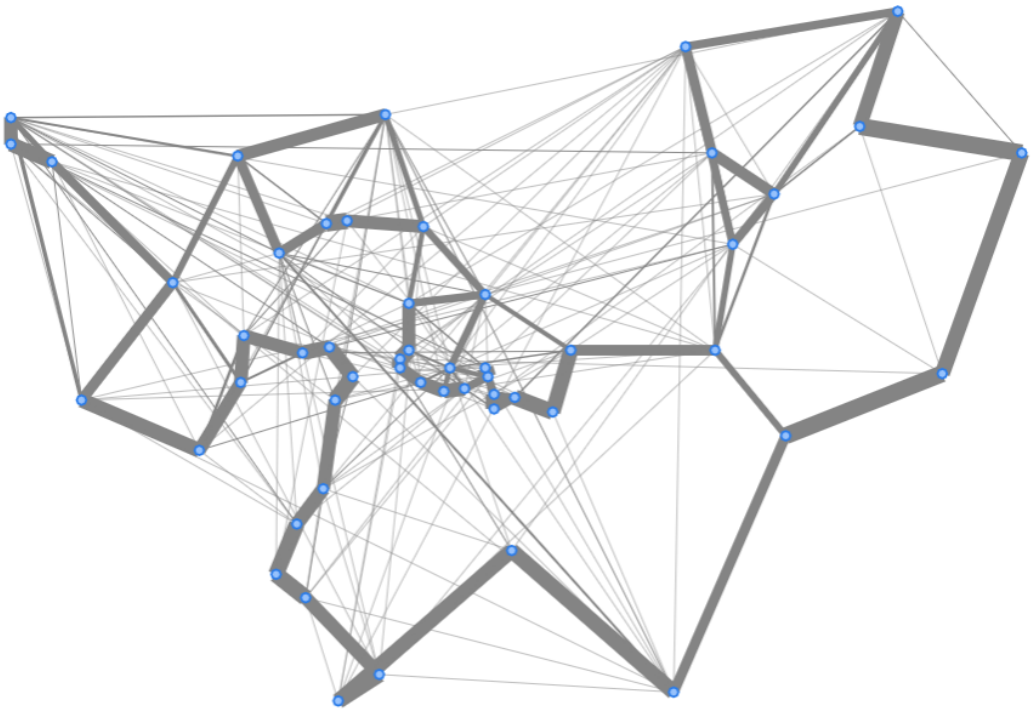
Visualizing and Evaluating the Working Principles of Nature-Inspired Optimization Metaheuristics. A Bachelor Project.

Phillip Løjmand s174252

May 2020



Phillip Løjmand

1 Abstract

We implement an extensible framework for visualizing and evaluating the working principles of nature-inspired optimization metaheuristics. Most note-ably the metaheuristics Simulated Annealing, Ant Colony Optimization and an Evolutionary Algorithm is implemented and used to solve The Travelling Salesman problem. We cover the development process of building the framework on the web stack with a frontend and backend using appropriate technologies like Nginx, React JS, Fast-CGI and more. The framework implements the algorithms through the use of cloud computing in order to speed up run-times and enhance ease of accessibility. Deployment and load-balancing is setup on virtual machines through the cloud service, Microsoft Azure. We implement bit-string benchmark problems as well and evaluate the metaheuristics with respect to their appropriateness and ability to solve given problems, but mainly TSP. Finally we show that our implementation of Simulated Annealing and Ant Colony Optimization can efficiently find solutions of high quality, or even the optimal solution, on certain TSPLib instances.

2 Glossary

ACO	Ant Colony Optimization.
API	Application programming interface.
Black-Box	Used to indicate a system where we do not know how the output is generated; Related in this project to a problem and its fitness value generation.
EA	Evolutionary Algorithm.
FCGI	Fast-CGI; Fast Common Gateway Interface.
Hill-climber	Local search optimization algorithm that iteratively finds better solutions.
HTML DOM	The object model for HTML.
MMAS	Max-Min Ant System.
MoSCoW	"Must have", "Should have", "Could have", "Won't have" prioritization technique.
NP-Hard	Problems that cannot be solved in polynomial time.
RLS	Randomized Local Search.
TSP	The Travelling Salesman Problem.

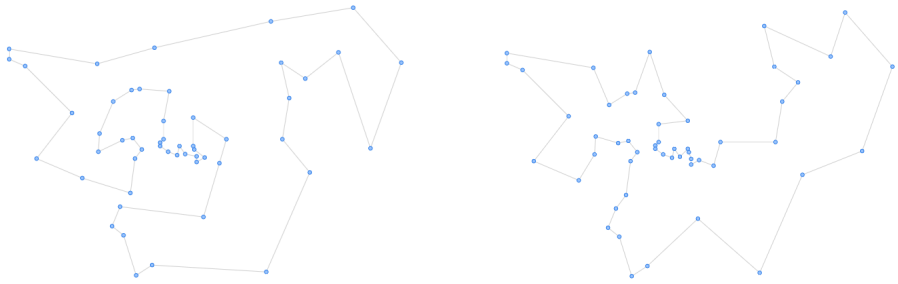
Contents

3 Introduction

For some optimization problems we do not have algorithms that can solve them efficiently by always reaching the optimal solution with an acceptable time complexity. Like the Travelling Salesman Problem, these problems can be NP-Hard and as such requires exponential time to guarantee an optimal solution with known methods [20].
This motivates the idea of alternative methods that may not necessarily find the optimal solution but one that is good enough for the specific use-case and hopefully comes close to the optimal.

3.1 Local search

Local search is a simple method for coming up with such solutions. Generally you start with any solution in the search space of the optimization problem. You then define strategies for how to change the solution in order to improve it, and apply those strategies until you no longer can improve the solution. At this point a local optima is reached. Such strategies are sometimes also known as heuristics and are defined on the given problem [36]. Local search can be efficient for the solution of certain problems, in fact, it is not unlike the pivot operation in the Simplex method. The idea is employed by many deterministic greedy algorithms that do indeed reach an optimal solution in an efficient manner. The issue with greedy local search methods however, is the fact that there may be better optima different from the attained local optimum [18][35]. An example of this is a mountain-climber aiming to climb the tallest mountain. His idea is simple, always move in an upwards direction. This strategy is a heuristic but not a very good one, he will reach a peak but very likely this peak will not be the tallest one in the mountain range, or even close to. And if he does not start in the Himalayas, he will certainly be far from reaching the optimal solution, Mt. Everest.



(a) A local minimum. Length: 8587.

(b) Global minimum. Length: 7544.

Figure 1: Comparison of local and global minimum for a TSP instance where local search works to make sure there are no crossing edges in the tour.

3.2 What is a metaheuristic?

We try to overcome some of these limitations with metaheuristics. What mainly sets metaheuristics apart, is the fact that they are not problem-specific but merely strategies that guide the search process in order to more efficiently explore the search space in hopes of attaining a near optimal solution. Metaheuristics are therefore an abstraction level higher, they describe concepts that fulfill desired properties in this regard. Instead of defining exactly how to generate or change a solution, a metaheuristic may instead describe what to do with neighboring solutions based on its fitness evaluation, or how to use information from previously constructed solutions

to construct a new one. So when implementing such a metaheuristic one needs to interpret this and for instance define what the neighboring solutions and fitness function is. [11].

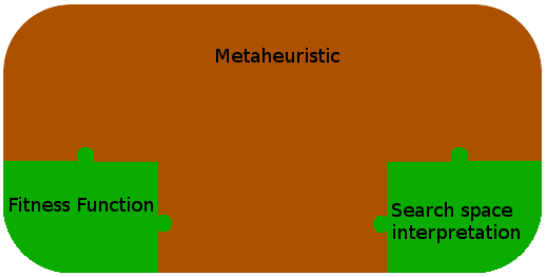


Figure 2: Problem specific pieces illustrates the abstraction. The formulation of a metaheuristic does not define an algorithm on its own.

In the above example of the mountain-climber, neighboring solution could be the possible steps, and the fitness function is the height at that point. However, it does not have to be that way, the metaheuristic is not invalidated if the mountain-climber jumps instead of taking steps, we just change, what is the neighboring solutions.
Instead the job of the metaheuristic can be to choose among the neighboring solutions.
For instance, we could choose to most of time walk upwards but sometimes allow walking downwards. This means we would not get stuck in a small peak. The metaheuristic we apply would in this case by itself not be concerned with notions such as "walking" or "downwards", those are search-space and problem specific. Instead, this would be the result of interpreting a more general formulation like, "transition to the best neighboring solution most of the time but sometimes also transition to a worse solution". This idea is also the foundation of the metaheuristic, Simulated Annealing, which we will later explore in depth.
While we mainly introduced the concept of metaheuristics through local search methods it is important to distinguish these from constructive methods where solutions are generated from scratch based on some information. An example of this is Ant Colony Optimization which we will also explore in depth [11].

3.2.1 Nature-inspired metaheuristics

Coming up with an idea for a new metaheuristic that efficiently explores the search space is not an easy feat. We need a good idea that has promising characteristics. Since metaheuristics are non problem specific and can be applied on a broad range of problem classes, we can do the reverse and look to existing good solutions to certain problems and draw inspiration from these. As it turns out, many of such ideas can be drawn from nature [40].
An example of this is the Evolutionary algorithm simulating evolution. In nature, evolution optimize species to be more fit for survival and more likely to pass on offspring. Within that context, the fitness function is the potential to pass on offspring, and the neighboring solutions is the offspring, or the mutated genes, describing the offspring. By means of natural selection the fit individuals survive.
However, with metaheuristics, the fitness function can be anything, and the neighboring solutions can also be different. If we borrow the idea of natural selection applied on the fitness function, we can describe an algorithm that optimize for other parameters and be used to solve other problems including hard optimization problems. But more generally, this idea can be thought of as a metaheuristic, namely the genetic algorithm. [26]

We can incorporate evolution with our mountain-climber example by imagining he instead repeatedly took some random steps from a location and selected the best one, as natural selection often would. It is then not hard to see, he on average would go towards a higher location and find a good solution. However, separate mountain ranges are still unlikely to be explored if he starts in one of them. In order to overcome that, we could either try to modify the local search heuristic, such that he does not only take steps, but we could also select a different metaheuristic, motivating the idea of evaluating different metaheuristics with respect to appropriateness for given problems.

3.3 Three nature-inspired metaheuristics

We give an introduction to the working principles of the three nature-inspired metaheuristics which we explore further in this project, Simulated Annealing, Ant Colony Optimization and an Evolutionary algorithm.

3.3.1 Simulated Annealing

Simulated Annealing is a simple metaheuristic. Its fundamental idea is drawn from annealing in solids, where heat treatment change physical or chemical properties of materials in order to make them more workable. While the material is hot it is more easily altered. Subsequent cooling then causes the material to freeze into a desired minimum energy configuration [17]. Simulated Annealing similarly relies on a cooling scheme and considers the hot and workable phase, a period where the search space is explored rapidly. A local search heuristic is also applied but when the temperature is warm, Simulated Annealing often pick a worse solution in order to explore more of the search space. Gradually as the cooling scheme lowers the temperature, the best choice of the local search heuristic gain more prioritization and the probability of choosing a worse neighboring solution decreases. When the temperature is 0, Simulated Annealing is akin to random local search and we end in a local optimum. However, due to the more rapid exploration during the warm phase, the probability that this local optimum is a good one, is higher. Again using the mountain-climber example, we imagine he initially runs around randomly while slowly prioritizing going upwards and finally honing in on a peak. We can see how the starting position won't matter as much any more if the duration of this warm phase is long enough such that it is now possible for him to find his way from his starting position to the Himalayas. However, if we cool too rapidly he would not have time to explore the entire earth, meaning we are not effectively exploring the search space. Simulated Annealing is particularly simple because it only considers one instance of the solution at a given time. From now on, we consider such an instance as an individual.

3.3.2 Evolutionary Algorithm

There are many variants of the Evolutionary Algorithm all exploiting the idea of generating offspring, evaluating them, and selecting the better ones in order to find better solutions to a given problem. A relatively simple idea is to consider a fixed population size generating a fixed number of offspring each generation. This is the $(\mu + \lambda)$ EA metaheuristic. Each individual in the population corresponds to a feasible solution in the search space and offspring are generated by applying search space specific mutations, like a bit-flip for bit-string problems, or reversing the order in a random range, for permutation problems. When applying the $(\mu + \lambda)$ EA, μ is the population size, so we start with μ feasible solutions. In each generation we then generate λ offspring by drawing λ times drawing a random individual from the population and applying mutations. Finally the population size is restored to μ by eliminating the weakest individuals,

according to the fitness function, and considered from both the original population and the generated offspring. [30]

3.3.3 Ant Colony Optimization

The final nature-inspired metaheuristic we consider is Ant Colony Optimization. It is different from the two others in the way that feasible solutions are generated from scratch every generation in a guided manner, instead of by applying a change to an already feasible solution. It is therefore a constructive metaheuristic.
The algorithm is based on the method ants use to find their way to food sources in nature. They do so by walking somewhat randomly and upon finding food they return to their colony while laying down a trail of pheromones on the way. Other ants are then drawn to follow this trail of pheromones rather than walking entirely random. Over time the trail evaporates which happens to a larger degree for longer trails where the ants cannot refresh it with new pheromones as frequently. Consequently, it is easier to build up strong trails of pheromones on shorter routes which then becomes dominating [27].
The same concept of laying out pheromones, evaporating them and finding good solutions through a dominant trail, can be exploited to solve optimization problems. In order to do so, we need to be able to represent each solution in the search space by a tour through a graph. While it is simple to see how this can be done for graph-problems, other types of problems can often be transformed into such a representation. We will later explore an example of this when implementing Ant Colony Optimization on bit-string functions.
After defining this graph-representation one way to run Ant Colony Optimization is by simulating virtual ants walking through the graph such that the tour is a feasible solution to the problem. The fitness value for the tour of each virtual is then evaluated, and pheromones are laid out accordingly on the edges of the tour. This is done so better fitness values results in more pheromones. In the next iteration, virtual ants walk again but this time, edges have different amount of pheromones, and chosen with different probabilities. Before laying out new pheromones, we evaporate the current ones by a factor. Laying pheromones based on the fitness value captures the natural effect of shorter tours being prioritized. While this idea may be more true to the actual behaviour of ants in nature, some variants of Ant Colony Optimization approaches the deposition of pheromones differently. Instead of laying a fitness value dependant amount for all ants, some approaches, like MMAS, lays a fixed amount of pheromones with respect to the best achieved tour so far [31], but still for each ant. Both of these approaches have their own strength as weaknesses.
For some problems, like the Travelling Salesman Problem, we can also incorporate a heuristic to include in the selection weight of each edge choice. For the Travelling Salesman Problem a good solution is likely to use shorter edges in general. We can combine the effect of the pheromones with a bias towards making shorter edges more likely to be chosen [12]. This usually causes the initial solutions to be better, thereby affecting the trail that later is dominated by pheromones.

3.4 Problem classes

Despite the fact that metaheuristics can be general enough to work on any problem with a fitness function and a search space, we limit our scope of different problems in this project.

3.4.1 The Traveling Salesman Problem

The Traveling Salesman Problem, TSP, is a combinatorial problem that is NP-Hard. It is highly relevant for the application of nature-inspired metaheuristics because these techniques

might provide the best approximate solutions we can get in an acceptable time-frame. TSP is the problem of finding the shortest path that connects all vertices in a graph to a cycle. The name is given by the practically application of a travelling salesman who visits a number of different cities and returns home. Finding the optimal order of cities to visit and reducing the travel distance is then the problem. Interestingly, the starting location does not matter to the optimal solution which is why we care about finding a cycle and not so much which city he starts in. Formally, the goal is to find the shortest Hamiltonian cycle in a graph. There are commonly two kinds of TSP problems, symmetric and asymmetric, the difference being that a given edge cost is the same in either direction for the symmetric case, and may be different in the asymmetric case [22]. In this project we largely focus on 2D TSP instances with edge cost derived from direct vertex distances which are symmetric. We are not too concerned with the practical implications of this since asymmetric TSP instances can be transformed into a symmetric equivalent [22]. We therefore limit our scope to the symmetric case, so from now on when referring to a TSP instance, we assume it is symmetric unless otherwise specified.

3.4.2 Bit-string benchmark problems

Apart from solving problems that are highly relevant for the application of nature-inspired metaheuristics we are also interested in a different kind of problem class that may provide us with some insight into the conditions that make a certain metaheuristic effective. It turns out bit-string problems are helpful in this aspect. The fitness function is defined on a bit-string where we already know the optimal solution, thereby making the techniques practically redundant but analysing how effectively a given algorithm can explores the bit-string search space can reveal some properties. For instance, we will mainly explore the OneMax and LeadingOnes problem. The fitness function of OneMax is defined as the number of ones. One of the motivations for studying Onemax is the fact that it is isomorphic to the function where the fitness value is given by the hamming distance to any choice of equivalent sized bit-string [16]. For LeadingOnes, the fitness value is given by the number of ones, before the first zero, from the left. Unlike OneMax, where we can always affect the fitness function by flipping a bit, a bit-flip will only affect the fitness value if the bit is one of the leading ones or the next zero [23]. Put in another way, the bits in LeadingOnes are much more dependant on each other. Recognizing these traits can be helpful as an effective way of handling such a property might generalize to other problem classes.
When studying the ability of a metaheuristic to solve a given bit-string benchmark problem, it is helpful with a short introduction to the black-box complexity.

3.4.2.1 Black-box complexity

Black-box complexity deals with how efficiently we can solve a problem in a black-box setting. From the perspective of the algorithm, this means we do not know the fitness function, it is only possible to query it to obtain a certain fitness value. When studying the performance of metaheuristics on the bit-string benchmark functions we put ourselves in this black-box setting. Otherwise it is pointless as we just know, for instance the optimum of OneMax is to set all the bits to 1. That is a different story for TSP, where we cannot just tell the optimum from knowing the definition of the fitness function.
A highly relevant property of a certain problem is its unrestricted black-box complexity. It tells us how efficiently we can hope to solve it to optimality in a black-box setting [16].

Lemma 3.2.2. *Let $\mathcal{F} \subseteq \{f : S \to \mathbb{R}\}$. For every collection $\mathcal{A}'$ of black-box optimization algorithms for $\mathcal{F}$, the $\mathcal{A}'$-black-box complexity of $\mathcal{F}$ is at least as large as its unrestricted black-box complexity.*

Figure 3: Unrestricted black-box complexity. The lemma is an extract from chapter 3 in the book, Theory of Evolutionary Computation [16].

We care about this because it allows us to reason about the efficiency of a given algorithm. For instance, a trivial idea to solve OneMax in a black-box setting is to flip one different bit at a time and query the fitness function. If the fitness value is better than the previous, we accept the bit-flip. Clearly this solves OneMax in n steps as each query fixes one bit, so we need one fitness evaluation per bit. However, because we know the unrestricted black-box complexity of OneMax is $\theta(n/log(n))$ [16], this idea is not optimal in terms of black-box complexity.

3.5 Aim of the project

The aim of this project is mainly to develop a framework, implementing and visualizing the nature-inspired metaheuristics, Simulated Annealing, Ant Colony Optimization and an Evolutionary Algorithm for solution of The Traveling Salesman Problem, TSP. We will evaluate the implementations on TSP, using instances from the TSPlib library, as well as on bit-string benchmark functions.
During these runs of metaheuristics we will need to be able to extract key figures, such as fitness and running time, which we can base our evaluations on.
While we limit our choice of metaheuristics, problem classes and visualizations in this project, more might be added later on. We keep this in mind and ensure the framework is extensible by focusing on modularity and flexibility during the development.
Apart from these mandatory requirements and in the context of evaluations, it is desirable for the metaheuristics to be configureable, so we can tune each run with different parameters.
As part of a secondary objective, we realise the Traveling Salesman Problem have profound practical applications. This encourages the aim of making the framework widely available and for the ability to manually setup TSP instances, so a user can find a good route for a given TSP request. Future work can then extend the framework in ways that enhances this motivation, for instance by allowing TSP instances to be defined by physical addresses along with a map visualization.
When designing the framework, we therefore have these applications in mind such that it is also practically useful and extensible.

4 Analysis

4.1 Main challenges

When developing the project there are some challenges to consider which we need to attend with special care. Approximation algorithms like implementations of metaheuristics are commonly analysed in terms of how much additional running time influence the quality of the solution. This is different than the usual asymptotic worst case time complexity commonly used to analyse deterministic algorithms, where achieving a better time complexity can overpower the benefits of writing high performant code. However, since we also have quality of solution as a dimension

in run-time analysis for metaheuristics, and because the nature of this analysis is often more practical, even conducted with actual test cases, we take special consideration to performance [38].
The implementation should therefore be written in a computationally efficient manner. However, the framework also needs to be extensible, modular and flexible such that it is later simple to add other problems or metaheuristics. This may result in performance versus modularity trade-offs which is a challenge we face multiple times during the development. These properties are not necessarily diametrically opposed and it is possible to write code that is both performant and modular. However, the black-box setting with which the metaheuristics considers the problem calls for high levels of abstraction in the implementation, which in turn will make the trade-offs particularly challenging. We will aim to achieve the best of both worlds while accepting some compromises, particularly in order to speed up our TSP solvers.

Another challenge to consider is the inherent disconnect between running an algorithm and visualizing it. We need to consider questions like; Do we visualize the data as it is generated? If so, given that the algorithms then needs to follow the playback of the visualization, can we even avoid tangled, highly coupled code? And how do we speed up the algorithms to the maximum potential without losing too much performance to data output and visualizations? What if we want to detach the visualization engine and just find a quick TSP solution? All of these questions and their answers are highly relevant, especially in order to build a modular and extensible product.

4.2 Solution proposal through choice of technologies

When developing such a project there is an array of different technologies available to assist with creating a product of high quality. We need to choose the programming languages involved, as well as other libraries, frameworks and dependencies. These choices are important because it will shape the project fundamentally and affect the future prospects as well. We could create a desktop application and make it platform independent using a language like Java with visualization libraries. Doing so would simplify some aspects, like communication pathways in the project being limited to one application and we would not need to learn a multitude of different technologies. However, while such an application might be ideal for researchers willing to download the source code and extend it with other metaheuristics or problem classes they are researching, we keep our second objective in mind and ensure the project can also be practically applied. In order to do so, we choose technologies that allows us to deploy the product on the web such that it is easily accessible, while also having an extensible backend, capable of evaluating the metaheuristics efficiently independently of visualizations. This frontend/backend architecture also provides the added benefit of intrinsic decoupling by enforcing communication patterns between the data model in the backend and inputs/visualizations in the frontend. While it may slow the development down, modularity is one of our most important considerations making this choice worth the potential time-investment.

4.2.1 The web stack

Another benefit of deploying the project on the web is the potential to choose entirely different technologies for the frontend and backend, streamlined for the individual purpose.

4.2.1.1 Backend - C++

We need a highly efficient compiled language to run the metaheuristics, so we choose C++, which allows for low level optimizations as well as the memory management needed to speed up our computation. We use Nginx as webserver with the FastCGI protocol as the interface between the web-server and our C++ backend. [4][13][39]. This means we will have a C++ program serving all API requests, potentially this program could also run the algorithms, but web requests should be responsive and return a result in a timely manner so we need another process handling the computations. By splitting the backend into two applications, a worker application and an application for serving API requests, we can create a responsive API capable of returning the current state of a given run of metaheuristics through communication with the worker application. This both increase modularity as well as even allowing the worker application to be independently extended and included in different products or projects.

4.2.1.2 Frontend - React JS

In the frontend we need a technology to easily create visualizations. It must be possible to create visualizations and various components like buttons, drop-downs and sliders for input control. This is commonly done using the markup language, HTML with CSS and JavaScript. Through many years of web development there now exists thousands of libraries with components assisting this kind of production. We utilize the modern JavaScript library React JS to speed up the development. React is named as such, because it "reacts" to state changes. The main design pattern when building a React application is to extend the component class in order to create components within the application. React then provides lifecycle event triggers, in particular the componentDidMount() and render() method is useful, where the idea is to load needed data in the former, and return HTML based on the current state in the latter. React is compatible with the JavaScript syntax extension JSX, which is translated into regular JavaScript. JSX makes the render() method particularly powerful because it allows for plain HTML to be put inside JavaScript and returned as any other type, eliminating the need for having markup and logic in separate files. In React, you would generally take advantage of this by having a main component returning the entire HTML through sub-components, and if state changes in any of these sub-components, React ensures only this change is applied to the HTML DOM [5].
By using React as intended, we will almost automatically enforce the Model-View-Controller design pattern, however it is important to never do any state changes in the render method because that would prompt it to run again and also violate this design principle.
All of these features of React are especially useful for our project since we will frequently update our current state with data as visualizations change and needs re-rendering.

4.3 Additional challenges

The choice of technologies means additional challenges needs to be tackled. One of the more immediate implications of this is the need for visualizations to be independent from the computations. If we tried to synchronize the visualizations with the computations, the communication needed to do so would become increasingly messy as more users utilize the backend worker application at the same time. The latency between server and client would potentially also pose a problem, as we would need frequent bursts of data to update visualizations multiple times a second. Keeping it separated may be a desirable trait regardless but it means we will need to generate and store data output on the backend server until requested by the frontend application.
If everything was handled by a single application we would not have to care as much about

the size of this data output since it would be generated and stored on the users machine. On the other hand, the backend server may generate data for multiple runs of metaheuristics and by many users, meaning we will need to have some system in place limiting this in order to not overwhelm the server storage capacity. Additionally we have to keep networking data transfer speeds in mind and ensure the product is not only use-able if the user have a fast fiber connection.
Another challenge in this regard, is the potential for multiple jobs to be started at the same time. If someone starts a large job requiring a lot of CPU-time it should not hinder someone else from getting a small job evaluated quickly or at all.
Finally, it should not be possible to start a job that runs forever. Even with a fair scheduling policy where large jobs get deprioritized after a while, we still do not want them to kick in and run forever when there is nothing else to do because they still use memory resources and this means we would gradually rack up such jobs potentially causing a memory leak. Also depending on the cloud hosting provider, some virtual machines collect credits when the CPU is idling, which is then used for higher performance when the load increases. Because of this and for general system health, we want to ensure no jobs are kept in memory and that the CPU idles when no one is using the service.
We can tackle these challenges by limiting the frequency of data output, make sure we do not send the same data multiple times, handle multiple jobs with a job scheduler, and enforce other restrictions on the backend while communicating these choices to the frontend.

4.4 Scope

In order to prioritize different features and requirements the MoSCoW method was applied. By grouping functionality into must have, should have, could have and would have categories, it was possible to better the time-investment and productivity during the development process. The final grouping is represented by the below list, and all the check-marked items made it into the final product. (Please note that the scope of this report is unable to extensively cover every single developed feature).

- **Must have:**
  - ☑ The ability to run a metaheuristic to solve TSP on the Berlin52 TSPlib instance.
  - ☑ A simple configuration of Simulated Annealing.
  - ☑ Ant Colony Optimization with preset parameters but multiple ants
  - ☑ The evolutionary algorithm (1 + 1) EA.
  - ☑ A graph visualization, highlighting the tours found as solutions.
  - ☑ Bit-string benchmark functions OneMax and LeadingOnes.
  - ☑ A bit-string visualization showing the bit-string solution as raw 0's and 1's.
  - ☑ Best so far fitness, number of function evaluations, running time, live key-figures.
- **Should have:**
  - ☑ Cloud hosting, so the website is always accessible and not limited to localhost.
  - ☑ Frequent data output updating visualizations with the current best solution.
  - ☑ Visualizations capable of handling, navigating and displaying multiple individuals at a given iteration when relevant to the metaheuristic.

- ☑ Configurable variations and parameters, $(\mu+\lambda)$ EA and α, β, ants, evaporation factor for AOC.
- ☑ Support for more TSPlib instances through a parser that supports the EUC_2D TSPLib instances.
- ☑ Enforced stopping criteria based on CPU-time such that runs of metaheuristics cannot prompt the backend server to run forever.
- ☑ A system capable of estimating and appropriately limiting the frequency of data output based on the given job parameters such that the data output does not overwhelm the server and network capacity.
- ☑ An API endpoint with frontend documentation serving all communication between frontend and backend which can also be used in other projects to substitute the frontend and customize all visualizations.
- ☑ A visualization of bit-strings showcasing the amount and average positions of ones as a dot in an onion like figure.
- ☑ A graph-visualization for TSP superimposing all edges from all tours when there are multiple individuals. This is particularly useful for ACO.
- ☑ The ability to change the evolutionary algorithm to RLS by supporting different choices of how mutations are applied.
- ☑ Max iterations stopping criteria.

- **Could have:**
  - ☑ Other bit-string benchmark functions, TwoMax, Jump and Needle.
  - ☑ A variation of ACO that also employs local search.
  - ☑ Multi-threaded concurrent evaluations to better support multiple jobs submitted at the same time.
  - ☑ Job scheduling, so jobs submitted by different users get fair treatment.
  - ☑ The ability to add/delete vertices in a given TSP instance by clicking on a canvas.
  - ☑ Support for TSP instances defined by edge weights in an XML file where vertex 2D locations are found through iterations of a physics engine.
  - ☑ User defined TSP instances based on defined vertices and edge weights.
  - ☑ Live updating pseudo-code of the resulting algorithm from the current selected metaheuristic, problem, and parameters.
  - ☑ Max iteratations without improvement stopping criteria.
  - ☑ Starting multiple independent runs of Simulated Annealing in parallel by taking advantage of the multi-individual visualization engine to quickly assess the variation in Simulated Annealing solutions.
  - ☑ Scripts that automatically maintains the uptime of all backend servers
  - ☑ Selecting reverse, insert or swap as the mutation of choice for the Evolutionary algorithm applied on permutation problems.
  - ☑ Give the user an estimate of the total run time for the given job.

- ☑ A red/green colour visualization for each bit in bit-strings.
- ☑ Raw text permutation visualization.
- ☑ A red/green colour gradient visualization for permutations.
- ☑ Adaptive cooling scheme of Simulated Annealing based on the difference between different simultaneous runs.
- ☑ Backend load balancing with multiple servers.
- ☐ Immediate processing to completion by the API endpoint, rather than the worker, if the job is deemed small enough.
- ☐ Support for asymmetric TSP instances.
- ☐ A route planner made possible by Google Maps API integration.
- ☐ Configurable mutation rates for the Evolutionary Algorithm.
- ☐ Compressed data output.

- **Won't have:**
  - ☐ Intelligent selection of the metaheuristic and configuration that is most applicable to solve the problem at hand.
  - ☐ Support for multiple independent runs of metaheuristics that already generates multiple individuals per iteration.
  - ☐ Other metaheuristics.
  - ☐ Other combinatorial NP-Hard problems.

Due to the agile development process, these groups where updated often with new functionality. Sometimes this also meant moving items between groups. In a few cases, items were removed if the functionality was deemed unnecessary or covered by other items. Categorizing lower prioritized items in "could have" and "won't have" was useful even early on in the process by maintaining a sense of direction for the project. By already considering these, the development of more important features was better steered towards modularity and flexibility in order to better support the lower prioritized features later on. Consequently, sometimes lower prioritized features would be implemented earlier than expected due to the small amount of time required to do so. So, while the prioritization in the MoSCoW model played a large role, it was better used as a general guideline with other parameters, like the size of the specific feature, also taken into consideration.
The must have features constitutes the minimum viable product. These are features that allow us to run and visualize the three different meta-heuristics with a simple configuration on predetermined TSP instances and bit-string functions. They are not useful for practical purposes since we cannot configure custom TSP instances, We would also only be able to evaluate the metaheuristics to a limited extend because the minimum viable product would not allow us to change problem instances/sizes and would also not allow different configurations of the metaheuristics. These features are important and some of them are necessary to fulfill our second objective of having a practical product as well. They are therefore all listed in the "should have" category.
The "could have" features are among other things refinements and additions that makes the user experience better. While these features are not essential they still greatly improve the quality of the product.
An example of a feature being moved to a lower prioritization during development is immediate

processing to completion by the API endpoint, rather than the worker, if the job is deemed small enough. The idea is to avoid API/worker communication overhead if the job can be completed within an acceptable response time. However, even though this overhead is dominating for small jobs, results are still returned quickly enough such that this would not have a great impact.

5 Implementation

5.1 Frontend/Backend networking architecture

The project consists of a frontend and a backend. We use the frontend to configure the run of metaheuristics as well as for visualizations. The backend is then used for computations. As seen in figure 4, communications conform to an API protocol.

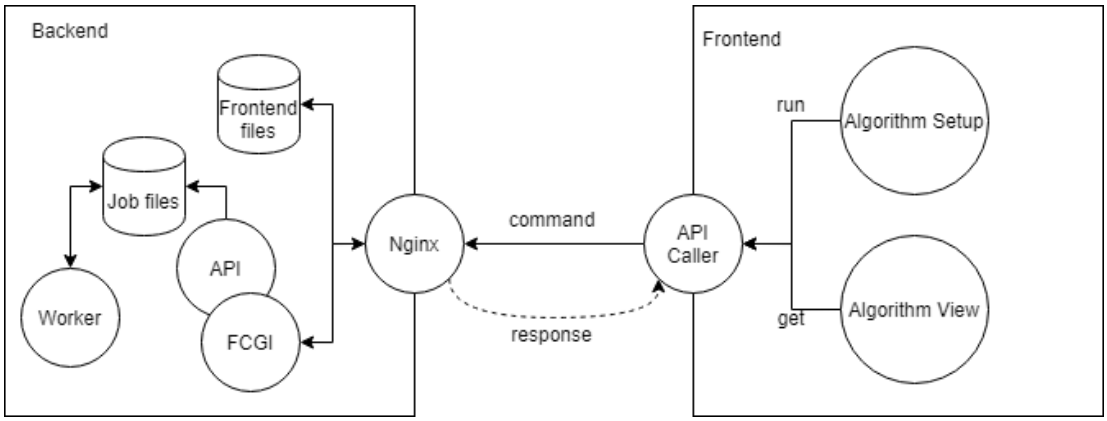


Figure 4: Frontend/Backend communication illustration

This means the frontend potentially could be replaced by other visualizations, for instance by creating a mobile app. By keeping the modules separated we ensure the project is extensible to a higher degree, we can also choose to extend different parts of the project. Through using the API and visualizing the returned data we are also geared more towards modularity in other design choices. By not making any assumptions about the returned data we can extend the project with more metaheuristics without even touching or updating the visualization engine.

5.1.1 A run of metaheuristics - The communication path

When starting a run of metaheuristics, it is first configured in the frontend. When the "run algorithm" button is clicked, the setup component constructs a run command of the current configuration and submits it to the API caller on the frontend side, which handles all API communication. The command is then sent to the server, first picked up by the web server application Nginx, then forwarded through the Fast-CGI protocol to a C++ application tasked with handling all web requests. This C++ application, API, analyse the command and if it is a valid job command, generates a random but unused ID and creates a job file named after the ID containing the command. A JSON with the ID and success status is then sent back, prompting the frontend to go into Algorithm View mode. Meanwhile, the worker, a separate application tasked with running all heavy computations, frequently scan the job folder, discovers the new command and starts working on it. The worker frequently dumps data output in the same job folder, also named after the ID. I.e. if the job it is working on is the job with ID 27, it outputs 27_output and 27_status. In Algorithm View mode, the frontend keeps requesting the data output which it use for visualizations. It does so through the "get" command along with the ID which the server originally returned. When the API receives a "get" command it looks for output files according to the provided ID, constructs a JSON response with the content and

sends it. In this JSON response, we also specify if the job is done. When it is, the frontend detects this and stop requesting new data. We make sure to only return newly generated data by sending back, in the JSON file, a pointer to the current last position in the output file. In the next "get" command, the frontend sends this pointer along, which the API now know to use as starting location in the output file. However, the frontend can still request the whole file, which we do to fall back on if something goes wrong, or if the user navigates away from the algorithm view tab and then back to it.

5.2 Output files and race conditions

The communication between the worker application and the API application is done through the file system. The worker may output a lot of data which must be available, as it is generated, to the API. This is done by only appending new data to the output file such that already appended data can be safely read.

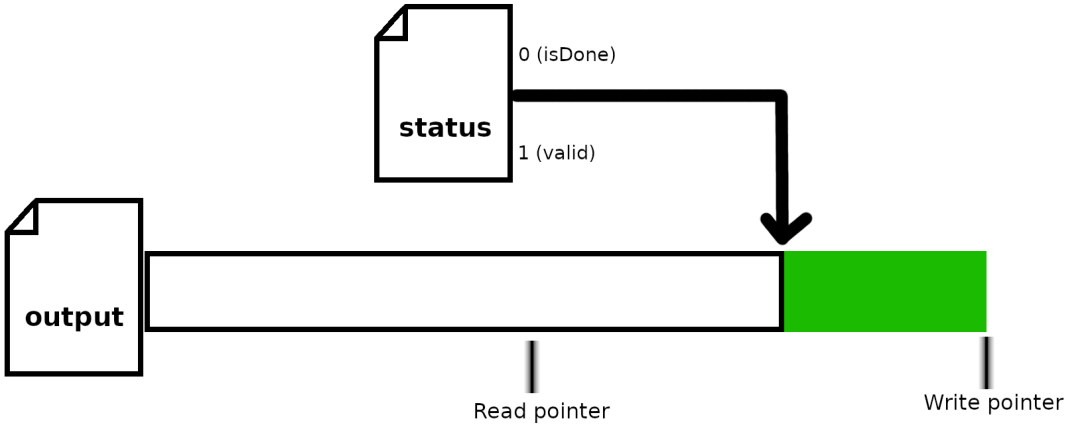


Figure 5: Job output files. Green area represents appended data. Blurry black bars represent data being read and written.

Figure 5 illustrates this with the green area representing newly generated data and appended by the worker, while the white data is read by the API. Normally we would not allow files to be written to and read from, at the same time. This would be handled by lock-files, but since we read/write so frequently, the use of lock-files would cause too much waiting, considering we can avoid it. We want the worker to output data freely without being hindered by external applications like the API. However, there is still one race condition we need to consider. When reading from the output file, it may be tempting to read the whole file. If we do so, it can then happen that we read from the file while we also append to it. If the read pointer then overtakes the write pointer, there is nothing more to read, so we do not get the full amount of newly appended data, only some of it. And this is a problem because the data may then be invalid.
We solve this by using a status file which can give us some information about the output. The status file contains an isDone value, it is 0 for not done, 1 for done. The worker can tell the API that the job is finished and all the data has been generated by setting this to 1. Status also contains a pointer to a position in the output file up to which, we can safely read data. The API uses this pointer to stop at when reading the data output. Finally the status file contains the number one, meaning valid, which is used to avoid the need for lock-files to access the status

file. Every time the worker has successfully appended a chunk of new data, it clears the status file and re-writes it such that the pointer points to the end of the output file.
Not using lock-files means we have to be diligent. For instance, it is crucial that the worker clears the status file before writing new information, because this means the valid number is not set in this interval. So if the API reads the status file at exactly the same time as it cleared and re-written, it will not read the valid number, meaning it will try again or fail safely if it happens multiple times in a row. We make sure the valid number is at the end of the status file and that the status file is written to sequentially such that, if the API reads the valid number, we are sure the pointer is also correct.

5.3 Core model - The Worker program

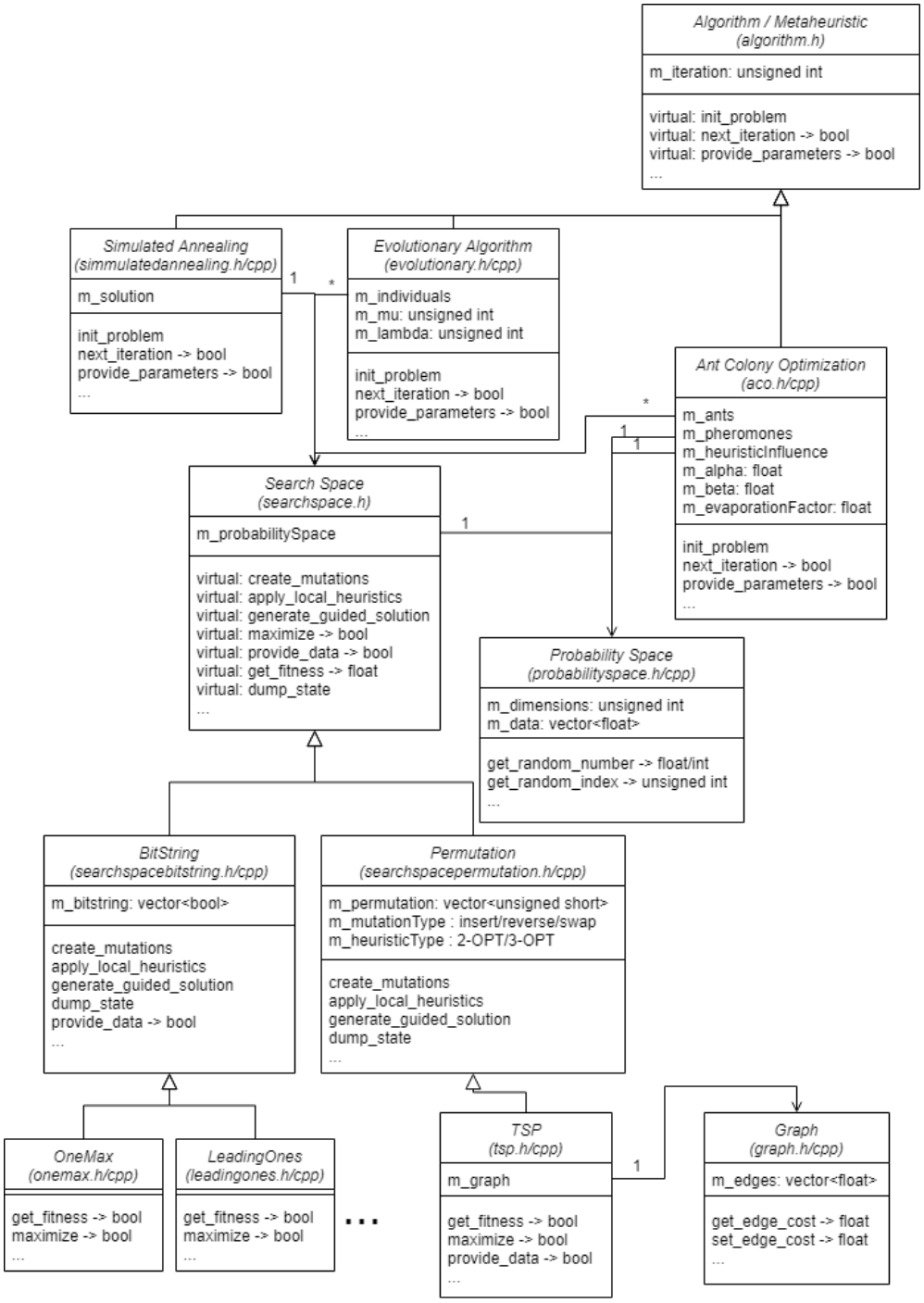


Figure 6: Core model class diagram

A central aspect of the framework is the engine running the heavy computations and generating the output as needed by the visualizations. It is the core of the project, without which we cannot run any metaheuristics and solve The Traveling Salesman Problem. The implementation of this core plays an important role in ensuring the framework is flexible, modular and extensible so we take extra care to design it in such a way that follows modular design principles, like "low coupling/high cohesion" and "don't repeat yourself", while making sure it is still performant.
Figure 6 is a class diagram of the core model. To keep it simple, and within one page, not all functionality for each class is represented and the actual source code may deviate insignificantly. The diagram provides understanding for how metaheuristics are setup to run a given problem and how we follow the design principles.

5.3.1 Search space and problem class abstraction

Metaheuristics are fundamentally separated from the problem so this needs to be reflected in the implementation in order to ensure it is flexible enough, such that we can easily implement more metaheuristics in the future as needed. By employing inheritance we create an abstract class, the SearchSpace class, through the use of C++ virtual functions. The idea is to represent any problem through a search space, which makes sense because metaheuristics guide the solution through the search space of the problem. So we can extend SearchSpace to implement specific search spaces as well as specific problems but from the perspective of the metaheuristic, we just call general virtual functions, like "create mutation", or "get fitness", that are defined as virtual functions and must be overridden in sub-classes.
This means the SearchSpace class, while also providing some limited amount of logic by itself, can be thought of like an interface, that a problem class must conform to in order to be used in a run of metaheuristics.
We realize that changes to feasible solutions, like when we generate a mutation or apply a local search heuristic in order to reach a new neighboring solution, are defined on the specific search space and not the specific problem. I.e. we can flip a bit regardless, when the problem we are trying to solve is OneMax or LeadingOnes, because they are both bit-string problems, but we cannot flip a bit when solving TSP. That we need to flip a bit is something the logic for specific metaheuristics are not allowed to care about because that would break flexibility and go against the black-box like way, with which metaheuristics considers the problem. So we write search space specific implementations for bit-string and permutations that extend the above-mentioned SearchSpace class. In this way, the metaheuristics can flip a bit through an abstraction that will result in another, but related, action if the problem is not a bit-string problem.
Even though these classes, SearchSpaceBitstring and SearchSpacePermutation, inherits from SearchSpace, they still do not override all required virtual methods, as some of these methods, like the fitness function, are problem specific. This is fine in C++ as long as we do not try to instantiate the classes, they are also abstract.
We extend each search space class with problem specific classes and override the fitness methods as well as one called maximize. Maximize simply returns true/false depending on whether or not a high or low fitness value is better. For instance maximize always returns false for TSP because a shorter tour is better and the fitness function is the length of the tour, while it returns true for OneMax. An interesting effect of this, is we can easily change TSP to do the reverse, namely finding the longest Hamiltonian cycle simply by changing the maximize function to return true.
When looking at the class diagram a trained eye would notice we provide definitions for the provide_data method in TSP, a problem specific class, while it is also provided in SearchSpace-Bitstring, a search space specific class and a different layer in the inheritance hierarchy. We can do so because the C++ compiler only requires all virtual methods to be defined somehow

before a given class can be instantiated and we only instantiate the problem classes. The reason for implementing it differently is the fact that the bit-string problems only needs to know the size of the bit-string in order to run the fitness function while TSP needs problem specific data, namely graph edge-cost data, which a different permutation problem may not need. However, if we suddenly also needed problem specific data to implement another bit-string function we would also put a provide_data definition in the implementation of it.

5.3.2 The Probability Space class

Some metaheuristics like Ant Colony Optimization work in ways that are particularly difficult to generalize across search spaces and problem classes. For instance, how do we properly define, update and evaporate pheromones, when pheromones are defined on the graph edges for TSP and differently for bit-string problems (We later explore how pheromones are updated on bit-string problems but it is essentially placed in two buckets of pheromones per bit, one for 0 and one for 1).
We do this by first realising the use of pheromones is the way Ant Colony Optimization handles probability, the pheromones carry some probabilistic idea. Since all metaheuristics employ randomness to some degree, we create a class general enough to handle these different kinds of ways to incorporate probability, the ProbabilitySpace class. It describes an n dimensional data-structure of floats with configure-able dimension size, so we can use it for applications like pheromones. For instance, a 20 bit bit-string would see the probability space class represented as a 20 by 2 matrix for use as pheromones, but a 20 vertex permutation would have a 20 by 20 matrix. We then put logic in the search space classes that initialize and use it accordingly. This logic is not represented in the class diagram but can be further explored in the source code. In C++ it is possible to define methods with variable number of arguments. We take advantage of this when accessing it. While we only use two dimensions for bit-strings and permutations, we can scale it to more dimensions with more arguments when needed. Further, we store the data-structure as a one dimensional array and calculate the access index from these arguments in such a way that values for the last argument is contiguous in memory. For the 2D case, this is the row-major order [9]. This is useful to remember as we use it to exploit memory locality and speed up processing.

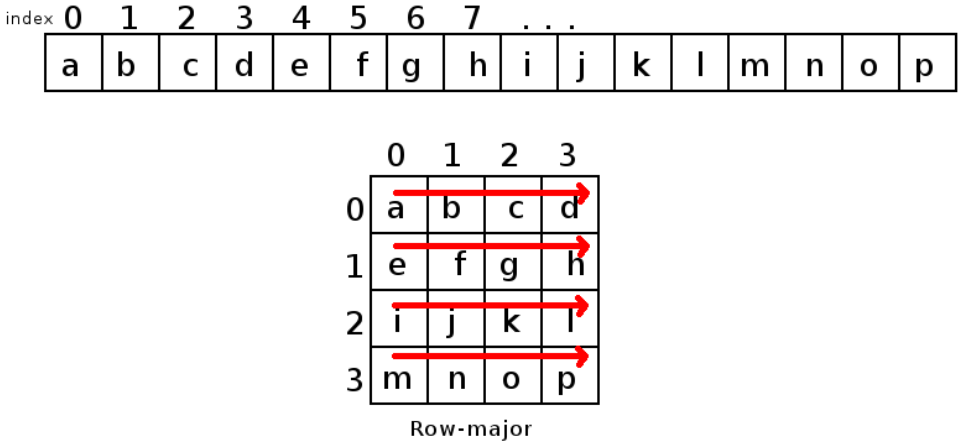


Figure 7: How the ProbabilitySpace data is stored and accessed in the 2D case.

A benefit of storing all the values in this data-structure in a single dimension array is the potential to iterate over all the values without caring about dimensions or other problem specific details. This is something we can do in the implementation of Ant Colony Optimization and we do so to evaporate the pheromones, but also to calculate probability weights from pheromones and the heuristic bias from TSP edge costs.

5.4 Specific metaheuristic implementations

All metaheuristics are implemented by extending the Algorithm class and defining the init_problem and next_iteration methods. next_iteration is then called when we progress the algorithm.

5.4.1 Simulated Annealing

Simulated Annealing is initialized with an initial solution and a temperature. The temperature is initially set to 1 for warm with 0 meaning completely cold. It is multiplied by 100 when returned to the frontend, giving the user a more familiar temperature range for what is warm and cold, but this is not something that matters to the implementation. We allow for multiple independent runs of Simulated Annealing in the same job, so the number of solutions is also specified through the provide_parameters method. This is not represented in the class diagram, as it is not an essential feature of Simulated Annealing but a lower priority feature we had time to implement.

5.4.1.1 Cooling scheme

In the next_iteration method we apply the cooling scheme and update the temperature. This is done in an exponential manner with a temperature half-time. The exponential cooling scheme is commonly applied and after testing a linear scheme, we can see how the lower temperatures potentially are not given enough time to reach a local minimum and if they are, we may spend a disproportionate amount of time in a too warm state.
Another way to see how an exponential cooling scheme is useful is by considering clustering in graphs when solving The TSP. If some edges in the optimal tour are much longer than others then higher temperatures are useful for finding those.

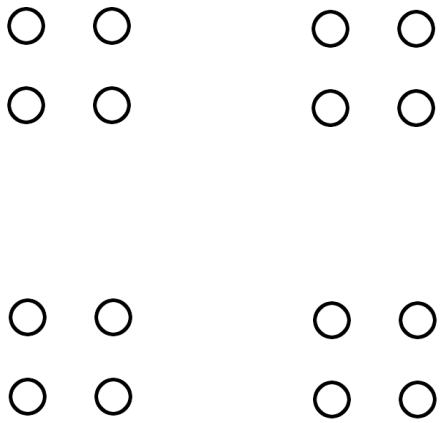


Figure 8: A graph of 4 clusters

Considering the graph in figure 8, there are four clusters. We can consider each of these a sub-problem when finding the optimal tour because the distance between the clusters is large enough to guarantee that an entire cluster should be visited before moving on to the next one. When each of the four sub-problems are solved, we just need to find the optimal interconnects between the clusters. The graph 8 is deliberately designed such that if we imagine the clusters are even more separated than pictured it is not so important which vertex within the cluster that connects to another cluster. So in this interconnect problem, each cluster can be represented as just one vertex making it the same as the sub-problems. While the problems are similar, the edge costs are proportionally different so the temperature range that is useful for solving the interconnect problem is different from the one that is useful for solving the clusters. For Simulated Annealing to be able to solve both with equal efficiency we need a cooling scheme that spend about the same amount of time in each of these temperature ranges, and this is a property the exponential cooling scheme provides. While this illustrative example is good to realise when the exponential cooling scheme is applicable, it may still not always be the best option. However since it allows us to care about a broad range of edge costs in a given run it provides a good general implementation.
In the implementation of the exponential cooling scheme we need to define a half-time. Since we have a stopping criteria defined from the beginning of a job, it is more useful to consider how many times the temperature should halve during a run of Simulated Annealing. Starting with a very high temperature and then having many half-times might be the most dynamic solution. We can input any graph and eventually this cooling scheme would get to a useful temperature for that problem, however it would also waste a lot of time in not very useful temperature ranges, so we need to run the algorithm for a longer time. On the other hand, if the temperature is only halved one or a few times during the run, the starting temperature needs to be very carefully picked and there would be more problems with too different edge costs in the optimal tour, which would then be more difficult to find.
We find that 10 half-times offers a good middle ground capable of handling most graph problems without wasting an unreasonable amount of time outside of appropriate temperature ranges.

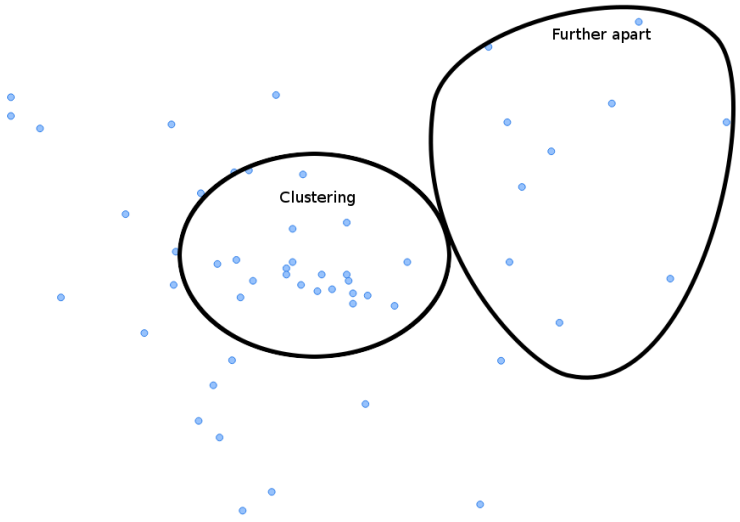


Figure 9: The Berlin52 TSPLib instance contains clustering

We test the cooling scheme on the Berlin52 TSPLib which is a good instance to test this on as it contains clustering, potentially widening the useful temperature range. As seen in figure 9 some vertices are much closer to their neighbours than others. The test reveals on figure 10 that most useful work happened in roughly half of the running time. Because we run the algorithm for one million iterations which is much more than enough to reach a local minimum for this problem, this interval is an effect of the influence of temperature and as such, the useful temperate range. We find it acceptable that half of the running time is essentially wasted because it acts as a buffer that ensures we can capture the useful temperature range for larger and more extreme clusters in other TSP instances as well.

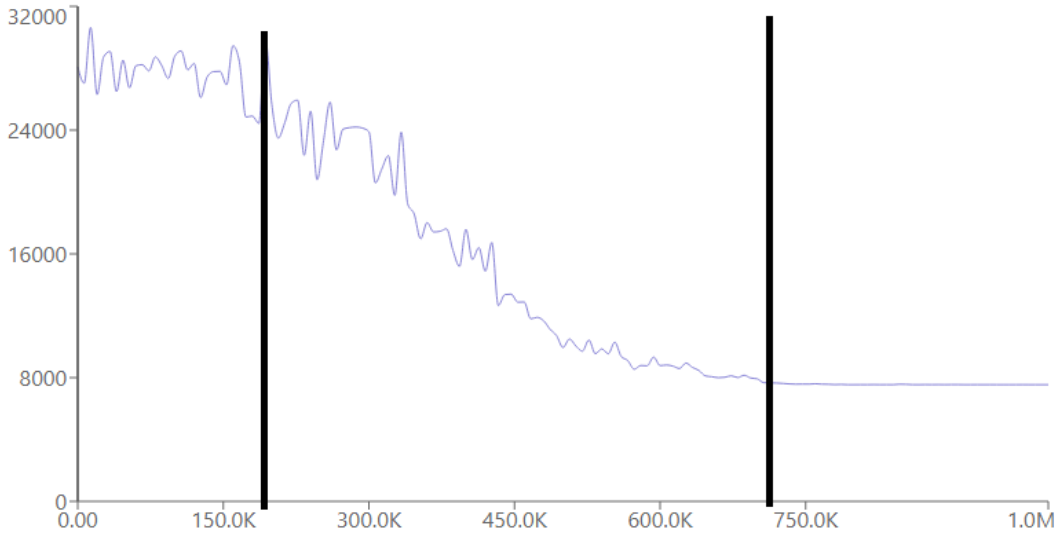


Figure 10: Fitness value of a run of Simulated Annealing on TSPLib instance Berlin52 for 1M iterations with a highlighted interval for when useful work happened

With this cooling scheme we never technically reach zero which is where Simulated Annealing no longer accepts worse solutions, causing the solution to go down to a local minimum. Normally and in the literature, this is not something we would be concerned about. Instead we could just run the algorithm for longer and with more half-times to make sure it becomes cold enough. However, because we limit the number of half-times we cannot do that. In order to better capture extreme cases where this would be useful, we force the temperature down to 0 by distributing the last $1/2^{10}$ of the cooling evenly throughout the run. Figure 11 shows how this only minimally influences the cooling scheme in general such that it does not take away from the exponential nature and effects of the scheme. It only shows the last 40 % of the cooling scheme where a difference is noticeable.

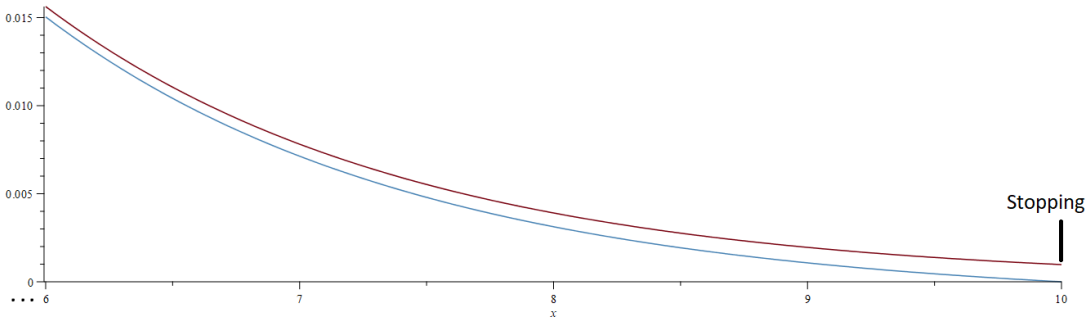


Figure 11: Temperature of the last 40% of a run. Blue is forced down to zero

5.4.1.2 Candidate solutions

After updating the temperature we call the apply_local_heuristic method on the current solution. This method generates a candidate solution depending on the given search space. Regardless of the search space, it also incorporates the acceptance probability, which is the probability of transitioning to the candidate solution. Even though the acceptance probability is defined generally by Simulated Annealing, we apply it in the implementation of the apply_local_heuristic method defined for the specific search space. This is for efficiency reasons as we otherwise would have to backup the solution, apply the local heuristic, compare them and apply the acceptance probability, which involves copying a lot of data unnecessarily. When incorporating the acceptance probability outside of the specific implementation of Simulated Annealing, we need to generalise it in a reasonable way such that Simulated Annealing can use it while it is still possible to apply a local heuristic as the name suggest, namely without the acceptance probability and the possibility of transitioning to a worse state. We create this generalization by applying the acceptance probability as a change to the ProbabilitySpace depending on a given difference in fitness value. We then implement the temperature through the ProbabilitySpace class. In this regard we can still disregard the acceptance probability if the ProbabilitySpace values are 0. The function for doing so is defined in the general SearchSpace class and then used in the implementation in the specific search spaces. We implement the acceptance probability as is done in the litterature [10]:

$$\begin{cases} 1 & e' < e \\ e^{-(e'-e)/T} & \text{otherwise} \end{cases}$$

Where e' is fitness value for the solution if the suggested change is applied, e is the fitness value for the current solution, and T is the temperature. We notice how this formula is only relevant when the objective is to minimize the fitness value but we can easily change it to also be applicable for maximization where it similarly is [10][16]:

$$\begin{cases} 1 & e' > e \\ e^{-(e-e')/T} & \text{otherwise} \end{cases}$$

5.4.1.3 Bit-flip, 2-OPT and 3-OPT moves

We need a way to generate the candidate solutions as used in Simulated Annealing. These are search space dependent and can potentially also be used in other algorithms which are free to

call the apply_local_heuristic. In fact, we do so to create a variant of Ant Colony Optimization with local search. This variants does local search at the end on the best solution found by ACO.
The candidate solution we implement for bit-strings is simple. It is a bit-flip, we select a random bit and flip it. While a simple idea like this might work for bit-strings, it is not as simple for permutation problems like TSP. We could use, Reverse, Insert and Swap mutations as implemented in the Evolutionary Algorithm. In fact, Reverse is the basis of the 2-OPT move. The K-OPT move is commonly used to solve TSP. While it can be used to generate candidate solutions for any permutation problem, it is defined on a graph tour. The idea is to cut the tour in K different places, generating K chains, and then stitching them together in the optimal way [33]. Notice how an n-OPT move solves TSP exactly as each chain then only contains one vertex and the optimal way of stitching chains of size one together is the optimal permutation. We should not implement the n-OPT move as it requires exponential time, in fact the running time of a K-OPT move is exponential in K, so we need to choose a small K.

K-OPT	2-OPT	3-OPT	4-OPT	5-OPT
Chain combinations	2	8	48	384

However, while we need to choose a small K, a larger K is also more powerful. This motivates the implementation of 2-OPT and 3-OPT moves. They respectively considers 2 and 8 ways of stitching the two and three chains together. If we were to choose a higher K it would be costly, and doing many 2 or 3-OPT moves instead might then be more worth it.

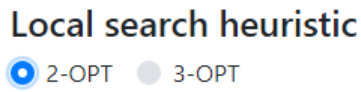


Figure 12: Configurable local search heuristic for permutations

We implement the candidate solution for 2 and 3-OPT by drawing respectively 2 and 3 random numbers used to cut the current tour into chains and by selecting the best chain combination that produce a different tour. This reduce the candidate chain combinations to 1 and 7. The last one is covered through the probability of not accepting the candidate. We now need a way to generate these other chain combinations by manipulating the corresponding permutation of the tour. This is simple for the 2-OPT case where there is just one candidate and we can do so by reversing the permutation order of either chain. The idea of using reverse operations can also be applied to the 3-OPT case in order to generate the other 7 combinations.

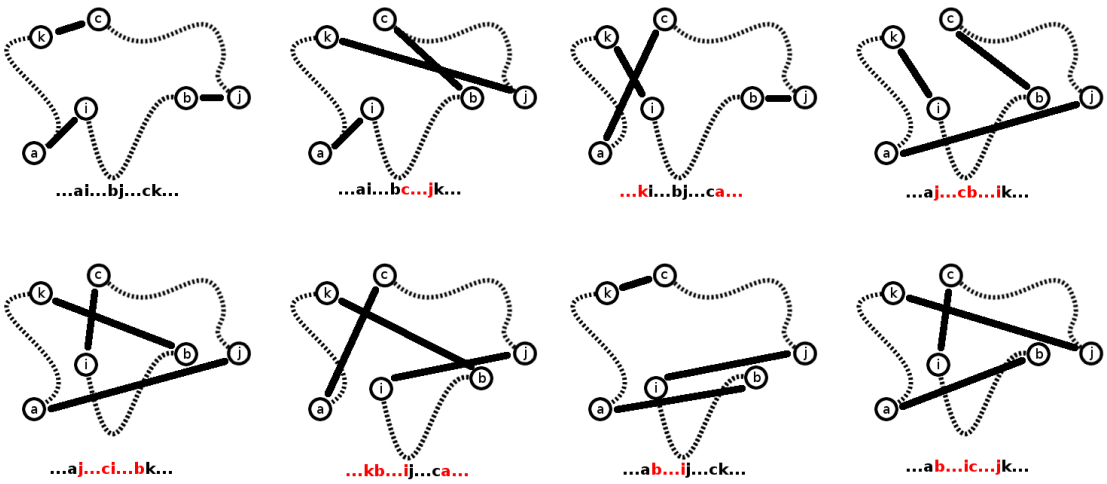


Figure 13: All 8 ways to reconnect the chains in a 3-OPT move.

In fact, we can hit all of the combination through combinations of this reverse function. Considering figure 13, where all chain endpoints are named, we just need to apply the reverse function accordingly. The following table considers the top-left tour, the operations in the corresponding spot for a tour can then be applied to produce it.

	reverse(c,j)	reverse(a,k)	reverse(a,k) reverse(c,j)
reverse(a,k) reverse(b,i) reverse(c,j)	reverse(a,k) reverse(b,i)	reverse(b,i)	reverse(b,i) reverse(c,j)

The listed permutation under each tour in figure 13 highlights in red which values in the permutation needs to be changed in order to generate the chain-combination. Particularly it is interesting to look at the bottom left tour which requires all three reverses, thereby changing all values in the permutation. However, because it is also possible to achieve this tour by reversing chain j...c and then swapping it with chain i...b, all the values are not highlighted in red. It reveals there are also other ways to achieve these combinations and in this bottom-left example, it would be slightly faster to generate the tour by introducing this chain-swap operation. We decide the benefit of this is too small to justify the added complexity and stick to generating all combinations through reverse functions.
When generating these 7 combinations for a 3-OPT move we can imagine how an acceptance probability could decide such that it is possible to transition to each of them. However, we follow the literature which specifies that the 6 worse are discarded leaving only the best as the candidate solution and subject to the acceptance probability [7]. It would be interesting to explore a variant of Simulated Annealing capable of considering more than one candidate but this is out of the scope of this project.

5.4.2 Evolutionary Algorithm

When implementing the evolutionary algorithm we take an approach that is versatile enough such that it is possible to configure it to be equivalent to common evolutionary algorithms like RLS, (1+1) EA and (μ+λ) EA. We can define each of these through the use of three parameters. How many mutations are created, how many offsprings are generated per iteration, λ, and what the population size is, μ. RLS, randomized local search, works by creating a mutation at random in each iteration, then accepting it, if it is better. It is not unlike (1+1) EA, which may create more mutations but also limited to $\mu = 1$ and $\lambda = 1$ [29]. These parameters and the choice of mutation distribution gives us all the possible combinations we need. In fact, we can also configure the evolutionary algorithm to other ideas like creating a RLS version with multiple offspring.
μ and λ are provided in the provide_parameters method. We initialize μ individuals in the init_problem method to random feasible solutions. These individuals are then mutated in the next_iteration method. This is done by selecting λ random individuals, cloning them and then calling create_mutation method on the clone. Finally, the original and cloned individuals are sorted based on fitness value and the μ best individuals are kept to the next iteration.
The create mutation method is defined in the specific search space implementation such that we can apply different mutations for different search spaces. For the permutation search space, we do reverse, insert, and swap mutations based on which mutation is configured. All of these selects two indices at random. The reverse mutation reverse the order between these indices. It is also the one used by Simulated Annealing to do the 2-Opt and 3-OPT moves. Insert takes the value at the first index and inserts it into the second index, moving all other indices in between over. Finally, the swap mutation simply swaps the two values at the two random indices. For bit-string we only implement the bit-flip mutation which simply flips a bit. The other configurable property that applies to the create_mutation method is related to how many of these operations we do, the above-mentioned choice of distribution. In genomes, every DNA base pair has a chance to mutate [8], it is not only limited to one random pair being mutated. We reflect this behaviour in our bit-string implementation by providing a change of 1/n to flip each bit, where n is the bit-string length. This way the expected number of bits we flip is one, the same as when selecting a random bit to flip, but we may flip more or none. Which of these behaviours to choose is then configurable as seen in figure 15.

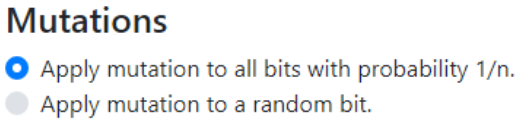


Figure 14: Frontend implementation of configurable mutation behaviour for bit-strings.

This type of behaviour can be generalized across search spaces. For the permutation search space we could do one of the three mutation operations on each pair combination with $1/n^2$ probability, with n being the permutation length. The expected number of operations in this case would also be one. This is simply realized by linearity of expectation, $1/n^2 \cdot n^2 = 1$. However, generating n^2 random numbers is not efficient when we only expect to do one mutation. So in order to speed it up we employ a commonly used technique, where we draw the number of operations to perform from a distribution and then select that many random pairs of indices to perform the operations on. This would be the binomial distribution with $n = n^2$ and $p = 1/n^2$. The problem with the binomial distribution is that we still have to generate the n^2 random numbers. To work around that, we approximate it with the poisson distribution which is very

similar when n is large and p is small. The benefit of using the poisson distribution is the fact we only need to generate a random number per operation we do. This actually means it is theoretically possible to do more than n^2 operations, in fact there is no limit to the largest value we can draw. However, this is not a problem since we utilize a poisson distribution with a low expectation, namely one, so the probability of drawing a large value is incredible slim. This is also part of the reason why the poisson distribution approximates the binomial distribution in our case.
Consequently we implement this similar option for the permutation search space.

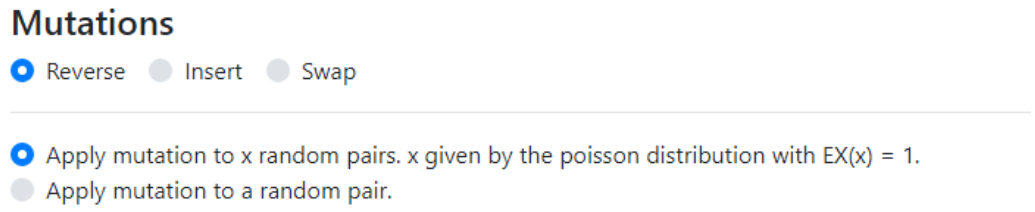


Figure 15: Frontend implementation of configurable mutation distribution for permutations.

5.4.3 Ant Colony Optimization

Before implementing ACO we need to consider the construction graph for bit-string problems on which for ants to walk an lay pheromones.

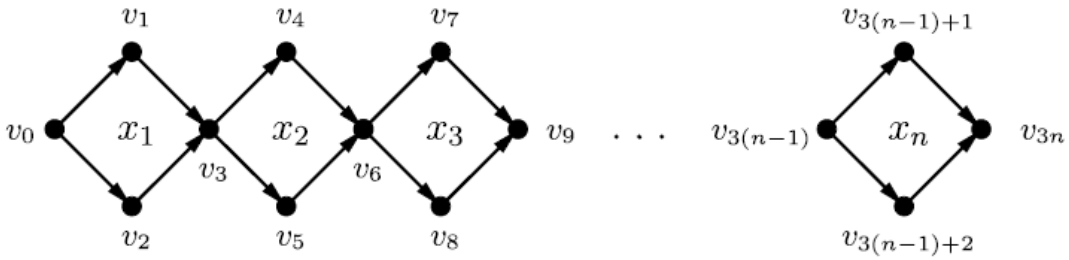


Figure 16: Construction graph for pseudo-Boolean optimization. The figure is an extract from "Analysis of different MMAS ACO algorithms on unimodal functions and plateaus" figure 2 [31].

This is commonly done as seen in figure 16 where a graph is constructed for the bit-string problem such that there are two choices per bit represented by edges. For instance, an ant stats in v_0, the choice of first bit is x_1 is then represented by whether the ant transitions to v_1 or v_2, after which it always goes to v_3 before repeating the process for the next bit. This is also how we contruct the bit-string graph for ACO. A more intuitive analogy, which is the same, is to consider two buckets for each bit. We can place pheromones in these buckets and the ants does a weighted choice depending on the pheromones in these two buckets for each bit.

When implementing ACO, each virtual ant is represented as a potential solution and initialized in the init_problem method. In this method we also initialize the pheromone and heuristic

influence/bias as a ProbabilitySpace. This heuristic influence is useful to guide the selections made by the ants when the pheromones are not yet dominating. For The Traveling Salesman Problem, it is a matrix of edge costs such that shorter edges are more likely chosen. It may not be relevant when applied to other problem classes. For instance, when considering the bit-string problems, choosing between 0 and 1 carries no backing information we can use as a bias so it needs to be left out. We can do so by still incorporating it but setting all values in the matrix to 1.
When generating a feasible solution for a virtual ant we use the following formula as acceptance probability for each potential edge-choice.

$$p_{xy}^{k} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{z \in \text{allowed}_x} (\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})}$$

This formula is commonly applied for ACO when solving TSP. It gives us control over how dominant the pheromones and the heuristic influence should be [12]. τ denotes the pheromone matrix while η is the above-mentioned heuristic, namely edge costs for TSP and all ones for bit-string problems. α and β are then control parameters which let us choose how sensitive the pheromones and heuristics should be to changes as well in relation to each other. We notice how it is also possible to disregard the heuristic influence with $\beta = 0$. We can similarly turn off the pheromones with $\alpha = 0$. The result p_{xy}^{k} is the probability that ant k walks from x to y during the tour. Consequently, x and y for τ and η, is the pheromone and bias on the edge from x to y.
When implementing this formula we realize it is not necessary to work out the denominator for each assessment by calculating $(\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})$ for all possible choices. Instead we precompute a matrix with values for $(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})$ then use these values as weights and ban all disallowed choices.

xy	0	1	2	3
0		2	2	2
1			10	2
2		3		4
3		4	1	

xy	0	1	2	3
0			2	2
1			10	2
2				4
3			1	

xy	0	1	2	3
0				2
1				2
2				4
3				

Figure 17: Example of edge selection probability implemented as weights. Blue is the current vertex ID, green is the randomly selected vertex to transition to. Black is banned choices. Cell values are weights.

Figure 17 shows how a permutation might be generated for TSP in a small graph of 4 vertices. We start with vertex 0 and generate the permutation 0123. Initially, we only ban transitions to the same vertex as well as the starting vertex 0. Then 1, 2 and 3 is selected with equal probability as they have the same weight. The green cell highlights the choice of the random generator, 1. We then ban 1 from the possible choices, as we only transition to new edges. The choices are now 2 and 3 but with the weights 10 and 2. This could be due to there being many pheromones on the edge from 1 to 2, or it being a short edge causing a large bias. We sum the row, 12 in this case, and generate a random number from 0 to 12. We look at the first possibility, 2, and subtract the corresponding weight, 10, from this random number. As in this case, if the result of this is 0 or less, we transition to vertex 2, otherwise we repeat this subtraction for the next possibilities, which would result in transitioning to 3.
The implementation of this matrix and these choices are done in the ProbabilitySpace space class. We implement a function for selecting weighted indices with disallowed choices. Recalling

figure 7, the ProbabilitySpace class use the row-major access order. We take advantage of memory locality by keeping the weights for all transition possibilities in the same row. A row may be scanned twice during such a choice, once for summing the weights for allowed choices, and once for figuring out which choice a random number corresponds to. This means these weights are stored next to each in memory, so multiple are fetched at a time, when a cache line is requested from memory.
When it comes to updating the pheromones, we do so in the next_iteration method after generation the permutation as described above. First, the pheromones are evaporated using the evaporation factor, ρ.

$$\tau_{xy} \leftarrow (1 - \rho)\tau_{xy}$$

We then implement two approaches for each virtual ant to discrete new pheromones.
For TSP it applies pheromones to all edges in the graph representation of the problem, that the ant used. We discrete more pheromones for better solutions following the formula [32]:

$$\tau_{xy} \leftarrow \tau_{xy} + Q/L_k$$

Where L_k is the fitness value of the tour for ant k. Q is a constant. It can be used to modulate the amount of pheromones produces that is applicable for the given problem instance. For TSP we realise that for each edge an ant uses, there are (n-1) it did not use. Consequently, there are (n-1) edges we are not putting new pheromones on. We therefore scale Q with n such that the proportion of pheromones on a given graph scales with the problem size. In theory this should make sense but determining whether this approach works best in practice requires some tests. We found it to give better results so we do not change it unless we see cases that cannot be explained or tuned by other parameters.
For bit-string problems we take a different approach to this and implement the Max-Min Ant System, MMAS, as described in "Analysis of different MMAS ACO algorithms on unimodal functions and plateaus" [31]. In this approach, the ants does not update the pheromones with respect to their own fitness value, they instead consider the best found solution instead and use that to update the pheromones. Furthermore, the amount of added pheromones is the the same as the evaporation factor ρ, so after evaporating pheromones we also do the following update w.r.t the edges in the tour of the best solution:

$$\tau_{xy} \leftarrow \tau_{xy} + \rho$$

Finally, MMAS implements pheromone bounds such that the effective probability of a bit choice does not evaporate to 0. These are commonly implemented as $1/n$ and $n - 1/n$, however we make that configurable such that the pheromone bounds can even be applied to the TSP case as well as turned off. The final pheromone update in an iteration becomes.

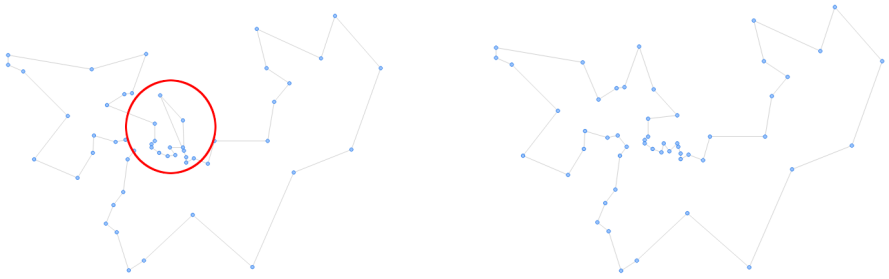
$$\tau_{xy} \leftarrow clamp(\tau_{xy}, \tau_{xy\_\mathrm{floor}}, \tau_{xy\_\mathrm{ceil}})$$

There is some advantages and disadvantages to this approach of updating pheromones. It is much better at hill-climbing, meaning it is good at gradually improving the fitness value, because the pheromone bounds prevents the solution from getting stuck. Also, because we only consider the best solution when updating pheromones, it is very similar to the evolutionary algorithms which are excellent hill-climbers. Consequently, this approach is good when solving the bit-string problems OneMax and LeadingOnes as we can hill-climb all the way to the optimal solution. However for a benchmark problem with a plateau, like Jump, we would most likely benefit from going with a different approach more similar to what we do for TSP. Doing so would allow different solutions to lay out pheromones in each iteration in order to prepare

for overcoming the plateau. We can also apply this reasoning to justify why we do not take a MMAS like approach for TSP. Because the fitness landscape of TSP is bumpy with many valleys a hill-climber might not be effective at approaching the optimal solution.

5.4.3.1 Local search variant

When solving TSP with Ant Colony Optimization we often see a certain issue. In the early process of laying out pheromones, while often finding good edges that are shared with the optimal tour, some other edges might accidentally be selected too many times, thereby receiving too many pheromones. To overcome this we implement a selectable variant of ACO that employs local search on the best found solution at the end of the run [25].



(a) Best solution found before local search. Red area highlights ACO inaccuracies. Length: 7951.

(b) Global minimum found with 10 3-OPT-moves on best found solutions. Length: 7544.

Figure 18: Comparison between best solution before/after local search on a run of ACO on TSPLib instance berlin52.

We can understand how this is effective because it combines the strengths of ACO and locals search. ACO is great at finding a good area/valley in the search space and local search is then employed in order to transition down to the local minimum of that valley. Figure 18(b) clearly shows it in action, where the best solution found in a run of ACO was actually quite good but contained some inaccuracies highlighted in red. 3-OPT based Local search was then able to fix that and actually find the optimal solution.
Because ACO might not find the optimal valley we do not always find the optimal solution with this local search, however it could be the case that this valley is often in the same region as the optimal valley. In this case it would be interesting to explore the effect of employing Simulated Annealing instead of local search, where we start from a very cold state such that Simulated Annealing is only able to explore a few valleys in the region. Perhaps we would then find the optimal solution more often. This idea is out of the scope of this project.

5.5 Frontend - Setup and Visualizations

Apart from keeping a modular architecture across the entire project and in the core computational model, we also take a flexible approach in the implementation of the frontend. We do so by building re-usable components using the Javascript library React JS.

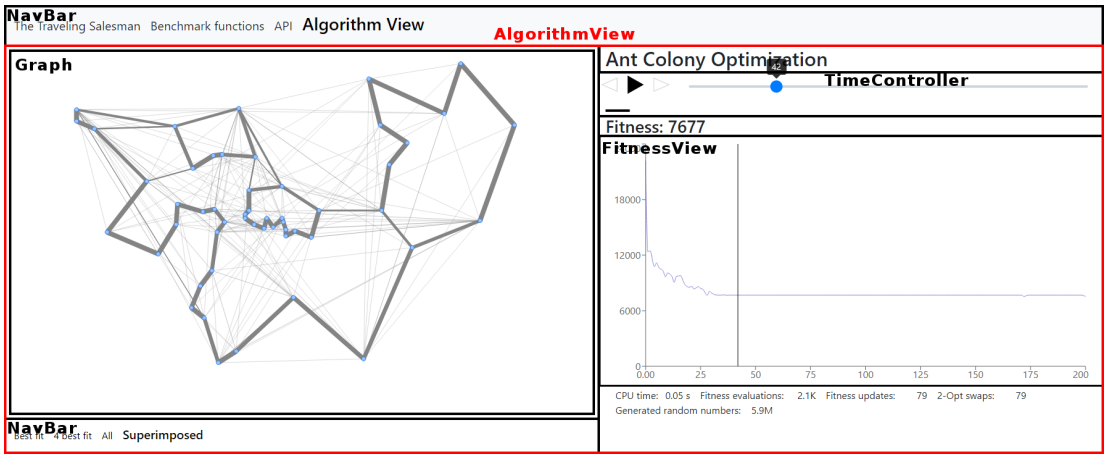


Figure 19: Algorithm view with selected React JS components highlighted.

Each component is in charge of its own functionality. While this is related to the idea of the class in object oriented programming, when reusing them, components are based around the principles of composition instead of inheritance [1]. This is ideal because components output associated HTML markup and the sum of all components render output is the entire HTML model. Similar to how HTML tags can be nested, it is also possible to nest components, which is how the composition is made. In figure 19 we see how the NavBar component is reused as both the main navigation bar but also to switch between individuals represented in the visualization. In the figure we see the AlgorithmView component is in charge of everything relating to visualizing the algorithm. Navigating the NavBar above changes the composition by switching out AlgorithmView with another one, for instance, the setup component.
A design pattern, when utilizing composition, is to always pass data downwards [14]. We can tell a sub-component what data to render, we pass data as props, but we should not ask a sub-component about its internal state and update something accordingly. Doing so would be an anti-pattern in React JS where the idea is to construct the composition with inline HTML in the render method. Then the lifetime of that constructed object is handled by React which the parent component should not make any assumptions about.

```
163   render() {
164     if (this.props.maxOutputIndex <= 0) return "";
165     return (
166       <div className="row">
167 >       <div className="col-2">···
175         <div className="col-10">
176           <RangeSlider
177             min={0}
178             max={this.props.maxOutputIndex}
179             tooltip={"on"}
180             tooltipPlacement={"top"}
181             tooltipLabel={(index) => this.props.iteration}
182             value={this.props.outputIndex}
183             onChange={(changeEvent) =>
184               this.onSliderDrag(changeEvent.target.value)
185             }
186           />
187         </div>
188       </div>
189     );
190   }
```

Figure 20: RangeSlider is handled inline with HTML.

Data is instead retrieved from a sub-component by passing down a callback function to it which the sub-component can call. Figure 20, which is a code snippet of the render method of the TimeController class as seen in figure 19, shows how the onSliderDrag method is passed down to the onChange function, which RangeSlider calls every time the slider value is changed.

5.5.1 Dependencies

A major benefit of implementing the frontend as a website using Javascript is the possibility to take advantage of specialized external libraries. We utilize the package mananger npm to download and maintain these dependencies. Some developers liberally include external libraries in their projects in order to do even relatively small tasks. The idea is to have one ideal module for everything. If there is a bug in one module, it can be fixed in one place, fixing the bug in potentially thousands of projects. It is the "don't repeat yourself" principle taken to the extreme. While this may be a noble pursuit it comes along with some pitfalls. An example of this is the library Left-Pad, which simply pads a string using 11 lines of code. It was unpublished by the developer, along with all of his other open-source projects, due some controversy. Consequently, thousands of projects could no longer build because they relied on it [34].
We limit the amount of dependencies to only those that are seriously time saving and allows us to focus on more relevant tasks, like a graph visualization library or a data plotter. The full list of dependencies we use from npm is Vis JS, Bootstrap, an XML parser, Rechart and the range slider as seen in figure 20. We also install a few others, like jQuery, that these dependencies rely on.

5.5.1.1 Bootstrap

Bootstrap is a styling library. We use it to make the website more visually appealing and responsive to different window sizes. An example of this can be seen in figure 19 where we give a div the class "row" and its contents, the classes "col-2" and "col-10". Row simply means its

content will be aligned horizontally. The "col" classes are used to split this horizontal space up into pieces of 12. So "col-2" means to occupy a sixth of the available horizontal space. Looking at figure 19, the effect of this can be seen as the TimeControllers buttons occupies this sixth, while the RangeSlider occupies the remaining portion of the horizontal space. Apart from this, we also use Bootstrap to style various input fields, buttons and dropdown controllers.

5.5.1.2 Vis JS

Vis JS is a visualization library. We use its module Graph Vis to visualize The Traveling Salesman Problem as a tour in a graph. It is also used during the setup phase where we for instance use Graph Vis functionality to add a new vertex to the graph when the user clicks on the canvas. It can be seen in action on figure 19 where it shows the Superimposed visualization for a run of Ant Colony Optimization.

5.5.1.3 Rechart

We use Rechart to plot our best fitness value for a given iteration. It is handled by the FitnessView as seen in figure 19.

5.5.2 Constructing custom TSP instances

Apart from running the metaheuristics on predetermined TSP instances from TSPLib, we also want the website and API to be practically useful. A useful feature is therefore the ability to construct custom TSP instances. From the API perspective when starting a job, we do so by sending the graph data along in the run command and reading it on the backend side. While this is simply done, building a user interface for constructing a custom TSP instance is a larger task.
We utilize callbacks provided by graph vis to add/delete vertices in a graph wherever the user clicks with a mouse. This is done to construct a quick 2D TSP instance where the edge costs are simply proportional to the 2D distance between the vertices. While doing so is useful for demonstrating the algorithms by creating quick and cool visualizations, it is not versatile enough to handle many use-cases where edge costs are not direct 2D distances. A common use-case would be route-planning, where the user is interested in a good route visiting a certain number of places. Getting from place to place may involve travelling down curvy roads which cannot be represented with the above-mentioned method. Edge cost might also be better represented differently than by distance. For instance, to consider different speed limits when travelling by car, we would like it to be based on the time it takes to travel the corresponding edges.

5.5.2.1 Abstract edge mode

In order to accommodate these use-cases we must be able to specify the edge costs directly rather than the vertex positions. We call this way to setup the TSP instance for Abstract edge mode and leave the other mode as default, which we call 2D - Coordinate mode. The abstract edge mode raises the challenge of figuring out where to position the vertices. An optimal TSP tour, with edge costs specified as the direct 2D distance between vertices, will contain no overlapping edges. This property makes visualizations neat because one can see how a solution slowly progresses to contain fewer of these overlapping edges. With abstract edge costs we cannot guarantee it is possible to find positions such that this property is maintained. In fact, it can be impossible to find perfectly representable vertex positions, for instance if a subset of vertices breaks the triangle inequality, like in figure 21.

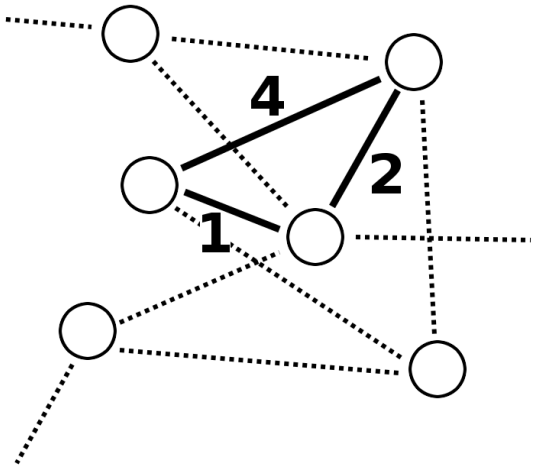


Figure 21: A Graph breaking the triangle inequality

However, while the 2D vertex positions cannot always be perfectly representable of the edge-costs, some positions are still better than others. Looking at the triangle in figure 21, at least the direct distance between the vertices for the edge with a cost of 4 is longer than the two other highlighted edges with lower cost. This raises the problem of finding positions that are as representable of the edge costs as possible. We need to come close in order for the visualizations to make sense. Interestingly, we could use our nature-inspired metaheuristics to solve this problem by formulating a fitness function based on how accurately the 2D distances between vertices represent the corresponding edge costs and by defining the neighbouring states through vertex moves. However, the Graph Vis library includes a physics engine that finds a good solution. It does so by using edges as springs and applying a force when a given edge is not its correct length. Through iterations of this physics engine an equilibrium is found where a good solution is often reached.
Now that we have a way to work out vertex positions we need still need a way to input the edge costs. We do so by creating syntax for adding vertices and setting edge costs. An example of this syntax can be seen as placeholder text in the input boxes for figure 22.

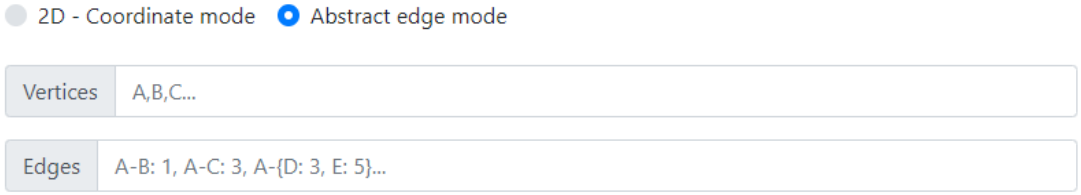


Figure 22: Frontend implementation of abstract edge mode offering syntax for specifying edge costs.

While creating vertices this way through a comma separated list is simple, specifying the edge costs using syntax can be tedious. We make this easier by creating input fields. The user can

click a vertex, highlighting it, and then edit all of its connecting edges through input fields as seen in figure 23.

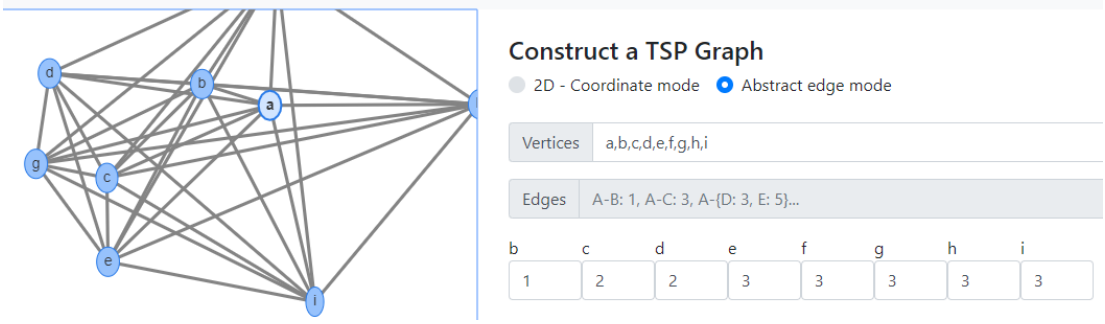


Figure 23: Frontend implementation of how edge costs can be directly edited with abstract edge mode.

When specifying edge costs through these input fields the edge syntax input box is disabled. We keep both options because the syntax is still useful since it allows the user to edit it in an external editor and copy it over, while the input field method is a more contained way of doing it.
Finally we also allow TSPLib instances that are specified by edge costs to be imported. They are available as such in XML format so we use an XML parser to parse the selected file such that we can construct the graph. This also allows the user to create custom XML files and import these.

5.6 Deployment

Since we have no interaction between users it could be sufficient to make the project available open-source and have any potential user deploy it locally on localhost in order to run it. While this may be alright for researchers looking to extend the backend with other metaheuristics and problem classes they are researching, we also want the project to be easily accessible. By having an API endpoint available through deployment it is also possible for developers to implement a different frontend without worrying about the backend.
We use the Microsoft Azure cloud service in order to avoid maintaining our own servers and network security. Microsoft Azure is a cloud hosting service which developers can use to deploy their projects. It offers a large variety of different features for doing so. We take a versatile approach and set up deployment on a virtual machine while avoiding the use of the more high level features offered. This ensures we can easily migrate to another provider or our own server in the future.
We setup the virtual machine with a Linux distribution ideal for hosting and access it through a ssh connection where we have root privileges. Through this connection we clone the repository, install Nginx and other dependencies, compile the backend and deploy exactly like we would on a physical machine.
This deployment can be accessed via the domain (last accessed 30st of May 2020): natureheuristics.northeurope.cloudapp.azure.com. This website will stay the same as a deployment of the source code of this project at least until the 16th of June 2020. As such it can be used to test and reproduce features and results in this report.

5.6.1 Tackling the difference between development and production environments

The fact that we develop on a Windows machine but deploy on a Linux server introduce some additional challenges. The frontend is simple to deploy due to Javascript being an interpreted language and run on the clients machine. It is invariant to these differences. However, we need to compile the backend applications differently for production. The backend is developed using the IDE, Visual Studio. Using Visual Studio speeds up development as we can take advantage of some integrated features like debugging and unit testing. Building is done by the MSBuild compiler which compile C++ into a Windows executable. However, on Linux we neither install Visual Studio nor MSBuild. Instead we use the g++ compiler and write a Makefile which builds both the FCGI API endpoint and the worker application. While both compilers should conform to the C++ standard they are still different, so in a few cases this means something that compiles perfectly fine with MSBuild does not with g++. It turned out to mostly be due to differences between how include files are handled and not the C++ syntax itself. The agile development method means we deploy frequently so we catch problems caused by these differences early and constantly ensure g++ support before we rack up an insurmountable list of server side compilation issues.
Apart from compiler differences there are also some platform specific differences we need to support. For instance, the file system is different on Windows and Linux. Even though file system support was recently introduced in the C++ standard library, g++ only implements experimental support for it. So we resort to writing two implementations of our file system functions, one for Windows and one for Linux. This is done by defining a compiler flag in Visual Studio and then using #ifdef and #ifndef directives to tell the compiler which implementation to use. When using the MSBuild compiler on Windows, the flag is then defined and the implementation wrapped in #ifdef is applied, while the flag is not defined for g++, resulting in the other implementation.

5.6.2 Automatic upkeep

Another thing to consider when deploying a website is the upkeep and maintenance. The backend worker application will potentially run for a very long time in the background without restarting. If it crashes the site becomes useless for everyone and not just one user. Then it would have to be restarted which we would not necessarily be aware of in an acceptable time-frame. We make sure to fix all noticeable bugs and also write unit tests, but despite this it is practically not possible to guarantee for certain that the application will run perfectly forever, and especially so as it grows in complexity. In order to avoid extended downtime such issues could cause, we write server scripts that automatically detects when the worker application is no longer running due to a crash, and restarts the backend within a minute. We also restart the backend anyways every midnight and clear all job files.

5.7 Scale-ability and performance

One of the challenges when implementing algorithms is to write performant code such that the runtime is reduced. This can often conflict with some of the higher level design principles and modularity but it is especially important in this project where performance is also related to scale-ability, the number of users we are capable of serving at the same time. There is a limit to how much we can speed up the algorithms, so we also tackle scale-ability requirements with other separate measures.

5.7.1 Buffered writes

The need for frequent and large data-outputs pose a substantial hurdle, namely overcoming the limitations of the memory hierarchy. Latencies to the CPU cache layers, which are mainly used during execution of individual iterations, are small and on the order of few nanoseconds. The less data we work on, the better caches and memory locality can ensure a high throughput. Even considering a relatively large job, 1000 bit bit-string, for Ant Colony Optimizations with 100 ants, we can keep the required data for processing in minimum about 100Kb. This can even fit in some processors L1 cache offering incredibly fast access timings. [24] However, we may need to output all of this data every, or every few iterations, and this is the main challenge. How do we keep the fast processing speed while outputting large amounts of data to the SSD? We take advantage of the high bandwidth the memory modules offers. A typical SSD is able to transfer about 0.5GB/s which is fast enough such that we have other issues, if we were to allow that much generated data. The problem is the access latency which is limiting [24]. So, we need to send large chunks of data at a time in order to not stall computations with the large latency required when writing to the SDD every iteration. Ideally, while writing a chunk of data, we generate a new one. This is the idea of buffered writes, we overlap computation with communication.
In order to do so, at the end of every iteration during a run of metaheuristics, we append output data for the current state to an output buffer. This buffer is stored in memory, so writing to it is orders of magnitudes faster than to the SSD, meaning the overhead of this is relatively small. We then append to the output file and clear the buffer every 250 ms of CPU-time. This is four times a second and chosen as such to match the same frequency with which the frontend requests new data.
Compared to other performance improvements, buffered writes is absolutely essential when the alternative is writing to disc at every iteration. The below table reports a simple test of the difference for a run of Simulated Annealing for 10K iterations on the Berlin52 TSPlib instance outputting data every iteration.

Output-buffer to disc frequency/time	Every iteration	Every 250 ms
Measured CPU-time	0.38 seconds	0.10 seconds
Actual time with stopwatch	164 seconds	0-1 seconds

Table 2: Performance improvement of buffered writes on a run of Simulated Annealing on berlin52 for 10K iterations.

Note, this test is absolutely still bottle-necked by memory because the computation done by Simulated Annealing for a given iteration is not comparable to outputting the state to a memory buffer. However it demonstrates how much we can alleviate the file-system bottle-neck. We later explore infrequent data-outputs for large jobs which tackles this DRAM bottle-neck. The actual time of the test as seen by table 2 is measured with stopwatch, while the CPU time is reported by the application. Interestingly, we see the CPU time even decrease when utilizing the output buffer. This is due to the fact the writes to the output buffer is counted as part of the iteration and included in the CPU time. For part of the operation to disc this buffer is locked, so the CPU stalls to access it at the end of each new iteration clocking up this additional time. However this 3-4x slowdown is clearly not the main problem, because after each iteration much more time is spent waiting for the output file to be unlocked and available for appending new data. This is where we see a slowdown of more than 100x. Usually extreme numbers like these should raise concern for other influences but it is reasonable, since the access latency difference between layers in the memory hierarchy are multiple orders of magnitudes. However we also

note the job outputting data every 250 ms did not even run for 250 ms, meaning it only got to output data to the SSD once. Running an even larger job could therefore influence these results but not meaningfully so compared to the otherwise extreme differences.

5.7.2 Problem specific logic

One of the main principles and reasons for utilizing metaheuristics is the ability to consider the fitness function like a black box. We do not care about the specifics of the problem in order for the metaheuristics to guide us through the search space and find a good solution. Some metaheuristics might be more applicable to some problems than others also depending on the specific neighbor selection mechanism, but generally speaking they all seek to improve the fitness value to some degree and regardless of the problem at hand.
It is therefore tempting to implement the different problems by simply providing a fitness function which the the metaheuristics can call. This is indeed what we do, we create a new problem by extending its corresponding search space and override the fitness function. It makes it very simple to add new problems, and for the bit-string benchmark functions this is ideal because we are not supposed to know any more information than the results of these black box function calls. However, take 2-OPT based Simulated Annealing applied on TSP as an example. At every iteration we may or may not do a reverse operation on the current solution depending on the new fitness value this operation will result in. By only treating the fitness function as a black box we then have to retrieve the current fitness value, do the operation, get the new fitness value and decide if we want to commit to this change or not. By doing so, we made two calls to the fitness function both of which were expensive evaluations of the cost of the corresponding TSP tour. But in fact, we only needed to evaluate the cost of the two random edges end-points being exchanged by the reverse function. This is only 4 edge cost calls if the algorithm is 2-OPT based instead of potentially hundreds or thousands.
Neither the implementation of Simulated Annealing or the permutation search space can know this fact if we want to keep our code modular and extensible, so we need to write problem specific logic in order to tap into these performance gains. We do this while keeping modularity by creating an over-writeable function, reveseHint(i,j) in our permutation implementation which is used in the 2-OPT function call. reveseHint return how much a given reverse operation will change the fitness value, or if we do not know. Overwriting this function is then optional and can be done in the problem implementation. If a given problem does not overwrite it, the default is to return "do not know" which will make the implementation proceed the black box way as described above. For TSP we overwrite the function and make sure to only assess the 4 required edges before returning the difference in fitness value, the reverse operation will cause. We do a simple test of the 2-OPT based Simulated Annealing run for 1 million iterations on the bier127 TSPlib instance with data-output every 128 iteration in order to evaluate the benefits of this measure.

Sim-Annealing, 1M, bier127	Without reverse hint	With reverse hint
CPU time	1.31 seconds	0.38 seconds

Table 3: Performance improvement gained on 2-OPT based Simulated Annealing run for 1M iterations on bier127 during development of the reverse hint.

It constitutes a substantial performance improvement of 3-4x which is unsurprising since the TSP tour in bier127 is 127 edges long, all of which would otherwise need evaluation twice during each iteration. We therefore expect this measure to improve the performance of larger problems even more and conclude the use of problem specific logic is a worthwhile investment for TSP.

However we do not write similar code for the bit-string problems to get a bitflip hint even though this could improve performance of these even more. The reason for this is the practical applications of TSP versus the theoretical of the benchmark functions. The bit-string problems embrace the black-box like nature of metaheuristics for theoretical analysis. We want to keep them simple and to make it as easy as possible to add more of them.
Unfortunately the reverseHint method is not powerful enough to allow us to do the same for 3-OPT based Simulated Annealing because some of the configurations require two or three reverses. We therefore implement a similar function for getting the fitness potential of a permutation pair, i.e. an edge cost in the TSP case, but we generalize it as such to be appropriate to the permutation search space which does not necessarily deal with graphs. In the 3-OPT implementation we can then use this function to get the required hints for all 3-OPT move combinations.

5.7.3 Multi-threaded job scheduling

Because multiple users can submit jobs for computations at the same time we need a system for properly handling these requests. Do we simply spawn processes/threads for all requested jobs and let the operating system do the work of handling prioritization, create a queue system, or make a custom job-scheduling mechanism.
With the first idea, we would most likely see too many processes competing for system resources causing disproportionate slowdowns [28]. We would lose important control over how long a thread is active at a time. The second idea is simple but allows for abuse. A user could submit very large jobs which another user then needs to wait in queue for. It seems more fair, if necessary slowdowns mostly affect the jobs which already received a lot of service, leading us into the third idea, service based job-scheduling.
We keep track of the CPU-time anyways as part of providing key figures for runs of metaheuristics, to limit the frequency with which we write to disc and also as a stopping criteria. The idea of the job scheduler is to always provide service to the job with least attained service, CPU-time. It should be configured such that we do not swap active jobs too often but often enough such that active users will see their job progress fairly frequently. Working on the same job for an extended period also increase performance due to the data used in computations being constantly hot in the processor cache. A great benefit of this scheduler is the capability to gracefully deal with a high load, also increasing the amount of load that is acceptable. It keeps the application responsive by, albeit potentially slowly, ensuring gradual progress is made across workloads, and if a user just want a quick job done, it can still be done rapidly.
To implement the scheduler, we maintain high control over the computation of algorithms through a job class. We create a do_work() method. From the perspective of the job, the worker can always call this method to get a little bit of work done, it does not care when or how frequently. This patterns also allows for different kind of tasks to be considered and run when the CPU has time to do so.
The scheduler is then implemented through a min-heap data structure based on attained service. This is ideal because we can quickly get the job with lowest service and insert it back in the heap when the worker decides it is time to switch jobs to work on. While we do not expect the scheduler to hold many jobs and fully take advantage of the O(log(n)) time complexity of heap inserts/deletes, we still take special care to performance, because insertion in the scheduler are bound by a critical region. In this way, the worker can retrieve a job from the scheduler, spawn a thread which then repeatedly calls the do_work() method before inserting the job back in the scheduler.
Thanks to the flexibility of the this setup whenever a job is available to work on, if a thread is also available, we spawn one to work on it for 2 seconds. This both takes care to the

above-mentioned cache implications but also ensures a low synchronization overhead. When all threads are active, the effects of the scheduler kicks in and then these 2 seconds might be a little too long for jobs to effectively make gradual progress. However, the main thread then starts working on jobs for shorter intervals of time effectively covering some of these concerns while still frequently scanning the job folder for new commands to run, and spawning new threads when available.

5.7.4 Load balancing

Even though we do not expect a large amount of users visiting the website at the same time, the load per user can be substantial due to the nature of cloud computing which this project essentially is. It is therefore a good idea to prepare for and keep additional scaling in mind. We do this by deploying on two different Azure cloud hosted virtual machines and employ load balancing through Nginx on the main one. We setup the load balancer to listen for http requests on the default port 80. In the case of serving on the same machine, we route the traffic that would otherwise go to port 80 to a different arbitrary port instead. When serving from the other machine, we simply route to that machines IP address on port 80 within the local network. One important considering is to ensure that different requests from the same user go to the same machine that is processing a given job. We setup the load balancer to use IP-hashing, a scheme deciding which machine to use by hashing the users IP-address. Additionally, we setup weighted load balancing because the main machine is more powerful with 2 virtual CPUs versus the other machines single CPU core. This is easily done through the Nginx configuration file where we just assign a weight to each choice [2]. We assign the main machine a weight of twice that of the other. Finally we test that load is indeed spread and balanced by submitting jobs from different IP-addresses and verifying that output files were generated on both machines.

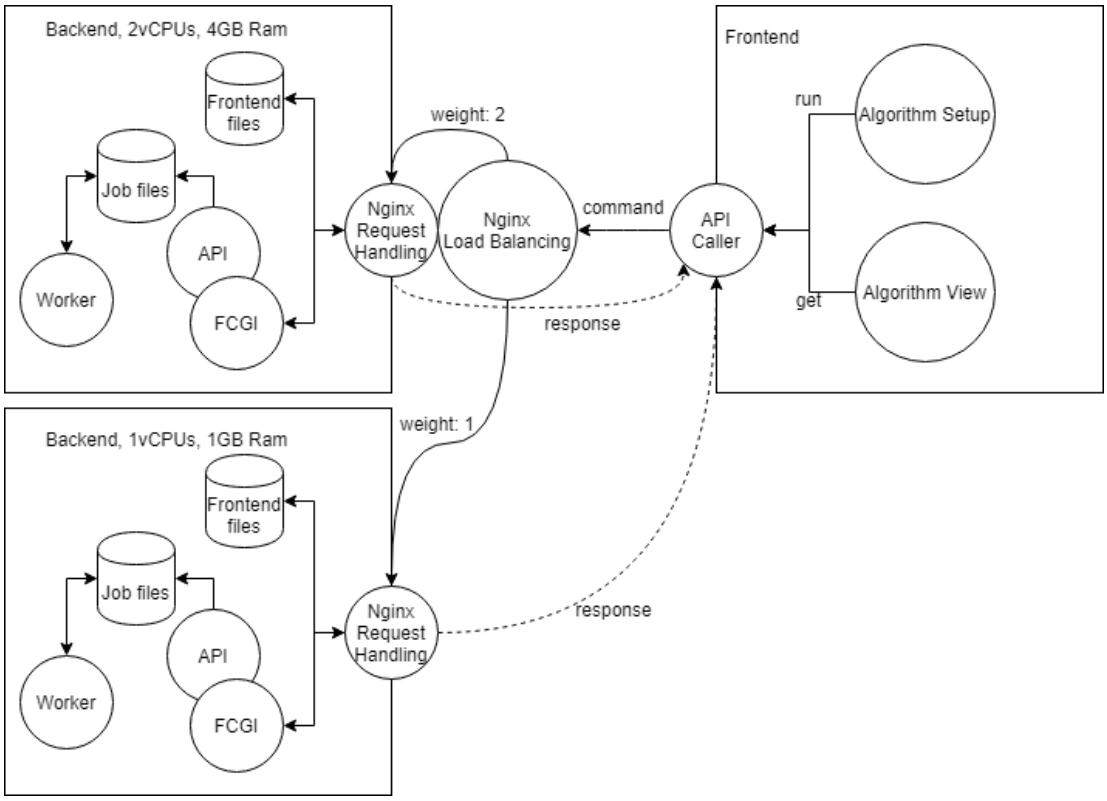


Figure 24: Load balancing addition to the networking architecture

This load balancing setup also provides additional flexibility in terms of request handling. We can use it in the future to setup micro-services and employ separate load balancing for each module. For instance, we can add another virtual machine that only run the FCGI/API and worker application without also providing the frontend. The load balancer would then only direct API commands to it and use another virtual machine to serve the frontend.

5.7.5 Random number generation

During a run of metaheuristics we need to generate a lot of random numbers to guide the decisions made by the algorithm. The number of random numbers used can be substantial. Apart from recent research in true random number generation, leveraging DRAM cells as an entropy source, computers does generally not have a true random source available to draw from [21]. Using an entropy source also requires sampling which can be hard to do efficiently enough when performance is also a priority. The random number generation used in almost all software is therefore pseudo-random, meaning they work deterministically such that the generated numbers are approximately random. Depending on the technique used, the generated sequences can be practically indistinguishable from true random sequences and depending on the use-case it is often good enough [3]. We will likewise use a pseudo-random number generator. We care about performance since we generate so many random numbers it could potentially bottleneck our computations, but we also care about the quality of the random number generator which influence the generated solutions. An inaccurate generator would also introduce more

noise than necessary into our evaluations. The Mersenne Twister is widely used, fast and still manages to pass numerous statistical tests for randomness [3].
We use a C++ implementation of the Mersenne Twister, mt19937, available in the standard library for the C++11 standard.
One of the characteristics of many pseudorandom number generators, including the Mersenne Twister, is the predictability of its output sequence if one knows the current state. While this may be a security issue when used in cryptography it is not a problem for our use-case. However we still need to consider it so we do not start with the same state for each job resulting in the same generated solution for multiple runs of the same metaheuristic on the same problem. The way to overcome this is by seeding the random generator. We provide a number, called a seed, which determines the initial state. As long as this number is different for different jobs, the generated sequences should be independent of each other. A common way to choose this seed is by using the Unix timestamp such that the seed is always different. We do so in C++ by using the chrono library providing access to the system clock and the current Unix timestamp in milliseconds.

5.8 Stopping criteria, job runtime estimation and maximum data output size

In order to visualize a run of metaheuristics the backend outputs a data representation for every state that should be visualized. Ideally that would be every iteration such that it is possible to follow the individual operations that occurs. However, a user may start a large job resulting in millions of iteration which could require on the order of gigabytes to store and transmit. In this case it is not feasible to output every iteration. While doing so would constitute a huge memory bottleneck for metaheuristics like Simulated Annealing or (1+1) EA with small amounts of computation per iteration, it more pressingly challenges the available server resources. We could run out of network bandwidth or storage space for data output files. Clearly we need a limit for how much data a given job of metaheuristics is allowed to output.
We consider 10MB a fitting limit which is large enough to contain data for every iteration of small jobs but will need infrequent data-output for large jobs. It also means we can store the output of thousands of jobs before running out of disc-space. However, whatever this limit is, it is not easy to enforce. While we can estimate the amount of data generated per iteration-output very accurately our implementation of stopping criteria makes it challenging to know how often we should output data in order to enforce the output limit.

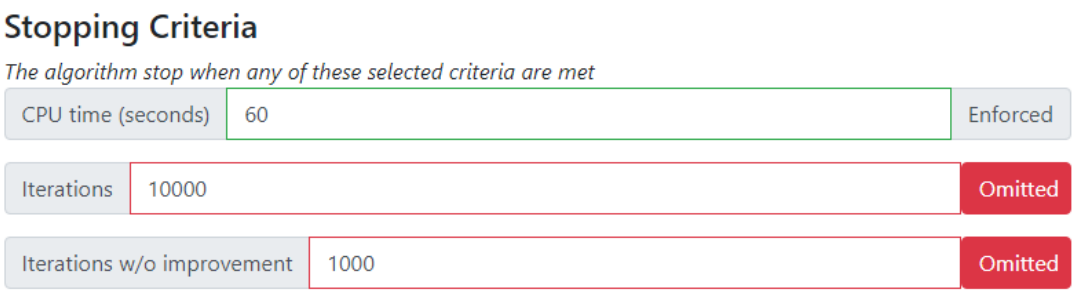


Figure 25: Frontend implementation of configurable stopping criteria

While using max iteration count as a stopping criteria is probably the default use-case, it is also a nice feature to be able to limit the run-time based on number of seconds. For instance, it

allows for more fair comparisons between 2-OPT and 3-OPT based Simulated Annealing where more work happens per iteration in the 3-OPT based case. We also use the CPU time stopping criteria to enforce it never exceeds 60 seconds such that users do not start jobs that run for too long, which also challenges memory resources as the scheduler could then handle too many jobs at a time.
The problem with the CPU time stopping criteria is that we do not know how many iterations of the metaheuristic we will end up generating, which we need to know in order to accurately limit the frequency of data output unless we also base this frequency on CPU time. However, an iteration count based frequency is desire-able as the number of operations between each data output for certain jobs would then be constant, making it easier to make sense of the visualization. Ideally we should also tell the user in advance when a given job does not output data every iteration and how often we then output data.

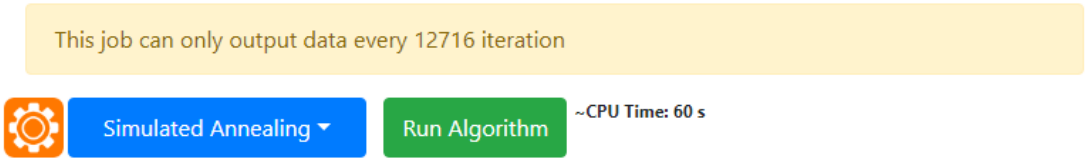


Figure 26: Frontend implementation of how the data output frequency for a large job of Simulated Annealing with stopping criteria configured as in figure 25 is reported.

5.8.1 The analyse command

In order to implement the report shown in figure 26 we need an additional API command and a way of estimating how many iterations a given run-time corresponds to. We could avoid this API command by implementing the estimation twice, both in frontend for reporting and in the backend to enforce it. Doing so would heavily violate the "don't repeat yourself" design principle which can sometimes be justified to avoid excessive complexity, however these estimations are complex and may be tweaked multiple times which could cause the implementation to become out of sync and the reporting to be false. So, apart from the "run" and "get" command used to start a run of metaheuristics and get its data-output, we also implement an "analyse" command. The idea of the analyse command is to analyse the job before starting it and tell the user useful information about it, most notably, the frequency of data output, but we also use it to report an estimation of running time and whether or not the proposed job is even valid. When we can estimate the number of iteration from a given CPU-time stopping criteria, we can easily implement the reverse and give an estimate of total run time, also if iteration count is the stopping criteria of choice.

5.8.2 Number of iterations from CPU-time estimation

Estimating the expected number of iterations is tricky. For instance, a permutation reverse operation have worst case O(n) time complexity as the number of swaps in the operation grows with the problem size. However, in our 2-OPT based Simulated Annealing implementation, when assessing whether or not we accept the reverse operation it takes constant time to check how much a reverse operation will change the quality of the solution. Consequently this leads a 2-OPT operation to sometimes take constant time and other times take O(n) time.
There is also the challenge of estimating when we are memory or compute bottle-necked which changes the relevance of different factors in the estimation. So in order to not over-complicate

the calculations we accept a certain margin of error. This also means we cannot enforce the data output limit exactly so we should set the limit lower than otherwise. However, as long as the estimation is somewhat accurate, even within a broad range like 30%-300%, the data output limit will still be sensible and the estimated run-time still useful.
We implement the estimations in the backend by declaring various virtual abstract functions in the Algorithm and Searchspace class that return how many of a certain operation we estimate is possible to do per second. For instance, the mutations_per_second method is overriden to mean bit-flips per second in the SearchSpaceBitString class. We do measurements of different problem sizes and configurations to find first sensible estimates for these functions. When implementing these estimates, the data output frequency change for some problem sizes which then changes the run-time. This effect can be especially large for some configurations with frequent data output that experienced a shift between being memory/compute bottle-necked. We therefore iteratively update the estimates through trial measurements. Both during the iterative process of updating these estimates and finally, we test the quality of the estimations by running a wide array of different configurations limited by iteration count and comparing the estimated CPU time with the actual CPU time.

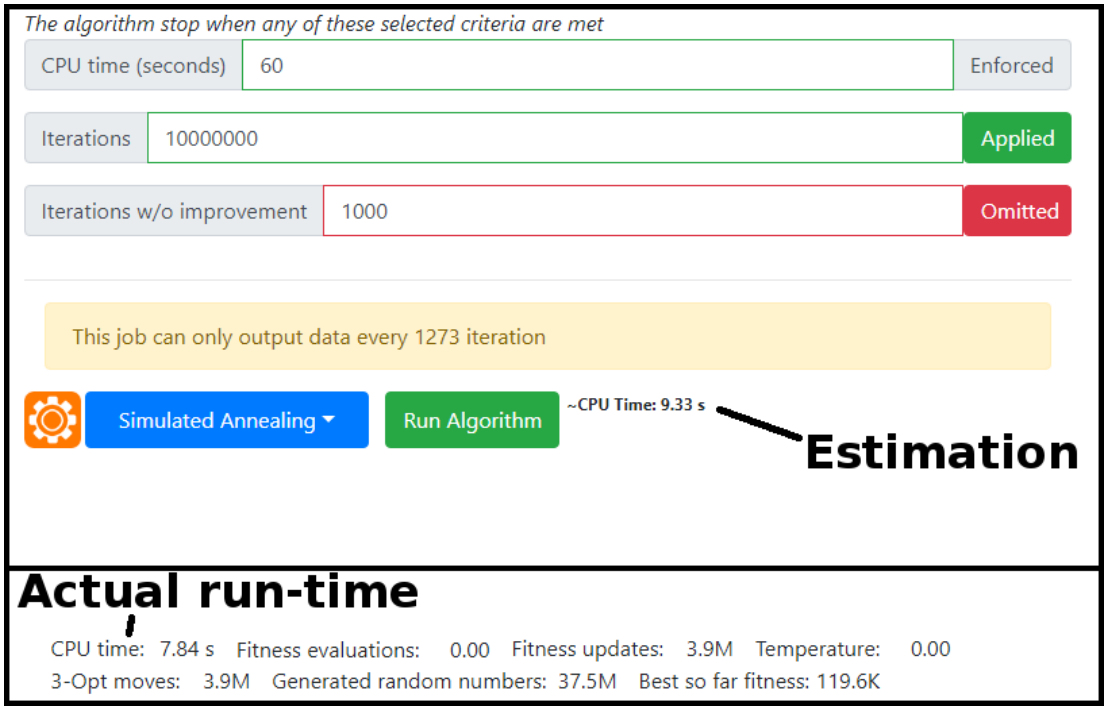


Figure 27: Testing the estimation on a run of 10M iterations of 3-OPT based Simulated Annealing on the bier127 TSPLib instance

Figure 27 shows an instance of this process where the estimation is good. The below table reports a representative selection of the final tests and the estimation accuracy of these. Because the method used to achieve these estimates is more akin to trial and error rather than a standard approach we make sure it rigorously covers a large variety of test cases and the entire range of various influencing parameters.

Test cases. Estimated CPU Time vs actual	Est.	Actual	Compare
Sim. Annealing. 2-OPT bier127. 1M.	0.4s	0.4s	100%
Sim. Annealing. 2-OPT bier127. 100M.	31.16s	28s	111%
Sim. Annealing. 3-OPT. berlin52. 100M.	41.25s	50s	82.5%
Sim. Annealing. 1000 bits. OneMax. 1M.	2.18s	4.53s	48%
Sim. Annealing. 1000 bits. LeadingOnes. 1M.	2.18s	1.6s	136%
(1+1) EA. Reverse/Poisson. bier127. 10M.	10.79s	11s	98%
(100+100) EA. Reverse/Poisson. bier127. 100K.	10.79s	13s	83%
(100+1) EA. Insert/Poisson. berlin52. 1M.	6.3s	8.28s	76%
(100+100) EA. Swap/One. berlin52. 100K.	5.37s	6.57s	82%
(1+1) EA. 1000 bits. OneMax. 1M.	14.94s	15s	100%
RLS. 1000 bits. OneMax. 1M.	3.86s	3.04s	127%
(100+100) EA. 10 bits. OneMax. 100K.	7.77s	5.37s	145%
(1+100) EA. 1000 bits. LeadingOnes. 10K.	14.94s	13s	115%
(100+1) EA. 1000 bits. LeadingOnes. 100K.	1.99s	1.81s	110%
ACO. 100 ants. Bier127. 1K.	8s	5.88s	136%
ACO. 1 ant. Berlin52. 100K.	29.64s	38s	78%
ACO 100 ants. 1000 bitstring. OneMax. 1K.	6.45s	7.54s	86%
ACO 1 ant. 10 bitstring. OneMax. 10M.	26.31s	25s	105%

Table 4: Comparing expected CPU-time with actual CPU-time.

These results are accurate enough such that we are satisfied with the estimations. In fact, they can be worse, but since we cannot cover every test configuration we have high expectations for the cases we can cover. One of these cases are noticeable more off that the others, the "Sim. Annealing. 1000 bits. OneMax. 1M." case. It is Simulated Annealing run on a 1000 bit bit-string for one million iterations to solve OneMax. It is somewhat inaccurate because the evaluation of the fitness function carries a larger influence compared to the other algorithms and we do not implement bit-string problem specific estimation logic. The OneMax fitness function usually takes longer to evaluate than the LeadingOnes, because OneMax needs to scan over all bits, while LeadingOnes can break at the first zero. Consequently, we see the LeadingOnes test for the same configuration took about 1/3 of the time, while the estimation was the same for both. We do not correct this because 1000 bits is the largest bit-string we allow, which is also where the fitness evaluation constitutes the largest portion of the running time, hence the difference between OneMax and LeadingOnes running time is lower for all other problem sizes. The estimation is still good enough for both cases and is in between, so correcting for one of the problems would make the other estimation worse, unless we introduce bit-string problem specific estimations.

5.9 Short remarks on other implemented features

Due to our modular and flexible approach throughout development some otherwise rather complex features were relatively simple to implement. Consequently we managed to implement all of our "must have" and "should have" features, and most of our "could have" features. The complete list is available in scope, it is large enough such that we cannot hope to cover all of them in this chapter. We provide short remarks about some of these features.

5.9.1 API documentation

The main frontend navigation bar contains a tab to navigate to API.

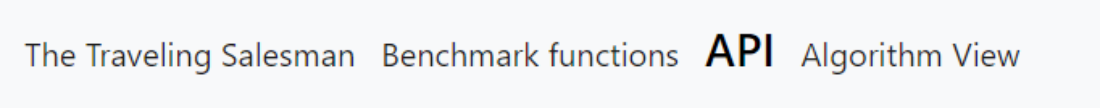


Figure 28: Main navigation bar

When selected the user is presented with interactive documentation for the three API commands which we implemented. They are selected such that the user can for instance click the "run" command and then see documentation for it. This documentation contains the required syntax to run the command specified as a context free language, as well as the data requirements and returned JSON format.

5.9.2 Bit-string benchmark functions, Jump, TwoMax and Needle

As well as OneMax and LeadingOnes, we also implemented the bit-string benchmark functions Jump, TwoMax and Needle.
The jump function is a modification of OneMax with a plateau around, but not on, the optimum and complement of a certain size [16]. In this plateau, the fitness values are 0, otherwise it is the same as OneMax. For our implementation the plateau size we use is $\sqrt{n}$ which can be considered a short jump. This function stops hill-climbers from reaching the optimal solutions due to the plateau, thereby requiring more sophisticated techniques.
TwoMax is similar to OneMax in the way that the fitness value for a bit-string is defined as the maximum of the hamming distance to either all ones or all zeros [15]. This causes the function to be bimodal instead of unimodal like OneMax, but hill-climbers can still effectively find the optimal solution as both peaks are optimal.
Needle is defined on the bit-string such that all fitness values are zero, except for the all-ones bit-string, which has a fitness value of one [16]. It is interesting because it has an unrestricted black-box complexity proportional to the size of the entire search space, namely exponential in n, as it is impossible to gather information about the optimum from a given fitness value of a solution. It can be used to sanity check that a given algorithm is not capable of exploring the entire search space making it redundant.

5.9.3 Parallel adaptive Simulated Annealing variant

We came up with an idea to improve Simulated Annealing by employing an adaptive cooling scheme. While adaptive cooling schemes are commonly implemented as variants to Simulated Annealing, this one is distinct in the manner that it considers multiple runs of Simulated Annealing at the same time and with the same temperature. The idea is then to measure the difference between the different solutions and cool them more rapidly when they are more different. By applying rapid cooling when the difference between solutions are large the idea is that should cause the different solutions to be forced down into different valleys meaning it is more likely that one of them is the optimal solutions. While we implemented it and it is selectable in the framework we discontinued further analysis of it due to two major issues.
1. Some optimization problems, like TSP, contains many valleys such that it is already likely that independent runs moves to different valleys.

2. It seemed to be extraordinarily sensitive to parameters such that it cannot effectively be used in different problems. TwoMax proved to be ideal in this regard for testing it, as it should supposedly cause two solutions to go to opposing directions in the bit-string boolean hypercube. However, we were mostly unsuccessful in this endeavor.
On the other hand, and the reason we keep it in the product, is the fact that it also did not seem to perform worse than independent runs.

6 Evaluation

One of the motivations for implementing and visualizing the metaheuristics is to evaluate the working principles of the nature-inspired metaheuristics. Such an evaluation requires the implementation to be functionally correct in order to be accurate. We therefore both evaluate the correctness of our implementation as well as the working principles of the metaheuristics. When evaluating the working principals of the metaheuristics we focus on their appropriateness to solve certain problems, mainly TSP. While our implementation of Simulated Annealing and ACO proved to be effective TSP solvers under certain circumstances, it was not as much the case for the Evolutionary Algorithm on which we therefore take a more reflective approach to the evaluation and consider the type of problem it may then be more suitable for.

6.1 Unit tests

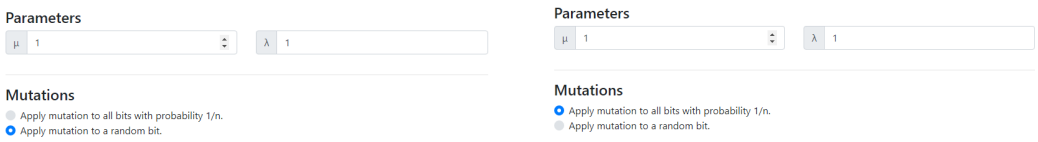
When developing a software project it is usually a good idea to introduce tests such that correctness certain functionality can be guaranteed and does not break when other aspects of the project are introduced. While regression tests are hard to introduce due to the non-deterministic nature of metaheuristics we can still incorporate unit test with some crucial features. The correctness of the core functionality in the worker application is most important. So we write unit tests to ensure correctness of various features like the 3-OPT operation, the scheduler, and fitness evaluations of TSP instances. Visual Studio implements integrated support for units tests which we use to create a unit test project and conduct our tests.

6.2 Comparing test cases to theoretical results

Before evaluating our implementation of the nature-inspired metaheuristics on TSP instances we compare test results on bit-string benchmark functions to theoretical expectations. We limit our selection of tests. For instance, since OneMax is a uni-modal function, Simulated Annealing does not benefit from its cooling scheme when solving it, so it may therefore make more sense to study variants of the evolutionary algorithm that are similar to Cold Simulated Annealing on benchmark functions like OneMax.

6.2.1 Solving LeadingOnes with (1+1) EA and RLS

We configure the benchmark function LeadingOnes to the largest allowed bit-string size of 1000 bits and setup two cases of the Evolutionary Algorithm to the equivalent of (1+1) EA and RLS.

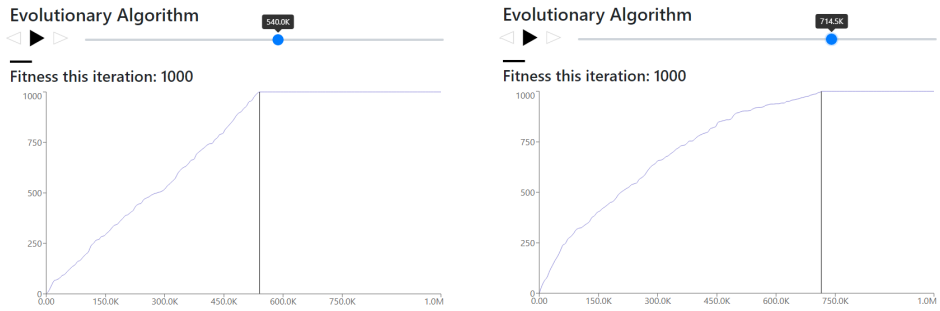


(a) RLS (b) (1+1) EA

Figure 29: How the Evolutionary Algorithm can be configured to RLS and EA(1+1).

Finding the theoretical expected iteration count for RLS to solve LeadingOnes is strait-forward. The probability of flipping the left most bit from a 0 to a 1 is 1/n. Expectantly it then takes n iterations to flip that bit. If the bit-string is randomly initialised as it is in our implementation, we must on average flip n/2 bits. Further, by means of linearity of expectation, the expected

time to flip all bits is then $n^2/2$. We therefore predict our RLS configuration to find optimum of LeadingOnes at around the 500K iteration mark. Note, even with a 0-initialization, we would still only need to consider about n/2 bit-flips of the first 0-bit as the bits after the leading ones are progressively randomized by the algorithm.
The same analysis is somewhat more complicated for (1+1) EA. This is because, while flipping the right bit is still done with 1/n probability, it may also flip any of the preceding bits thereby ruining the fitness value. The probability of increasing the fitness value by correctly flipping the bit is then between $1/n$ and $\sim 1/en$ depending on the current number of Leading Ones [19]. The expected time to flip a bit is then between n and en, and finally, the expected time to flip all bits is therefore between $n^2/2$ and $en^2/2$, or [500K;1.36M]. We therefore predict our (1+1) EA configuration to find optimum of LeadingOnes after RLS does but not significantly after 1.36 million iterations.



(a) RLS solved LeadingOnes in 540K iterations

(b) (1+1) EA solved LeadingOnes in 741.5K iterations

Figure 30: Testing RLS and (1+1) EA on LeadingOnes

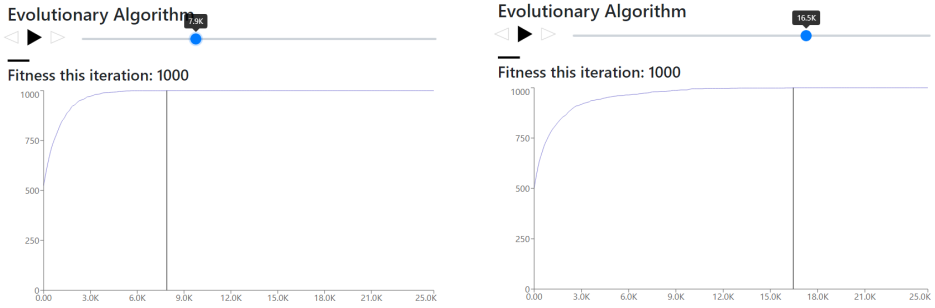
As seen in figure 30, both of our predictions were accurate. We can also see the from the figures that RLS did in fact seem to increase the fitness value linearly by means of a constant probability of increasing of flipping the left-most 0-bit. This is unlike (1+1) EA, where the fitness function increased more slowly due to the probability of correctly flipping the left most 0-bit slowly falling from $1/n$ towards around $1/en$.

6.2.2 Solving OneMax with (1+1) EA and RLS

We do a similar analysis of the ability of (1+1) EA and RLS to solve the OneMax benchmark function. When finding the expected time to solve RLS on OneMax, we realize it is the same as the coupon collectors problem [23]. For each randomly selected bit, it will be fixed to 1. So, like collecting coupons, there is no utility in getting the same again. With random initialization this means we need to flip on average half the bits, we start with half of the coupons. In coupon collector, the probability of getting a new coupon depends on how many we already collected, in fact it is $(n-i)/n$ if the current collections contains i different coupons [19]. The expected time to collect a new coupon is therefore $n/(n-i)$. Through linearity of expectation, we can then find the total expected time to solve the coupon collectors problem given that we start with half of the coupons.

$$\sum_{i=n/2}^{n-1} n/(n-i) \xrightarrow{n=1000} \sum_{i=500}^{999} 1000/(1000-i) = 6793$$

As such, it is our prediction that our test will spend 6793 iterations to solve OneMax with RLS. We do not do the same analysis for (1+1) EA. Instead, it can be shown it is factor e slower for large n [37], so our prediction for (1+1) EA is $6793 \times e = 18465$.



(a) RLS solved OneMax in 7.9K iterations

(b) (1+1) EA solved OneMax in 16.5K iterations

Figure 31: Testing RLS and (1+1) EA on OneMax

As seen in figure 31 our results were again fairly accurate. Counter-intuitively, we do not expect it to match the expectation exactly, it is more likely that our results are somewhere close. In fact, after 5.6K iterations, RLS only needed to flip one bit which it then spend 2.3K iterations doing. If it just happened to do that quickly, like in a few hundred iterations, which would not be unlikely, we would see a noticeable different result.

6.2.3 Collapsing MMAS to (1+1) EA

It is a well known result that the Ant Colony Optimization algorithm, MMAS, collapses to (1+1) EA when configured with 1 ant, a large ρ, and pheromone bounds $(1/n, n - 1/n)$ such that the bounds are hit immediately for all bits [31]. We test whether our implementation of ACO on bit-strings does so by seeing if we can replicate the above result of (1+1) EA solving OneMax with ACO.

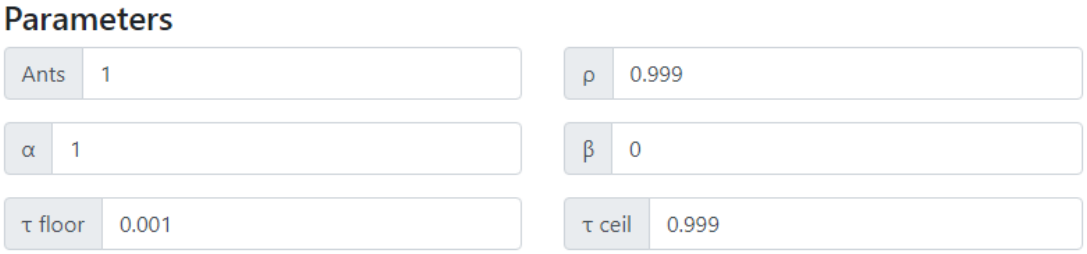


Figure 32: ACO configuration that is the same as (1+1) EA when run on a 1000 bit bit-string.

After running our test we achieve the result in figure 33.

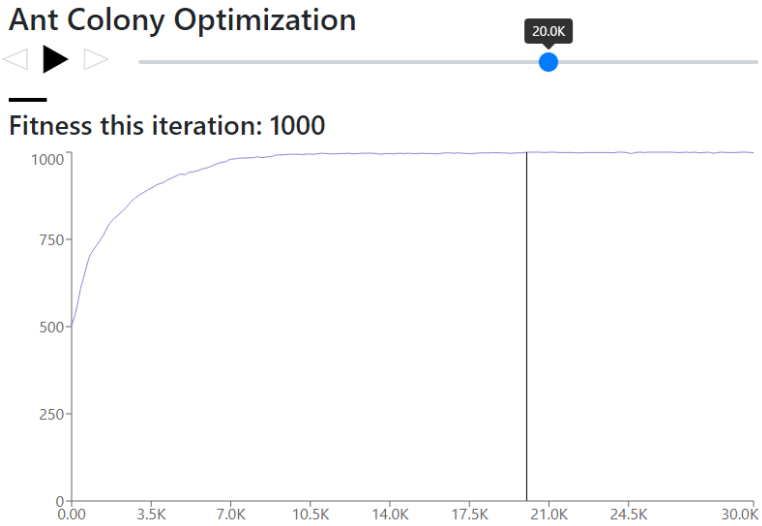


Figure 33: ACO collapsed to (1+1) EA solved OneMax 20K iterations.

This time it took 20K iterations which means we are on the other side of our expectation, namely 18465. It is close and the fitness value also progressed like it did for the (1+1) EA test case as seen by the fact Figure 33 looks very similar to figure 31(a), so we consider it a success.

6.3 Solving TSP with Simulated Annealing

The main feature that distinguish Simulated Annealing is the acceptance probability of transitioning to a worse state in hopes of finally transitioning to a local optimum that is more likely to be close to the global optimum. We test the ability of our Simulated Annealing implementation to attain this property by comparing it to a configuration of the evolutionary algorithm that is fundamentally the same as cold 2-OPT based Simulated Annealing where a worse neighbor is never selected. This configuration is seen in figure 34 and mimics the cold 2-OPT move by applying the reverse mutation to a random pair thereby generating an offspring that is selected only if it is better.

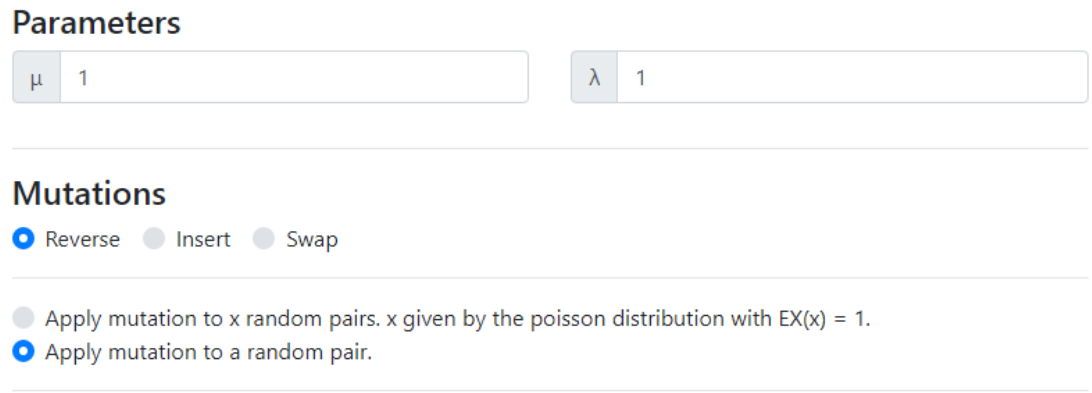


Figure 34: A configuration of the Evolutionary Algorithm that is the same as 2-OPT based Cold Simulated Annealing

We compare 30 independent runs of the two algorithms on the berlin52 TSPLib instance for 100K iterations where we know the optimal solution is 7544 (The optimal solution is listed as 7542 on the official TSPLib website. That is due to a different rounding behaviour in the calculation of edge costs from vertex positions. We are unconcerned with this difference.). For the Simulated Annealing case, the parallel instances feature is helpful in this setup to start all of these runs at the same time.

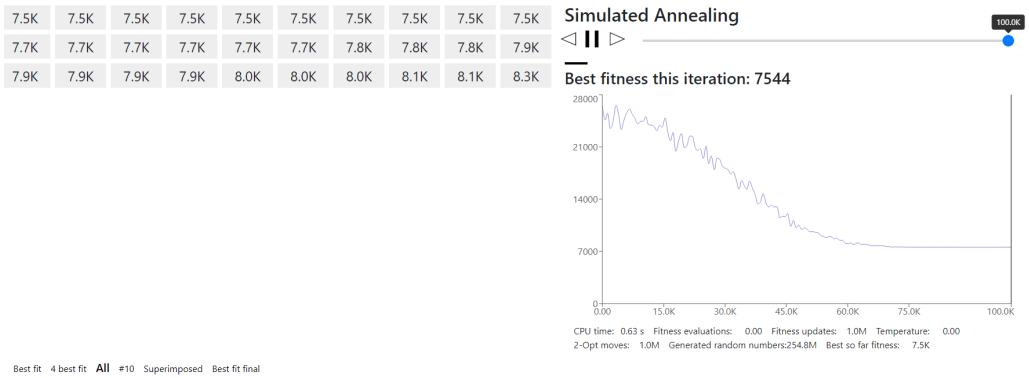


Figure 35: After running for 0.63 seconds, 10 out of the 30 Simulated Annealing instances found the optimal solution.

After manually also running 30 individual test cases of the evolutionary algorithm, we can compare the differences.

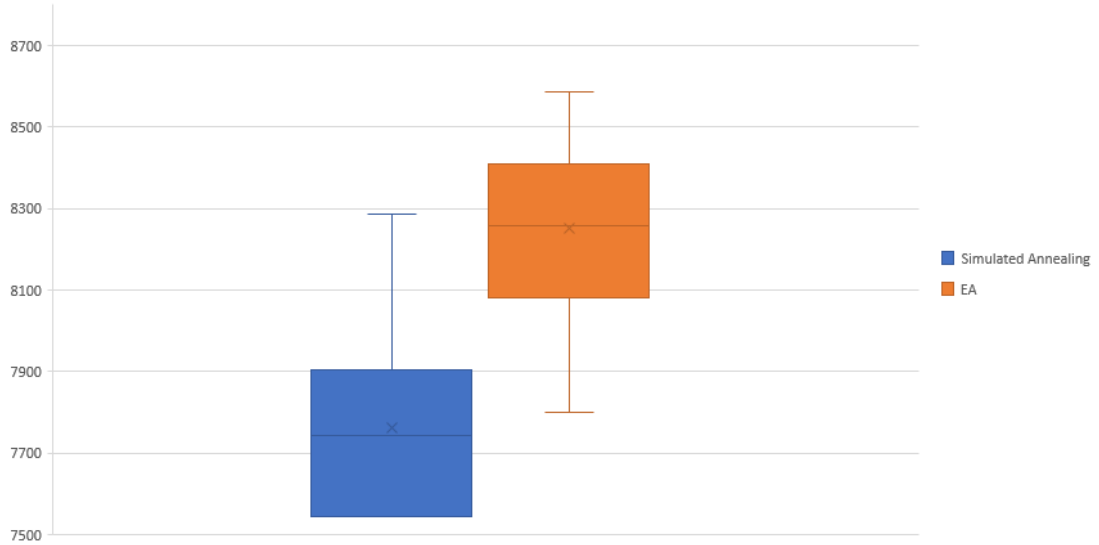


Figure 36: Box-plot of fitness value distributions founds by 30 independent runs of Simulated Annealing and an Evolutionary Algorithm that is identical to Simulated Annealing without the cooling scheme. The Simulated Annealing cooling scheme clearly marked an improvement

Figure 36 is a box-plot of the fitness value distributions for the two configurations. In this case, we can conclude that Simulated Annealing clearly benefits from its cooling scheme. However, what is not apparent from this data, is the fact that the Evolutionary Algorithm usually attains its minimum and final fitness after around 7-10K iterations causing it to do nothing for about 90% of the running time. This raises the question of whether or not the comparison was even fair in the first place. Arguably yes, because we are testing the very property of Simulated Annealing that allows it to not get stuck early in a sub-optimal minimum. But if we have strict requirements in terms of running time it could potentially be more beneficial to attain the local minimum as quickly as possible. In this case, the Evolutionary Algorithm/Cold Simulated Annealing could be desired. While we do not expect Simulated Annealing to be able to reach a local minimum as quickly since it also sometimes accepts worse solutions, it could still be the case that it likely is in a deeper place in another valley around the same time as Cold Simulated Annealing would otherwise reach local minimum. We conduct a quick test of Simulated Annealing run for 10K iterations to see the effect of limiting the running time to around the time a local minimum is otherwise achievable.

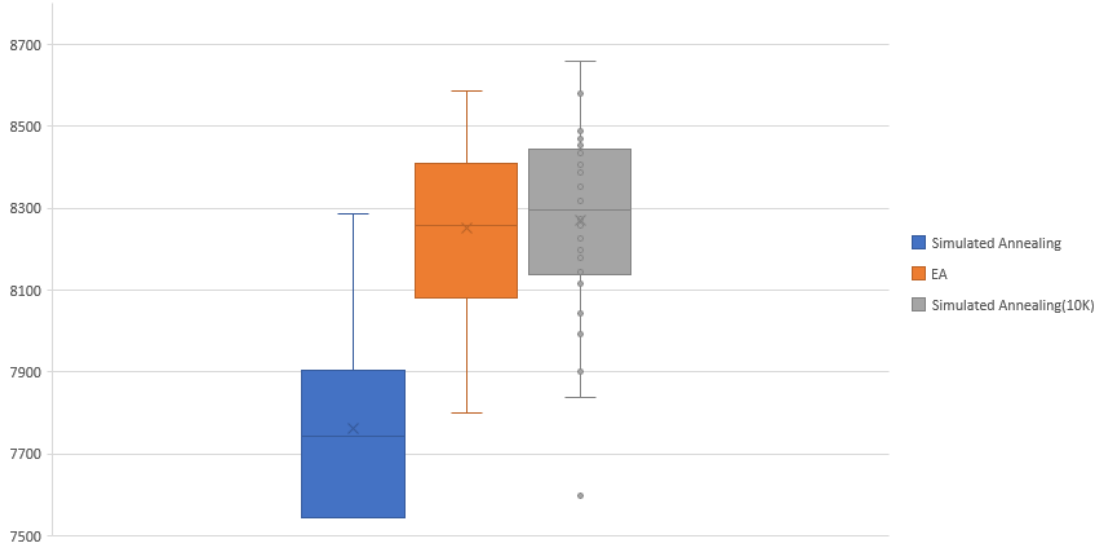


Figure 37: Box-plot similar to figure 36 with a now restricted version of Simulated Annealing to 10K iterations.

While, apart from one outlier, it did not perform better than the Evolutionary algorithm, it can still be considered a success because it is only slightly worse. The difference can be up to statistical inaccuracy but regardless, it is otherwise insignificant enough such that we can conclude that the cooling scheme of Simulated Annealing can be powerful enough to warrant potentially not needing a separate implementation when under strict running time requirements. The raw fitness values used to plot figure 37 is listed in appendix.

6.4 2-OPT vs 3-OPT based Simulated Annealing for TSP

We test the difference between basing Simulated Annealing on the 2-OPT and 3-OPT local search heuristics. Intuitively, the 3-OPT version could be better because it does more work per iteration. In fact, this extra work comes relatively cheap. While it checks 7 combinations of stitching 3-OPT combination chains back together and the 2-OPT only checks one, checking these cases is a constant operation and accepting one of the combinations uses 1-3 O(n) reverse operations. Consequently, the factor 7 of extra work is diminished by the acceptance of different neighbours. Quick tests of our implementation reveal the 3-OPT version is about 2x-3x slower than 2-OPT per iterations which is not surprising.
While the 3-OPT move may do more work, unlike the 2-OPT move, it can be unable to explore the entire search space. This is easily demonstrated by imagining a case, where the longest TSP tour is unique. The 3-OPT move will never accept this tour, even in warmest phase of Simulated Annealing, because it will be filtered out among the 6 discarded combinations before being subject to the acceptance probability. While the ability to accept the longest tour is not something we are concerned about, it could be the case that even the warm phase is not exploring tours that are necessary to get to the deepest valley in the search space.

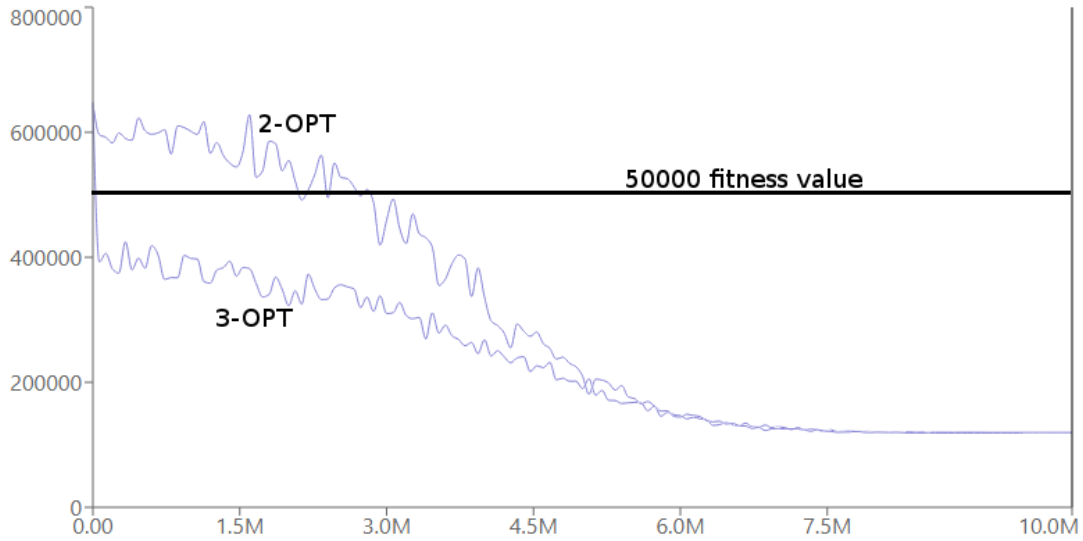


Figure 38: Running 2-OPT and 3-OPT based Simulated Annealing on bier127 for 10M iterations. Fitness values overlayed.

We run the two configurations of Simulated Annealing on bier127 for 10M iterations and overlay their fitness values. We can clearly see how 3-OPT immediately accepts better neighbouring solutions than 2-OPT, even in the very warm phase where candidate solutions are nearly always accepted. We can understand the concern of this by imagining two local optima in the search space, one of which is also the global optimum. If all possible combinations of moves to get from the worse local optimum to the global one involves selecting 3-OPT candidates that causes the fitness value to go above 50000, we can see from figure 38 how 3-OPT would clearly struggle to overcome this barrier and find the deepest valley where the global minimum lies. However, it may also be the case that this bar is too high to impose a barrier between any two local minimum in the search space such that it is not a problem for 3-OPT. While analysing the position of this barrier is out of the scope of this project, it tells us that even though a 3-OPT move may do more work per iteration, we do not know for sure it outperforms 2-OPT even when run for the same number of iterations. We therefore test 2-OPT vs 3-OPT both in terms of equal number of iterations and equal allocated CPU-time.
We conduct these test on three different TSPLib instances and for different amount of relevant iterations. Table 5 reports statistical results of these tests. Fitness values including the R-script used to process the data is included in appendix.

Welch's t-test, 50 samples. Means and p-value	3-OPT	2-OPT	p-value
berlin52. 1M Iterations. (3-OPT took 21s)	7552	7566	0.28
berlin52. 21s. (2-OPT made 2.6M iterations)	7552	7551	0.91
bier127. 1M Iterations. (3-OPT took 24 s)	120005	120680	0.00053
bier127. 24s (2-OPT made 2.6M iterations)	120005	120192	0.28
lin318. 2M Iterations. (3-OPT took 59s)	43283	44380	2.2^{-16}
lin318. 59s. (2-OPT made 5M iterations)	43283	43374	0.17

Table 5: Statistical analysis of 2-OPT vs 3-OPT test instances.

We used the parallel instances to setup each run of Simulated Annealing with 50 independent instances of the given test cases. That it took 21 seconds to run 3-OPT for 1 million iterations means all 50 runs cumulatively spend 21 seconds. Running a single case would therefore take less than a second. The number of iterations is selected such that we see a variety of effects. For the berlin52 test case, 1 million iterations was enough to find the global optimum in more than 90% of the cases for both 2-OPT and 3-OPT. In this case it is interesting to analyse the risk of not finding the optimal solution when given plenty of CPU-time. We also ran 2-OPT for 21 seconds, because that is the time it took to run 3-OPT for 1 million iterations. In both cases we do not see a p-value small enough to allow us to conclude any difference between 2-OPT and 3-OPT. When looking at the other test cases, we realise this is most likely due to the fact that too many of the cases found the optimal solution, thereby reducing the potential for variability. 1 million iterations was on the other hand too few for the TSPLib instance lin318, as there would usually be crossing edges in the best found solution, so we increase this test instance to 2 million which confidently finds local minima. As such, this case is our other extreme where we test the difference between 2-OPT and 3-OPT when we barely are able to find a local minimum. Finally 1 million iterations for bier127 is a good middle ground, where we find good solutions but still a different between most cases, meaning the optimal solution is hard to find. We see a statistical significance between 2 and 3-OPT run for the same amount of iterations for both of these TSPLib instances. 3-OPT performs significantly better in both cases, and as such it is evidence against the barrier-concern outlined through figure 38. However, since we cannot test all TSP instances, we do not rule out this concern generally.
Testing the 2-OPT vs 3-OPT based Simulated Annealing when allocating the same amount of CPU-time interestingly did not prove any statistical significance even though 3-OPT performed slightly better. We would most likely be able to see a statistical significance in either direction by expanding our sample size or run different TSPLib instances as well. However, while that would allow us to say there is a difference, it does not take away from the fact that the difference is small which is the main point of these results. We note again, that our selected TSPLib instances may not be representative of all possible TSP instances, and as such there may still be cases where one of the two is clearly preferred. Perhaps we could benefit from incorporating both into Simulated Annealing, and selecting at random whether or not a given iteration performs a 2-OPT or 3-OPT move, but that is out of the scope of this project.

6.5 The Evolutionary Algorithm and its applications

While evaluating the ability of Simulated Annealing to solve TSP we already tested it against an Evolutionary Algorithm, an RLS implementation on permutations which is equivalent to Cold Simulated Annealing. Not surprisingly it was outperformed by Simulated Annealing because it got stuck in a local minimum too quickly preventing it from exploring other valleys as well. Unfortunately we expect this to also be the case with the (1+1) EA permutation version where we draw the number of mutations to do from the Poisson distributions. While this could cause it to mutate enough to get out of a sup-optimal local minimum in one iteration, it is still unlikely to do more than a few mutations which is often times necessary. That is unless provided with liberal amounts of CPU-time such that we can more likely get the exact mutations required to get out of a local minimum. Since the TSP search space contains many valleys it is therefore not surprising that our implementation is more appropriate on unimodal functions like the bit-string OneMax problem.
Another property of the Evolutionary Algorithm is the ability to quickly adapt to changing environments. We can imagine that a solution to TSP is already found and suddenly 10 extra vertices is introduced in the graph. The Evolutionary Algorithm would then quickly attain a

new local minimum in this new graph while a metaheuristic like Ant Colony Optimization would already have a dominant trail of pheromones in place which it would likely not deviate from. In this example the Evolutionary Algorithm may be desirable, however we do not implement such dynamically changing problems that allows it to shine.
We try to mitigate the problem of the Evolutionary Algorithm finding sub-optimal local minimum by introducing a larger population size. We tune it with the μ parameter as well as also drawing the numbers of mutations we do from the Poisson distribution. To this end we set μ to the largest allowed value, 100, and run berlin52 for 1 million iterations too make sure it is enough for all individuals too attain a local minimum.

Fitness values.
7797,8347,7544,7955,7747,7544,7876,7757,7837,7598,

Table 6: Best fitness value of 10 quick runs of (100 + 1) EA on berlin52. for 1M iterations.

With a mean of 7800 and the global optimum found twice, this test was definitely an improvement on the local minima found by the above-mentioned Evolutionary Algorithm which we compared to Simulated Annealing through figure 36 and 37. However, Simulated Annealing is still attains better solutions in much less time. A main issue with increasing the population size for solving TSP seems to be the fact that the entire population gets replaced by the best solution too quickly. In our case, after 100 iterations of no improvement which happens frequently during the later stages of the run, the entire population is replaced. The potential for separate individuals to explore the search space and finding different valleys is therefore reduced. There are methods for tackling this problem but those are out of the scope of this project.

6.6 Solving TSP with Ant Colony Optimization

We evaluate the ability of Ant Colony Optimization to solve TSP by studying a run on the TSPLib instance berlin52.

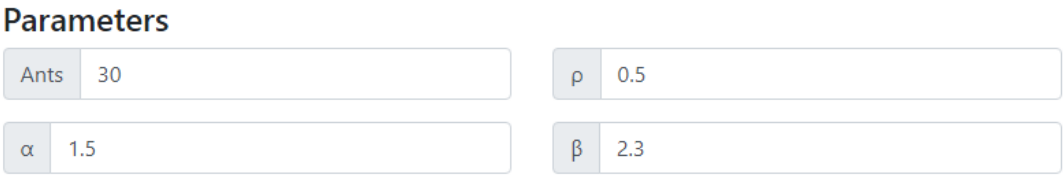
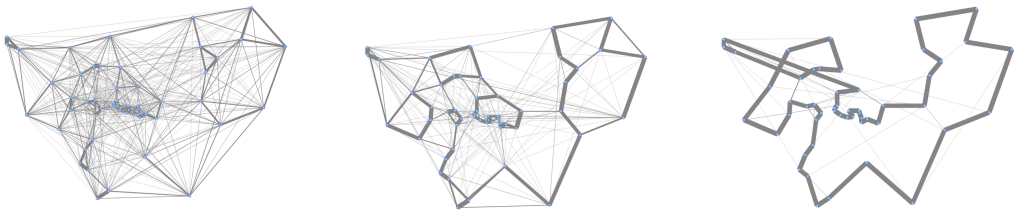


Figure 39: Parameters found to be fitting for demonstrating ACO on berlin52.

Evidently, ACO proves to be very sensitive to the choice of parameters. We find the parameters in figure 42 to provide reasonably good results which is used in this run we study. We will later reason about how to change some of the parameters and tune them even more.
The superimposed visualization is helpful to understand the effects of pheromones. It overlays all the routes made by the ants in a given iteration.



(a) Iteration 1 (b) Iteration 10 (c) Iteration 100

Figure 40: Effect of pheromone progression in ACO on TSPLib instance berlin52. Tours of 30 ants overlayed.

As seen in figure 40(a), the ants initially wander uninfluenced by pheromones, meaning they end up selecting wildly different tours. However, it is not completely random, we can still make out the effect of the bias towards selecting shorter edges. Already in iteration 10 it is apparent that some edges are reinforced with pheromones as more ants use them in their route. It is promising that many of these edges are shared with the optimal tour which is actually the tour seen in figure 1(b). Finally, in iteration 100, almost all ants walk the same tour dominated by pheromones. In this iteration, almost all pheromones have evaporated of off the other edges. From figure 40(c) it is strikingly apparent that the dominant pheromone trail is not optimal because of the way the edges cross near the top-left of the graph. Mishaps like this seems to be hard to avoid with ACO, in fact local search is commonly employed to fix such crossing tours with 2 and 3-opt moves [25], which is the reason why we also implemented such a variant of ACO.
When looking at the fitness value chart for the run we see another interesting effect.

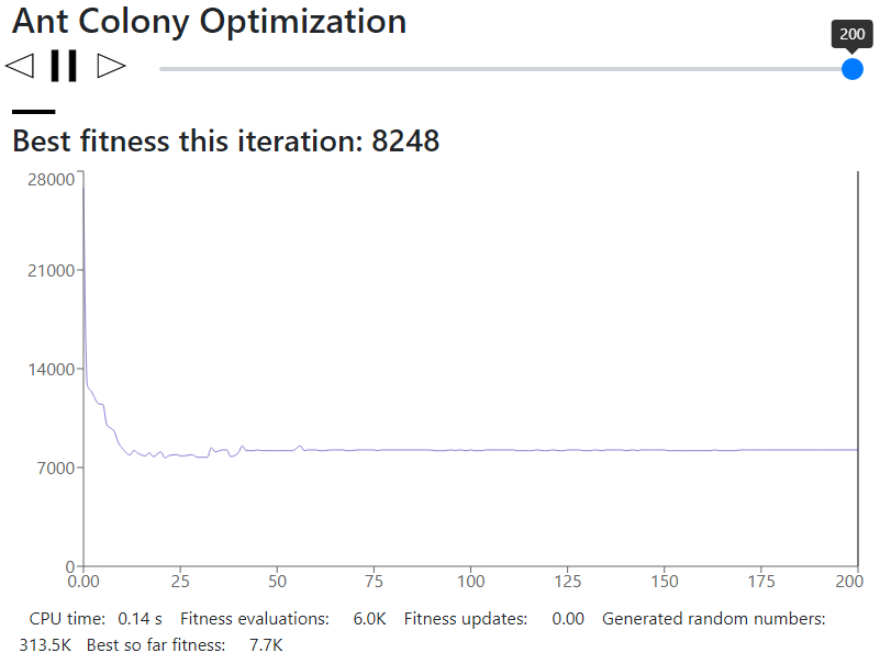


Figure 41: Fitness value of best ant per iteration

The best fitness value stays relatively constant starting at around iteration 50. This is due to the effect of the dominant pheromone trail which ants are unlikely to deviate from. However, just before the trail became too dominant the best fitness value dips below that of the dominant trail. We can see this happen at around iteration 25.

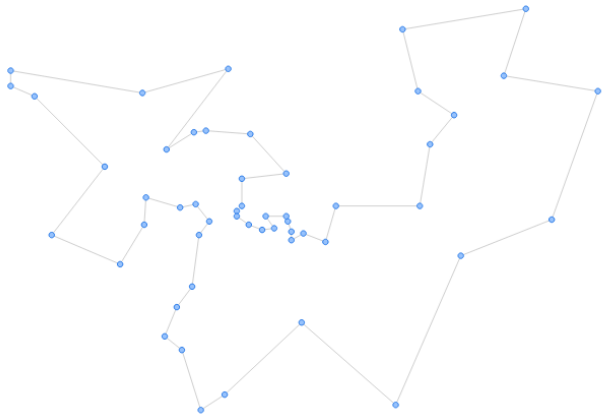


Figure 42: The best found solution has a 7.7K fitness value and is almost optimal.

It is not hard to understand why this happens. When the pheromone trail is not overly dominant, ants may deviate more from it and potentially find better tours. But if too few ants find better tours, and if these tours are not much better, then the new pheromones on such tours may not be enough to cause a shift of the dominant trail.

6.6.1 Tuning the parameters

We try a configuration of parameters that allows the ants to deviate more from the dominant trail in order to achieve a higher chance of finding a tour that is similar but better. In order to do so we increase the number of ants to 100, thereby increasing the chance some of the ants deviate from the dominant trail. We also decrease the pheromone influence α to 1 such that the ants becomes less sensitive to the dominant pheromones.

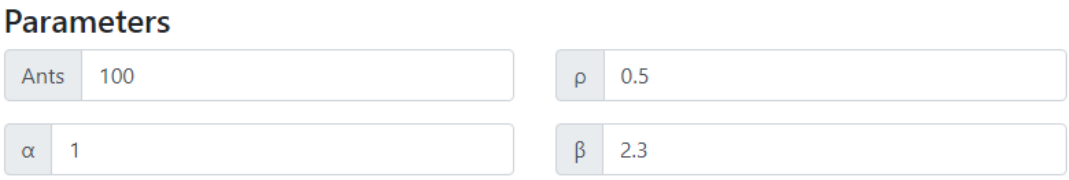


Figure 43: Adjusted parameters

When running this configuration we achieve better results on the berlin52 instance.

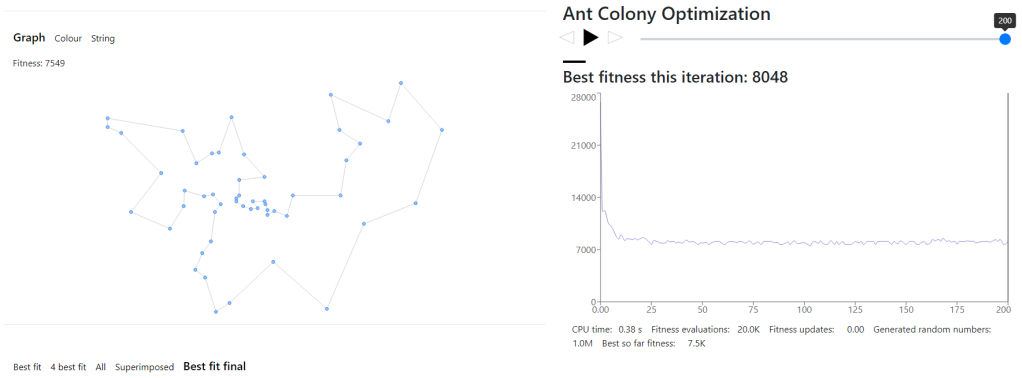


Figure 44: Running ACO on berlin52 with parameters as in figure 43.

In figure 44 we can see how the best fitness value no longer completely flat-lines after 50 iterations. The best achieved fitness value is now 7549 which is within 0.1% of the optimal solution. While this seems to be better than the other configuration, Simulated Annealing seems to reliably find better results. If we run Simulated Annealing for the same amount of time (0.38s) we will often find the optimal solution, and it is not as sensitive to parameters. On the other hand, figure 44 also reveals that we already have good solutions with ACO after about 20 iterations which is quick enough to compete with Simulated Annealing in this time-frame. It corresponds to about 30ms. Roughly speaking, the equivalent 2-OPT based Simulated Annealing job would be about 5000 iterations which produce fitness values in the range of 8000-10000 for berlin52. As seen in table 8, this ACO implementation often provides sub-8000 values within the first 20 iterations so it is able to beat that.

ACO 20 Iterations.	Sim. Annealing. 5K Iterations.
7725, 7887, 7916, 8070	8411, 8902, 9002, 9147, 9312

Table 7: Best fitness value of 5 quick runs to compare ACO with Simulated Annealing on berlin52 under very strict running time requirements.

We can improve the performance of ACO even more by also employing local search which insignificantly affects the running time. Table 8 similarly lists fitness values for 5 quick runs of the same ACO configuration, except local search is also employed at the end on the best solution. Three of these is the optimal solution (7544) meaning this configuration of ACO is the fastest we have been able to somewhat reliably solve berlin52 to optimality.

20 Iteration ACO with local search at the end.
7544,7722,7630,7544,7544

Table 8: Best fitness value of 5 quick runs of ACO with local search on berlin52 under very strict running time requirements.

Consequently, it seems to be the case that both of these metaheuristics are effective at solving TSP under different circumstances but if we were to select one of these implementations as the default TSP solver, we would still choose Simulated Annealing as it is less sensitive to parameters.

7 Future work

It is our belief that the framework currently provides an expandable baseline for evaluation and visualisation of three nature-inspired metaheuristics Simulated Annealing, Ant Colony Optimization and an Evolutionary Algorithm. Future work might therefore include implementing and evaluating new metaheuristics or variants of the three that are already there, potentially with a larger focus on research. The following list compiles some of these ideas as provided throughout the report.

- A 3-OPT based Simulated Annealing variant capable of transitioning to each of the 7 other 3-OPT move combinations.
- A hybrid method based on ACO and Simulated Annealing where Simulated Annealing is used in place of local search as otherwise already employed alongside ACO.
- Simulated Annealing variant where 2-OPT and 3-OPT moves are selected at random.

One of the disadvantages of the Evolutionary Algorithm we implemented is the tendency to get stuck early in a local minimum. This is even the case when we increase the population size as the population quickly becomes monotone through replacement by the strongest individual. Future work might tackle methods that prevents this from happening. For instance, it could more closely imitate evolution through the introduction of species that do not compete as much thereby potentially exploring separate areas of the search space.
Future work can also focus on the practical aspects of the framework and implement a route planner based on physical addresses. The Google Maps API implement a distance matrix function call which allows one to obtain road distances and expected travelling times between an array of different cities [6]. It would be ideal to incorporate this with our project in order to find the shortest route in a real world scenario. We de-prioritized this feature during development partly because the distance matrix API call requires a costly API-license and we would then also have to charge the users of our framework. However, this is also a direction the project could take. Future work could enhance the use-ability of the API with extended features like the route planner and the possibility to turn off the visualization data. We could then charge a fee per API call as well.

8 Conclusion

A framework for visualizing and evaluating the working principles of the nature-inspired optimization metaheuristics, Simulated Annealing, Ant Colony Optimization and an Evolutionary Algo-rithm was developed. The framework was developed on the web stack in a modular manner such that it can be extended with other problem classes, metaheuristics and visualizations. Through the framework it was possible to import instances of The Travelling Salesman Problem and obtain solutions for these. It was shown that particularly Simulated Annealing and Ant Colony Optimization can be effective solvers, often times achieving the optimal solution in a split-second for a selected TSP instance of 52 vertices. It was further made possible to run the metaheuristics on benchmark bit-string problems as well as extracting key-figures from a run such that the algorithms can be properly analyzed. We successfully replicated analytical results through runs of the Evolutionary Algorithms on the bit-string problems OneMax and LeadingOnes.

References

[1] Composition vs inheritance. `reactjs.org/docs/composition-vs-inheritance.html`. Accessed: 2020-05-24.

[2] Load balancing with nginx. In *Nginx*. Apress.

[3] Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. In *Acm Transactions on Modeling and Computer Simulation (tomacs)*. ACM.

[4] Nginx modules. In *Nginx*. Apress.

[5] React.component. `reactjs.org/docs/react-component.html`. Accessed: 2020-05-24.

[6] Web services, distance matrix api. 2020. `https://developers.google.com/maps/documentation/distance-matrix/start`. Accessed: 2020-05-31.

[7] Heragu Sundresh S. Alfa, Attahiru Sule. and Mingyuan Chen. A 3-opt based simulated annealing algorithm for vehicle routing problems. In *Computers and Industrial Engineering*, pages 635–639. 1991.

[8] Samuel J. Balin and Marilia. Cascalho. The rate of mutation of a single gene. 2010.

[9] Eli Bendersky. Memory layout of multi-dimensional arrays. 2015. `eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays`. Accessed: 2020-05-24.

[10] P. Bettinger, K. Boston, and J. Sessions. Simulated annealing. In Sven Erik Jørgensen and Brian D. Fath, editors, *Encyclopedia of Ecology*, pages 3255 – 3261. Academic Press, Oxford, 2008.

[11] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. volume 35, pages 268–308. 01 2001.

[12] Ivan Brezina Jr and Zuzana Čičková. Solving the travelling salesman problem using the ant colony optimization. volume 6, pages 10–14. 2011.

[13] Mark R. Brown. Fastcgi specification. 1996. `http://www.mit.edu/~yandros/doc/specs/fcgi-spec.html`. Accessed: 2020-05-24.

[14] Mark Clow. More components. In *Angular 5 Projects*, chapter 12, pages 159–210. 2018.

[15] Benjamin. Doerr and Frank Neumann. The benefits of population diversity in evolutionary algorithms: A survey of rigorous runtime analyses. In *Theory of Evolutionary Computation*, chapter 8, pages 359–404. Springer International Publishing, 2019.

[16] Carola Doerr. Complexity theory for discrete black-box optimization heuristics. In *Theory of Evolutionary Computation*, chapter 3, pages 133–206. Springer International Publishing, 2019.

[17] Swamy M. N. S. Du, Ke-Lin. Simulated annealing. In *Search and Optimization by Metaheuristics*, pages 29–36. Springer International Publishing, 2016.

[18] T. C. Hu and Andrew B. Kahng. Introduction to the simplex method. In *Linear and Integer Programming Made Easy*, chapter 4, pages 39–60. Springer International Publishing, 2016.

[19] Thomas Jansen. Analysing stochastic search heuristics operating on a fixed budget. In *Theory of Evolutionary Computation*, chapter 5, pages 249–271. Springer International Publishing, 2019.

[20] Dieter Jungnickel. A hard problem: The tsp. In *Graphs, Networks and Algorithms*, pages 481–526. Springer Berlin Heidelberg, 2012.

[21] Patel Minesh. Hassan Hasan. Orosa Lois. Kim, Jeremie S. and Onur Mutlu. D-range: Using commodity dram devices to generate true random numbers with low latency and high throughput. IEEE Computer Society, 2019.

[22] Ratnesh Kumar and Haomin Li. On asymmetric tsp: Transformation to symmetric tsp and performance bound. 2015.

[23] Johannes Lengler. Drift analysis. In *Theory of Evolutionary Computation*, chapter 2, pages 89–131. Springer International Publishing, 2019.

[24] David Levinthal. Performance analysis guide for intel® core™ i7 processor and intel® xeon™ 5500 processors.

[25] Baykan Ömer Kaan. Mahi, Mostafa. and Halife Kodaz. A new hybrid method based on particle swarm optimization, ant colony optimization and 3-opt algorithms for traveling salesman problem. Elsevier Ltd, 2015.

[26] John McCall. Genetic algorithms for modelling and optimisation. In *Journal of Computational and Applied Mathematics — 2005, Volume 184, Issue 1*, pages 205–222. ELSEVIER SCIENCE BV, 2005.

[27] Nicolas Monmarché. Artificial ants. In *Metaheuristics*, pages 133–212. Springer International Publishing, 2016.

[28] Thomas. Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. 2007.

[29] Frank. Neumann and Andrew. Sutton. Parameterized complexity analysis of randomized search heuristics. In *Theory of Evolutionary Computation*, chapter 4, pages 213–249. Springer International Publishing, 2019.

[30] Pourhassan Mojgan. Neumann, Frank. and Vahid Roostapour. Analysis of evolutionary algorithms in dynamic and stochastic environments. In *Theory of Evolutionary Computation*, pages 323–357. Springer International Publishing, 2019.

[31] Sudholt Dirk. Neumann, Frank. and Carsten Witt. Analysis of different mmas aco algorithms on unimodal functions and plateaus. Springer US, 2009.

[32] Yindee. Oonsrikaw and Arit Thammano. Enhanced ant colony optimization with local search. Institute of Electrical and Electronics Engineers Inc., 2018.

[33] Lapalme Guy Potvin, Jean-Yves. and Jean-Marc Rousseau. A generalized k-opt exchange procedure for the mtsp. In *Infor: Information Systems and Operational Research*, pages 474–481. Oxford, 1989.

[34] James Sanders. Why it's finally time for developers to address the chaos of node.js and npm. 2018. `https://www.techrepublic.com/article/why-its-finally-time-for-developers-to-address-the-chaos-of-node-js-and-npm/`. Accessed: 2020-05-29.

[35] Bart Selman. Greedy local search. 2009.

[36] Thomas. Stützle and Rubén. Ruiz. Iterated local search. In Pardalos Panos M Martí, Rafael and Mauricio G. C Resende, editors, *Handbook of Heuristics*, chapter 19, pages 579–605. Springer International Publishing, 2018.

[37] Chi Wan1 Sung and Shiu Yin Yuen. Analysis of (1+1) evolutionary algorithm and randomized local search with memory. In *Evolutionary Computation*, pages 287–323. MIT PRESS, 2011.

[38] Sevaux Marc. Sörensen, Kenneth. and Fred Glover. A history of metaheuristics. In *Handbook of Heuristics*. Springer International Publishing, 2018.

[39] Christopher Wu. Writing hello world in fcgi with c++. 2012. `chriswu.me/blog/writing-hello-world-in-fcgi-with-c-plus-plus/`. Accessed: 2020-05-24.

[40] Xin-She Yang. Nature-inspired metaheuristic algorithms. 07 2010.

9 Appendix

9.1 Simulated Annealing cooling scheme test

Below is the raw fitness values for 30 tests of 2-OPT based Simulated Annealing vs Evolutionary Algorithm with $\mu = 1$, $\lambda = 1$, reverse mutation, mutations applied to one random pair, on the berlin52 TSPLib instance and run for 100K iterations. This data is the basis of figure 36

Simulated Annealing(100K)	Simulated Annealing(10K)	EA(100K)
7544	7598	7801
7544	7837	7956
7544	7901	7989
7544	7992	8005
7544	8004	8029
7544	8044	8061
7544	8114	8070
7544	8145	8084
7544	8157	8119
7544	8177	8176
7682	8198	8184
7696	8225	8211
7702	8225	8211
7736	8256	8220
7737	8273	8257
7747	8316	8258
7764	8351	8299
7773	8360	8326
7814	8387	8337
7854	8407	8361
7861	8416	8377
7883	8434	8405
7903	8440	8409
7904	8452	8415
7955	8468	8454
7966	8488	8460
8017	8581	8470
8060	8589	8472
8073	8590	8488
8287	8657	8587

9.2 Welch's t-test of 2-OPT vs 3-OPT based Simulated Annealing

The following script is used to generate the results of table 5

```
1 TWO_OPT_berlin52_1M <- as.numeric(unlist(strsplit("7544,
     7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
     7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
     7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
     7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
     7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7598,
     7682, 7736, 7797, 8006", ", ")))
2 THREE_OPT_berlin52_1M <- as.numeric(unlist(strsplit("
     7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
     7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
```

```
   7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
   7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
   7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
   7544, 7544, 7544, 7717, 7778", ", ")))
TWO_OPT_berlin52_21s <- as.numeric(unlist(strsplit("7544,
    7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
    7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
    7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
    7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
    7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544, 7544,
    7544, 7598, 7683, 7717", ", ")))

TWO_OPT_bier127_1M <- as.numeric(unlist(strsplit("118486,
    118814, 118877, 119117, 119274, 119322, 119346,
   119439, 119582, 119650, 119736, 119748, 119875,
   119905, 119989, 119991, 120103, 120111, 120320,
   120491, 120517, 120562, 120596, 120637, 120674,
   120809, 120829, 120907, 120947, 121082, 121116,
   121136, 121139, 121148, 121196, 121241, 121245,
   121291, 121325, 121602, 121679, 121689, 121753,
   121805, 121816, 122192, 122208, 122783, 122785, 123126
   ", ", ")))
THREE_OPT_bier127_1M <- as.numeric(unlist(strsplit("
   118482, 118622, 118679, 118799, 118838, 118861,
   118895, 119000, 119104, 119141, 119190, 119470,
   119497, 119521, 119702, 119734, 119735, 119782,
   119790, 119857, 119907, 119947, 119963, 119973,
   119990, 120007, 120055, 120117, 120159, 120217,
   120228, 120280, 120309, 120358, 120469, 120574,
   120579, 120608, 120701, 120761, 120780, 120833,
   120872, 120877, 120905, 120914, 121169, 121197,
   121342, 121440, ", ", ")))
TWO_OPT_bier127_24s <- as.numeric(unlist(strsplit("
   118357, 118645, 118736, 118738, 118840, 119008,
   119065, 119118, 119134, 119263, 119325, 119444,
   119535, 119537, 119542, 119558, 119731, 119780,
   119894, 119908, 119926, 119976, 119980, 120030,
   120242, 120292, 120316, 120317, 120341, 120465,
   120483, 120569, 120631, 120693, 120710, 120747,
   120787, 120911, 120930, 120938, 121063, 121213,
   121280, 121441, 121455, 121569, 121587, 121722,
   121723, 122080", ", ")))

TWO_OPT_lin318_2M <- as.numeric(unlist(strsplit("43601,
   43680, 43694, 43707, 43843, 43873, 43969, 43990,
   43992, 43995, 44010, 44043, 44051, 44060, 44066,
   44132, 44164, 44179, 44195, 44263, 44291, 44315,
   44353, 44358, 44363, 44367, 44402, 44424, 44424,
   44469, 44490, 44517, 44533, 44585, 44633, 44659,
```

```
   44662, 44675, 44693, 44709, 44721, 44826, 44838,
   44845, 44893, 44919, 44944, 45033, 45121, 45453", ", "
   )))
THREE_OPT_lin318_2M <- as.numeric(unlist(strsplit("42724,
    42816, 42866, 42886, 42915, 42953, 42979, 43010,
   43011, 43016, 43030, 43035, 43053, 43063, 43139,
   43142, 43156, 43165, 43166, 43175, 43185, 43192,
   43230, 43235, 43242, 43292, 43297, 43298, 43311,
   43312, 43333, 43341, 43347, 43386, 43408, 43426,
   43481, 43485, 43511, 43516, 43543, 43588, 43591,
   43617, 43638, 43664, 43665, 43687, 43957, 44061, ", ",
    ")))
TWO_OPT_lin318_59s <- as.numeric(unlist(strsplit("42425,
   42797, 42944, 42957, 43012, 43016, 43065, 43084,
   43089, 43124, 43149, 43150, 43182, 43190, 43195,
   43198, 43207, 43211, 43230, 43246, 43266, 43277,
   43282, 43294, 43296, 43317, 43323, 43325, 43344,
   43353, 43356, 43360, 43395, 43482, 43491, 43505,
   43506, 43536, 43570, 43583, 43608, 43663, 43722,
   43846, 43908, 43966, 43998, 44037, 44056, 44576, ", ",
    ")))

t.test(THREE_OPT_berlin52_1M, TWO_OPT_berlin52_1M)
t.test(THREE_OPT_berlin52_1M, TWO_OPT_berlin52_21s)

t.test(THREE_OPT_bier127_1M, TWO_OPT_bier127_1M)
t.test(THREE_OPT_bier127_1M, TWO_OPT_bier127_24s)

t.test(THREE_OPT_lin318_2M, TWO_OPT_lin318_2M)
t.test(THREE_OPT_lin318_2M, TWO_OPT_lin318_59s)
```