes the experiment if all batches have been processed.

5.3 Algorithm Implementations

This section details the implementation of the algorithms discussed in Section 2, with each implementation adhering to the pseudocode specifications outlined therein. All algorithm implementations adhere to the Algo interface, which is responsible for defining the core functionalities utilized by these algorithms. See block 15.

Block 15: Algo.java

```
public interface Algo {
    void initialize();
    void runAlgorithm();
    void setListener(Consumer<RunInfo> listener);
}
```

The initialize() method is responsible for preparing the algorithm's initial state. The runAlgorithm() method encapsulates the primary optimization loop, where the core computational work is performed. Furthermore, the setListener() is responsible to provide real-time feedback to the visualization classes during execution.

We will not be providing a detailed description of every implemented algorithm. As the fundamental implementation of each follows the pseudocode presented in Section 2, the focus will instead be placed on algorithms where notable implementation details or design choices were necessary to

ensure robust operation.

During the implementation, certain TSP algorithms, notably the (1+1) EA using Poisson-distributed Mutation (with 2-opt) and Simulated Annealing for TSP, were observed under testing to show oscillatory behavior when reaching near optimal fitness. This issue was apparent through the visualization of the framework, which highlights edge exchanges during 2-opt operations. When algorithms entered an oscillatory state, the repeated visualization of the same edge swaps created the impression of the framework's execution being stalled or lagging. As these algorithms often reached near-optimal solutions relatively early in the search process approximately after 45 iterations, and given the implementation of a specialized 'NoImprovementFound' stopping condition for TSP algorithms with a threshold of 10,000 iterations, users would observe the algorithm repeatedly attempting the same swapping cities for an extended period from approximately iteration 45 up to the 10,000-iteration limit, leading to a problematic user experience. To address this issue, we decided to implement a further mechanism within these algorithms to detect oscillatory behavior and terminate execution in some cases because further progress towards the optimal fitness is unlikely.

5.3.1 Mechanisms to escape oscillation

The mechanism implemented in the (1+1) EA with Poisson-distributed mutation tracks a history of edge swaps during the algorithm's execution. It detects when the same edge swaps occur repeatedly in a cyclical pattern, which indicates that the algorithm is trapped in a local optimum. If this pattern is observed more than MAX_SAME_EDGE_SWAPS times (set to 10), the algorithm concludes that it is oscillating and terminates the search process. At this point, the algorithm returns the best solution found. See block 16.

Block 16: OnePlusOneEA_Poisson_TSP.java

```
// Check for oscillation by looking at the edges that were swapped
String currentEdgeSwap = createEdgeSwapString(offspring);
if (!currentEdgeSwap.isEmpty()) {
    if (currentEdgeSwap.equals(previousEdgeSwap)) {
        // Same swap as two iterations ago - possible oscillation
        sameEdgeSwapCount++;

        if (sameEdgeSwapCount >= MAX_SAME_EDGE_SWAPS) {
            System.out.println("Algorithm appears stuck in oscillation after " +
                               generation + " generations. Ending search.");
            stoppingConditionMet = true;
            break;
        }
    }
    // Update edge swap history
    previousEdgeSwap = lastEdgeSwap;
    lastEdgeSwap = currentEdgeSwap;
}
```

The mechanism implemented in Simulated Annealing takes a different approach, focusing on avoiding oscillations to allow the algorithm to terminate smoothly. To prevent repeating the same move, the algorithm attempts up to three different 2-opt moves until it finds one that differs from the previous iteration. If all three attempts result in the same move, the algorithm proceeds but then checks whether the resulting tour length is identical to the previous one. If it is, the evaluation is skipped,

and the algorithm tries again in the next iteration. The code implementation of the mechanism can be seen here 10.5.

5.4 Graphics

The framework was built using JavaFX. We decided to choose JavaFX because it offers several key features that make it well-suited for showing how algorithms work in real-time. JavaFX provides strong drawing capabilities, allowing the creation of custom visualizations components such as fitness graphs, hypercube representations, and animated TSP tours. These visualizations update smoothly, even when algorithms run quickly and generate large amounts of data points. See Figure 21, which illustrates the visualization of two million generations plotted along the x-axis, rendered within a 20-second timeframe.

A major advantage of JavaFX is its ability to handle multiple tasks at the same time, which is important for our implementation. The algorithms implemented in this framework require a lot of computing power, and if not managed well, they can make the interface unresponsive. JavaFX solves this by running these algorithms in background threads, keeping the interface responsive and updating visualizations smoothly without delay.

To simplify building the interface, Scene Builder was used alongside JavaFX. Scene Builder is a visual tool where we design screens by dragging and dropping elements, instead of writing code for each part. This saved considerable time and made it easier to try different layouts. Scene Builder creates FXML files that separate how the interface looks from how it works. This separation means we can change the screen layout without changing the code that handles the logic for algorithm visualizations. This became very valuable as we added more visualization features.

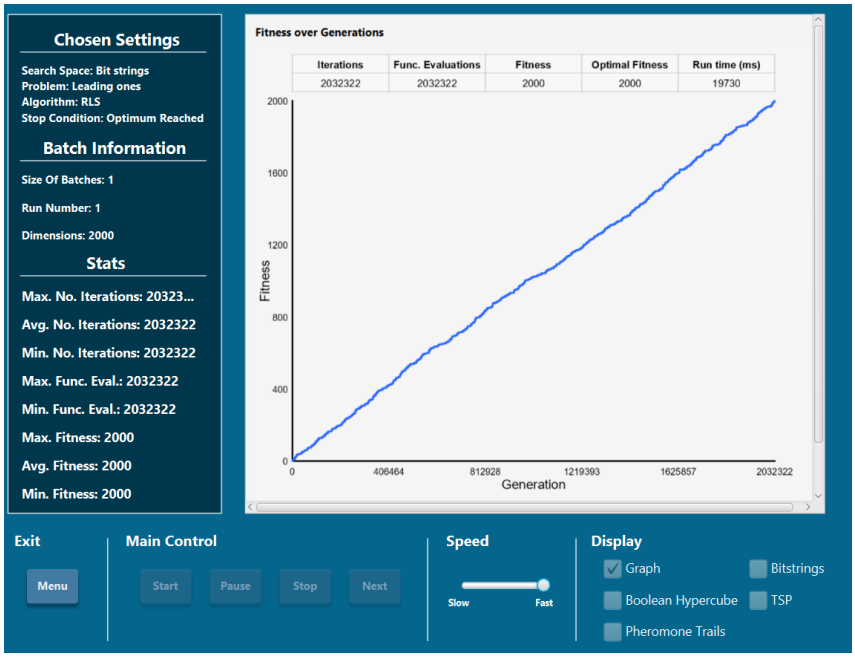

Figure 21: Illustrating two millions of generation in x-axis done in 20 seconds.

6 Evaluation and Results

This section evaluates the performance of the algorithms discussed earlier through a series of experiments. Each experiment tests different parameter settings to observe how they affect the behavior and outcomes of the algorithms. Once the best-performing configuration for each algorithm is identified, a final comparison is made to assess their relative strengths. The experiments are conducted on three benchmark problems: OneMax, LeadingOnes, and the Traveling Salesman Problem (TSP).

For the OneMax problem, experiments measure the average number of iterations required for each algorithm to find the optimum across bit string lengths of 500, 1000, 1500, 2000, 2500, and 3000. For each problem size, 100 runs are conducted to calculate the average optimization time.

Similarly, for the LeadingOnes problem, experiments are conducted by measuring the average number of iterations required for each algorithm to find the optimum across bit string lengths of 100, 200, 400, 600, 800, and 1000. For each problem size, 100 runs are conducted to calculate the average optimization time.

Finally, for the Traveling Salesman Problem (TSP), the measurement of computational cost requires a more nuanced approach. While iterations provide a general sense of algorithmic progress, the primary measure of cost, particularly from a theoretical perspective, is the number of function evaluations. This is because function evaluations are typically the most computationally expensive operation. For algorithms like the $(1+1)$ EA, RLS or Simulated Annealing (SA), a single function evaluation often occurs per iteration, making iterations and function evaluations largely equivalent. However, for Ant Colony Optimization (ACO) algorithms, such as MMAS and AS each ant performs a function evaluation for the tour it constructs within a single iteration. This means that ACO algorithms will make multiple function evaluations per iteration, making a direct comparison based solely on iterations potentially misleading. Therefore, when analyzing the optimization time for TSP, especially concerning ACO algorithms, we will primarily focus on the total number of function evaluations.

6.1 Evolutionary Algorithms and Randomized Local Search

6.1.1 OneMax

We begin by examining the optimization time of the (1+1) Evolutionary Algorithm (EA) on the OneMax problem, using a mutation probability of $p = \frac{1}{n}$. According to Witt 2.3.1 the expected optimization time is given by $e\, n\ln(n) + O(n)$. For the purpose of comparison with empirical results, we disregard the lower-order terms and focus on the leading term $e\, n\ln(n)$, which is plotted as a reference line. To provide a baseline, we also evaluate Randomized Local Search (RLS) on OneMax. As discussed in the background section 2.4.1, RLS is expected to outperform the $(1+1)$EA by approximately a factor of e. The graph below illustrates the average optimization time as the number of iterations required to reach the optimum for both the $(1+1)$ EA and RLS. The theoretical expectations for both algorithms are included as reference lines.

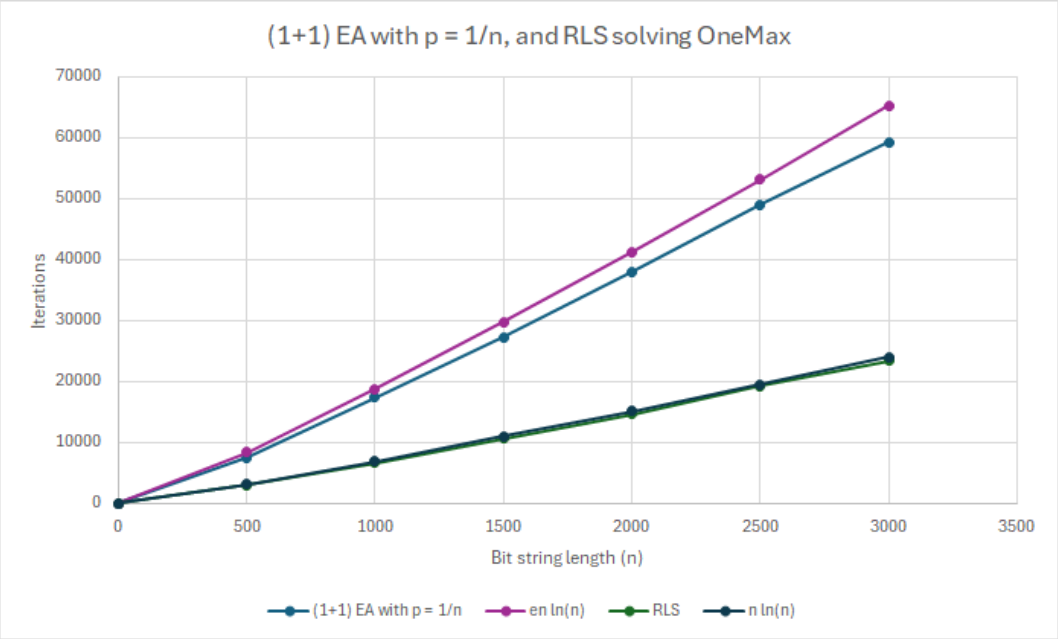

Figure 22: Averaged over 100 runs.

The graph confirms the theoretical expectations for both algorithms. The $(1+1)$ EA with $p = \frac{1}{n}$ closely follows the $en\ln(n)$ reference line, indicating its expected performance on the OneMax problem. Similarly, the RLS algorithm's performance aligns well with its theoretical expectation of $n\ln(n)$. As predicted, RLS outperforms the $(1+1)$ EA, demonstrating a lower average number of iterations to reach the optimum across all tested bit string lengths.

6.1.2 LeadingOnes

Building on our discussion in the background section 2.3.1 regarding the impact of mutation probability on optimization time, this experiment investigates the performance of the $(1+1)$ EA on LeadingOnes with two different mutation probabilities: the standard $p = \frac{1}{n}$ and an optimized $p = \frac{1.59}{n}$. As shown by Böttcher, Doerr, and Neumann, the optimized mutation probability is theoretically expected to yield a faster optimization time. We will compare our empirical results against the known expected optimization times for both configurations: $0.86n^2$ for $p = \frac{1}{n}$ and $0.77n^2$ for $p = \frac{1.59}{n}$.

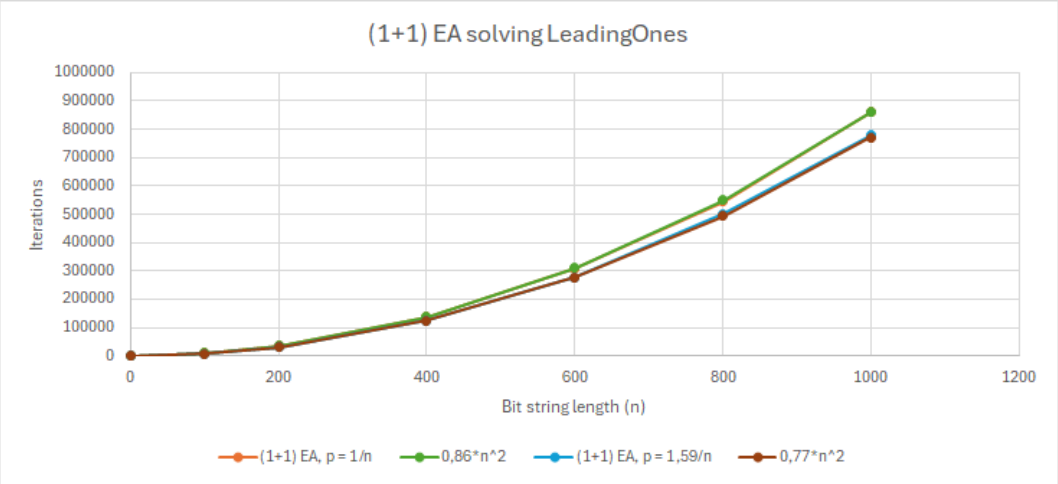

Figure 23: Averaged over 100 runs.

The graph illustrates the impact of the mutation probability on the optimization time of the $(1+1)$ EA for the LeadingOnes problem. As predicted by Böttcher, Doerr, and Neumann, the empirical results show that using a mutation probability of $p = \frac{1.59}{n}$ leads to a faster optimization time compared to the standard $p = \frac{1}{n}$, particularly as the bit string length n increases. Both empirical curves closely track their respective theoretical expectations, $0.86n^2$ for $p = \frac{1}{n}$ and $0.77n^2$ for $p = \frac{1.59}{n}$, confirming the accuracy of these theoretical analyses.

Our next comparison focuses on the performance of RLS and the $(1+1)$ EA with the optimized mutation probability $p = \frac{1.59}{n}$ when solving the LeadingOnes problem. As we discussed here 2.4.2, the theoretical expected optimization time for RLS on LeadingOnes is given by $\frac{n^2}{2} + O(n^2)$. In our comparison, we will again only consider the leading term, $\frac{n^2}{2}$, as a benchmark for RLS performance.

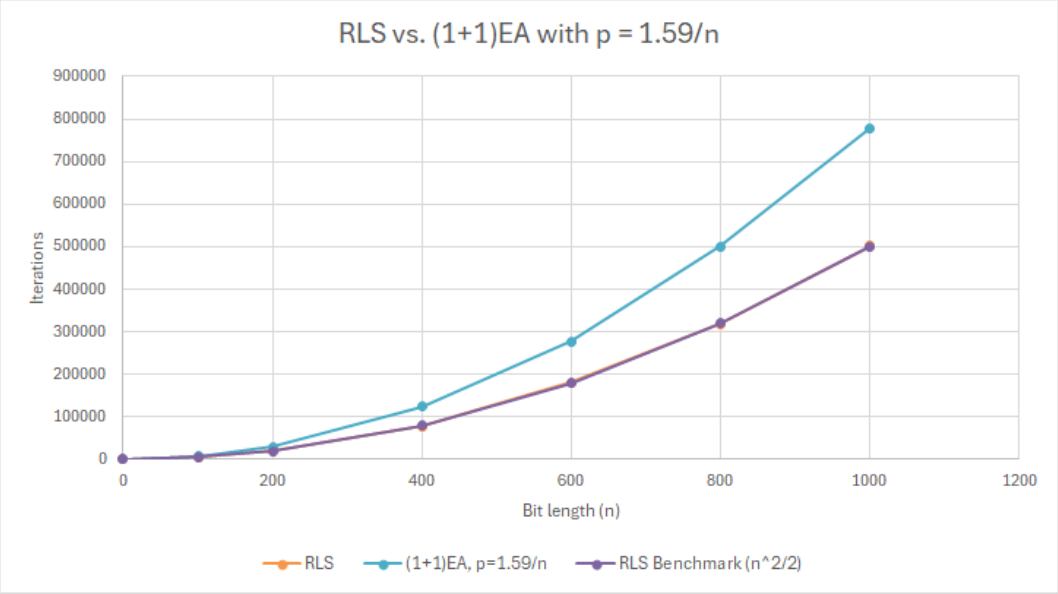

Figure 24: Averaged over 100 runs.

The graph shows how the empirical performance of RLS aligns remarkably well with the theoretical

expected time $\frac{n^2}{2}$ across all tested bit string lengths. The graph also shows that the $(1+1)$ EA with $p = \frac{1.59}{n}$, while optimized, still requires a significantly higher number of iterations (778,460 iterations) to find the optimum compared to RLS (501,693 iterations). This outcome empirically supports the theoretical prediction that RLS is more efficient than the $(1+1)$ EA, even with an optimized mutation rate, for solving the LeadingOnes problem.

6.1.3 TSP

(1+1) EA solving TSP using 2-opt and 3-opt

As explained in the implementation section 5.2.6, we implemented a stopping condition class called "NoImprovementFound" that has a maximum iteration count of 10,000 as a termination criterion for the TSP algorithms. This experiment aims to investigate the influence of varying this iteration limit on the ability of the (1+1) EA to reach its optimum for the TSP. We will conduct a systematic study, testing the following iteration limits: 10,000, 20,000, 40,000, 60,000, 80,000, and 100,000. For each tested limit, the average fitness will be computed across 100 runs.

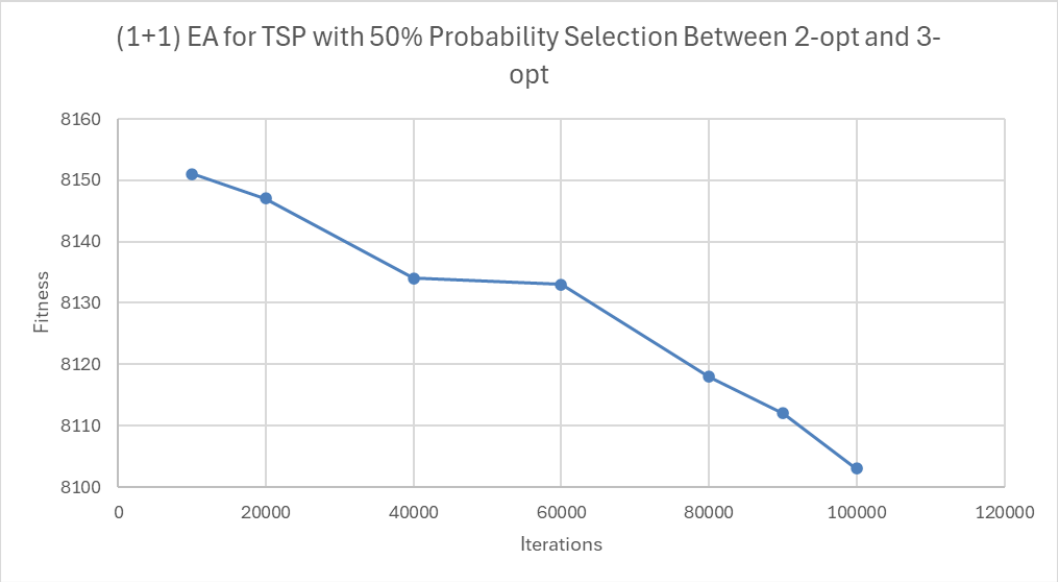

Figure 25: Averaged over 100 runs on the Berlin52 tour.

As shown in Table 1, the known optimal fitness for the Berlin52 TSP instance is 7542. Our experimental results, shown in Figure 25, demonstrate that increasing the number of iterations significantly improves the fitness achieved by the $(1+1)$ EA. For instance, with an iteration limit of 10,000, the $(1+1)$ EA reached an average fitness of 8151. This value represents a deviation of approximately 8.07% from the optimal fitness. However, when the iteration limit was extended to 100,000, the $(1+1)$ EA achieved an average fitness of 8103. This improved fitness represents a deviation of approximately 7.44% from the optimal, representing 0.63% reduction in deviation compared to the performance at 10,000 iterations.

We will now investigate another critical parameter of the $(1+1)$ EA for TSP. As noted in our Background section 2.3.3, this version of the $(1+1)$ EA uses 2-opt and 3-opt operators with equal probability. In this experiment, we will test the impact of changing the probability of using the 3-opt operator. Specifically, the experiment will cover probabilities of 10%, 30%, 50%, 75%, and 100%

for the 3-opt operator (with the 2-opt probability being $1 - P_{3-opt}$). The experimental setup will largely mirror the previous one, with iteration limits on the x-axis and fitness on the y-axis, allowing us to observe the fitness evolution for these different 3-opt probabilities.

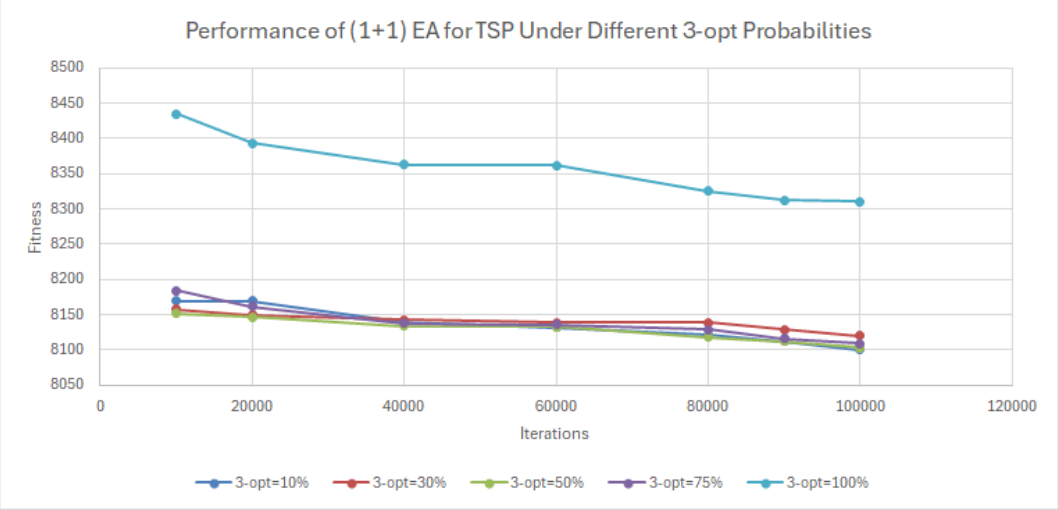

Figure 26: Averaged over 100 runs on the Berlin52 tour.

A key finding from this experiment is the significant impact of the 3-opt probability on the algorithm's ability to find high-quality solutions. As illustrated in figure 26, when using a 100% 3-opt probability yields the highest/worst fitness across all iteration counts. This demonstrates that relying solely on the 3-opt operator is not optimal for the (1+1) EA.

In contrast, all other tested probabilities $(10\%, 30\%, 50\%,$ *and* $75\%)$ achieve better fitness, with their performance curves clustering at lower fitness values. For instance at 100,000 iterations, the 10% 3-opt probability delivers the best average fitness of 8100. The 50% 3-opt (default setting) and 75% 3-opt configurations perform very similarly, achieving average fitness values of 8103 and 8109 respectively. This shows how a balanced application of 2-opt and 3-opt, or even a higher bias towards the 2-opt operator, is more effective, while very high probabilities of 3-opt can be detrimental.

(1+1) EA solving TSP using Poisson-distributed Mutation (2-opt)

We will now investigate a distinct variant of the (1+1) EA. This version of the (1+1) EA employs Poisson-distributed 2-opt mutation, an approach where the number of 2-opt moves applied in each mutation step is drawn from a Poisson distribution. This experiment aims to evaluate the performance of this alternative mutation strategy in solving the Traveling Salesman Problem.

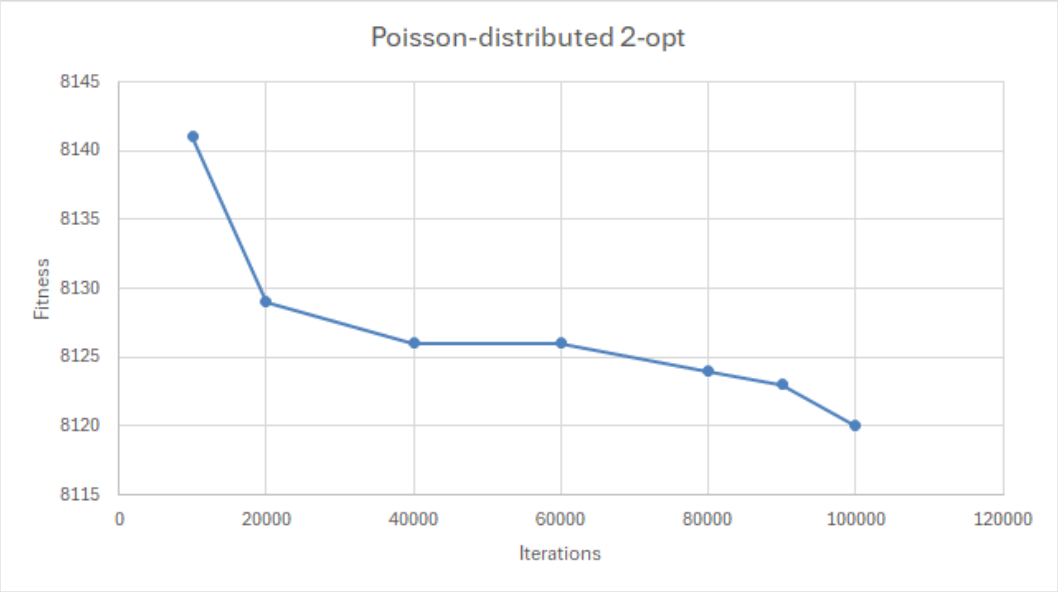

Figure 27: Averaged over 100 runs on the Berlin52 tour.

Figure 27 illustrates the optimization progression of the (1+1) EA utilizing Poisson-distributed 2-opt mutation on the Berlin52 TSP instance. As expected from an optimization process, the graph shows a consistent improvement in fitness as the number of iterations increases. The algorithm begins with an average fitness of approximately 8141 after 10,000 iterations and further refines the solution to an average fitness of around 8120 at 100,000 iterations.

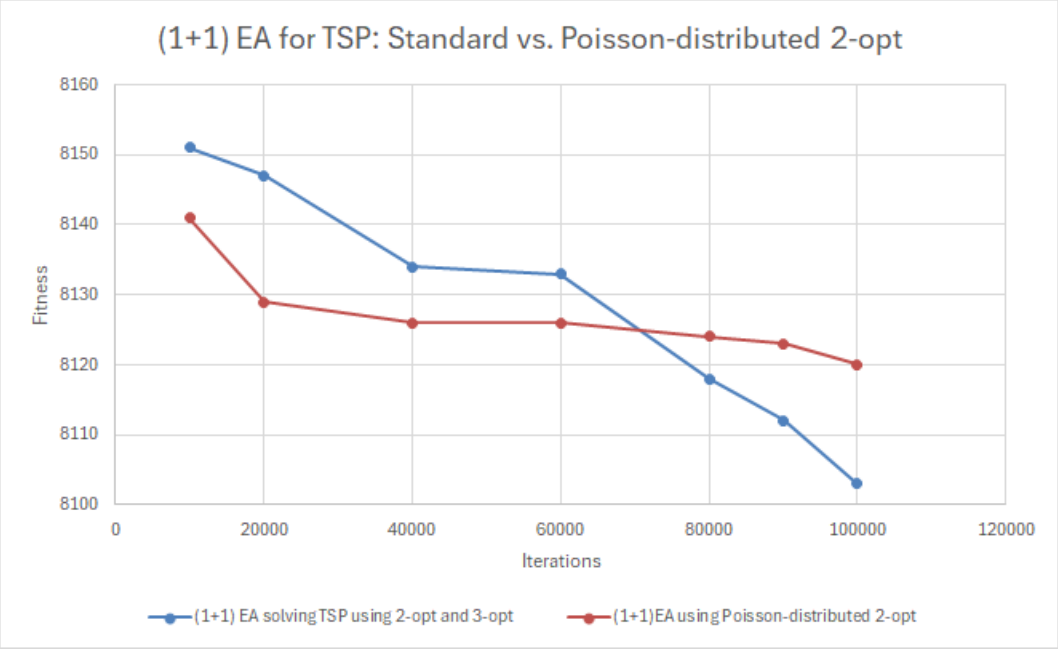

Figure 28: Averaged over 100 runs on the Berlin52 tour.

Figure 28 provides a comparison of the optimization for the two variants of the $(1+1)$ EA on the Berlin52 TSP instance: the "Standard" version using 2-opt and 3-opt operators with equal probabil-

ity, and the variant employing Poisson-distributed 2-opt mutation.

We can see in figure 28 that up to approximately 60,000 iterations, the $(1+1)$ EA with Poisson-distributed 2-opt mutation demonstrates a slightly faster convergence, achieving better fitness values compared to the standard 2-opt/3-opt variant. However, as the optimization progresses towards 100,000 iterations, the trend reverses. The standard $(1+1)$ EA, which incorporates the 3-opt operator, eventually overtakes and achieves a final fitness of 8103 at 100,000 iterations, while the Poisson-distributed 2-opt variant converges to approximately 8120.

This outcome shows how the 3-opt operator is capable of making more significant changes to the tour helping the algorithm to escape local optima and explore the search space more effectively, leading to better solutions in the long run.

6.2 $(\mu+\lambda)$ EA

In the following experiments, we investigate how varying the values of μ and λ affects the performance of the $(\mu+\lambda)$ Evolutionary Algorithm. The benchmark problems used for these experiments are OneMax and LeadingOnes. In the first experiment, we vary μ from 1 to 10 while keeping λ fixed at 1. In the second experiment, we reverse this setup by keeping μ constant and increasing λ from 1 to 10.

6.2.1 OneMax

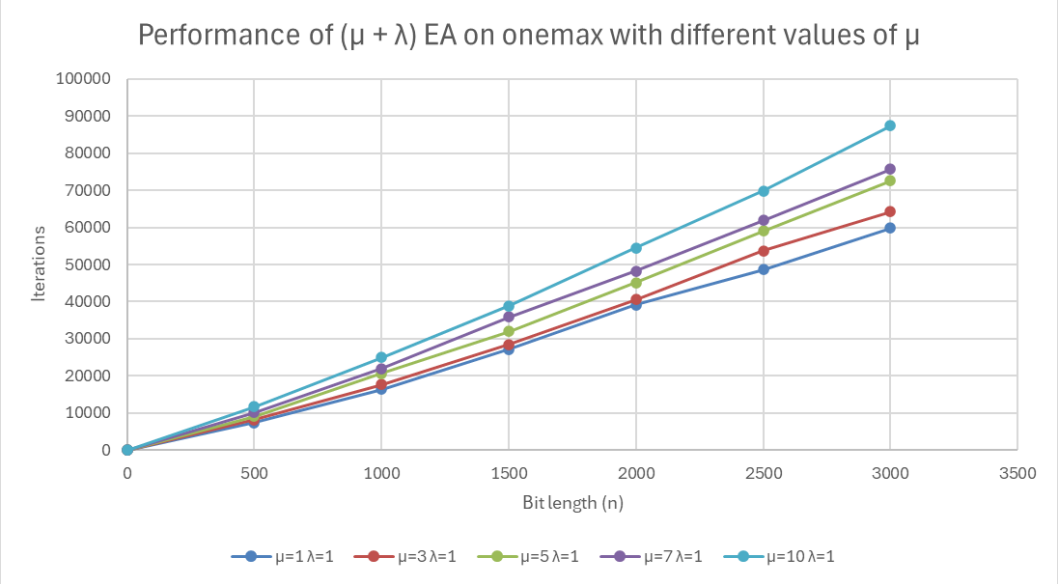

Figure 29: Averaged over 100 runs

Figure 29 illustrates the performance of the $(\mu+\lambda)$ EA on OneMax, specifically investigating the impact of varying the parent population size μ while keeping the number of offspring λ constant at 1. As the bit string length (n) increases, the number of iterations required for the algorithm to reach the optimum consistently rises across all tested μ values. A clear trend observed is that increasing the parent population size μ leads to a notable increase in the average number of iterations required for optimization. The configuration with $\mu = 1$ (i.e., the $(1+1)$ EA) demonstrates the most efficient

performance, requiring the fewest iterations (59,855 iterations) to reach the optimum. Conversely, as μ increases to 3, 5, 7, and especially 10, the optimization time progressively increase, with $\mu = 10$ requiring the highest number of iterations (87,418 iterations).

This increase in optimization time can be explained by how the $(\mu + \lambda)$ EA behaves when $\lambda = 1$. In this setup, only one new solution (offspring) is created in each generation. If μ is large, this single offspring has to compete with many parents for a spot in the next generation. For example, with $\mu = 10$ and $\lambda = 1$, the offspring is one of 11 candidates, but only 10 can be selected. This means even good offspring might get rejected, slowing down progress. As μ increases, it becomes harder for improvements to take over the population, leading to longer optimization times.

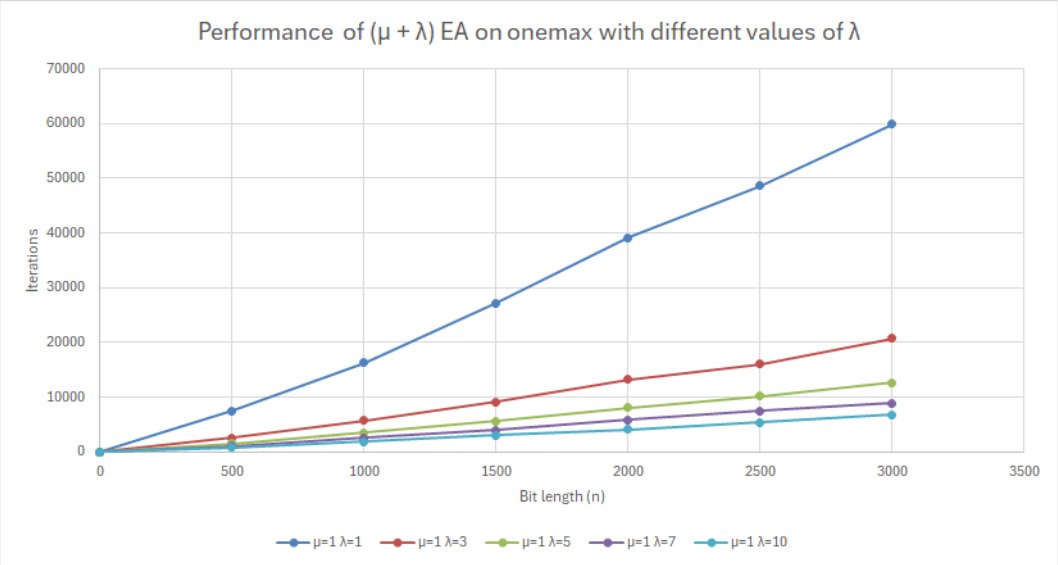

Figure 30: Averaged over 100 runs

Figure 30 illustrates the results of the second experiment for the $(\mu + \lambda)$ EA on OneMax, focusing on the impact of varying the number of offspring λ while keeping the parent population size μ constant at 1. Similar to previous observations, the required number of iterations to find the optimum increases with the bit string length (n) for all tested λ values. A significant inverse relationship is observed between the number of offspring λ and the optimization time. Specifically, increasing λ leads to a reduction in the average number of iterations. The configuration with $\mu = 1, \lambda = 1$ (the $(1+1)$ EA) is the least efficient, requiring the highest number of iterations (59,855 iterations). As λ increases to 3, 5, 7, and particularly to 10, the algorithm's performance markedly improves, with $\mu = 1, \lambda = 10$ demonstrating the fastest optimization time (6761 iterations) which is a reduction of 88.69%.

This reduction in optimization time can be explained by how the $(\mu + \lambda)$ EA behaves when $\mu = 1$. In this setup, multiple offspring are generated from a single parent each generation. With more offspring, there's a higher chance of producing a better solution. This increases the chances of improvement in each step, allowing the algorithm to find the optimum faster and with fewer iterations.

6.2.2 LeadingOnes

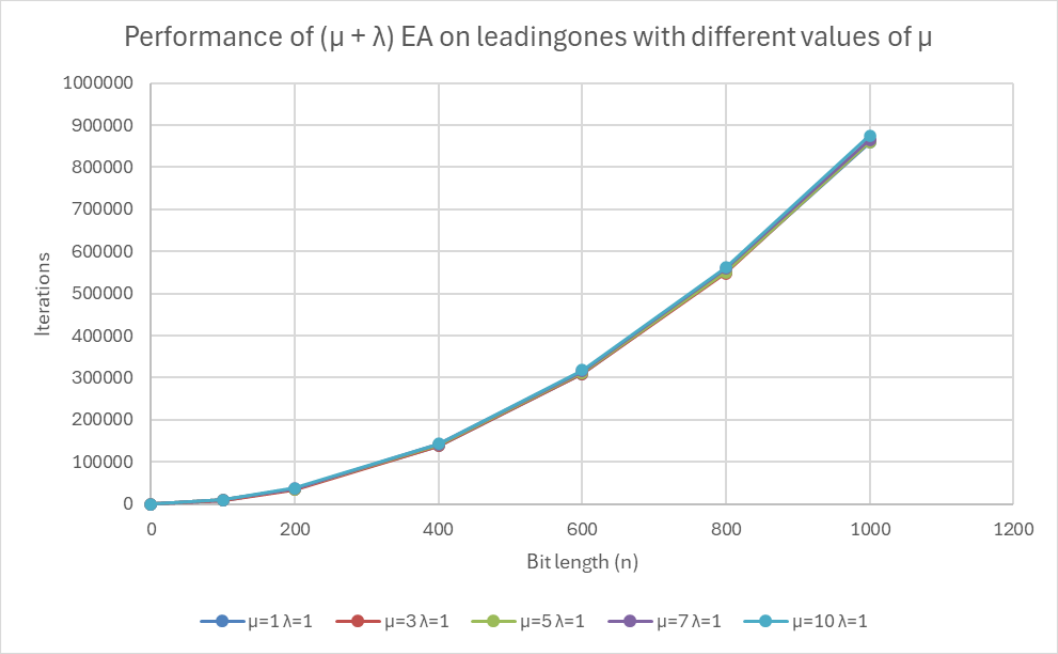

Figure 31: Averaged over 100 runs

Figure 31 illustrates the performance of the $(\mu + \lambda)$ EA on LeadingOnes, investigating the impact of varying the parent population size μ while keeping the number of offspring λ constant at 1. While the number of iterations consistently increases with bit string length (n). The same trend observed on OneMax. Increasing the parent population size μ from 1 to 10 has a minimal impact on the algorithm's optimization time. The performance curves for all tested μ values (e.g., $\mu = 1, 3, 5, 7, 10$ with $\lambda = 1$) are almost identical, indicating that the value of μ plays a minimal role in the convergence speed when $\lambda = 1$. The following table shows the number of iterations at $n = 1000$:

μ	Iterations at $n = 1000$
1	859,342
3	863,755
5	859,715
7	864,803
10	875,895

Table 3: Iterations for $(\mu + 1)$ EA on LeadingOnes at $n = 1000$

The difference in behavior between OneMax and LeadingOnes can be explained by looking at how these problems are structured. In OneMax, having a larger μ with $\lambda = 1$ makes it harder for small improvements to take over, which slows the optimization time down. But LeadingOnes is harder in a different way, it needs the exact right bit to change each time to make progress. Since only one offspring is created per generation, the main challenge is finding that correct mutation. In this case, having more parents (μ) doesn't matter as much because progress depends more on getting lucky with the right mutation than on which individuals are selected.

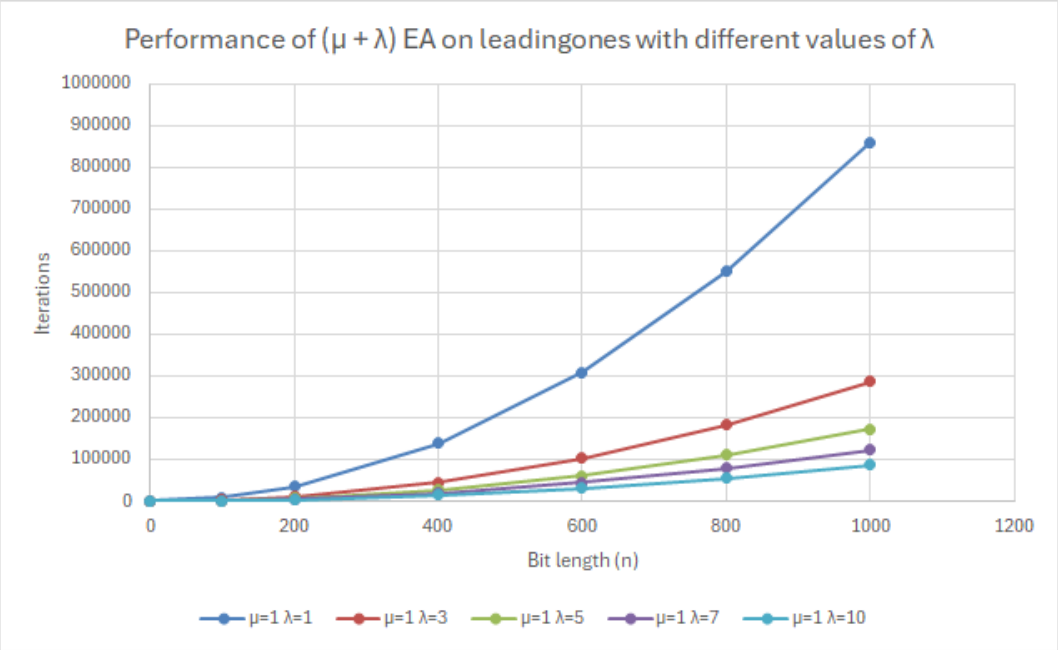

Figure 32: Averaged over 100 runs

Figure 32 illustrates the performance of the $(\mu + \lambda)$ EA on LeadingOnes, investigating the impact of varying the number of offspring λ while keeping the parent population size μ constant at 1. Consistent with the OneMax results, the number of iterations required to reach the optimum for LeadingOnes also increases with the bit string length (n).

Increasing λ from 1 to 10 leads to a reduction in the average number of iterations required to solve LeadingOnes. The configuration with $\mu = 1, \lambda = 1$ (the $(1+1)$ EA) is the least efficient, requiring the highest number of iterations. As λ increases, the algorithm's performance significantly improves, with $\mu = 1, \lambda = 10$ demonstrating the fastest optimization time. For instance, at $n = 1000$, the iterations are reduced from 859,342 for $\lambda = 1$ to around 86,150 for $\lambda = 10$. This is a reduction of $89,96\%$.

This significant reduction in optimization time for LeadingOnes, similar to that observed for OneMax, is explained by the fundamental behavior of the $(\mu + \lambda)$ EA when $\mu = 1$. Generating multiple offspring from a single parent in each generation increases the probability of producing a fitter solution. This opportunity for improvement at each step allows the algorithm to propagate beneficial mutations more rapidly, leading to faster optimization time and fewer total iterations.

6.3 Simulated Annealing (SA)

In the following experiments, we investigate how varying the values of the cooling rate α and the initial temperature T_0 affect the performance of the simulated annealing algorithm. The benchmark problems used for these studies are OneMax, LeadingOnes, and the Traveling Salesman Problem (TSP).

In the first experiment, we will vary the cooling rate α. We will test values suggested by Cicirello (0.9, 0.95, and 0.99) [25], as well as some other extreme values (0.1, 0.5, and 0.9999) to study their effect on the algorithm's optimization behavior. This particular experiment will be conducted on

both OneMax and LeadingOnes.

6.3.1 OneMax

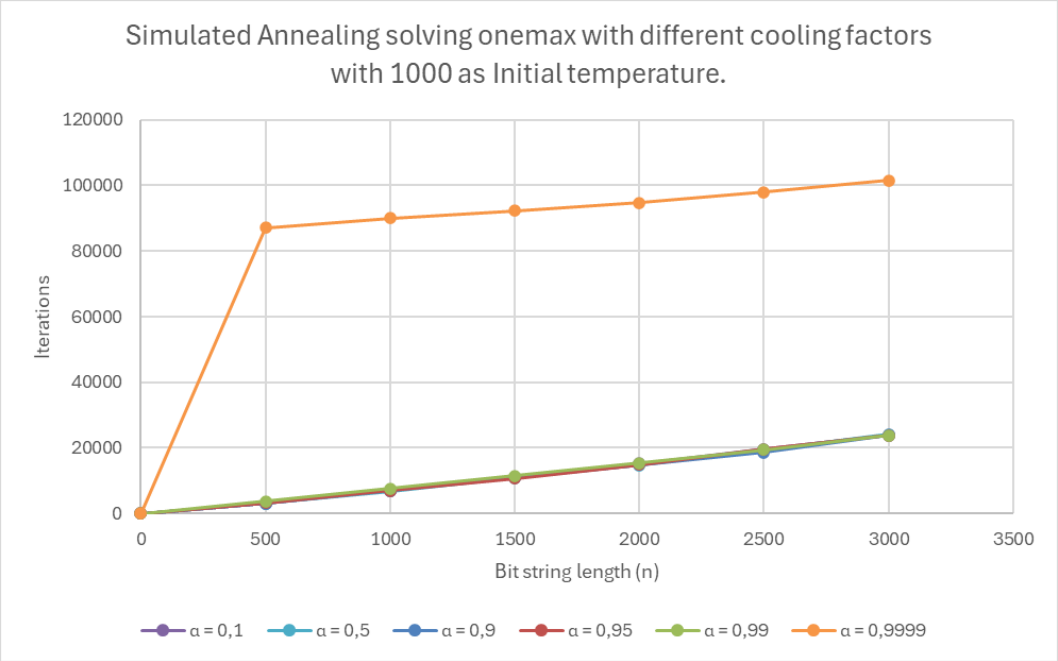

Figure 33: Averaged over 100 runs

Figure 33 shows how the Simulated Annealing algorithm performs on OneMax when using different cooling rates (α), starting from an initial temperature of 1000. As expected, the number of iterations needed to reach the optimal solution increases with the bit string length (n) for all tested values of α.

The results highlight a clear dependence on the choice of α. Very high cooling rates, such as $\alpha = 0.9999$, lead to much longer optimization time over 100,000 iterations for $n = 3000$. In contrast, lower values like $\alpha = 0.1$ and $\alpha = 0.5$ result in much faster convergence, with around 24,000 iterations for the same bit string length. Intermediate values (e.g., $\alpha = 0.9$, 0.95, 0.99) show a gradual increase in runtime as α gets closer to 1.

Cooling Rate (α)	**Iterations**
0.1	23,933
0.5	24,172
0.9	23,721
0.95	23,728
0.99	23,779
0.9999	101,499

Table 4: Number of iterations for $n = 3000$ with varying cooling rates α

This trend is due to how α controls the cooling schedule. A high α causes the temperature to decrease very slowly, keeping the probability of accepting worse solutions high for longer. This leads to a prolonged random search, slowing down convergence. On the other hand, lower α values cool the system more quickly, reducing the chance of accepting worse solutions early on.

6.3.2 LeadingOnes

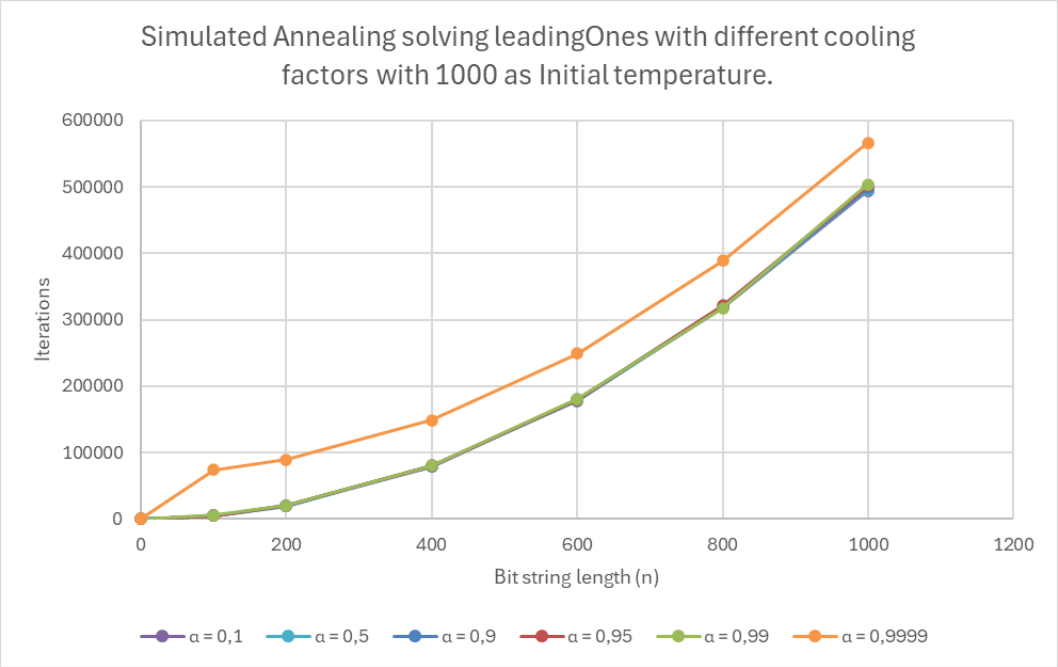

Figure 34: Averaged over 100 runs

Figure 34 illustrates the performance of the Simulated Annealing algorithm on LeadingOnes, examining the influence of different cooling rates (α) with a fixed initial temperature of 1000. As consistently observed, the number of iterations required to reach the optimum increases with the bit string length (n).

A critical finding for LeadingOnes is the performance degradation when α is extremely high (e.g., $\alpha = 0.9999$), resulting in significantly longer optimization times, nearing 567,000 iterations for $n = 1000$. In contrast, a wide range of lower α values (0.1, 0.5, 0.9, 0.95, 0.99) yield considerably better and similar performance, clustering closely together, see table 5. This suggests that for LeadingOnes, as long as the cooling is not slow, the precise α value within a reasonable range has a less impact on convergence speed compared to what we have observed from OneMax.

Cooling Rate (α)	**Iterations for** $n = 1000$
0.1	494,184
0.5	495,902
0.9	498,389
0.95	501,279
0.99	503,873
0.9999	566,612

Table 5: Number of iterations for $n = 1000$ with varying cooling rates α

This trend can be explained by the balance between exploration and exploitation controlled by the cooling rate α. A very high α slows down the cooling process, keeping the algorithm in a exploratory phase for too long. This leads to extended random behavior, which is especially inefficient for LeadingOnes, where progress requires finding the next correct bit in sequence. Once α is low enough

to allow faster cooling (e.g., between 0.1 and 0.99), the algorithm quickly shifts toward exploitation. In this phase, performance is limited more by the difficulty of finding the next correct move than by differences in the acceptance probability explaining why these α values perform similarly.

6.3.3 TSP

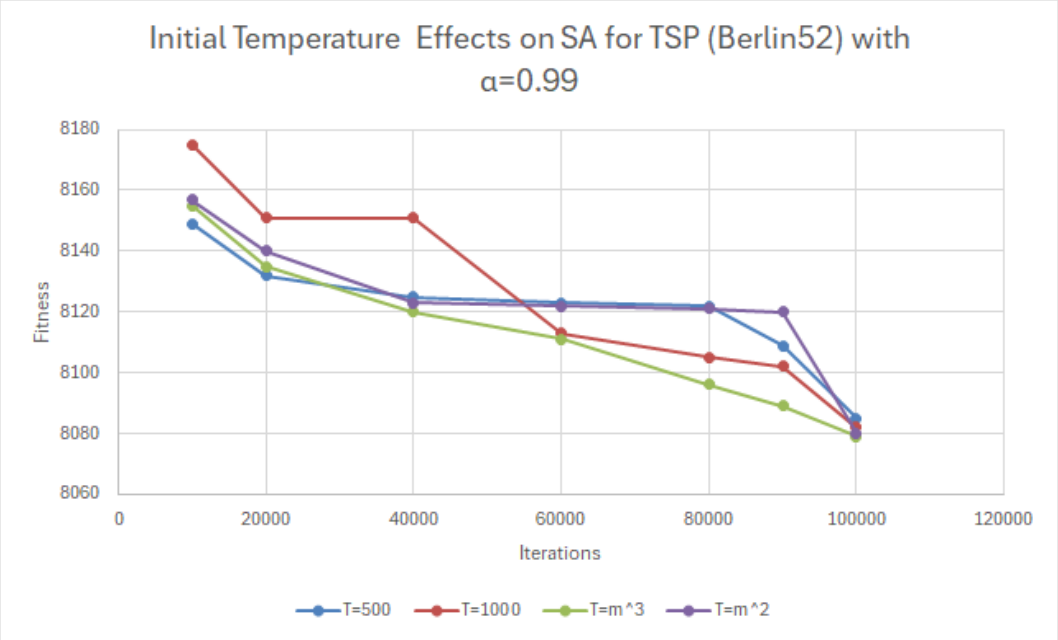

Figure 35: Averaged over 100 runs on the Berlin52 tour

Figure 35 illustrates the performance of the Simulated Annealing algorithm on TSP (Berlin52) instance, investigating the impact of different initial temperatures (T_0) while keeping the cooling rate α constant at 0.99. All tested initial temperatures demonstrate the algorithm's ability to reduce the tour length over iterations, with all configurations eventually converging to competitive solutions within the range of 8079 to 8085 fitness at 100,000 iterations.

Specifically, the configuration with $T = m^3$ consistently yields the lowest final fitness of 8079 at 100,000 iterations. Closely following in performance are $T = m^2$ and $T = 500$, achieving final fitness values of 8082 and 8085 at 100,000 iterations. The $T = 1000$ configuration, while starting with a higher initial fitness (8175 at 10,000 iterations), ultimately also converges to a competitive solution of 8082 fitness at 100,000 iterations.

However, despite these performances, the observed final fitness values are quite similar across the different tested initial temperatures, approximately 7.16% above the known optimal fitness of 7542 for the Berlin52 instance. To gain a better understanding of these initial temperature settings, a further experiment will be conducted. This analysis will involve 100 runs for each initial temperature, all executed in 100,000 iterations. For each configuration, we will record the average fitness achieved, the lowest fitness achieved, the highest fitness achieved, and the number of times the algorithm successfully reaches the optimal fitness of 7542.

	$T_0 = 500$	$T_0 = 1000$	$T_0 = m^2$	$T_0 = m^3$
Optimal Reached (times)	2	3	0	0
Minimum Fitness	7542	7542	7619	7598
Average Fitness	8080	8083	8087	8092
Maximum Fitness	9176	8800	8675	8737

Table 6: Performance over 100 runs, executed in 100,000 iterations.

The results shown in table 6 reveal nuanced differences not apparent when examining only the average fitness over iterations. While the average fitness for all tested initial temperatures converged to a similar range of values (approximately 7.16% above the known optimal fitness of 7542 for the Berlin52 instance) at 100,000 iterations, a key finding from this more detailed analysis is their differing abilities to reach the true optimal fitness of 7542. Only $T_0 = 500$ and $T_0 = 1000$ successfully converged to the optimal solution, doing so 2 and 3 times out of 100 runs. In contrast, the higher initial temperatures, $T = m^2$ and $T = m^3$, never reached the optimal fitness of 7542 in any of the 100 runs, with their best observed fitness being 7619 and 7598.

Regarding the average performance across all 100 runs, $T = 500$ achieved the best average fitness of 8080, closely followed by $T = 1000$ at 8083. $T = m^2$ and $T = m^3$ with average fitness values of 8087 and 8092, indicating less efficient convergence on average across multiple runs.

$T_0 = 500$ demonstrated the widest spread of results, with a maximum fitness of 9176, suggesting that while it can achieve the optimal, it also has a higher probability of performing poorly in some runs. In contrast, $T = m^2$ showed the most consistent performance by having the lowest maximum fitness of 8675, meaning its worst runs were not as extreme as those of $T_0 = 500$.

These observations suggest that for the TSP (Berlin52) instance with $\alpha = 0.99$, lower initial temperatures like $T_0 = 500$ or $T_0 = 1000$ are more likely to find the global optimum, although they may get higher variability in performance. Higher initial temperatures, such as $T_0 = m^2$ and $T_0 = m^3$, tend to yield more consistent, however they have sub-optimal, average results and never reached the true global optimum. This indicates that while a very high initial temperature (like m^3) might offer thorough exploration and good average performance over iterations, it doesn't guarantee reaching the global optimum when compared to more moderate initial temperatures.

6.4 Ant Colony Optimization (ACO)

This section details experiments involving ACO algorithms. For the OneMax and LeadingOnes, we will investigate the performance of the 1-Ant algorithm and the Max-Min Ant System (MMAS), focusing on how different values of the evaporation rate (ρ) impact their effectiveness. For the Traveling Salesman Problem (TSP), we will study the performance of both the standard Ant System (AS) and MMAS, examining the impact of varying ρ values and the number of ants (m).

6.4.1 1-Ant

OneMax

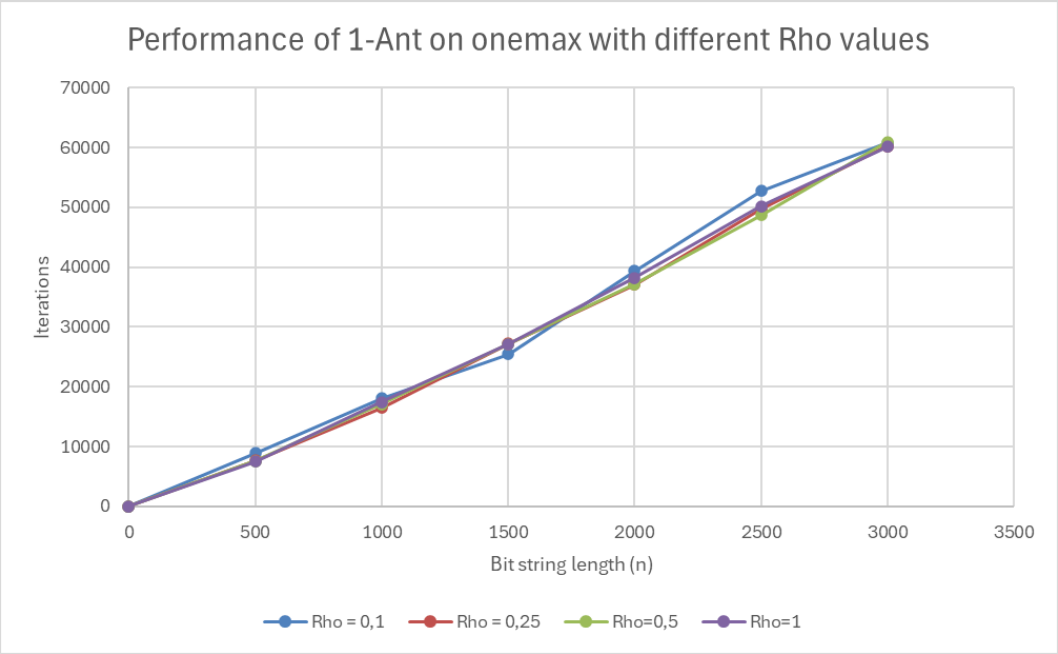

Figure 36: Averaged over 100 runs.

Figure 36 illustrates the performance of the 1-Ant algorithm on OneMax, examining the impact of varying the evaporation rate (ρ) on the number of iterations required for optimization. As expected, for all tested ρ values, the number of iterations consistently increases with the bit string length (n).

A primary observation from this experiment is that the choice of ρ has a relatively minor influence on the 1-Ant algorithm's convergence speed for the OneMax problem. The performance curves for $\rho = 0.1, 0.25, 0.5$, and 1 are tightly clustered, indicating that the algorithm's behavior is largely robust to changes in this parameter within the tested range.

$\rho = 0.1$		$\rho = 0.25$		$\rho = 0.5$		$\rho = 1$	
n	**Iterations**	**n**	**Iterations**	**n**	**Iterations**	**n**	**Iterations**
0	0	0	0	0	0	0	0
500	8916	500	7677	500	7566	500	7456
1000	18015	1000	16500	1000	17101	1000	17444
1500	25417	1500	27243	1500	27164	1500	27164
2000	39344	2000	37076	2000	37178	2000	38247
2500	52706	2500	49649	2500	48737	2500	50166
3000	60798	3000	60244	3000	60934	3000	60113

Table 7: Iterations for 1-Ant on OneMax with different ρ values (corresponding to Figure 36).

Upon closer inspection of the detailed iteration counts (Table 7), some differences can be observed. While no single ρ value consistently outperforms all others across the entire range of bit string lengths. For smaller problem sizes, such as $n = 500$, $\rho = 1$ shows slightly better performance (7456 iterations). For intermediate lengths like $n = 1000$ and $n = 2000$, $\rho = 0.25$ shows an advantage (16500 and 37076 iterations). At the largest problem size, $n = 3000$, $\rho = 1$ and $\rho = 0.25$ again perform comparably well (60113 and 60244 iterations). Conversely, $\rho = 0.1$ generally requires slightly more iterations for larger problem instances, such as $n = 3000$ (60798 iterations).

Overall, these results show that for the 1-Ant algorithm on OneMax, the specific setting of the evaporation rate ρ does not drastically affect the optimization time. The algorithm generally scales as expected with problem size, and a moderate ρ value (e.g., between 0.25 and 1) appears to offer better or comparable performance across varying bit string lengths.

LeadingOnes

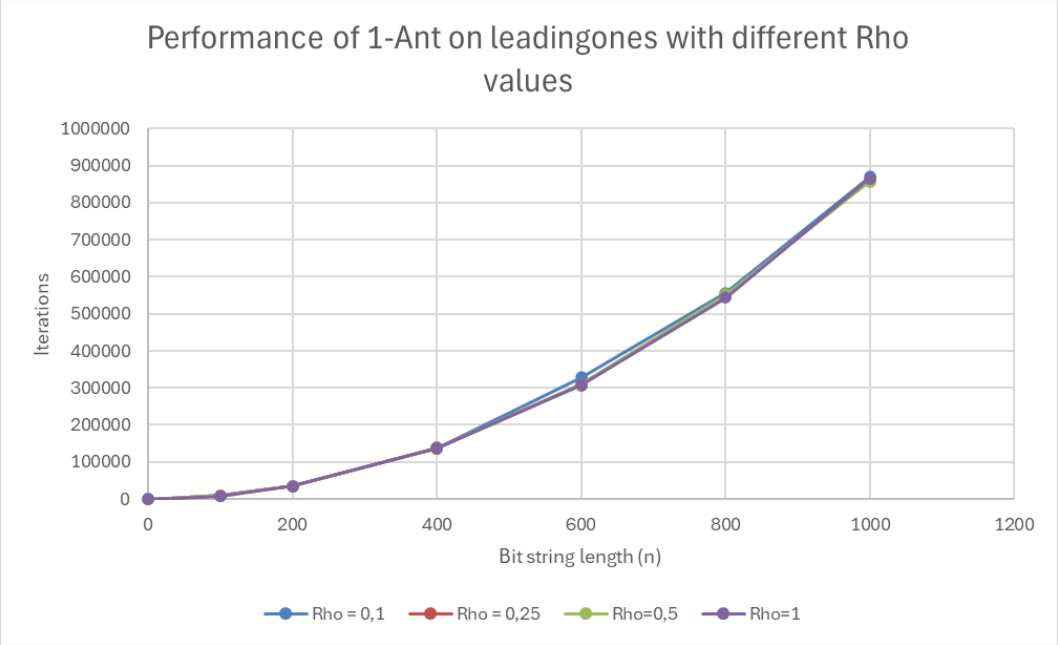

Figure 37: Averaged over 100 runs.

Figure 37 illustrates the performance of the 1-Ant algorithm on LeadingOnes, investigating the impact of varying the evaporation rate (ρ) on the number of iterations required for optimization. As observed with the OneMax problem, the number of iterations consistently increases with the bit string length (n).

Unlike the OneMax experiment where ρ had a minor influence, the choice of ρ demonstrates a more noticeable impact on the 1-Ant algorithm's convergence speed for the LeadingOnes problem. Generally, for larger problem sizes, higher values of ρ lead to better performance.

$\rho = 0.1$		$\rho = 0.25$		$\rho = 0.5$		$\rho = 1$	
n	**Iterations**	**n**	**Iterations**	**n**	**Iterations**	**n**	**Iterations**
0	0	0	0	0	0	0	0
100	8693	100	8884	100	8586	100	8422
200	34224	200	34821	200	35090	200	35362
400	138560	400	138680	400	137291	400	136948
600	328524	600	307318	600	309594	600	308333
800	556312	800	548176	800	552726	800	543161
1000	869279	1000	862733	1000	856987	1000	865437

Table 8: Iterations for 1-Ant on LeadingOnes with different ρ values (corresponding to Figure 37).

Detailed iteration counts (Table 8) reveal specific trends. For the smallest tested length ($n = 100$),

$\rho = 1$ was slightly more efficient (8422 iterations). Interestingly, for $n = 200$, lower ρ values like $\rho = 0.1$ (34224 iterations) and $\rho = 0.25$ (34821 iterations) showed an advantage over higher ρ values. However, for all problem sizes $n \geq 400$, $\rho = 1$ consistently emerged as the most efficient evaporation rate, requiring the fewest iterations (e.g., 136948 at $n = 400$, 865437 at $n = 1000$). Conversely, $\rho = 0.1$ generally performed the worst for $n \geq 400$, requiring the highest number of iterations (e.g., 869279 at $n = 1000$).

These results show that for the 1-Ant algorithm on the LeadingOnes problem, a higher evaporation rate (like $\rho = 1$) is generally beneficial for larger problem instances. This can be explained by the fact that a higher ρ leads to more aggressive learning. It causes pheromones on less promising paths to fade faster and gives more weight to recent successful solutions. As a result, the algorithm can adapt more quickly, focus on promising areas, and concentrate pheromones on the key path segments needed to find the leading ones, especially as the string length increases.

6.4.2 MMAS

As observed in the previous experiments, the performance of the 1-Ant algorithm remains relatively similar for ρ values within the range $\rho \in [0.1, 1]$. Therefore, in the upcoming experiments, we focus on testing more extreme values, specifically $\rho = 0.001$ and $\rho = 0.01$. These will be compared against $\rho = 0.1$ and $\rho = 1$ to better understand the impact of very low evaporation rates.

OneMax

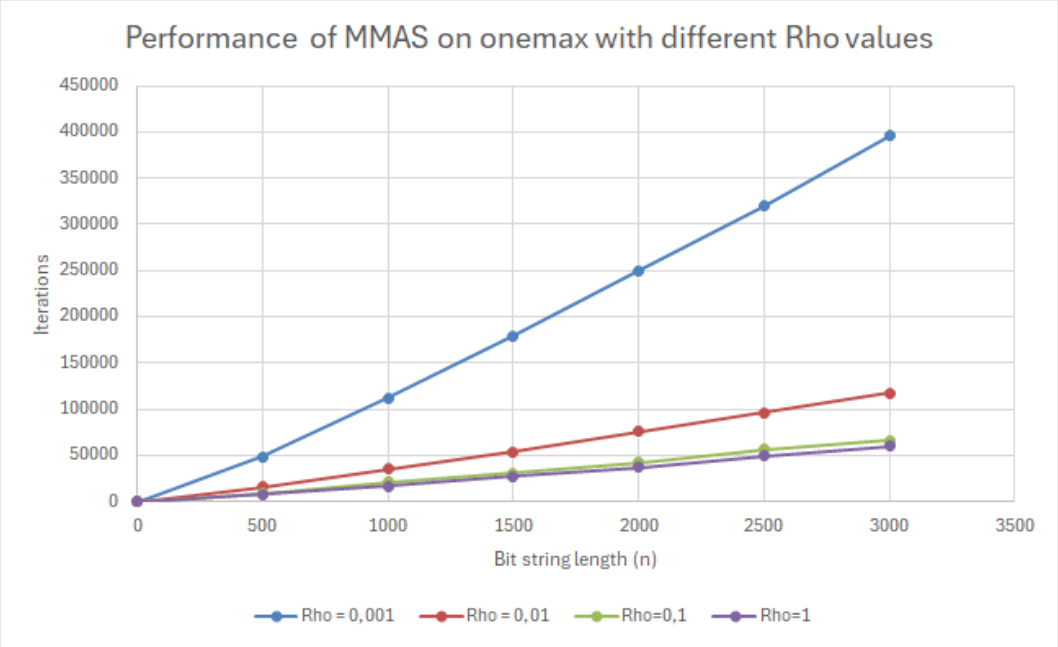

Figure 38: Averaged over 100 runs.

Figure 38 shows the performance of the MMAS on OneMax for various evaporation rates (ρ). As expected, the number of iterations required to reach the optimum increases with the bit string length (n) across all tested ρ values.

Unlike the 1-Ant algorithm, where performance was relatively similar for $\rho \in [0.1, 1]$, MMAS shows a much stronger sensitivity to the choice of ρ. In particular, extremely low evaporation rates lead to significantly slower convergence. As shown in both Figure 38 and Table 9, $\rho = 0.001$ performs the worst by a large margin, requiring nearly 400,000 iterations for $n = 3000$. $\rho = 0.01$ also underperforms, though much better than $\rho = 0.001$.

ρ	**Iterations (n = 3000)**
0.001	395,628
0.01	116,912
0.1	66,063
1	59,878

Table 9: MMAS Iterations for OneMax at $n = 3000$ with different ρ values (corresponding to Figure 38)

In contrast, higher evaporation rates specifically $\rho = 0.1$ and $\rho = 1$ lead to much faster convergence. For $n = 3000$, MMAS converges in 66,063 iterations with $\rho = 0.1$ and just 59,878 iterations with $\rho = 1$. This represents a major improvement over $\rho = 0.01$ (116,912 iterations) and $\rho = 0.001$ (395,628 iterations).

These results highlight the importance of using a high evaporation rate in MMAS when solving the OneMax problem. Very low values of ρ cause pheromone to persist too long on suboptimal paths, which can lead to exploration or even stagnation. In contrast, higher values allow the algorithm to quickly reinforce promising solutions, leading to more efficient convergence.

LeadingOnes

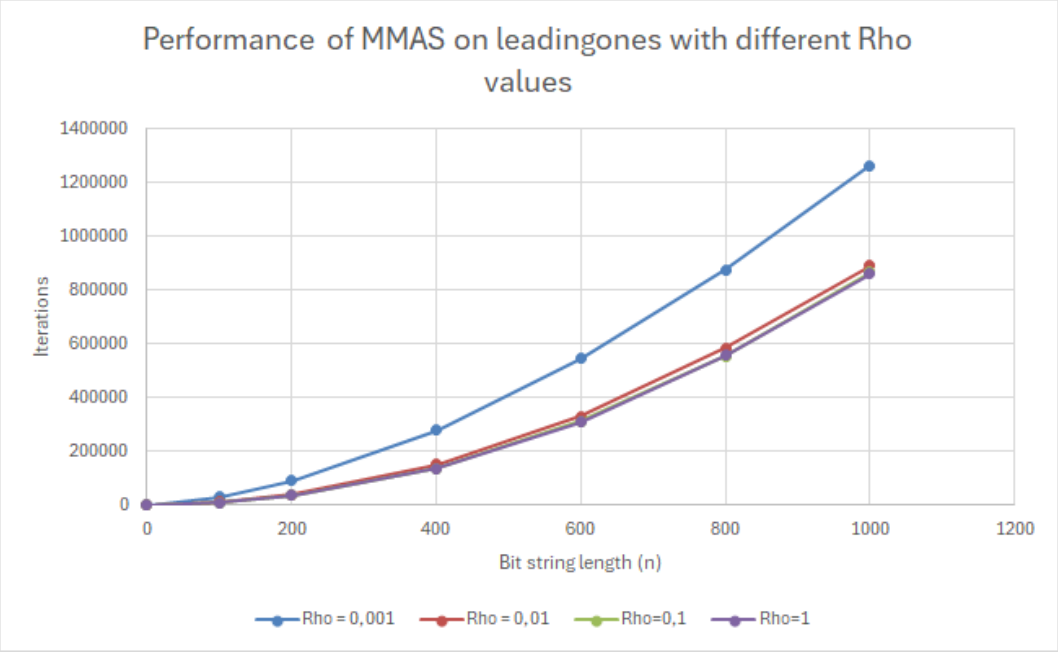

Figure 39: Averaged over 100 runs.

Figure 39 shows the performance of the MMAS on LeadingOnes across different evaporation rates

(ρ). Extremely low values particularly $\rho = 0.001$ and $\rho = 0.01$ result in substantially slower convergence. At $n = 1000$, MMAS requires over 1.26 million iterations for $\rho = 0.001$ and nearly 900,000 iterations for $\rho = 0.01$, as shown in Table 10. In contrast, higher evaporation rates, $\rho = 0.1$ and $\rho = 1$, lead to much faster and comparable performance, with convergence achieved in 865,810 and 860,221 iterations, respectively.

ρ	**Iterations (n = 1000)**
0.001	1,263,629
0.01	891,235
0.1	865,810
1	860,221

Table 10: MMAS Iterations for LeadingOnes at $n = 1000$ with different ρ values (corresponding to Figure 39)

When comparing MMAS to the 1-Ant algorithm on LeadingOnes, both favor higher ρ values at larger problem sizes. MMAS is sensitive to very low evaporation rates, where performance declines more sharply. This highlights the importance of maintaining a high ρ in MMAS to avoid stagnation and ensure that pheromone updates remain focused on promising regions of the search space. The similar performance of $\rho = 0.1$ and $\rho = 1$ further suggests that MMAS benefits from a dynamic balance of evaporation and reinforcement, which helps it discover and exploit the correct bit sequence.

TSP

As discussed in the introduction of Section 6, Ant Colony Optimization algorithms such as MMAS and AS perform multiple function evaluations per iteration when solving the TSP, due to the use of multiple ants. Therefore, in the following experiments, we will be looking at the number of function evaluations and the achieved fitness for MMAS and the AS algorithm. The experimental parameters is summarized in Table 2, where we focus specifically on varying two parameters: the evaporation rate ρ and the number of ants m.

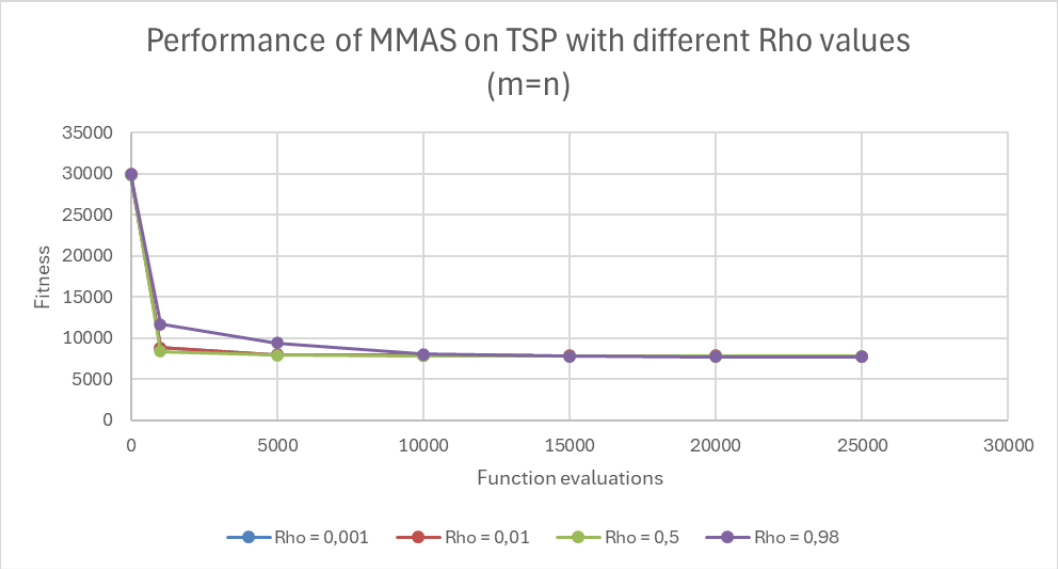

Figure 40: Averaged over 100 runs on the Berlin52 tour.

Figure 40 illustrates the performance of MMAS on TSP (Berlin52) instance, examining the impact of varying the evaporation rate ρ on the fitness achieved over a fixed number of function evaluations. Initially, lower ρ values (e.g., $\rho = 0.001$ and $\rho = 0.01$) lead to a rapid decrease in fitness, suggesting quick convergence to a local optimum. For instance, at 1000 function evaluations, $\rho = 0.5$ achieves the lowest fitness of 8342, followed closely by $\rho = 0.01$ (8810) and $\rho = 0.001$ (8850).

As the number of function evaluations increases, higher ρ values tend to produce better solutions. As shown in Table 11, at 25,000 evaluations, $\rho = 0.98$ achieves the best fitness (7693), followed closely by $\rho = 0.5$ (7759). In contrast, very low ρ values (0.001 and 0.01) reach fitness values of 7805 and 7809, respectively, and show slower improvement.

ρ	**Fitness (25,000 Func Eval)**
0.001	7805
0.01	7809
0.5	7759
0.98	7693

Table 11: MMAS Fitness for TSP (Berlin52) at 25,000 Function Evaluations with different ρ values (corresponding to Figure 40)

Higher ρ values (such as 0.5 and 0.98) support better exploration, helping the algorithm avoid premature convergence and escape local optima. With enough function evaluations, this leads to higher-quality solutions. This is especially clear with $\rho = 0.98$, which, although slower to improve at first, ultimately achieves the best result.

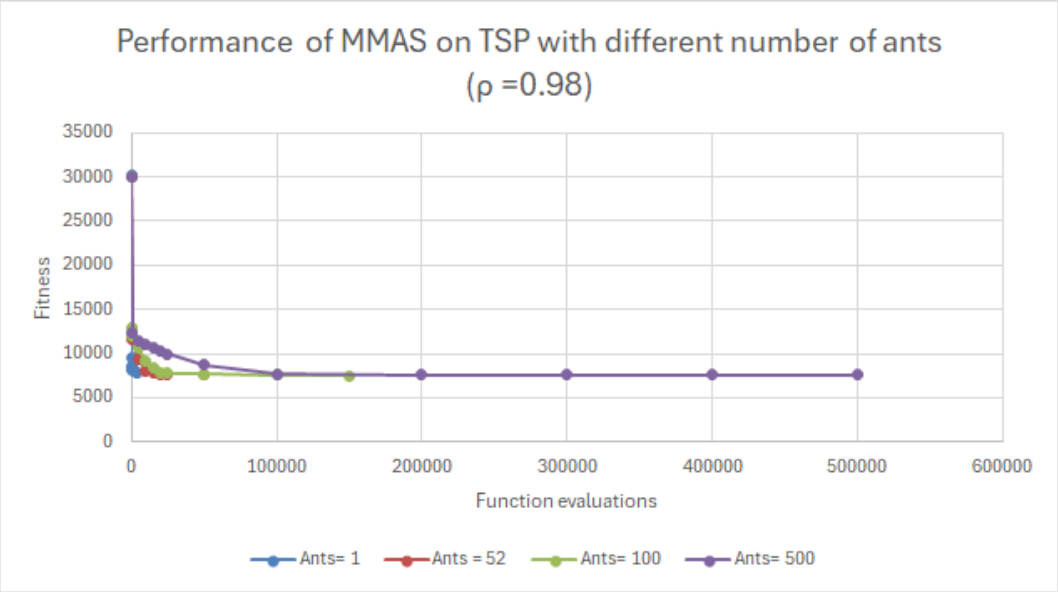

Figure 41: Averaged over 50 runs on the Berlin52 tour.

Figure 41 illustrates the performance of MMAS on TSP (Berlin52) problem, investigating the impact of the number of ants (m) on the fitness achieved, with the evaporation rate fixed at $\rho = 0.98$. It is important to note that the total number of function evaluations is directly correlated with the number of ants, which means that a higher number of ants leads to a greater number of function evaluations.

The results reveal a clear trade-off between convergence speed (in terms of function evaluations) and the quality of the final solution. Fewer ants, such as $m = 1$, achieve rapid initial convergence, reaching a fitness of 7920 within 10,000 function evaluations. However, this configuration appears to settle for a slightly higher local optimum compared to configurations with more ants.

Conversely, increasing the number of ants, particularly to $m = 52$ and $m = 100$, allows the algorithm to explore the solution space more thoroughly, leading to better final fitness values, though often requiring more function evaluations to reach them. As shown in Table 12, $m = 100$ ultimately achieves the best recorded fitness of 7542 (at 150,000 function evaluations), while $m = 52$ reaches 7652 (at 50,000 function evaluations). A very high number of ants, such as $m = 500$, significantly slows down initial convergence and does not necessarily lead to a better final solution, settling at 7619 fitness even with a much larger number of function evaluations (up to 500,000).

Number of Ants (m)	**Max Func Eval**	**Fitness**
1	10,000	7920
52	50,000	7652
100	150,000	7542
500	500,000	7619

Table 12: MMAS Fitness for TSP (Berlin52) with different numbers of ants (at max recorded function evaluations) ($\rho = 0.98$)

This suggests that an optimal number of ants, balancing exploration (more ants, more diverse paths) and exploitation (fewer ants, faster pheromone consolidation on promising paths), is crucial for

efficient optimization on the TSP. For the Berlin52 instance, $m = 52$ or $m = 100$ appear to offer a strong balance between solution quality and computational effort.

6.4.3 AS

TSP

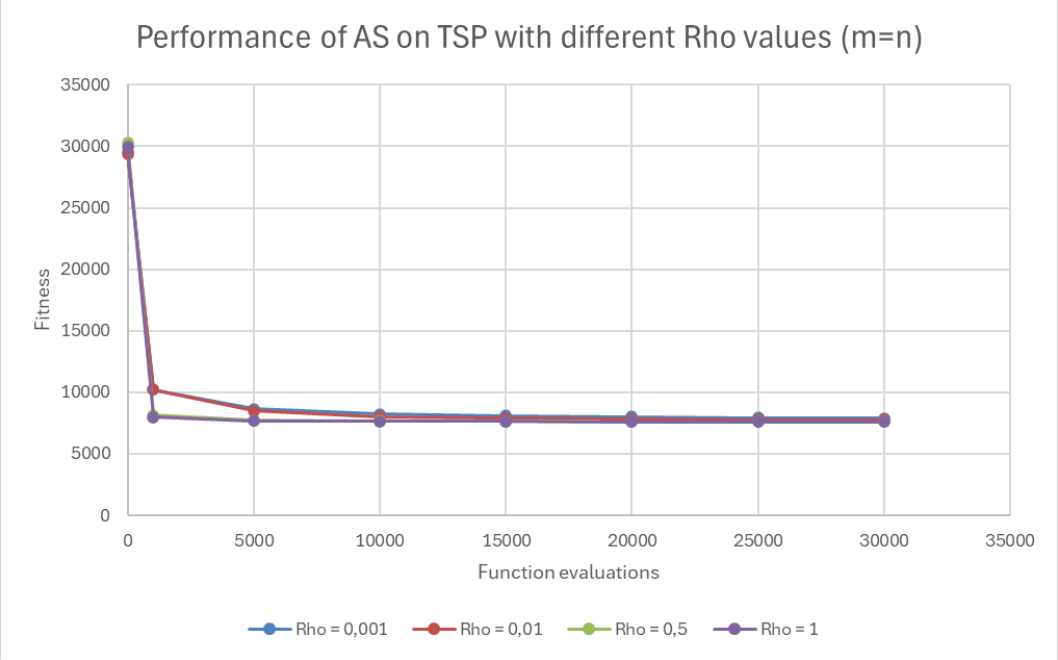

Figure 42: Averaged over 100 runs on the Berlin52 tour.

Figure 42 illustrates the performance of AS on the TSP (Berlin52) instance, examining the impact of varying the evaporation rate (ρ) on the fitness achieved over a fixed number of function evaluations. Similar to the MMAS results, the evaporation rate significantly affects AS's ability to find good solutions.

Consistently, higher ρ values demonstrate better performance. As shown in Figure 42 and detailed in Table 13, $\rho = 1$ and $\rho = 0.5$ achieve significantly better fitness values than the lower rates of $\rho = 0.001$ and $\rho = 0.01$. For instance, at 30,000 function evaluations, $\rho = 1$ yields the best fitness of 7590, closely followed by $\rho = 0.5$ at 7596. In contrast, $\rho = 0.01$ and $\rho = 0.001$ converge to higher fitness values of 7750 and 7880, respectively.

ρ	**Fitness (30,000 Func Eval)**
0.001	7880
0.01	7750
0.5	7596
1	7590

Table 13: AS Fitness for TSP (Berlin52) at 30,000 Function Evaluations with different ρ values (corresponding to Figure 42)

Comparing AS to MMAS on the TSP (Berlin52), it appears that AS, particularly with $\rho = 0.5$ and $\rho = 1$, can achieve better final fitness values within a similar range of function evaluations. For instance, AS with $\rho = 1$ reaches 7590 fitness, which is better than MMAS's best of 7693 with $\rho = 0.98$ within the 25,000-30,000 function evaluation range. This indicates that, for this specific TSP instance, the simpler AS algorithm can be highly competitive, and even slightly outperform MMAS, particularly when using a high evaporation rate that promotes effective exploration and discourages stagnation.

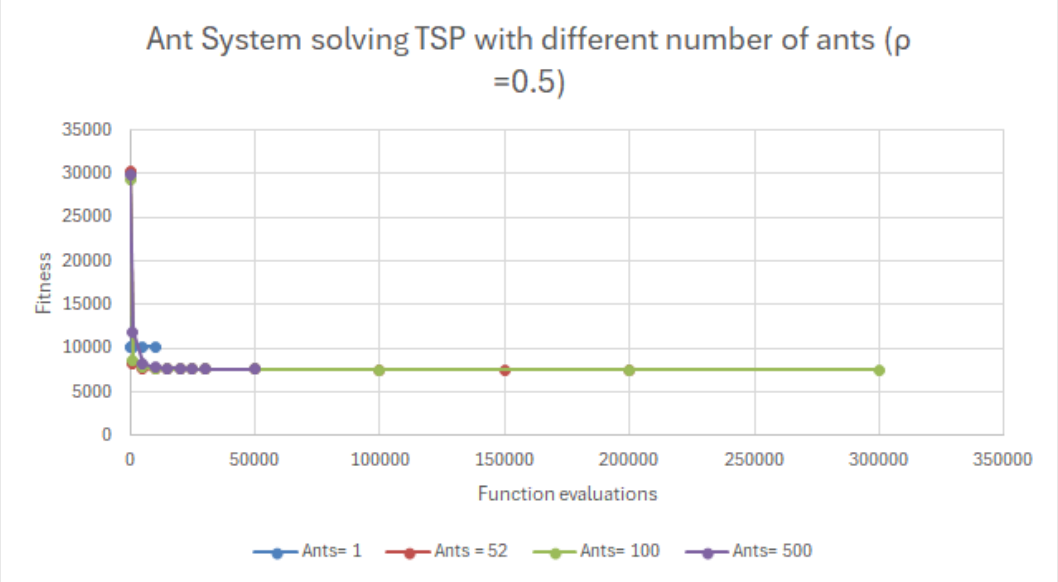

Figure 43: Averaged over 50 runs on the Berlin52 tour.

Figure 43 presents the performance of the AS on the TSP (Berlin52) problem, evaluating the impact of varying the number of ants (m), with the evaporation rate fixed at $\rho = 0.5$. As previously noted, an increase in the number of ants directly corresponds to a higher number of function evaluations per iteration, thereby influencing the overall computational cost to reach convergence.

The results highlight a critical dependency on the number of ants for effective optimization with AS. While a single ant ($m = 1$) shows rapid initial convergence, it quickly stagnates at a very high fitness value of 10,208, indicating an inability to effectively explore the solution space or escape poor local optima with the given ρ. In contrast to MMAS, where $m = 1$ achieved a significantly better fitness of 7920, demonstrating MMAS's greater robustness for very small colony sizes, likely due to its min-max pheromone limits.

Conversely, configurations with a sufficient number of ants (e.g., $m = 52$, $m = 100$, and $m = 500$) allow AS to achieve the best recorded fitness of 7542 for this problem instance. However, reaching this optimal fitness demands increasingly more function evaluations as the number of ants increases: $m = 52$ achieves 7542 fitness by 300,000 function evaluations, while $m = 100$ requires 390,000, and $m = 500$ requires 490,000 function evaluations.

Number of Ants (m)	**Max Func Eval**	**Fitness**
1	10,000	10,208
52	300,000	7542
100	390,000	7542
500	490,000	7542

Table 14: AS Fitness for TSP (Berlin52) with different numbers of ants (at max recorded function evaluations) ($\rho = 0.5$)

MMAS demonstrates better efficiency in reaching high-quality solutions. For example, MMAS with $m = 100$ achieves a fitness of 7542 in just 150,000 function evaluations, which is half the number of evaluations required by AS with $m = 52$ to reach 7542. Furthermore, while both algorithms show diminishing returns with excessively large colonies, AS with $m = 500$ also plateaus at 7542 fitness after 490,000 function evaluations, whereas MMAS with $m = 500$ reaches a slightly worse 7619 fitness after a comparable number of evaluations.

In conclusion, for AS on TSP (Berlin52) with $\rho = 0.5$, an optimal balance between solution quality and computational effort is achieved with $m = 52$, as it reaches the best fitness with significantly fewer function evaluations compared to larger ant populations. While AS can find solutions of comparable or slightly better quality than MMAS, MMAS often demonstrates greater efficiency in achieving high-quality solutions, particularly for intermediate colony sizes, and shows more robustness for single-ant configurations.

6.5 Comparison

Now that we have studied the individual performances of the implemented algorithms, we will proceed to a comparative analysis of their most effective configurations.

For the binary problems, OneMax and LeadingOnes, our comparison will include:

- $(1+1)$ EA.
- RLS.
- $(\mu+\lambda)$ EA with $\mu = 1$ and $\lambda = 10$.
- Simulated Annealing, using a cooling rate $\alpha = 0.9$ for OneMax and $\alpha = 0.1$ for LeadingOnes.
- 1-Ant algorithm, with an evaporation rate (ρ) of 1 for OneMax and 0.5 for LeadingOnes.
- MMAS, with an evaporation rate (ρ) of 1 for both OneMax and LeadingOnes.

For TSP on Berlin52, to ensure a fair comparative analysis, we will conduct a new experiment to evaluate the following implemented algorithms with their best configurations:

- $(1+1)$ EA solving TSP using 2-opt and 3-opt, with a probability of $\frac{1}{2}$ for choosing between them.
- $(1+1)$ EA solving TSP using Poisson-distributed Mutation (2-opt).
- Simulated Annealing, using a cooling rate $\alpha = 0.99$ and an initial temperature $T_0 = 1000$.
- Ant System, with an evaporation rate $\rho = 0.5$ and $m = 52$ ants.

- MMAS, with an evaporation rate $\rho = 0.98$ and $m = 100$ ants.

The experiment will be running each of these algorithms for 100,000 iterations, with results averaged over 100 runs. We will then evaluate their performance by how frequently they reach the known optimal solution for Berlin52, and by recording their maximum, average, and minimum fitness values.

6.5.1 OneMax

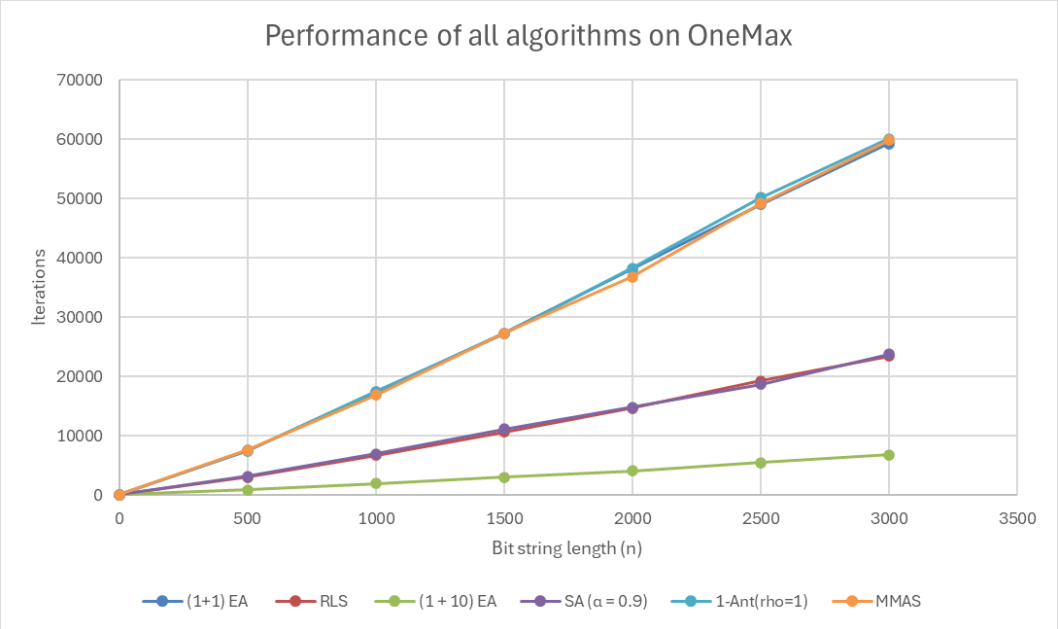

Figure 44: Averaged over 100 runs.

Figure 44 illustrates the comparative performance of all implemented algorithms on OneMax, displaying the number of iterations required to reach the optimum as the bit string length (n) increases.

Upon close examination, the results, also summarized in Table 15, reveal different performance levels. The $(\mu + \lambda)$ EA, specifically configured as $(1 + 10)$ EA, stands out as the most efficient algorithm by a significant margin. It consistently requires the fewest iterations to converge across all tested bit string lengths, taking only 6761 iterations for $n = 3000$.

Algorithm (Optimal Configuration)	**Iterations** ($n = 3000$)
$(1+1)$ EA	59261
RLS	23378
$(1+10)$ EA ($\mu = 1, \lambda = 10$)	6761
SA ($\alpha = 0.9$)	23721
1-Ant ($\rho = 1$)	60113
MMAS ($\rho = 1$)	59878

Table 15: Comparative Performance of Algorithms on OneMax at $n = 3000$

Following the $(1 + 10)$ EA, RLS and Simulated Annealing with $\alpha = 0.9$ demonstrate very competitive performance. For $n = 3000$, RLS converges in 23378 iterations, while SA requires 23721

iterations, indicating very similar efficiency between these two algorithms for OneMax.

The remaining algorithms, namely the $(1+1)$ EA, 1-Ant (with $\rho = 1$), and MMAS (with $\rho = 1$), show slower convergence. For $n = 3000$, the $(1+1)$ EA completes in 59261 iterations, MMAS in 59878 iterations, and 1-Ant in 60113 iterations. While these three algorithms perform similarly to each other, they are less efficient than the $(1+10)$ EA, RLS, and SA.

Overall, for OneMax, algorithms with stronger search strategies are the $(1+10)$ EA, RLS, and SA that tend to perform better than the basic (1 + 1) EA and ACO variants.

6.5.2 LeadingOnes

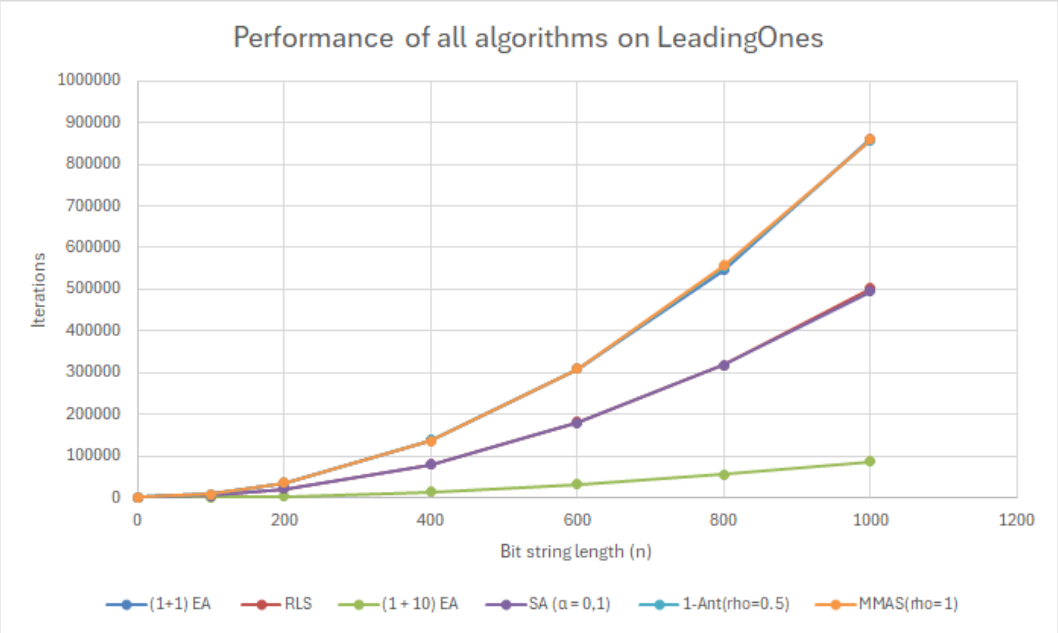

Figure 45: Averaged over 100 runs.

Figure 45 represents the comparative performance of the implemented algorithms on LeadingOnes, displaying the iterations required to reach the optimum for varying bit string lengths (n).

As summarized in Table 16, the $(1+10)$ EA again demonstrates exceptional performance, and is the most efficient algorithm for LeadingOnes. For $n = 1000$, the $(1+10)$ EA converges in 86150 iterations, significantly outperforming all other algorithms.

Algorithm (Optimal Configuration)	**Iterations** ($n = 1000$)
$(1+1)$ EA	869094
RLS	501693
$(1+10)$ EA ($\mu = 1, \lambda = 10$)	86150
SA ($\alpha = 0.1$)	494184
1-Ant ($\rho = 0.5$)	856987
MMAS ($\rho = 1$)	860221

Table 16: Comparative Performance of Algorithms on LeadingOnes at $n = 1000$

Following the $(1+10)$ EA, we have RLS and Simulated Annealing with $\alpha = 0.1$. SA slightly edges out RLS on LeadingOnes, requiring 494184 iterations for $n = 1000$ compared to RLS's 501693 iterations. Both algorithms are considerably slower than the $(1+10)$ EA but faster than the remaining algorithms.

The $(1+1)$ EA, 1-Ant (with $\rho = 0.5$), and MMAS (with $\rho = 1$) constitute the slowest group of algorithms for LeadingOnes. Their performance is similar: for $n = 1000$, $(1+1)$ EA requires 869094 iterations, 1-Ant requires 856987 iterations, and MMAS requires 860221 iterations.

In conclusion, for both OneMax and LeadingOnes, the $(1+10)$ EA proves to be the most effective algorithm, highlighting the benefits of a larger offspring population for exploration. RLS and SA also offer strong performance, particularly when the problem landscape allows for effective local search. Conversely, (1 + 1) EA and Ant Colony Optimization algorithms perform less efficient for these binary optimization problems.

6.5.3 TSP

Algorithms	**Optimal Reached (Times)**	**Min. Fitness**	**Avg. Fitness**	**Max. Fitness**
$(1+1)$ EA (2-opt & 3-opt, $p = 1/2$)	0	7696	8108	8679
$(1+1)$ EA (Poisson Mutation, 2-opt)	1	7542	8096	8943
SA ($\alpha = 0.99, T_0 = 1000$)	3	7542	8083	8800
AS ($\rho = 0.5, m = 52$)	53	7542	7553	7677
MMAS ($\rho = 0.98, m = 100$)	61	7542	7633	8036

Table 17: Performance of Algorithms on Berlin52 over 100,000 Iterations (based on 100 runs)

Table 17 presents the results of the comparative experiment for solving the TSP (Berlin52) instance using the implemented algorithms, each running for 100,000 iterations and averaged over 100 runs. The analysis focuses on their ability to reach the known optimal solution (7542), along with their minimum, average, and maximum fitness achieved.

The results clearly indicate a significant performance difference between the Ant Colony Optimization (ACO) algorithms and the other metaheuristics tested. Both Ant System (AS) and Max-Min Ant System (MMAS) demonstrate exceptional capabilities in finding the optimal solution for Berlin52. MMAS successfully reached the optimal tour 61 times out of 100 runs, while AS achieved it 53 times. This highlights their effectiveness on this complex combinatorial optimization problem.

In contrast, Simulated Annealing showed limited success, finding the optimal solution only 3 times. The two variants of the $(1+1)$ Evolutionary Algorithm performed even worse, with the Poisson-distributed Mutation variant reaching the optimal solution only once, and the 2-opt and 3-opt combined variant failing to find it at all within the given iteration budget.

Beyond optimal solution frequency, the average and maximum fitness values further reinforce the superiority of ACO algorithms. AS achieved an impressive average fitness of 7553 and a maximum (worst-case) fitness of 7677, indicating highly consistent and high-quality solutions. MMAS also performed strongly with an average fitness of 7633 and a maximum fitness of 8036. While both achieved the true minimum fitness of 7542 in their best runs, AS's tighter distribution of results

(lower average and maximum fitness) suggests slightly more stable convergence to near-optimal solutions.

Conversely, SA and both $(1 + 1)$ EA variants yielded significantly higher average and maximum fitness values (ranging from approximately 8000 to 8943), implying they frequently converged to much worse local optima and were less reliable in finding high-quality solutions for the TSP.

In conclusion, for the TSP (Berlin52) instance, Ant Colony Optimization algorithms, particularly AS and MMAS, are demonstrably more effective and robust than Evolutionary Algorithms or Simulated Annealing in terms of both consistently finding optimal solutions and achieving high-quality average performance.

7 Future work

Throughout our work on this project, several potential extensions and improvements were considered. These included exploring additional problem domains, enhancing the implemented visualization more, incorporating AI-related features, implementing further algorithmic refinements, and conducting additional experiments. However, to stay within the project's time constraints and scope, it was necessary to prioritize the current implementation. The following sections present some directions for future work that could enhance and expand the project.

7.1 Visualization

Effective visualization of algorithm performance has been a consistent focus throughout this project. While three distinct visualization techniques were developed for binary problems, and an optimized technique for TSP was implemented, opportunities for further enhancement remain. For instance, the visualization optimized for 2-opt and 3-opt mutations in TSP solutions is effective. For Ant Colony Optimization variants, the visualization of pheromone trails has been implemented to represent their unique search processes. However, despite this initial implementation, observing the solution evolution for ACO algorithms through the current UI could be further enhanced to provide more insight.

7.2 Algorithms

While the implemented algorithms consistently demonstrate expected behavior, opportunities for further refinement and code optimization exist. For instance, several algorithms designed for the TSP currently exceed 200 lines of code. This presents an opportunity for significant refactoring to enhance modularity, readability, and maintainability, potentially leading to more efficient implementations in the future.

7.3 AI-Assisted Analysis

Integration of Artificial Intelligence models is continuously expanding. While developing this framework, we thought of an idea with the aim of modernizing the framework's capabilities and providing enhanced insights regarding the performance of all algorithms. Specifically, the integration of a generative AI model, such as Google Gemini, could serve to provide details of the graphical outputs generated by the framework.

Consider a scenario where a user executes multiple batches of experiments. Upon completion, a dedicated "Feedback from AI" button could be made available. Upon a click of this button, the underlying data points of the generated graphs, along with contextual information, could be transmitted to the AI model via an API key. The AI's analytical response could then be presented to the user. This functionality would empower users, including those with a more basic background in metaheuristics, to gain a good understanding of the data trends and algorithmic behaviors illustrated by the visualizations.

8 Conclusion

This project successfully developed and implemented a software framework dedicated to the visualization and evaluation of nature-inspired optimization metaheuristics. The intuitive graphical user interface (GUI) enables users to configure and execute experimental schedules, offering selections across various combinatorial optimization problems and a suite of algorithms with configurable parameters.

The framework supports pseudo-boolean functions including OneMax and LeadingOnes, and a set of Traveling Salesman Problem (TSP) instances from the search space of permutations, specifically: a280, berlin52, eil101, kroa101, lin318, pcb442, pr1002, rat99, rd400, and st70. To provide insightful observations, the framework integrates various visualization techniques: graph representations showing iteration over fitness, bitstring snapshots illustrating solution evolution, and a Boolean hypercube mapping for binary problems. For TSP, an optimized 2D map visualization tracks tour path improvements, including highlighting 2-opt and 3-opt changes, augmented by a real-time statistics panel. Pheromone trail visualization is also provided for Ant Colony Optimization algorithms. The implemented algorithms include the $(1+1)$ Evolutionary Algorithm (EA), Randomized Local Search (RLS), and $(\mu+\lambda)$ EA, alongside Simulated Annealing (SA), 1-Ant, and Max-Min Ant System (MMAS) for binary problems. Specialized TSP algorithms include $(1+1)$ EA variants utilizing 2-opt and 3-opt mutations, (1+1) EA solving TSP using Poisson-distributed Mutation, SA with 2-opt, Ant System (AS), and MMAS. Performance figures are extracted and used to evaluate these algorithms through extensive experiments, providing insights into their performance.

The experimental investigations yielded clear insights into the varying effectiveness of the metaheuristics across problem domains. For the pseudo-boolean functions, OneMax and LeadingOnes, the $(\mu+\lambda)$ EA, specifically the $(1+10)$ configuration, consistently emerged as the most efficient algorithm, achieving significantly faster convergence. Local Search (RLS) and Simulated Annealing (SA) also showed a competitive performance, closely following the $(1+10)$ EA. Conversely, the standard $(1+1)$ EA, 1-Ant, and Max-Min Ant System (MMAS) demonstrated slower convergence rates for these problems. Their performance, while similar to each other, was notably less efficient than the leading algorithms. This highlights that for pseudo-boolean functions, algorithms leveraging larger offspring populations or local search strategies tend to outperform simpler EAs and ACO variants.

For the Traveling Salesman Problem on Berlin52, we observed that Ant Colony Optimization algorithms, namely Ant System and Max-Min Ant System, proved to be more effective and robust than Evolutionary Algorithms or Simulated Annealing. MMAS achieved the optimal tour in 61 out of 100 runs, with AS following closely at 53 optimal solutions. Furthermore, AS demonstrated superior consistency with the lowest average and maximum fitness values among all algorithms, indicating more stable convergence to high-quality solutions. Conversely, SA and the $(1+1)$ EA variants showed limited success in finding optimal solutions and frequently converged to suboptimal tours, yielding significantly higher average and maximum fitness values. This highlights the effectiveness of ACO algorithms' pheromone-based mechanisms for the complex combinatorial structure of TSP.

Building upon the current framework, several key areas for future work have been identified. Significant improvements to visualization capabilities can be done, especially for ACO algorithms on TSP. From an algorithmic perspective, opportunities for refactoring and code optimization exist, partic-

ularly for complex TSP algorithms, to enhance modularity and maintainability. Finally, integrating AI models, such as generative AI, into the framework can be a plus where AI-assisted analysis could explain complex graphical outputs, providing accessible insights to users regardless of their background in metaheuristics gaining a better understanding of algorithmic performance.

9 References

References

[1] Ahmed Mrabet, (02 October 2023). Solving Travelling Salesman Problem With Variable Neighborhood Search. *Medium*. Accessed: 24 March 2025. Available at https://shorturl.at/cPknW.

[2] Frank Neumann and Carsten Witt (2010). Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity. *Bioinspiredcomputation*, Accessed: 25 March 2025 . Available at http://www.bioinspiredcomputation.com/.

[3] Christian Blum, Andrea Roli (01 September 2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Digital Library*. Page 269-272. Accessed: 25 March 2025 . Available at https://dl.acm.org/doi/10.1145/937503.937505.

[4] Fernando Peres and Mauro Castelli (21 June 2021). Combinatorial Optimization Problems and Metaheuristics: Review, Challenges, Design, and Development. *MDPI*. Page 3-4. Accessed: 25 March 2025 . Available at https://www.mdpi.com/2076-3417/11/14/6449.

[5] A.E. Eiben and J.E. Smith (2015). Introduction to Evolutionary Computing .Accessed: 26 March 2025. Available at https://ndl.ethernet.edu.et/bitstream/123456789/88556/1/2015_Book_IntroductionToEvolutionaryComp.pdf.

[6] Marco Dorigo and Thomas Stützle (2004). Ant Colony Optimization. Accessed: 26 March 2025. Available at https://www.researchgate.net/publication/36146886_Ant_Colony_Optimization.

[7] Thomas Jansen (2013). Analyzing Evolutionary Algorithms: The Computer Science Perspective. Accessed: 27 March 2025. Available at https://link-springer-com.proxy.findit.cvt.dk/book/10.1007/978-3-642-17339-4.

[8] *stats_noob* (Feb 3, 2022). Counting the Number of Paths in the "Travelling Salesman Problem". *Stack Exchange*. Accessed: 27 March 2025. Available at https://math.stackexchange.com/questions/4373325/counting-the-number-of-paths-in-the-travelling-salesman-problem.

[9] *PengYM*. Permutations.*pymoo*. Accessed: 27 March 2025. Available at https://pymoo.org/customization/permutation.html.

[10] Timothy Lanzone and Robert A. Bosch. Travelling salesman problem.*Wikipedia*. Accessed: 27 March 2025. Available at https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[11] Marco Chiarandini (2020). Local Search for Traveling Salesman Problem.*University of Southern Denmark*. Accessed: 27 March 2025. Available at https://dm865.github.io/assets/dm865-tsp-ls-handout.pdf.

[12] 2-opt.*Wikipedia*. Accessed: 28 March 2025. Available at https://en.wikipedia.org/wiki/2-opt.

[13] Kamil Rocki and Reiji Suda (July 2012). Accelerating 2-opt and 3-opt Local Search Using GPU in the Travelling Salesman Problem.*ResearchGate*. Page 489-490. Accessed: 28 March 2025. Available at https://www.researchgate.net/publication/229476110_Accelerating_2-opt_and_3-opt_Local_Search_Using_GPU_in_the_Travelling_Salesman_Problem.

[14] Jingyan Sui, Shizhe Ding, Ruizhi Liu, Liming Xu (2021). Learning 3-opt heuristics for traveling salesman problem via deep reinforcement learning. Page 4-6. Accessed: 28 March 2025. Available at https://proceedings.mlr.press/v157/sui21a/sui21a.pdf.

[15] André Karge. Traveling Salesman Problem Introduction. Accessed: 28 March 2025. Available at https://www.uni-weimar.de/fileadmin/user/fak/medien/professuren/Intelligente_Softwaresysteme/Downloads/Lehre/SBSE/LabClass/SS19/ex01_introduction.html.

[16] Carsten Witt. Tight Bounds on the Optimization Time of a Randomized Search Heuristic on Linear Functions. Page 295-296. Accessed: 31 March 2025. Available at http://dx.doi.org/10.1017/S0963548312000600.

[17] Suntje Bottcher, Benjamin Doerr, and Frank Neumann. Optimal Fixed and Adaptive Mutation Rates for the LeadingOnes Problem. Accessed: 31 March 2025. Available at https://link.springer.com/chapter/10.1007%2F978-3-642-15844-5_1.

[18] Yu Shan Zhang and Zhi Feng Hao (2010). Runtime Analysis of (1+1) Evolutionary Algorithm for a TSP Instance. Page 298. Accessed: 01 April 2025. Available at https://link.springer.com/chapter/10.1007/978-3-642-17563-3_36.

[19] Jens Scharnow, Karsten Tinnefeld and Ingo Wegener (2004). The Analysis of Evolutionary Algorithms on Sorting and Shortest Paths Problems. Page 350-353. Accessed: 08 April 2025. Available at https://link.springer.com/article/10.1023/B:JMMA.0000049379.14872.f5.

[20] Denis Antipov and Benjamin Doerr (2020). A Tight Runtime Analysis for the $(\mu + \lambda)$ EA. Page 1059. Accessed: 08 April 2025. Available at https://link.springer.com/article/10.1007/s00453-020-00731-5.

[21] Gerhard Reinelt. TSPLIB 95. Page 9-11. Accessed: 08 April 2025. Available at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf.

[22] Carola Doerr and Johannes Lengler. OneMax in Black-Box Models with Several Restrictions. Page 7-11. Accessed: 09 April 2025. Available at https://as.inf.ethz.ch/people/members/lenglerj/ElitistOM_Algorithmica.pdf.

[23] Maxim Buzdalov and Arina Buzdalova. Can OneMax Help Optimizing LeadingOnes using the EA+RL Method?. Accessed: 09 April 2025. Available at https://ctlab.itmo.ru/~mbuzdalov/papers/cec15-leadingones-onemax.pdf.

[24] Anne Auger and Benjamin Doerr. Theory of randomized search heuristics. Page 171-174. Accessed: 10 April 2025. Available at https://books.google.dk/books?hl=da&lr=&id=Y2OtGyL6gyUC&oi=fnd&pg=PR7&dq=Foundations+and+recent+

developments,+volume+1.+World+Scientific,+2011.&ots=cIhosfPhhD&sig=gnIDHrQa2qlg1otNtsVNHj7VMbY&redir_esc=y#v=onepage&q=onemax&f=false.

[25] Vincent A. Cicirello (2020). Optimizing the Modified Lam Annealing Schedule. Page 1-2. Accessed: 10 April 2025. Available at https://www.cicirello.org/publications/eai.16-12-2020.167653.pdf.

[26] P. H. Gunawan and Iryanto (2022). Simulated Annealing – 2 Opt Algorithm for Solving Traveling Salesman Problem. Page 44-45. Accessed: 10 April 2025. Available at https://computingonline.net/files/journals/1/archieve/IJC_2023_22_1_06.pdf.

[27] Frank Neumann, Dirk Sudholt and Carsten Witt (2008). Analysis of different MMAS ACO algorithms on unimodal functions and plateaus. Accessed: 11 April 2025. Available at https://www.researchgate.net/publication/47843232_Analysis_of_Different_MMAS_ACO_Algorithms_on_Unimodal_Functions_and_Plateaus

[28] Frank Neumann and Carsten Witt (2007). Runtime Analysis of a Simple Ant Colony Optimization Algorithm. Page 228. Accessed: 11 April 2025. Available at https://link.springer.com/article/10.1007/s00453-007-9134-2

[29] Thomas Stützle and Holger H. Hoos. MAX –MIN Ant System. Page 898-901. Accessed: 11 April 2025. Available at https://www-sciencedirect-com.proxy.findit.cvt.dk/science/article/pii/S0167739X00000431

[30] Patrick Briest, Dimo Brockhoff, Bastian Degener, Matthias Englert, Christian Gunia, Oliver Heering, Michael Leifhelm, Kai Plociennik, Heiko Roglin, Andrea Schweer, Dirk Sudholt, Stefan Tannenbaum, Thomas Jansen and Ingo Wegener (2003). FrEAK (Free Evolutionary Algorithm Kit). Accessed: 12 April 2025. Available at https://sourceforge.net/projects/freak427/files/FrEAK/0.1/

[31] Emil Lundt Larsen (2018). Visualizing and Evaluating the Working Principles of Nature-Inspired Optimization Metaheuristics. Accessed: 12 April 2025. Available at https://findit.dtu.dk/en/catalog/5b45e5d35010df018032450d

[32] Marcus Pihl, Jakob Kildegaard Hansen, and Victor Winther Saldeen (2024). Visualizing and evaluating the working principles of nature-inspired optimization metaheuristics. Accessed: 12 April 2025. Available at https://findit.dtu.dk/en/catalog/668c7fe3715defdc488b7202

[33] Mermaid Live Editor. Class diagram for the visualization package. Created: 30 April 2025. Available at Viewing the live class diagram of the visualization package

10 Appendix

10.1 Decisions made while development

This list does not represent all the decisions, but only the most essential ones.

1. **Batch Management Buttons**
   Implemented new buttons to manage batches (Add, Remove, Copy), inspired by Emil's implementation.
2. **Step Highlighting**
   Added a yellow line to visually indicate the steps reached by the user, improving clarity and navigation. Inspired by Emil as well.
3. **Info Message Placement**
   Positioned important system messages at the top center of the app to ensure high visibility of critical information.
4. **Batch Count Display**
   Displayed the number of created batches to keep users informed.
5. **Search Space Locking**
   When creating a batch and selecting a search space (e.g., bitstrings), the selection becomes fixed until the batches are run.

   Reason: The application uses two different canvases for displaying content. Allowing multiple search spaces simultaneously could cause frequent and confusing view changes. This limitation ensures a consistent display experience by enforcing the use of a single search space at a time.
6. **Display Options for Binary Problems**
   For binary problem types:
   - Available displays: Graph (default), Boolean Hypercube, Bitstrings.
   - Disabled: TSP visualization (not applicable) and Pheromone trails.
7. **Display Options for TSP Problems**
   For TSP problem types:
   - Default display: TSP graph.
   - Optional: Pheromone trail (only for ACO variants).
   - Disabled: Graph, Boolean Hypercube, and Bitstrings (not relevant).
8. **Tour Name as Dimension**
   When TSP is selected, the tour name is shown as the "dimension" to remind users of the selected tour.
9. **Stopping Condition for TSP Algorithms**
   When "optimum reached" is the stopping condition:
   - Most algorithms stop after approximately 10,000 iterations.

- **Exceptions:** `OnePlusOneEA_Poisson_TSP` and `SimulatedAnnealing_TSP`. These can get stuck in loops or oscillations.
- A custom stopping condition was added: if the algorithm fails to find a new move different from the last within 3 tries, the stopping condition variable is triggered to end execution.

10.2 getOptionsForStep

MainInteraction.java

```
private List<String> getOptionsForStep(int step) {
    switch (step) {
        case 1:
            return List.of("Bit strings", "Permutation");
        case 2:
            // Return problems based on selected search space
            String searchSpace = selectedOptions.stream()
                    .filter(s -> s.startsWith("Search Space:"))
                    .findFirst()
                    .orElse("Search Space: Bit strings");

            if (searchSpace.contains("Bit strings")) {
                return List.of("Leading ones", "One max");
            } else if (searchSpace.contains("Permutation")) {
                return List.of("Traveling Salesman Problem");
            }
            return List.of();
        case 3:
            // Return algorithms based on selected search space and problem
```

10.3 getOptimalTourLengthForTspFile

ScheduleInteraction.java

```java
/**
 * Returns the known optimal tour length for common TSP instances.
 *
 * @param tspFile The TSP file name
 * @return The optimal tour length if known, -1 otherwise
 */
private double getOptimalTourLengthForTspFile(String tspFile) {
    // Clean the filename (remove path and extension)
    String filename = tspFile.toLowerCase();
    if (filename.contains(s:"/") || filename.contains(s:"\\")) {
        filename = filename.substring(Math.max(
            filename.lastIndexOf(ch:'/'),
            filename.lastIndexOf(ch:'\\')
        ) + 1);
    }
    if (filename.contains(s:".")) {
        filename = filename.substring(beginIndex:0, filename.lastIndexOf(ch:'.'));
    }

    // Known optimal tour lengths for common instances
    switch (filename) {
        case "berlin52": return 7542.0;
        case "eil101": return 629.0;
        case "lin318": return 42029.0;
        case "rd400": return 15281.0;
        default:
            System.out.println("Unknown optimal tour length for: " + filename);
            return -1; // Unknown or no optimal value
    }
}
```

10.4 FitnessBoundReached

FitnessBoundReached.java

```java
/**
 * Creates a stopping condition that stops when a fitness bound is reached.
 *
 * @param fitnessTarget The target fitness value
 * @param isMinimizationProblem True if the problem is a minimization problem (like TSP)
 */
public FitnessBoundReached(int fitnessTarget, boolean isMinimizationProblem) {
    this.fitnessTarget = fitnessTarget;
    this.isMinimizationProblem = isMinimizationProblem;
}

@Override
public boolean isMet(int currentFitness, int dimension, int generation) {
    boolean isBoundReached;

    if (isMinimizationProblem) {
        // For minimization problems (like TSP), lower values are better
        isBoundReached = currentFitness <= fitnessTarget;
    } else {
        // For maximization problems (like OneMax), higher values are better
        isBoundReached = currentFitness >= fitnessTarget;
    }

    if (isBoundReached) {
        System.out.println("FitnessBoundReached condition met! " +
                           "Current fitness: " + currentFitness +
                           ", Target: " + fitnessTarget +
                           (isMinimizationProblem ? " (minimization)" : " (maximization)"));
    }

    return isBoundReached;
}
```

10.5 SA's mechanism to escape oscillation

SimulatedAnnealing_TSP.java

```java
// Try up to 3 times to find a move that's different from the last one
for (int attempt = 0; attempt < 3; attempt++) {
    newSolution.apply2OptMove();

    // Generate a hash of the move to compare with the last one
    String moveHash = "";
    if (newSolution.getEdge1From() != null) {
        moveHash = newSolution.getEdge1From().getCityId() + "-" +
                   newSolution.getEdge1To().getCityId() + "-" +
                   newSolution.getEdge2From().getCityId() + "-" +
                   newSolution.getEdge2To().getCityId();
    }

    // If this move is different from the last one, use it
    if (!moveHash.equals(lastMoveHash) || attempt == 2) {
        lastMoveHash = moveHash;
        break;
    }
}

// Avoid exact repeats by checking tour length
if (newTourLength == lastMoveTourLength) {
    // Skip this evaluation
    continue;
}
lastMoveTourLength = newTourLength;
```

10.6 EA

OnePlusOneEA.java

```java
@Override
public void runAlgorithm() {
    while (!stoppingMet && !Thread.currentThread().isInterrupted()) {
        // Create offspring by mutating parent
        String candidate = searchSpace.mutate(bitString);
        int candidateFitness = problem.evaluate(candidate);
        functionEvaluations++;

        // Accept candidate if fitness is at least as good
        if (candidateFitness >= bestFitness) {
            bitString = candidate;
            bestFitness = candidateFitness;
            noImprovementCount = 0;
        } else {
            noImprovementCount++;
        }

        // Notify visualization components about progress
        if (generation % 100 == 0 && listener != null) {
            RunInfo info = new RunInfo(generation, bestFitness, bitString);
            listener.accept(info);
        }

        // Check stopping condition
        if (stoppingCondition.isMet(bestFitness, n, generation)) {
            stoppingMet = true;
            break;
        }

        generation++;
    }
}
```

10.7 2-opt method

TspSolution.java

```java
/**
 * Apply a 2-opt move to potentially improve the tour.
 * A 2-opt move removes two edges and reconnects the tour in a different way.
 */
public void apply2OptMove() {
    int size = tour.size();
    double bestImprovement = -0.1;  // Use threshold to avoid floating point issues
    int bestI = -1, bestJ = -1;

    // Try all possible combinations of edges
    for (int i = 0; i < size - 2; i++) {
        City city1 = tour.get(i);
        City city2 = tour.get(i + 1);
        double dist1 = city1.distanceTo(city2);

        for (int j = i + 2; j < size; j++) {
            City city3 = tour.get(j);
            City city4 = tour.get((j + 1) % size);
            double dist2 = city3.distanceTo(city4);

            // Calculate new distances if we reconnect
            double newDist1 = city1.distanceTo(city3);
            double newDist2 = city2.distanceTo(city4);

            // Calculate improvement
            double improvement = (dist1 + dist2) - (newDist1 + newDist2);

            // Keep track of best improvement
            if (improvement > bestImprovement) {
                bestImprovement = improvement;
                bestI = i;
                bestJ = j;
            }
        }
    }
```

10.8 3-opt method

TspSolution.java

```java
public void apply3OptMove() {
    previousTour = new ArrayList<>(tour);
    int size = tour.size();

    // Select three random indices
    int i = random.nextInt(size);
    int j = random.nextInt(size);
    int k = random.nextInt(size);

    // Make sure they're all different and not adjacent
    while (j == i || j == (i + 1) % size || i == (j + 1) % size) {
        j = random.nextInt(size);
    }

    while (k == i || k == j || k == (i + 1) % size || k == (j + 1) % size ||
           i == (k + 1) % size || j == (k + 1) % size) {
        k = random.nextInt(size);
    }

    // Sort indices
    if (i > j) {
        int temp = i;
        i = j;
        j = temp;
    }
    if (j > k) {
        int temp = j;
        j = k;
        k = temp;
    }
    if (i > j) {
        int temp = i;
        i = j;
        j = temp;
    }

    // Get cities at the cut points
    City a = tour.get(i);
    City b = tour.get((i + 1) % size);
    City c = tour.get(j);
    City d = tour.get((j + 1) % size);
    City e = tour.get(k);
    City f = tour.get((k + 1) % size);

    // Try all possible 3-opt moves
    ThreeOptCase bestCase = ThreeOptCase.CASE_0;
    double bestGain = 0;

    for (ThreeOptCase optCase : ThreeOptCase.values()) {
        if (optCase == ThreeOptCase.CASE_0) continue;

        double gain = calculate3OptGain(a, b, c, d, e, f, optCase);
        if (gain > bestGain) {
            bestGain = gain;
            bestCase = optCase;
        }
```