FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

David Ruda

Traffic Signal Optimization

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: RNDr. Jiří Fink, Ph.D.
Study programme: Computer Science

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....
Author's signature

I would like to thank my supervisor, RNDr. Jiří Fink, Ph.D., for his patience, guidance, and valuable feedback throughout the development of this thesis.
I would also like to thank my family and friends for their unwavering support and encouragement during this journey.

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

The clarity and readability of this thesis were enhanced using tools based on Large Language Models (LLMs).

Title: Traffic Signal Optimization

Author: David Ruda

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Fink, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: As cities grow and traffic congestion worsens, traffic signal optimization is becoming an increasingly important real-world problem for ensuring efficient urban mobility. This thesis explores a simplified version of this problem, originally presented in the Google Hash Code competition. The task involves optimizing the schedules of traffic lights at city intersections to maximize the number of cars reaching their destinations before a deadline, while minimizing the overall time spent in traffic. We develop a fast and efficient simulator to evaluate solutions for the task. We then integrate this simulator into an optimization pipeline with three heuristic algorithms: Genetic Algorithm, Hill Climbing, and Simulated Annealing. We experimentally compare these algorithms on the provided datasets, achieving new best scores on two datasets.

Keywords: traffic signal optimization, Google Hash Code, genetic algorithm, hill climbing, simulated annealing

Název práce: Optimalizace světelných křižovatek

Autor: David Ruda

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jiří Fink, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: S rostoucí velikostí měst a zhoršující se dopravní situací je optimalizace světelných křižovatek stále důležitějším problémem pro zajištění efektivní dopravy ve městech. Tato práce zkoumá zjednodušenou verzi tohoto reálného problému, která byla původně zadána v soutěži Google Hash Code. Cílem je optimalizovat nastavení semaforů na městských křižovatkách tak, aby co nejvíce aut dorazilo do cíle před stanoveným časovým limitem a zároveň se minimalizoval celkový čas strávený v dopravě. Vyvíjíme rychlý a efektivní simulátor, který umožňuje vyhodnocovat navržená řešení pro tento problém. S pomocí tohoto simulátoru následně optimalizujeme nastavení semaforů pomocí tří heuristických algoritmů: genetického algoritmu, horolezeckého algoritmu a simulovaného žíhání. Tyto algoritmy experimentálně porovnáváme na zadaných datových sadách a dosahujeme nových nejlepších výsledků na dvou z nich.

Klíčová slova: optimalizace světelných křižovatek, Google Hash Code, genetický algoritmus, horolezecký algoritmus, simulované žíhání

Contents

Introduction

Due to its significant impact on urban mobility and the environment, Traffic Signal Control (TSC) is a widely studied problem [1]. As the number of vehicles on the road continues to increase [2], its importance is growing even further. Traffic signal optimization is known to be a cost-effective method for reducing congestion without physically changing the road infrastructure [3]. Optimized traffic lights reduce delays at intersections [4], which directly translates into time savings for commuters and improves the overall efficiency of transportation networks. Additionally, TSC plays a critical role in reducing vehicle emissions [5], helping to reduce pollution and promote environmental sustainability.

There are many approaches to solving TSC [6]. Early static, fixed-time designs have evolved into real-time adaptive systems and data-driven algorithms, utilizing data from various detectors and sensors. In practice, commonly used are adaptive traffic control systems (ATCS) such as SCOOT, SCATS, and SURTRAC [7], which continuously adjust splits, offsets, and cycle lengths network-wide. In addition, optimization-based methods—such as Genetic Algorithm [8], Simulated Annealing [6], and Model Predictive Control [9]—have also been employed for traffic signal planning and coordination. More recently, learning-based methods, particularly those based on Reinforcement Learning and Deep Reinforcement Learning, are increasingly being explored as alternatives to traditionally used methods [10, 11].

In this thesis, we focus on the Traffic signaling problem from the Google Hash Code competition [12]. It serves as a simplified version of the real-world problem of traffic signal optimization in a city. First, we implement a fast and efficient C++ simulator for the problem and wrap it as a Python package to enable easy use and integration with the broader Python ecosystem, without compromising on performance. We then utilize the simulator as a black-box fitness function for three heuristic algorithms to optimize the traffic light schedules.

Hill Climbing (HC) is the simplest of the methods and it has been used by some participants both during and after the competition. Genetic Algorithm (GA) is a more complex method that, to the best of our knowledge, has not been applied to this particular competition problem before. As a third method, we choose Simulated Annealing (SA), a metaheuristic that can be viewed as a simple extension of HC. However, it appeared to be a suitable choice after initial tests suggested that GA's broader search capabilities may not be so beneficial for this problem. We then experimentally compare the performance of these algorithms on the provided competition datasets, which vary in size and structure.

The thesis is structured as follows: Chapter 1 presents the Traffic signaling problem in detail, along with preprocessing steps, datasets, and the simulator. Chapter 2 covers the theory of the optimization methods used in the thesis. Chapter 3 describes the application of the optimization methods to our specific problem, including initialization, algorithm operators, and hyperparameters. Chapter 4 presents the experimental results comparing the performance of the algorithms on the provided datasets. Appendix A provides a user guide detailing how to use the simulator, run optimization, and execute the scripts replicating our experiments. Appendix B contains developer documentation briefly describing implementation details of the simulator and optimization.

1 Problem description

This chapter introduces the optimization problem addressed in this thesis. Section 1.1 provides a brief overview of the competition setting where the problem was initially assigned. In Section 1.2, the original problem statement is presented and explained in detail. Section 1.3 explains preprocessing steps that we apply to simplify the problem for optimization. Section 1.4 describes the datasets provided with the problem. Finally, Section 1.5 presents the custom simulator tool that we developed to evaluate solutions to the problem.

1.1 Competition overview

The problem we address in this thesis was originally assigned in the qualifying round of Google Hash Code 2021. Google Hash Code was a global team programming competition, running between 2014–2022, where teams of 2–4 competitors solved an optimization problem with the goal of achieving the best score in a limited time of 4 hours. Despite its discontinuation, the competition remains a valuable resource for a wide range of optimization problems and has been the subject of various follow-up studies and articles [13, 14].

The usual solution procedure for our problem was as follows: Read the input data, create a trivial solution and write it to the output file in the specified format, and upload the solution to the evaluation system. The evaluation system not only displayed the score, but also some informative statistics, such as the number of cars that reached the finish before the deadline, the cars that arrived earliest and latest, the average cycle length of traffic lights at intersections, etc. For some datasets, an interactive visualization of the evaluation process was also available, allowing to see the structure of a particular dataset. Although a local simulator was not needed to solve the problem, the manual upload of the solution to the evaluation system was slow and cumbersome and therefore not suitable for the use of efficient optimization algorithms.

Most teams were able to construct trivial solutions, which they then tried to improve by randomly changing the values. However, the best teams were able to write their own local simulator and use it to run multiple heuristics to get a better total score. Still, there was no time for anything more complex than a simple random search.

The total score was the sum of the scores of all 6 datasets (A–F). The first dataset (A) served as a “toy problem” mainly for debugging purposes, but the rest of the datasets were large enough for optimization. It should be noted that the distribution of points among the datasets is uneven, so the contestants mostly focused on the datasets with the highest possible score (D, F) and pragmatically skipped optimizing the rest (B, C, E).

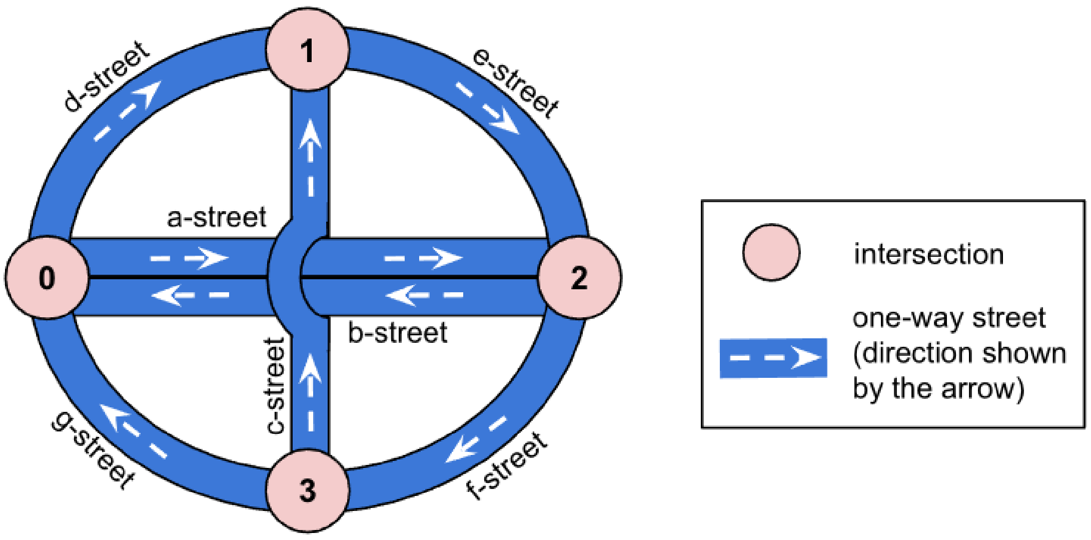

Figure 1.1 Example of a city plan [12].

1.2 Original problem statement

The full problem statement[1] is available in the Google Coding Competitions archive [12]. Here we describe only the parts that are important for understanding this work. In short, the task is as follows:

> *Given a city plan describing intersections and streets and cars with planned paths through the city, optimize the schedule of traffic lights to minimize the total amount of time spent in traffic, and help as many cars as possible reach their destination before a specified deadline.*

In terms of graph theory, the city plan is a *directed graph*. The intersections are *vertices* and the streets are *directed edges* (see Figure 1.1). The planned path for each car is indeed a *path* in this graph because it has a different start and end, and no intersection is repeated.

1.2.1 Streets and intersections

In the city, we have a set of intersections I, where $|I| \in [2, 10^5]$, and a set of streets $S \subseteq \{(u, v) | u, v \in I \wedge u \neq v\}$, where $|S| \in [2, 10^5]$. Each street $s \in S$ is a unique one-way connection between two different intersections u, v; two distinct streets (u, v) and (v, u) in opposite directions between the same two intersections are allowed. Each street $s \in S$ has a fixed time $l(s) \in \mathbb{N}_+$ that it takes a car to get from the beginning to the end of the street, independently of the other cars on the street. Each intersection $i \in I$ has a set of incoming streets $S_i^+ \subset S$, where $\left|S_i^+\right| \geq 1$, and a set of outgoing streets $S_i^- \subset S$, where $\left|S_i^-\right| \geq 1$; thus each intersection has at least one incoming street and at least one outgoing street.

[1] `https://github.com/google/coding-competitions-archive/blob/main/hashcode/hashcode_2021_qualification_round.pdf`

1.2.2 Traffic lights and schedules

In each intersection $i \in I$, there is a traffic light at the end of each *incoming* street $s^+ \in S_i^+$. The traffic light has two states—green and red. Green means the cars from this street can pass through the intersection and continue to any *outgoing* street $s^- \in S_i^-$ in their path. Red means the cars must stop until the light turns green again. At most one traffic light can be green at each intersection at any time.

When the light is red, cars arriving at the end of a street queue up and wait for the light to turn green. The queue does not take up any space and does not change the distance cars have to travel. When the light is green, one car can pass through an intersection every second. Passing through an intersection, i.e., moving from the end of an incoming street to the beginning of an outgoing street, takes no additional time.

For each intersection $i \in I$, we can set a traffic light schedule. This schedule determines the order and duration of green light for the incoming streets of the intersection. The schedule repeats in a cycle until the end of the simulation (see Figure 1.2). Each street can appear at most once in the schedule. If a street is not included in the schedule, it is red the whole time, and any waiting cars are blocked. By default, intersections have no schedule and all streets are red.

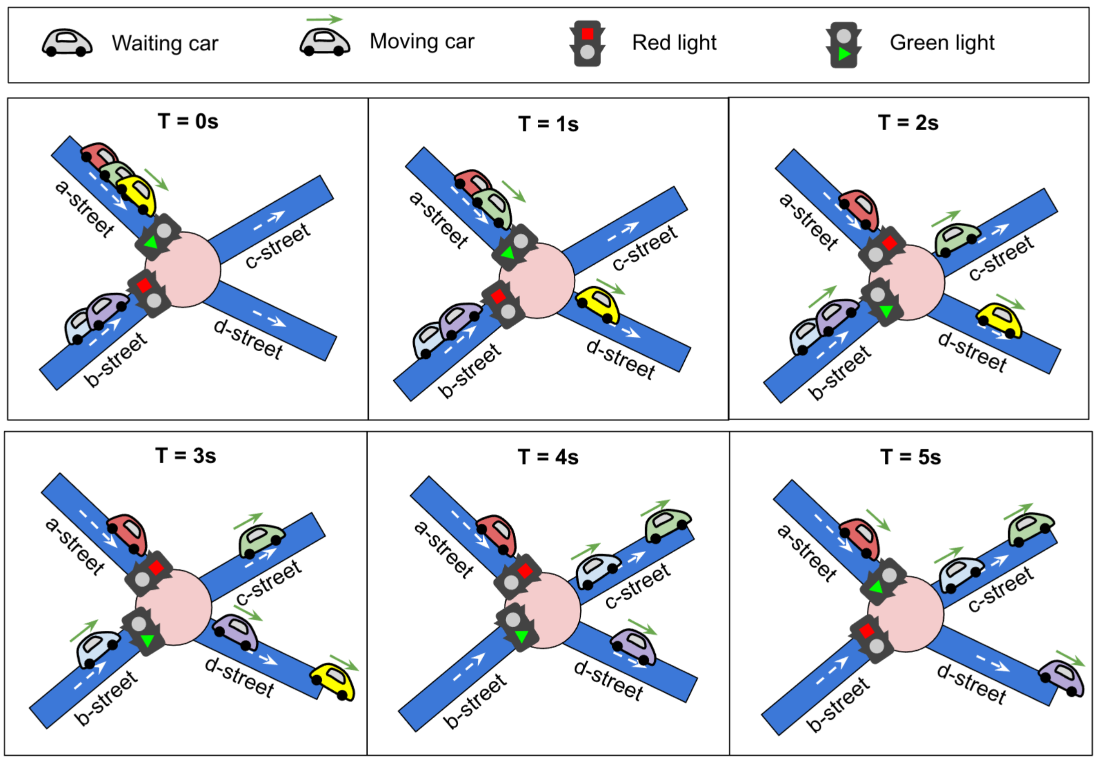

Figure 1.2 This figure shows how the traffic light schedule works for an intersection with two incoming streets [12]. The schedule is as follows: First *a-street* for 2 seconds, then *b-street* for 3 seconds. We can see the first two cars from *a-street* pass in the first two seconds, then the green light switches to *b-street* for three seconds, allowing the two cars from *b-street* pass. The last car from *a-street* waits till the beginning of the next cycle and then passes.

1.2.3 Cars

Furthermore, we have a set of cars C, where $|C| \in [1, 10^3]$. Each car $c \in C$ has a given path $\boldsymbol{p_c}$ through the city. The path is a sequence of streets the car has to drive through. The number of streets in each path $\boldsymbol{p_c}$ is in the range $[2, 10^3]$. No intersection or street can be repeated in a path.

At the beginning of the simulation, all cars are at the end of the first street in their path. They either wait if the light is red or are ready to move if the light is green. If more cars start at the end of the same street, they queue up according to their IDs in the input file (see Figure 1.3). When a car reaches the end of the last street in its path, it is immediately removed from the street.

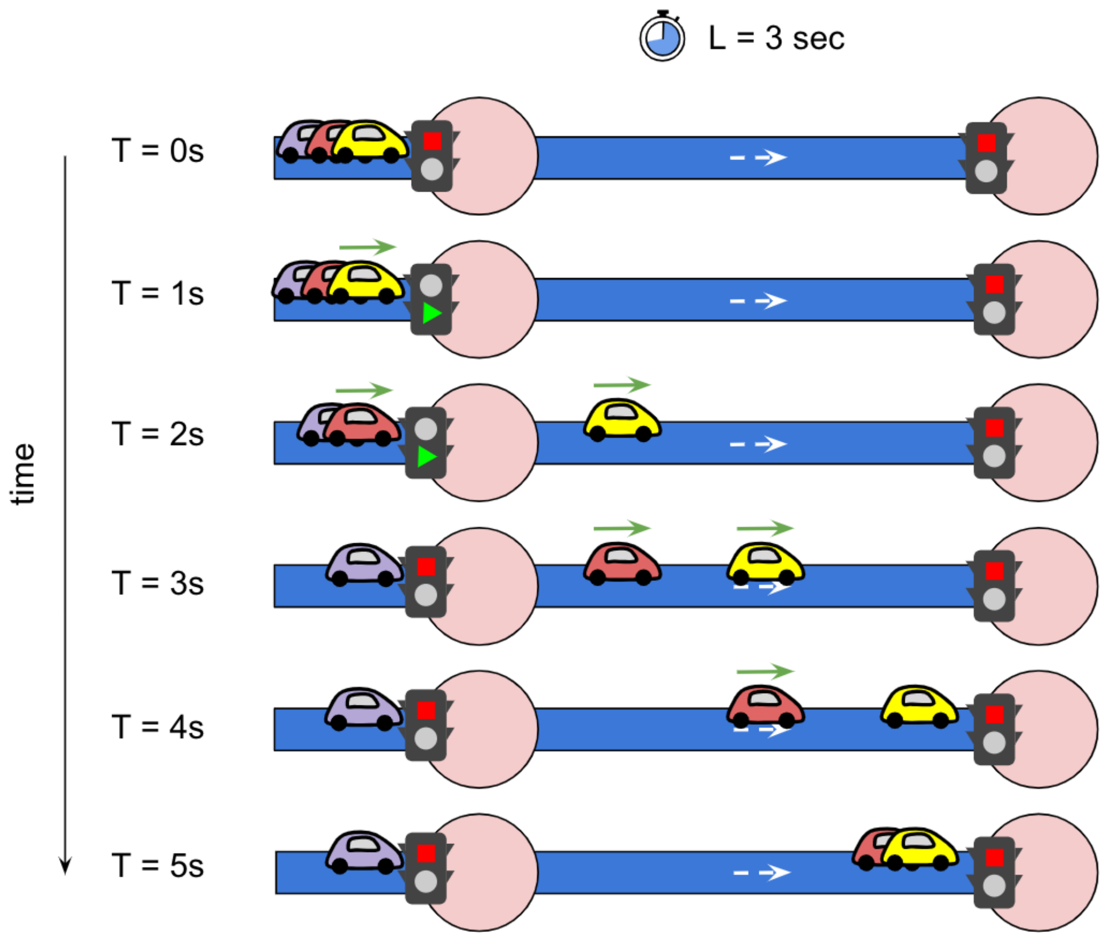

Figure 1.3 This figure shows the first five seconds of a simulation [12]. For simplicity, only one street is shown in the figure; when the light is red for this street, it is green for another street in the same intersection. When the light turns green at $T = 1s$, the first (yellow) car immediately passes the intersection and moves to the next street, reaching the end of the street at $T = 4s$. At $T = 2s$, the light is still green, so the second (red) car passes the intersection and moves to the next street, reaching the end of the street at $T = 5s$. From $T = 3s$ to $T = 5s$, the light is red, so the third (purple) car cannot pass the intersection and has to wait for the next cycle.

1.2.4 Score

The score reflects the quality of the traffic light schedules: the more cars reach their destination, and the sooner they arrive, the higher the score. The objective is to maximize this value. In the context of optimization, the score serves as a *fitness function*.

The score is determined as follows. Let $\boldsymbol{\theta}$ denote the traffic light schedules, and let C be the set of cars. Given a simulation duration $\mathrm{D} \in [1, 10^4]$ in seconds, and a fixed bonus awarded for reaching the destination $\mathrm{F} \in [1, 10^3]$, let $t(c; \boldsymbol{\theta}) \in \mathbb{N}$ be the time when a car $c \in C$ reaches its destination. The score of a single car c under the schedules $\boldsymbol{\theta}$ is defined as

$$score(c; \boldsymbol{\theta}) = \begin{cases} \mathrm{F} + (\mathrm{D} - t(c; \boldsymbol{\theta})), & \text{if } t(c; \boldsymbol{\theta}) \leq \mathrm{D}, \\ 0, & \text{otherwise.} \end{cases} \tag{1.1}$$

Then, the total score of the schedules $\boldsymbol{\theta}$ is defined as

$$SCORE(\boldsymbol{\theta}) = \sum_{c \in C} score(c; \boldsymbol{\theta}). \tag{1.2}$$

1.3 Preprocessing of a problem instance

In this section, we introduce preprocessing steps that we apply to reduce the total number of parameters we need to optimize. These are our own observations and are not part of the original problem statement. After preprocessing, a substantial amount of intersections and streets is removed from optimization, which allows us to save resources and focus only on the important parameters. We also describe the format in which we represent the schedules.

1.3.1 Preprocessing steps

Throughout this section, we introduce some additional terms to help us better describe the steps.

Used and unused streets *Unused street* is a street that is the final destination of all cars that have it in their path (i.e., a traffic light for this street is not needed). *Used street* is a street that is not the final destination of at least one car that has it in its path.

Used and unused intersections *Unused intersection* is an intersection where all incoming streets are unused streets. *Used intersection* is an intersection with at least one used incoming street.

Step 1: Remove unused intersections and unused streets
Remove unused intersections and unused incoming streets from used intersections.

Step 1 is an obvious one; unused intersections are never used and only add unnecessary complexity. Unused incoming streets in used intersections prolong the traffic light cycle for the whole intersection, which very frequently leads to longer waiting times and thus a lower score. We now proceed to the next step.

Trivial and non-trivial intersections There are two types of used intersections: trivial and non-trivial. *Trivial intersection* is an intersection with exactly one used incoming street. *Non-trivial intersection* is an intersection with two or more used incoming streets.

Trivial and non-trivial streets For completeness, we also split the used streets into trivial and non-trivial ones. *Trivial street* is a used incoming street in a trivial intersection. *Non-trivial street* is a used incoming street in a non-trivial intersection.

Step 2: Fix the schedule for trivial intersections; optimize only schedules of non-trivial intersections For each trivial intersection, the schedule can be fixed by assigning a green light to its only used incoming street for the entire duration. Trivial intersections can then be removed from optimization, and only schedules of non-trivial intersections are optimized.

Let us think about Step 2 in more detail. For trivial intersections, we have two meaningful schedule options:

1. Keep the light green the whole time.
2. Keep the light red the whole time, effectively blocking all cars there.

Empirically speaking, the vast majority of trivial intersections should be kept green, otherwise many cars would not be able to reach their destination at all.

However, suppose that keeping the light red for some trivial intersection improves the score. This means that there must be a problematic car passing through this intersection. Such a car must eventually reach some non-trivial intersection via a street, and this street can still be set to have a red light the whole time during optimization. We therefore leave this up to the optimization algorithm, hoping that it explores this option if it is indeed beneficial.

To demonstrate the usefulness of preprocessing, Tables 1.1 and 1.2 show statistics of intersections and streets for each dataset. After applying both steps, we are only left with non-trivial intersections and non-trivial streets for optimization. This enables us to skip optimizing thousands of parameters.

Intersections

Dataset	Unused	Trivial	Non-trivial
A	1 (25%)	2 (50%)	**1 (25%)**
B	777 (11%)	4,977 (70%)	**1,319 (19%)**
C	2,340 (23%)	4,468 (45%)	**3,192 (32%)**
D	0 (0%)	0 (0%)	**8,000 (100%)**
E	0 (0%)	263 (53%)	**237 (47%)**
F	30 (2%)	332 (20%)	**1,300 (78%)**

Table 1.1 Intersection statistics across all datasets.

From now on, whenever we refer to schedules, we mean the *schedules of non-trivial intersections with non-trivial streets*, as these intersections and streets are the only ones that are left after preprocessing.

	Streets		
Dataset	Unused	Trivial	Non-trivial
A	1 (20%)	2 (40%)	**2 (40%)**
B	1,138 (12%)	4,977 (55%)	**2,987 (33%)**
C	23,558 (67%)	4,468 (13%)	**7,004 (20%)**
D	12,054 (13%)	0 (0%)	**83,874 (87%)**
E	42 (4%)	263 (26%)	**693 (70%)**
F	4,667 (47%)	332 (3%)	**5,001 (50%)**

Table 1.2 Street statistics across all datasets.

1.3.2 Schedules format

Each schedule of an intersection consists of two parts: *order* and *times*. Order is an array of street indices that defines the order in which the streets have the green light. Times is an array of integers that defines the duration of the green light with respect to the street order. For illustration, recall the schedule from Figure 1.2; street-a (ID 0) has a green light for 2 seconds, and then street-b (ID 1) has a green light for 3 seconds. This schedule is represented in our format as follows:

$$\begin{array}{cc} \text{Order} & \text{Times} \\ \Big([0,1], & [2,3]\Big). \end{array}$$

Then, the schedules are collectively represented as

a **list of pairs**, where each pair consists of **order** and **times** arrays.

1.4 Datasets

This section briefly describes all datasets provided with the problem. The datasets vary in the size and structure of their city plans. In the previous section, we already showed statistics of intersections and streets (see Tables 1.1 and 1.2). Here we compare the datasets by the number of parameters we need to optimize. That is, twice the number of non-trivial streets—one parameter for the order and one for the time. Note that this is only one way to compare the datasets, and it does not account for the actual paths cars must take or how convoluted those paths may be.

A - An example: 4 parameters Simple toy problem dataset used for debugging (see Figure 1.4).

B - By the ocean: 5,974 parameters Dataset based on a real city plan of Lisbon, Portugal (see Figure 1.5).

C - Checkmate: 14,008 parameters Dataset with a chessboard-like pattern and regular structure of intersections and streets (see Figure 1.6).

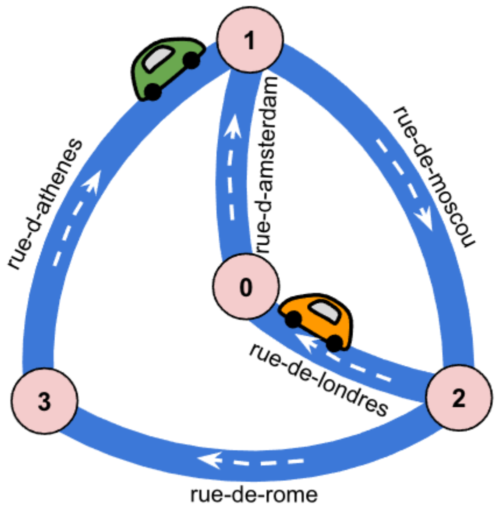

Figure 1.4 Visualization of dataset A [12].

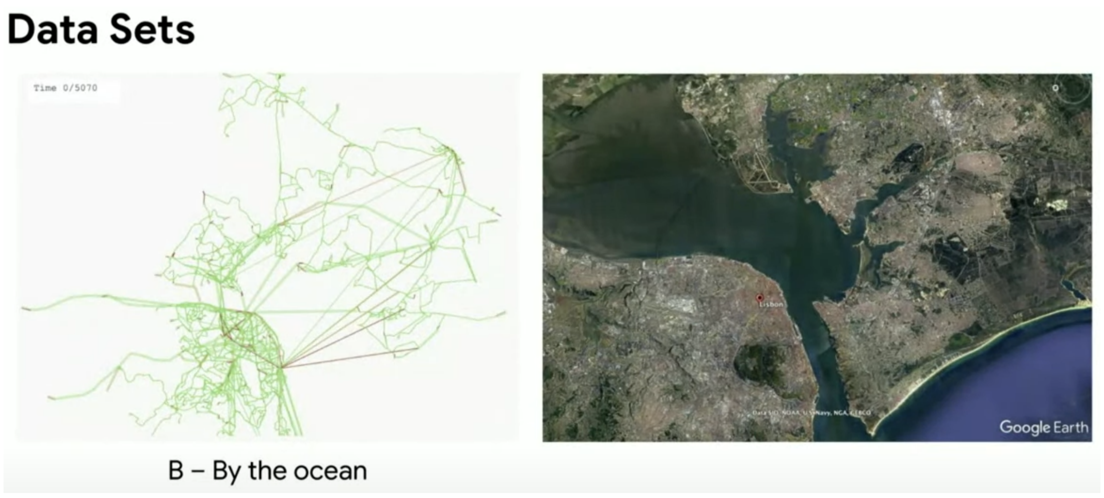

Figure 1.5 Dataset B based on the real data of Lisbon on the right. Screenshot from Hash Code 2021: Online Qualification Round Livestream.

D - Daily commute: 167,748 parameters By far the largest dataset with a challenging-to-navigate network from the *Barabási-Albert* distribution [15].

E - Etoile: 1,386 parameters Nicknamed *Etoile*[2], this dataset is a one big star, meaning there is one very important intersection in the middle with hundreds of incoming streets (see Figure 1.6).

F - Forever jammed: 10,002 parameters Medium sized dataset but again with a complex network difficult to optimize.

1.4.1 Score normalization

As mentioned in Section 1.1, each dataset yields an absolute score within a different range. To compare the performance across all datasets, we normalize the

[2]*Étoile* means star in French.

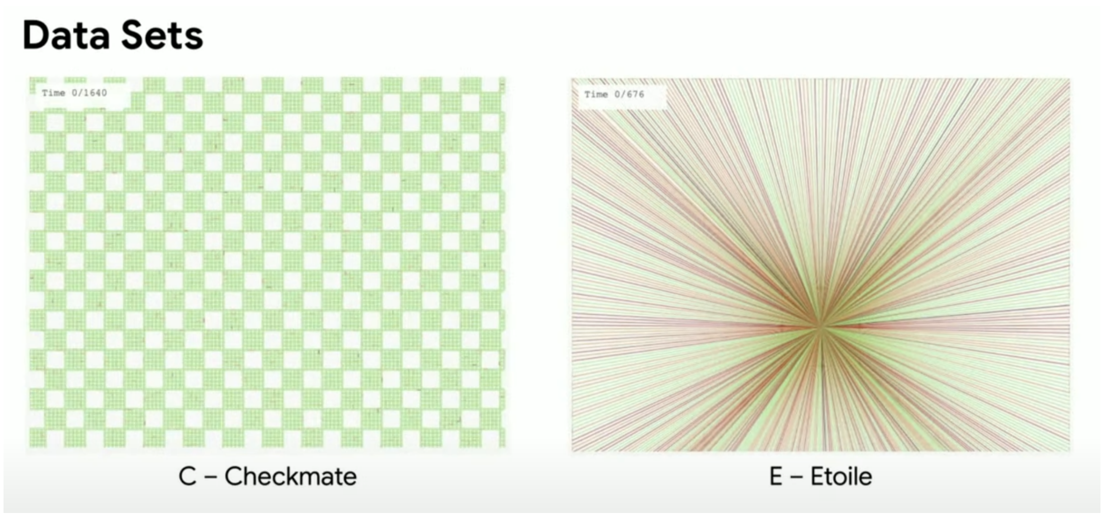

Figure 1.6 Visualization of datasets C and E. Screenshot from Hash Code 2021: Online Qualification Round Livestream.

scores to a 0–1 scale, where 0 represents our baseline solution (see Section 3.1 for details), and 1 corresponds to the maximum known score[3] for the dataset. Note that the baseline is already a good solution, so there may be limited room for improvement—for example, in dataset B.

1.5 Simulator

This section provides a brief overview of the custom simulator that we developed to serve as a black-box fitness function for the optimization algorithms. We discuss the motivation behind its design and describe the functionality relevant to the optimization process. For further information on using the simulator or regarding its implementation, please refer to the user guide in Appendix A and the developer documentation in Appendix B.

Our simulator replaces the original competition evaluation system, which was accessible only through the competition website via a user interface and is no longer available. It also adds several features on top of the original system. The simulator is written in C++ to be as fast and efficient as possible, and is wrapped as a Python package using the powerful pybind11 library [16], making it very convenient to use. Inspired by libraries such as NumPy[4] and PyTorch[5], our goal is to provide an easy-to-use interface and seamless integration with the vital Python ecosystem, without compromising on top-tier performance.

Following are the features of the simulator package that are important for optimization. These features are implemented in C++ and exposed through a lightweight Python API:

- Reading and storing the input data.

[3]See the maximum known scores at `https://github.com/sagishporer/hashcode-2021-qualification#score`.

[4]`https://numpy.org/`

[5]`https://pytorch.org/`

- Creating initial schedules using several different options listed in Section 3.1.
- Running the simulation to evaluate given schedules.
- Retrieving and updating schedules in a format described in Section 1.3.2.

The only case where larger data are passed between C++ and Python is when retrieving and updating the schedules. Since the optimization loop is implemented in Python using parts of the DEAP library [17], it must retrieve the schedules from the simulator, modify them externally, and then send them back into the simulator to evaluate their score.

2 Theory of optimization methods

In this chapter, we cover the theory of the methods we use to optimize the Traffic signaling problem. Specifically, we describe *hill climbing* (Section 2.1), *simulated annealing* (Section 2.2) and *genetic algorithm* (Section 2.3). In Section 2.4 and Section 2.5 we discuss common methods for initializing and parallelizing the algorithms.

In optimization, the goal is to find the best state according to an objective function, also known as a *fitness function*. In other words, the fitness function is a problem-specific function that defines how well a solution solves a particular problem. Because the state space of optimized problems can be very large or even infinite, systematic search methods are often unsuitable. Instead, we can use the so-called *heuristic algorithms*. These algorithms do not guarantee to find the best solution but are frequently used in practice because they can find good solutions in reasonable time. Our three chosen methods are all examples of heuristic algorithms.

2.1 Hill climbing

Hill climbing [18, 19] is one of the simplest local search algorithms. It keeps track of the current state and on each iteration moves to the neighboring state with the highest value, i.e., it moves in the direction that provides the biggest improvement. If there are no neighboring states with a higher value, the algorithm terminates. The greedy nature of hill climbing is both its strength and its weakness. It can make rapid progress towards a good solution but once it gets stuck in a local optimum it cannot escape it.

A lot of variants of this algorithm exist. In our experiments, we use a version called *first-choice hill climbing* (see Algorithm 1). This version randomly generates next states until it finds one with a higher value than the current state and then moves to this state. This strategy works well when there are many neighboring states or when no defined neighborhood structure exists, which is exactly our case.

```
function First-Choice-Hill-Climbing(problem)
    current ← problem.Initial
    for $t = 1$ to Max_iterations do
        next ← a randomly generated next state
        if Value(next) > Value(current) then
            current ← next
    return current
```

Algorithm 1 First-choice hill climbing

2.2 Simulated annealing

Simulated annealing [18, 19] builds up on the idea of hill climbing and solves the problem of getting stuck in local optima (see Algorithm 2). In each iteration, it randomly generates a next state. If the state is better, it is always accepted. Otherwise, the worse state is accepted with some probability. The probability decreases exponentially with how much worse the state is and with the current value of the "cooling" schedule. By allowing moves to worse states, the algorithm can escape local optima and therefore find better solutions.

The idea of the cooling schedule is inspired by the process of annealing in metallurgy, where metals are heated to a high temperature and then slowly cooled down to reach a strong, crystalline structure. Analogously, we want to allow worse moves more often at the beginning during the exploration phase and then gradually decrease that chance to hopefully converge to the best solution. Obviously, the choice of the schedule is absolutely crucial for the performance of the algorithm and has to be tuned specifically for each problem.

```
function SIMULATED-ANNEALING(problem, schedule)
    current ← problem.INITIAL
    for $t = 1$ to MAX_ITERATIONS do
        $T$ ← schedule($t$)
        next ← a randomly generated next state
        $\Delta E$ ← VALUE(next) − VALUE(current)
        if $\Delta E > 0$ then
            current ← next
        else
            current ← next with probability $e^{\Delta E/T}$
    return current
```

Algorithm 2 Simulated annealing

2.3 Genetic algorithm

Initially proposed by Holland [20] and Goldberg [21], genetic algorithm [18, 22] belongs to the group of *evolutionary algorithms*. Those algorithms are inpired by the seminal Darwin's theory of evolution, particularly by the idea of *natural selection* [23]. It states that the individuals with better traits suited to their environment are more likely to adapt, survive and reproduce. Over successive generations, these advantageous traits become more common in the population, leading to the emergence of new species.

This biological motivation is translated into the context of evolutionary algorithms in the following way. We work with a set of individuals called a *population*. Each individual in this population represents a solution to a given problem. The algorithm runs in iterations called *generations*. In each generation, it selects some individuals, modifies them using the so-called *genetic operators* and then selects which individuals survive to the next generation. The selection process prefers individuals that are better at solving the problem, i.e., individuals with higher fitness. There are many different evolutionary algorithms with even more variations that follow the previous description. Let us now focus specifically on the genetic algorithm (see Algorithm 3).

```
function GENETIC-ALGORITHM(population)
    population ← INITIALIZE(population)
    for $t = 1$ to MAX_ITERATIONS do
        population ← EVALUATE(population)
        selected ← SELECT(population)
        offspring ← REPRODUCE(selected)
        population ← offspring
    return population

function REPRODUCE(selected)
    offspring ← empty list
    for parent1, parent2 in selected do
        child1, child2 ← CROSSOVER(parent1, parent2)
        if small random probability then
            child1 ← MUTATION(child1)
        if small random probability then
            child2 ← MUTATION(child2)
        add child1, child2 to offspring
    return offspring
```

Algorithm 3 Genetic algorithm

2.3.1 Selection

For the selection of parents for the next generation, common methods include the *roulette wheel selection* and the *tournament selection* [22]. Roulette wheel selection selects individuals with a probability directly proportional to their fitness. Its downside is that it does not work for negative fitness and can be sensitive to changes in fitness values. Tournament selection works by repeatedly sampling a few individuals and selecting the best one among them. Thus, it is not dependent on specific values and can be used with any fitness function.

2.3.2 Crossover and mutation

The offspring is created from the selected parents using genetic operators *crossover* and *mutation.* Crossover combines two parents to produce two new children. There are many different ways to perform crossover, the most common being the *one-point crossover.* Since individuals are usually represented as strings or arrays of numbers, one-point crossover simply picks a point in the individual and creates two new individuals by copying the first part from one parent and the second part from the other parent and vice versa. This can be extended to the *two-point crossover* or the *k-point crossover* in general. Other examples of crossover, e.g., if an individual represents a permutation and simple swapping of parts cannot be used, include the *order crossover (OX)* or the *partially mapped crossover (PMX)* [22], also known as the *partially matched crossover (PMX)*.

Mutation is a small random change in an individual. It is used to introduce variability into the population and to prevent the algorithm from getting stuck in local optima. Mutation is usually more problem-specific than crossover and is often customized to the problem at hand. Some common examples are the *bit-flip mutation* for binary strings, the *uniform* and the *gaussian* mutations, i.e., a mutation replacing a value with a number drawn from one of these distributions, or the *index shuffle* mutation for permutations.

The created offspring directly replaces the old population and the process repeats. Unfortunately, this mechanism does not guarantee that the best individual will be preserved in the next generation. To prevent this, a technique called *elitism* can be used. Elitism always transfers a few of the best parents to the next generation, ensuring that the best solution found so far is not lost. The rest of the population is then filled with the offspring as previously mentioned.

2.4 Initialization

In all of the previously mentioned algorithms, we start with some initial state or population of initial states. The simplest and most straightforward way is to generate the initial state randomly. This is often good enough, but if the state space is large or the problem is complex, the algorithm can be very inefficient. Instead, we can try to come up with some smarter heuristic method to start in a more promising region of the search space. This is especially beneficial if we have some prior knowledge about the problem because we can help the algorithm skip exploring irrelevant parts of the search space, effectively reducing the number of iterations needed to find a good solution. An example of such a heuristic initialization is the *nearest neighbors* initialization for the *Traveling Salesman Problem (TSP)*, where the next node in the sequence is chosen as the closest unvisited node instead of generating the sequence randomly.

However, using heuristics for initialization can be a double-edged sword as it can lead to *premature convergence* [22]. This means losing the diversity of solutions too early and getting stuck in a local optimum. When using optimization algorithms, it is always important to balance the trade-off between *exploration* and *exploitation* [19]. Exploration is a process of searching the search space for new solutions, while exploitation is a process of refining the current solutions.

2.5 Parallelization

As hinted in the previous sections, heuristic algorithms can run for a long time. It is not uncommon to compute hundreds of thousands of fitness evaluations—imagine a genetic algorithm with a population size of 200, run for 1000 generations. These evaluations can take a while, especially if they are simulations, games, neural network training, or similar. As a result, we may want to use parallelization to speed up this process and fully utilize our computational resources.

Single-state methods like hill climbing and simulated annealing are inherently sequential and thus difficult to parallelize. But we can at least run multiple independent instances of the algorithm in parallel. Note that this is different from methods like *beam search*, where useful information is shared between the instances.

For population methods like genetic algorithm, parallelization makes much more sense. The most common way is to run the fitness evaluations of individuals in parallel. This is great because the evaluations are completely independent of each other and the fitness computation is often the biggest bottleneck of the whole algorithm. Other parts, such as mutation and crossover, can also be parallelized across individuals, but these are usually much faster than the evaluation and may not provide much speedup, if any.

3 Application of optimization algorithms

In this chapter, we describe the problem-specific details of our chosen optimization algorithms. Section 3.1 presents different options for creating initial schedules and explains which ones were chosen for optimization. Section 3.2 gives a high-level overview of how the schedules are modified by the algorithms. Sections 3.3, 3.4, and 3.5 cover the exact configuration of genetic algorithm, hill climbing, and simulated annealing, respectively, including their hyperparameters. Section 3.6 explains how we searched for the best hyperparameters and summarizes the final settings for each algorithm and dataset.

3.1 Initial schedules

As mentioned in Section 1.5, the simulator offers several options for generating initial schedules. Here we describe the different options for initializing both *order* and *times* (see Section 1.3.2). The order initialization options include:

- *default* - simply uses the order given by street IDs in the input file
- *random* - takes a random permutation of the streets
- *adaptive* - determines the order during a simulation run—each street is assigned to the earliest free position in the order array when used for the first time (only compatible with the *default times* initialization; with other options, it may result in an inconsistent format and fail to generate schedules)

The times initialization options include:

- *default* - all times are set to 1 second
- *scaled* - time for each street is a total number of cars using this street divided by a single given constant (the divisor)

Both *order initialization* and *times initialization* are hyperparameters. To determine the best settings for optimization, we experimentally compared the performance of different initialization options for each dataset. Specifically, we compared the following options:

- **default (baseline)** - default order and default times
- **adaptive** - adaptive order and default times
- **random** - random order and default times
- **scaled** - default order and scaled times

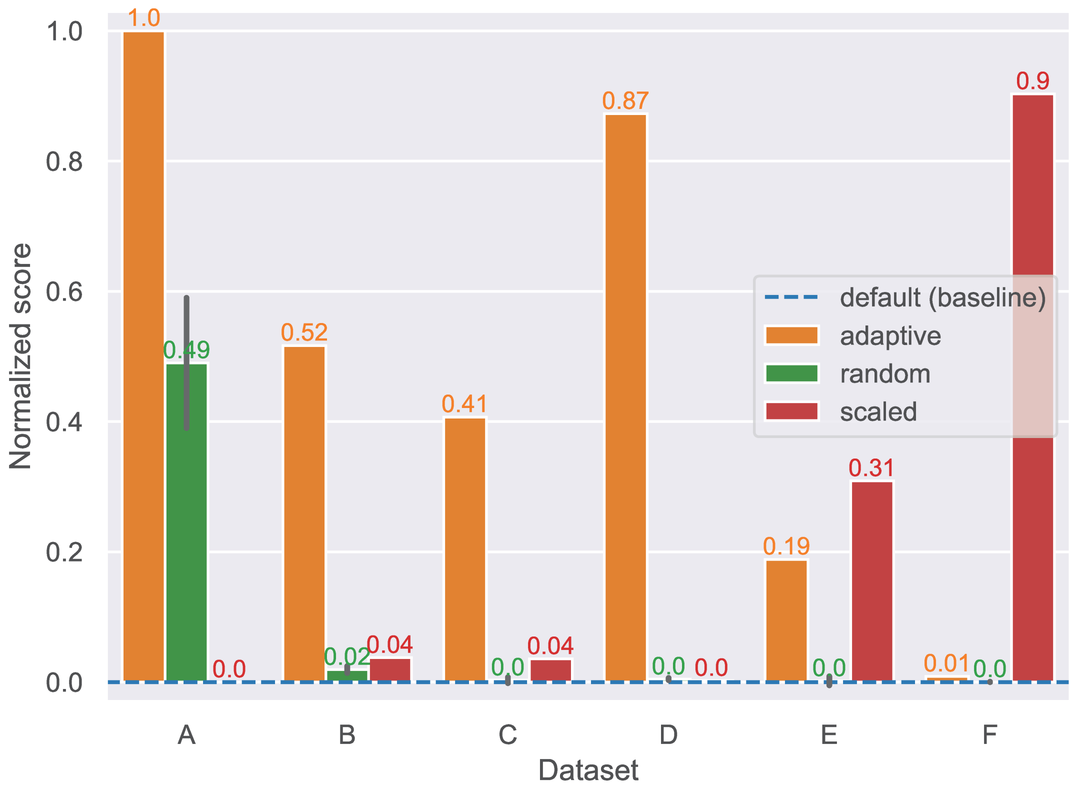

Figure 3.1 Comparison of different initialization options for each dataset. Y-axis shows the normalized score (see Section 1.4.1). The black error bars indicate the 95% confidence intervals.

Scaled option uses the best divisor between 1–100 for each dataset. Random option is stochastic and is therefore averaged over 100 trials.

The results are shown in Figure 3.1. The **random** option performs similarly to the **default** option—since the default order is neither optimized nor intentionally poor, it is reasonable that the random order has comparable performance on average. The **adaptive** option achieves strong results, especially on datasets B, C, and D. This is likely because streets in these datasets are mostly used by one or a few cars, making the specific order very important and the default times a suitable choice. The **scaled** option performs best on datasets E and F. Unsure about the reason for dataset E, but dataset F contains many streets used by hundreds of cars, so it makes sense to increase the green light time for these streets.

We see that using other "smarter" initialization methods can significantly improve the baseline solution, enabling the optimization process to begin from a much better starting point. We decided to use *adaptive order* and *default times* for datasets B, C, and D, and *random order* and *scaled times* for datasets E and F (see Table 3.3).

3.2 Modifying the schedules

Once the initial schedules are created, we can start modifying them using the proposed optimization algorithms. Since the schedules format (see Section 1.3.2) is quite complex and each intersection's schedule may have a different number of streets, operations such as crossover or mutation are applied separately for each intersection. Moreover, because the *times* array depends on the *order* array, any modifications to the order array must also be applied to the corresponding times array.

3.3 Genetic algorithm

In this section, we apply the theory introduced in Section 2.3 and describe our design of the genetic algorithm, including the selection, crossover, and mutation operators, as well as the corresponding hyperparameters. We begin with this method, as the other algorithms reuse its mutation operator to generate new states.

3.3.1 Selection

For selecting individuals for reproduction, we use *tournament selection* combined with *elitism*. The corresponding hyperparameters are *tournament size* and *elitism*, which defines the percentage of top-performing individuals that are directly selected for reproduction.

3.3.2 Crossover

For each intersection, we randomly decide whether to crossover only the order, only the times, or both. We use *order crossover (OX)* for modifying the order, and *two-point crossover* for modifying the times. Crossover is performed with a probability defined by the hyperparameter *crossover probability*.

3.3.3 Mutation

For each intersection, we randomly decide whether to mutate only the order, only the times, or both. We use *index shuffle* for modifying the order, and for the times, we add or subtract one to some values at random. Mutation is performed with a probability defined by the hyperparameter *mutation probability*. When mutation is applied, the hyperparameter *mutation bit rate* controls how likely each individual value is to be modified. It can be given either as a probability or as an integer specifying the expected number of modified values in the state.

The remaining hyperparameters of the genetic algorithm are *population size* and *generations*—their names are quite self-explanatory. Together with the crossover and mutation probabilities, these hyperparameters control the total number of fitness evaluations (i.e., simulation runs) performed by the algorithm. Additionally, utilizing the approach from Section 2.5, we run fitness evaluations within each population in parallel.

3.4 Hill climbing

As explained in Section 2.1, we use a variant of the algorithm that generates next states randomly because there is no explicit neighborhood structure. We simply apply the mutation operator from the genetic algorithm (see Section 3.3.3) to generate the next state. Note that we again use the hyperparameter *mutation bit rate* but not the mutation probability because the mutation is always applied. The only other hyperparameter is *iterations*, which sets the number of iterations the algorithm runs for.

3.5 Simulated annealing

Introduced in Section 2.2, this algorithm uses the same strategy to generate next states as presented in Section 3.4. That means it has the same hyperparameters *mutation bit rate* and *iterations*. The only additional hyperparameter is *initial temperature*, which sets the initial temperature of the cooling schedule. For the cooling schedule, we use a linear decay. It is defined as

$$schedule(t) = \tau_{init} \cdot \left(1 - \frac{t}{T}\right) + \varepsilon, \tag{3.1}$$

where τ_{init} is the *initial temperature*, t is the current iteration, and T is the total number of *iterations*.

3.6 Hyperparameter search

Performance of all three aforementioned algorithms largely depends on the setting of their hyperparameters. To find the best hyperparameters for the experiments, we use a form of *greedy search*. That is, we only focus on optimizing one hyperparameter at a time and try to find the best value for it. Other hyperparameters are currently fixed—either heuristically or set to their already optimized values. We test a number of reasonable values for each hyperparameter and perform runs with additional values if the results are not satisfactory. Each setting is tested on 10 different fixed seeds and the results are averaged.

To maintain comparability, we allocate an equal budget of fitness evaluations to all three algorithms. For hill climbing and simulated annealing, this budget is easily enforced via the *iterations* hyperparameter. For genetic algorithm, the number of evaluations is stochastic and cannot be precisely set; however, we always set the *generations* hyperparameter so that the expected total number of fitness evaluations matches the predefined budget.

For each dataset, we use the best initialization options that we selected in Section 3.1.

Crossover probability	$\{0.2, 0.3, \ldots, 0.8\}$
Mutation probability	$\{0.1, 0.2, \ldots, 1.0\}$
Tournament size	$\{2, 3, \ldots, 10\}$
Elitism	$\{0, 0.05, 0.1, \ldots, 0.3\}$
Population size	$\{10, 20, \ldots, 100\}$
Mutation bit rate	$\{1, 2, \ldots, 20\}$
Initial temperature	many values between 0.1 and 500

Table 3.1 Ranges of tested hyperparameter values.

To give a clearer picture of the values explored, Table 3.1 summarizes the ranges used in the hyperparameter search. The genetic algorithm parameters—*crossover probability*, *mutation probability*, *tournament size*, and *elitism*—were tested only on smaller datasets E and B to reduce search complexity, because they seemed generalizable across datasets. Other parameters, especially the *mutation bit rate* and *initial temperature*, are highly dataset-dependent and were tuned separately for each algorithm and dataset.

Finally, Tables 3.2 and 3.3 provide a detailed overview of the selected hyperparameters for the experiments. For clarity, the first table lists the hyperparameters that are shared across all datasets, while the second table shows dataset-specific hyperparameters.

Genetic Algorithm	
Crossover probability	0.6
Mutation probability	0.4
Tournament size	3
Elitism	0.05
Hill Climbing	
Iterations	450,000
Simulated Annealing	
Iterations	450,000

Table 3.2 Shared hyperparameters for all datasets.

Dataset E

Order initialization	random
Times initialization	scaled
Genetic Algorithm	
Population size	90
Generations	6,667
Mutation bit rate	9
Hill Climbing	
Mutation bit rate	15
Simulated Annealing	
Mutation bit rate	5
Initial temperature	275

Dataset B

Order initialization	adaptive
Times initialization	default
Genetic Algorithm	
Population size	10
Generations	60,000
Mutation bit rate	2
Hill Climbing	
Mutation bit rate	4
Simulated Annealing	
Mutation bit rate	1
Initial temperature	0.25

Dataset F

Order initialization	random
Times initialization	scaled
Genetic Algorithm	
Population size	50
Generations	12,000
Mutation bit rate	2
Hill Climbing	
Mutation bit rate	5
Simulated Annealing	
Mutation bit rate	3
Initial temperature	15

Dataset C

Order initialization	adaptive
Times initialization	default
Genetic Algorithm	
Population size	20
Generations	30,000
Mutation bit rate	2
Hill Climbing	
Mutation bit rate	4
Simulated Annealing	
Mutation bit rate	2
Initial temperature	0.1

Dataset D

Order initialization	adaptive
Times initialization	default
Genetic Algorithm	
Population size	40
Generations	15,000
Mutation bit rate	2
Hill Climbing	
Mutation bit rate	2
Simulated Annealing	
Mutation bit rate	2
Initial temperature	0.1

Table 3.3 Dataset-specific hyperparameters. The order and times initializations are the same for all three methods for each dataset.

4 Experimental results

This chapter presents the results of our experiments. Specifically, we compare the performance of three optimization algorithms:

- *genetic algorithm (GA)*,
- *hill climbing (HC)*,
- *simulated annealing (SA)*,

on datasets B, C, D, E, and F. Section 4.1 explains the setup of the experiments and format of the results shown in the figures and tables. The following sections present the results for each dataset, including plots and tables summarizing the performance of each algorithm. The datasets are presented in ascending order of their size (see Section 1.4 for more details).

4.1 Experimental setup

As previously mentioned, Tables 3.2 and 3.3 summarize the hyperparameters used for each algorithm and dataset. The experiments were performed on either AMD EPYC 9454 or AMD EPYC 9474F CPUs. Each setting was run with 10 different fixed seeds, and the results were averaged to ensure reliability. As explained in Section 3.6, all three algorithms were run for the same fixed budget of fitness evaluations.

Note that comparing by number of evaluations is somewhat disadvantageous for the genetic algorithm, which inherently performs a broader search of the state space. Additionally, fitness evaluations within each population are computed in parallel, which can result in significantly faster runtimes compared to single-state methods. Nevertheless, we chose the number of fitness evaluations as the comparison metric instead of runtime, as it better reflects algorithmic efficiency and is independent of implementation details and the specific hardware used.

For each dataset, we provide a plot illustrating the performance of all three methods throughout the optimization process. The x-axis indicates the number of fitness evaluations, while the primary y-axis shows the normalized score (see Section 1.4.1). For completeness, the secondary y-axis displays the absolute score. For genetic algorithm, the plotted curve represents the best score within the current population. For hill climbing and simulated annealing, the curve shows the current score at each step. All curves are averaged over 10 runs, with the shaded areas indicating the standard deviation.

For each dataset, we also include a table reporting the overall best score achieved by each algorithm, averaged over 10 runs, along with the total runtime. The runtime is included to give a general idea of how long each method took to run, but it should not be interpreted as an exact metric.

Finally, we summarize in Table 4.6 the best scores achieved by each algorithm in a single run (out of 10 seeds), and compare them to the maximum known score for each dataset. The schedules corresponding to the best scores achieved for each dataset during optimization are included in the thesis attachment.

4.2 Dataset E

For dataset E, the smallest optimized dataset, *SA* not only outperformed *GA* and *HC*, but even exceeded the best known score, which explains why its normalized score in Table 4.1 is greater than 1. While *GA* and *HC* quickly reached good scores above 80% of the best known score, they converged early and improved only slightly afterwards. In contrast, *SA* started off slower and volatile, but continued to steadily improve its score throughout the optimization process (see Figure 4.1).

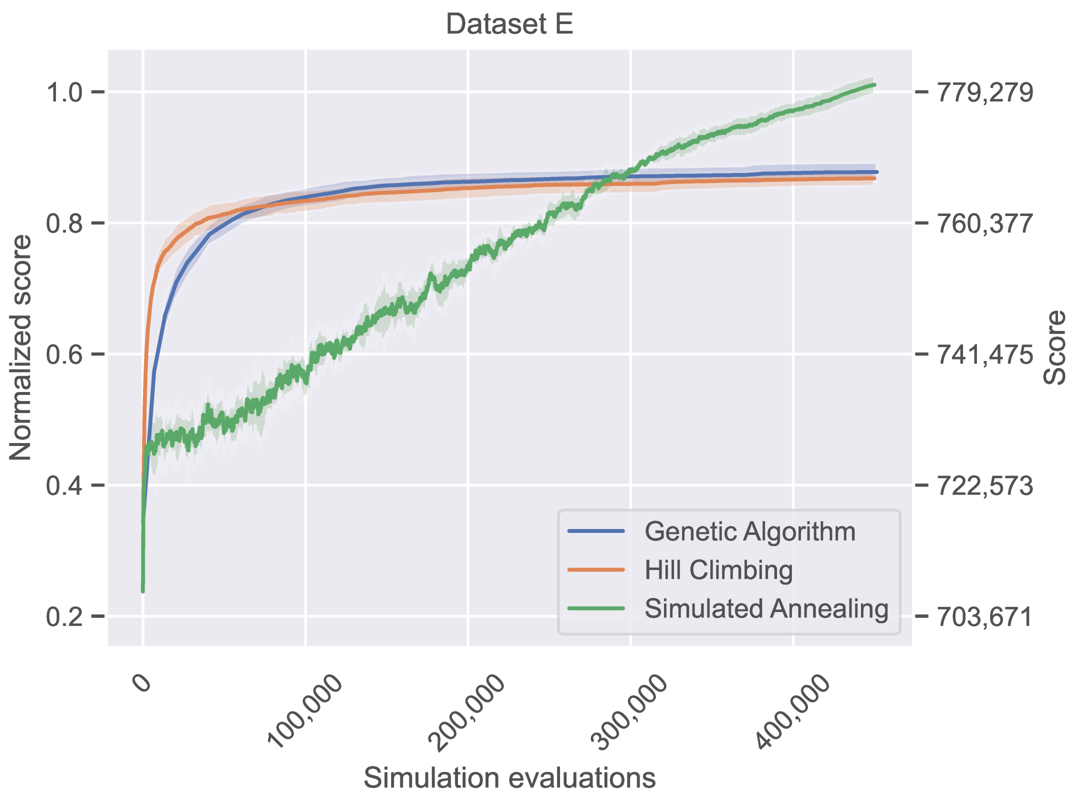

Figure 4.1 Performance of the algorithms on dataset E.

	Normalized Score	Score	Runtime (mm:ss)
Genetic Algorithm	0.88	767,768	**06:58**
Hill Climbing	0.87	766,798	07:03
Simulated Annealing	**1.01**	**780,299**	07:10

Table 4.1 Final statistics for dataset E.

4.3 Dataset B

For dataset B, *SA* again achieved the best score among the three algorithms. This time, it performed the best since the very beginning of the optimization process (see Figure 4.2). Although close, it fell short of the best known score, reaching 97% of it (see Table 4.2). *HC* started off similarly to *SA*, but again converged early and ultimately performed the worst—though it still achieved a good score. *GA* performed noticeably better than *HC* but did not reach the performance of *SA*. Note that the parallel fitness evaluation in *GA* starts to show its advantage, with the *GA* runtime being approximately one-third shorter than the other two algorithms (see Table 4.2).

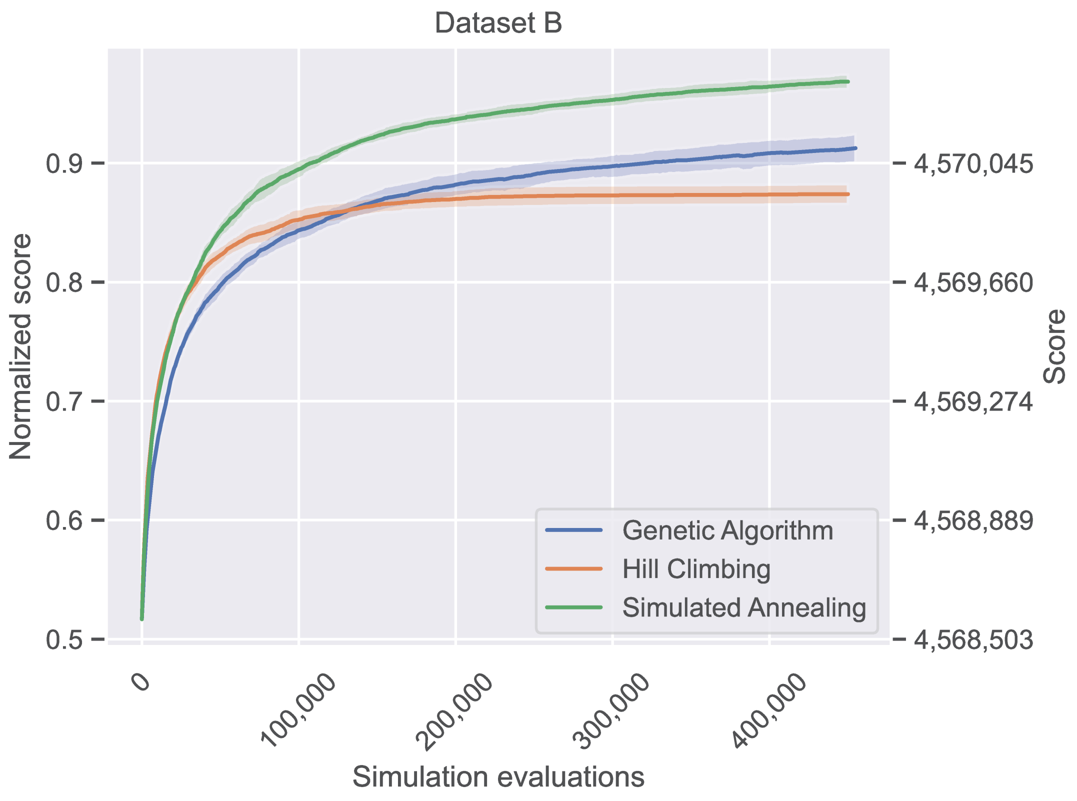

Figure 4.2 Performance of the algorithms on dataset B.

	Normalized Score	Score	Runtime (h:mm:ss)
Genetic Algorithm	0.91	4,570,095	**0:42:29**
Hill Climbing	0.87	4,569,945	1:00:08
Simulated Annealing	**0.97**	**4,570,309**	0:59:07

Table 4.2 Final statistics for dataset B.

4.4 Dataset F

For dataset F, the performance of all three algorithms is much closer than in previous datasets, due to starting from a really good initial solution. Nonetheless, *SA* achieved the highest score, *HC* performed the worst, and *GA* was in between (see Table 4.3). As shown in Figure 4.3, the shaded areas representing standard deviation overlap quite a lot, indicating that there is not a big difference between the algorithms. *GA* again demonstrated a significantly shorter runtime.

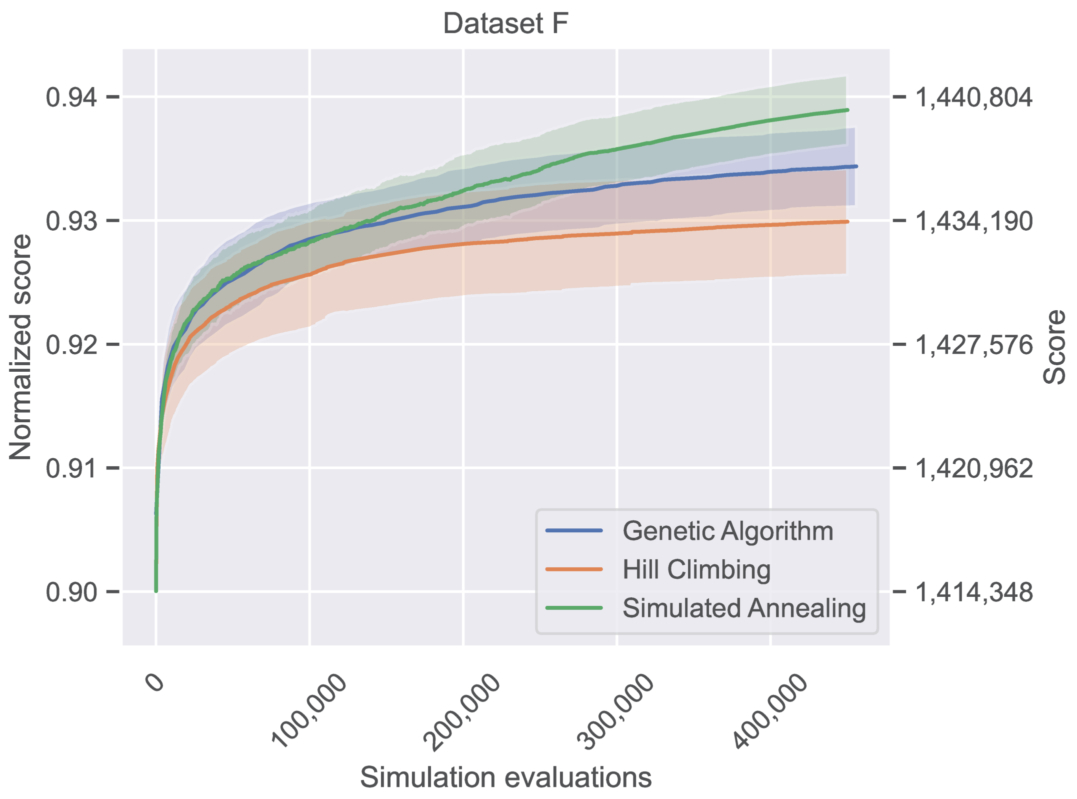

Figure 4.3 Performance of the algorithms on dataset F.

	Normalized Score	Score	Runtime (h:mm:ss)
Genetic Algorithm	0.93	1,437,086	**1:48:33**
Hill Climbing	0.93	1,434,129	2:41:04
Simulated Annealing	**0.94**	**1,440,097**	2:28:03

Table 4.3 Final statistics for dataset F.

4.5 Dataset C

For dataset C, the curves shown in Figure 4.4 resemble those observed for dataset B (see Figure 4.2). *SA* consistently outperformed the other algorithms from the start and again exceeded the best known score, with the normalized score above 1 (see Table 4.4). *GA* and *HC* performed similarly, with *GA* narrowly outperforming *HC*. As in previous datasets, *GA* finished in a significantly shorter time (see Table 4.4).

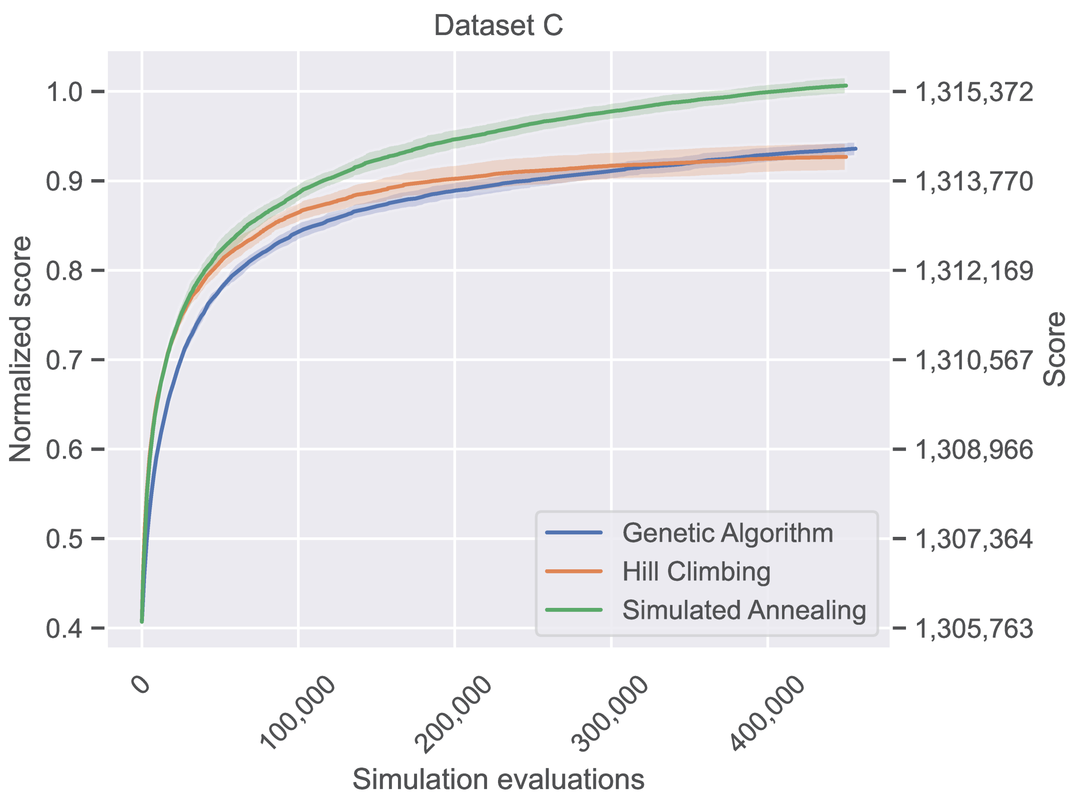

Figure 4.4 Performance of the algorithms on dataset C.

	Normalized Score	Score	Runtime (h:mm:ss)
Genetic Algorithm	0.94	1,314,347	**2:02:52**
Hill Climbing	0.93	1,314,200	3:33:10
Simulated Annealing	**1.01**	**1,315,476**	3:26:31

Table 4.4 Final statistics for dataset C.

4.6 Dataset D

For dataset D, the largest dataset by far, the results are different from all previous datasets. *HC*, the weakest algorithm in the previous cases, performed the best here, although only slightly ahead of *SA* (see Figure 4.5). It is likely that the algorithms would have benefited from running for more evaluations, as their performance curves had not yet fully converged. *GA*, in particular, performed the worst here, but completed in less than half the time of the other two algorithms (see Table 4.5), fully utilizing its parallel evaluation. However, the runtimes for this dataset were already much longer than for the rest of the datasets combined, showing how large and demanding dataset D really was.

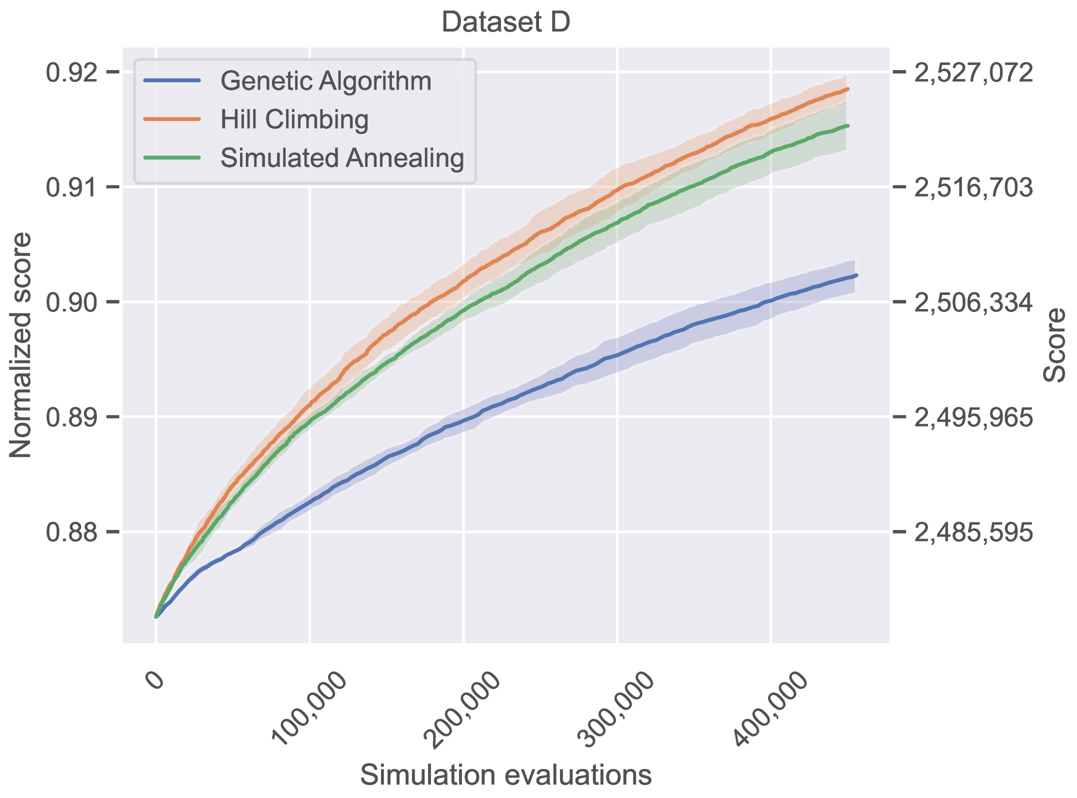

Figure 4.5 Performance of the algorithms on dataset D.

	Normalized Score	Score	Runtime (hh:mm:ss)
Genetic Algorithm	0.90	2,508,730	**09:41:52**
Hill Climbing	**0.92**	**2,525,531**	20:59:59
Simulated Annealing	0.92	2,522,204	22:39:17

Table 4.5 Final statistics for dataset D.

Dataset	GA	HC	SA	*Max known score*
B	4,570,168	4,569,994	4,570,346	*4,570,431*
C	1,314,597	1,314,584	**1,315,702**	*1,315,372*
D	2,512,355	2,528,954	2,525,797	*2,610,027*
E	771,025	768,443	**782,044**	*779,279*
F	1,440,172	1,439,639	1,443,333	*1,480,489*

Table 4.6 Best scores achieved in a single run (out of 10 seeds) by each algorithm, compared to the max known score. Bold values indicate newly achieved best scores that outperform the max known score. The schedules corresponding to the best scores achieved for each dataset during optimization are included in the thesis attachment.

Conclusion

In this thesis, we addressed the Traffic signaling problem from the Google Hash Code competition, which serves as a simplified version of the real-world problem of traffic signal optimization. We began by implementing a fast and efficient simulator in C++, which we wrapped as a Python package. We then integrated the simulator into a Python optimization pipeline as a black-box fitness function. This setup enabled quick evaluation of solutions and allowed us to perform many iterations of our three chosen optimization algorithms: *Genetic Algorithm (GA)*, *Hill Climbing (HC)*, and *Simulated Annealing (SA)*. We then compared the performance of these algorithms on the provided competition datasets of different sizes and structures.

Our experiments showed that all three algorithms achieved good results across all datasets. However, *SA* consistently outperformed the others on every dataset except the largest one, dataset D, where the otherwise weakest *HC* performed slightly better—likely because the algorithms had not yet fully converged. Furthermore, *SA* was able to surpass the max known scores in datasets C and E, achieving new best results. *GA* generally performed better than *HC*, but only by a small margin. The performance of all algorithms was highly dependent on the choice of hyperparameter values.

The superior performance of *SA* compared to *HC* is unsurprising, as *SA* is a more capable algorithm in theory. Its ability to move to worse states obviously broadens the search and helps to escape local optima. Nonetheless, based on our experimental experience, we believe that simply always moving to a state with the same fitness value as the current state is the key factor behind *SA*'s success.

On the other hand, *GA*'s performance was somewhat below expectations, especially considering the additional complexity involved compared to the single-state methods. There are several possible reasons for this. As previously hinted, comparing the algorithms by the number of evaluations is disadvantageous for *GA*. Moreover, greedily optimizing one hyperparameter at a time may be less optimal for *GA* because it has more hyperparameters—some form of grid search could be more appropriate. Additionally, the initial population might have lacked diversity or been completely homogeneous. Unfortunately, we were unable to come up with initializations that would increase diversity and still achieve good results. Lastly, we think that our optimization problem does not benefit much from the broader search capabilities of *GA* and is rather suited to methods that refine a single solution.

For future work, we could try to improve the performance of *GA* by exploring additional initializations, performing a wider hyperparameter grid search, or using some form of informed crossover or mutation. It would also be interesting to run *GA* for the same amount of time as *SA* to see if it can catch up. This approach could maybe better reflect the real-world scenario where we are constrained by time and aim to fully utilize *GA*'s parallel fitness evaluation. Additionally, we could test other single-state optimization methods such as *Iterated Local Search* [24] or *Tabu Search* [25].

Bibliography

1. Zhao, Dongbin; Dai, Yujie; Zhang, Zhen. Computational Intelligence in Urban Traffic Signal Control: A Survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews).* 2012, vol. 42, no. 4, pp. 485–494. issn 1558-2442. Available from doi: `10.1109/tsmcc.2011.2161577`.
2. Caves, R. W. *Encyclopedia of the City.* Routledge, 2004. isbn 9781134528479. Available from doi: `10.4324/9780203484234`.
3. Wang, Jindong; Jiang, Shengchuan; Qiu, Yue; Zhang, Yang; Ying, Jianguo; Du, Yuchuan. Traffic Signal Optimization under Connected-Vehicle Environment: An Overview. *Journal of Advanced Transportation.* 2021, vol. 2021, pp. 1–16. issn 0197-6729. Available from doi: `10.1155/2021/3584569`.
4. Mu, Rui; Nishihori, Yasuhide; Kato, Hideki; Ando, Ryosuke. Traffic Signal Optimization Models Minimizing Vehicles' Average Delay in an Isolated Signalized Intersection. *Journal of the Eastern Asia Society for Transportation Studies.* 2022, vol. 14, pp. 1906–1918. Available from doi: `10.11175/easts.14.1906`.
5. Gunarathne, Dinakara; Amarasingha, Niranga; Wickramasighe, Vasantha. Traffic Signal Controller Optimization Through VISSIM to Minimize Traffic Congestion, CO and NOx Emissions, and Fuel Consumption. *Science, Engineering and Technology.* 2023, vol. 3, no. 1. issn 2831-1043. Available from doi: `10.54327/set2023/v3.i1.56`.
6. Qadri, Syed Shah Sultan Mohiuddin; Gökçe, Mahmut Ali; Öner, Erdinç. State-of-art review of traffic signal control methods: challenges and opportunities. *European Transport Research Review.* 2020, vol. 12, no. 1. issn 1866-8887. Available from doi: `10.1186/s12544-020-00439-1`.
7. Smith, Stephen F.; Barlow, Gregory; Xie, Xiao-Feng; Rubinstein, Zachary B. SURTRAC: Scalable Urban Traffic Control. *Carnegie Mellon University.* 2013. Available from doi: `10.1184/R1/6561035.V1`.
8. Costa, Nator Junior Carvalho da; Maia, Jose E. B. An Intersection Traffic Signal Controller Optimized by a Genetic Algorithm. *International Journal of Computer Applications.* 2020, vol. 176, no. 40, pp. 9–13. issn 0975-8887. Available from doi: `10.5120/ijca2020920521`.
9. Ye, Bao-Lin; Wu, Weimin; Ruan, Keyu; Li, Lingxi; Chen, Tehuan; Gao, Huimin; Chen, Yaobin. A survey of model predictive control methods for traffic signal control. *IEEE/CAA Journal of Automatica Sinica.* 2019, vol. 6, no. 3, pp. 623–640. issn 2329-9274. Available from doi: `10.1109/jas.2019.1911471`.
10. Zhao, Haiyan; Dong, Chengcheng; Cao, Jian; Chen, Qingkui. A survey on deep reinforcement learning approaches for traffic signal control. *Engineering Applications of Artificial Intelligence.* 2024, vol. 133, p. 108100. issn 0952-1976. Available from doi: `10.1016/j.engappai.2024.108100`.

11. Saadi, Aicha; Abghour, Noureddine; Chiba, Zouhair; Moussaid, Khalid; Ali, Saadi. A survey of reinforcement and deep reinforcement learning for coordination in intelligent traffic light control. *Journal of Big Data*. 2025, vol. 12, no. 1. issn 2196-1115. Available from doi: 10.1186/s40537-025-01104-x.
12. Google. *Google Coding Competitions Archive* [online]. 2023. [visited on 2025-03-21]. Available from: https://github.com/google/coding-competitions-archive.
13. Rodrigues, Pedro Miguel Duque. *Principled Modeling of the Google Hash Code Problems for Meta-Heuristics*. 2023. Available also from: https://hdl.handle.net/10316/110471. MA thesis. University of Coimbra.
14. Li, Wenyu; Sun, Yilin; Zhao, Xiangyu. Building a Simulator and Emulator for Traffic Signaling [online]. 2022 [visited on 2025-03-21]. Available from: https://victorzxy.github.io/project/traffic-sim/.
15. Albert, Réka; Barabási, Albert-László. Statistical mechanics of complex networks. *Reviews of Modern Physics*. 2002, vol. 74, no. 1, pp. 47–97. issn 1539-0756. Available from doi: 10.1103/revmodphys.74.47.
16. Jakob, Wenzel; Rhinelander, Jason; Moldovan, Dean. *pybind11 – Seamless operability between C++11 and Python* [online]. 2017. [visited on 2025-04-22]. Available from: https://github.com/pybind/pybind11.
17. Fortin, Félix-Antoine; De Rainville, François-Michel; Gardner, Marc-André Gardner; Parizeau, Marc; Gagné, Christian. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*. 2012, vol. 13, pp. 2171–2175. issn 1532-4435. Available also from: https://dl.acm.org/doi/abs/10.5555/2503308.2503311.
18. Russell, Stuart; Norvig, Peter. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson, 2020. isbn 978-0134610993. Available also from: https://aima.cs.berkeley.edu/.
19. Luke, Sean. *Essentials of metaheuristics: A set of undergraduate lecture notes; Online Version 2.0*. 2nd ed. Lulu, 2013. isbn 9781300549628. Available also from: https://cs.gmu.edu/~sean/book/metaheuristics/.
20. Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, 1975. isbn 0472084607. Available from doi: 10.7551/mitpress/1090.001.0001.
21. Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989. isbn 9780201157673.
22. Vanneschi, Leonardo. *Lectures on Intelligent Systems*. Ed. by Silva, Sara. Cham: Imprint: Springer, 2023. Natural Computing Series. isbn 9783031179228. Available from doi: 10.1007/978-3-031-17922-8.
23. Darwin, Charles. *On the Origin of Species by Means of Natural Selection*. London: John Murray, 1859.

24. Lourenço, Helena Ramalhinho; Martin, Olivier C.; Stützle, Thomas. Iterated Local Search: Framework and Applications. In: *Handbook of Metaheuristics*. Springer International Publishing, 2018, pp. 129–168. ISBN 9783319910864. ISSN 2214-7934. Available from DOI: 10.1007/978-3-319-91086-4_5.

25. Glover, Fred; Laguna, Manuel. Tabu Search. In: *Handbook of Combinatorial Optimization*. Springer US, 1998, pp. 2093–2229. ISBN 9781461303039. Available from DOI: 10.1007/978-1-4613-0303-9_33.

A User guide

This guide is intended for users interested in running the code associated with this thesis. It is logically divided into two parts:

- **Simulator** - a standalone tool for evaluating the Traffic signaling problem (described in Chapter 1) with various handy features, wrapped into the `traffic-signaling` Python package
- **Optimization and experiments** - scripts for running the optimization and experiments, which use the simulator

A.1 Prerequisites

When listing version requirements for the tools below, we specify the *minimum* supported version. As of July 2025, the latest available versions (e.g., Python 3.13) are also compatible. However, we cannot guarantee compatibility with all future versions.

To build and install the `traffic-signaling` package, you need:

- **C++ compiler** with C++20 code support; e.g., gcc 11, clang 16, or MSVC 19.29 (or later versions)
- **Python** 3.10 or later
- **Python headers** (most likely already installed with Python)
  - You can verify that the headers are available by inspecting their expected location with e.g.

    ```
    python3 -c "import sysconfig;
        print(sysconfig.get_path('include'))"
    ```
  - If not available, install the headers with e.g.

    ```
    sudo apt install python3-dev
    ```

To easily set up the environment for optimization and run the experiments, you further need:

- **GNU Make**

Optionally, if you want to run all unit tests for both C++ and Python, you additionally need:

- **Git**
- **CMake** 3.24 or later

A.2 Simulator

If you only want to run the optimization and do not want to use the simulator as a standalone tool, feel free to skip this section.

After satisfying the prerequisites, you can simply build and install the `traffic-signaling` Python package by running the following command in the top-level directory:

```
pip install ./traffic_signaling
```

This will install the package into your Python environment (preferably into a virtual environment), making it available for use in your Python scripts.

To run Python unit tests for the package using Python's `unittest` built-in framework, use the following command:

```
make test_package_python
```

To run both C++ and Python unit tests using CMake, use the following command:

```
make test_package_cmake
```

Note that running the make commands will create a virtual environment `.venv` and install all required packages there using pip.

A.2.1 Code example

The following code snippet demonstrates some of the package's functionality. For the full API reference, see the `traffic_signaling/traffic_signaling` subdirectory, which contains the files `city_plan.pyi`, `simulation.pyi`, and `utils.py`. These files include extensive docstrings for every class and method, effectively serving as the package documentation.

```python
from traffic_signaling import *

# Load the city plan for a specific dataset
plan = create_city_plan(data='e')
# Create a simulation for the given city plan
sim = Simulation(plan)

# Initialize the traffic light schedules
sim.create_schedules(order='default', times='default')
# Save the schedules to a file in the competition format
sim.save_schedules('schedules.txt')
# Load the schedules from a file
sim.load_schedules('schedules.txt')

# Calculate the score
score = sim.score()
# Show a summary report of the simulation
sim.summary()
```

A.3 Optimization

To quickly set up the environment for optimization, run the following command in the top-level directory:

```
make setup
```

This will create a virtual environment `.venv`, build and install the simulator and other necessary packages using pip into it, and compile `operators.py` file using Cython[1] for better performance during optimization. Do not forget to activate the virtual environment before running any scripts with e.g.

```
source .venv/bin/activate
```

If you encounter any issues, try running `make clean` and then `make setup` again.

Now you can run the optimization algorithms using the `optimizer.py` script. The script has two required positional arguments:

- `algorithm` - algorithm to use for optimization; possible values are `ga`, `hc`, `sa`
- `data` - input dataset to use; possible values are `a`, `b`, `c`, `d`, `e`, `f`

After specifying the required arguments, you can easily run the script with e.g.:

```
python3 optimizer.py hc e
```

This will run the Hill Climbing algorithm on dataset E using default values. However, you probably want to explicitly set some parameters, especially the hyperparameter values. If you prefer, you can run

```
python3 optimizer.py --help
```

to see the full usage. Below is a concise list of the parameters:

- `--order_init` - *order initialization* hyperparameter; possible values are `adaptive`, `random`, `default`
- `--times_init` - *times initialization* hyperparameter; possible values are `scaled`, `default`
- `--mutation_bit_rate` - *mutation bit rate* hyperparameter
- `--population` - *population size* hyperparameter (GA only)
- `--generations` - *generations* hyperparameter (GA only)
- `--crossover` - *crossover probability* hyperparameter (GA only)
- `--mutation` - *mutation probability* hyperparameter (GA only)
- `--elitism` - *elitism* hyperparameter (GA only)
- `--tournsize` - *tournament size* hyperparameter (GA only)

[1]`https://cython.org/`

- `--iterations` - *iterations* hyperparameter (HC and SA)
- `--temperature` - *initial temperature* hyperparameter (SA only)
- `--seed` - value of the random seed for reproducibility
- `--threads` - number of threads for parallel evaluation
- `--logdir` - custom name of the directory with results and logs
- `--verbose` - whether to print detailed output during optimization
- `--no-save` - skip saving results to the log directory

An example of running the script with more parameters could look like this:

```
python3 optimizer.py ga e \
    --order_init random --times_init scaled \
    --mutation_bit_rate 5 --population 100 --generations 200 \
    --threads 16 --seed 21 --verbose
```

When the optimization finishes (and if the `--no-save` option was not used), the optimizer will save the following files in the log directory:

- a CSV file containing statistics for each iteration / generation of the algorithm
- a file with the best schedules found, stored in the competition format
- a PDF file visualizing the optimization process
- an information file listing all parameters, their values, and additional details

Optionally, you can run unit tests for the optimizer with the following command:

```
make test_optimizer
```

A.3.1 Running the experiments

To replicate the experiments presented in this thesis, ensure that you have already run the `make setup` command, then navigate to the `experiments` directory. We recommend reading the `experiments/README.md` file for more details, but for convenience, we also include the list of commands below. We strongly suggest running each algorithm and dataset separately.

- `make test` - run a simple sanity check experiment
- `make init_experiment` - run a quick experiment comparing different initialization methods
- `make run_{b,c,d,e,f}_{ga,hc,sa}` - run a specific algorithm on a specific dataset (using 10 runs with different fixed seeds)
- `make run_{b,c,d,e,f}` - run all algorithms on a specific dataset

- `make plot_{b,c,d,e,f}` - plot the results of a specific dataset
- `make plots` - plot all datasets
- `make all` - run everything and plot all results

The schedules corresponding to the best scores achieved for each dataset during optimization are stored in the `experiments/best_solutions` directory.

B Developer documentation

B.1 Simulator

The source code for the simulator is located in the `traffic_signaling` directory. Below is an outline of the directory structure:

- `include/` - C++ header files of the simulator
  - `city_plan/` - headers of *city plan* part
  - `simulation/` - headers of *simulation* part
- `src/` - C++ source files of the simulator
  - `city_plan/` - source files of *city plan* part
  - `simulation/` - source files of *simulation* part
  - `bindings/` - python bindings for *city plan* and *simulation* parts
- `tests/` - Unit tests for both C++ and Python verifying the simulator functionality
- `traffic_signaling/` - Contents of the Python package when installed with pip
  - `utils.py` - provides extra utilities and helper functions for the simulator
  - `data/` - contains the datasets provided with the competition
- `pyproject.toml`, `setup.py` - Python configuration files for building and installing the `traffic-signaling` package using pip
- `CMakeLists.txt` - CMake configuration file for building the C++ code and the Python package using CMake

Both the C++ and Python sources are well documented with comments and docstrings to help you understand the code and its functionality.

The simulator is logically divided into two main parts:

- *city_plan* - responsible for loading and storing input data; it essentially represents all "static" data known ahead of the simulation; its main class is `CityPlan`
- *simulation* - responsible for running the simulation and working with the traffic light schedules; it represents the "dynamic" data that change during the simulation; its main class is `Simulation`

The motivation behind this separation is to initialize a single `CityPlan` object at the beginning and reuse it across multiple simulations, possibly running in parallel—which is especially useful for the genetic algorithm.

These two parts are compiled into two Python extension modules—`city_plan` and `simulation`—using pybind11[1]. They are then bundled together with the files

[1] `https://github.com/pybind/pybind11`

in the `traffic_signaling` subdirectory into the `traffic-signaling` Python package. For more details about the bindings, see the `bindings/simulation.cpp` and `bindings/city_plan.cpp` files.

Note that when calling C++ code that takes a non-trivial amount of time to run from Python, we release the Python Global Interpreter Lock (GIL)[2]. This allows us to evaluate multiple simulations in parallel using regular threads without blocking the Python interpreter. It eliminates the need for slower and cumbersome multiprocessing, which is the typical approach to run parallel code in Python.

B.1.1 Simulation algorithm

Not only is the simulator implemented in C++ for performance reasons, but it also uses a **custom event queue algorithm** to ensure the simulation is evaluated as efficiently as possible. Rather than checking all cars every second of the simulation, we use a priority queue of street events sorted by their time of occurrence.

Let us briefly explain how the algorithm works. As explained in Section 1.2.3, at the beginning of the simulation, all cars are at the end of the first street in their path, waiting for the green light. If there are more cars at the same street, they queue up according to their IDs.

We are at time 0. For each car, we calculate the *earliest time* it can get the green light on its current street and add it to the street's car queue. The calculated time depends on the *traffic light schedules* and *other cars already waiting* in the car queue. For the street, we add an event occurring at the calculated time to the event queue. The event indicates that the front car in the queue can now move to the next street.

Then, we iterate over the event queue until it is empty. We pop the first event from the queue and process it—that is, move the car to the next street and schedule another event at the time the car can get the green light on the next street. If the next street is the car's destination, we can immediately calculate the arrival time and remove the car from the simulation.

Using the event queue allows us to skip all unnecessary checks and reduce the operations to the required minimum, because we only process cars at the times when they are actually moving. The main loop of the algorithm is implemented in the `simulation/simulation.cpp` file in the `Simulation::run` method.

B.2 Optimization

The source code for optimization is in two files:

- `optimizer.py` - the main script providing the command-line interface for running the optimization algorithms
- `operators.py` - contains the implementation of the optimization algorithms and their operators

[2]`https://docs.python.org/3/glossary.html#term-global-interpreter-lock`

To implement the optimization algorithms, we used the DEAP[3] library, which provides a modular framework for evolutionary algorithms. However, we tweaked and rewrote some of its functions to suit our needs.

The `optimizer.py` file contains the `Optimizer` class, which is responsible for running the optimization (`run` method). It processes the command-line arguments (described in Section A.3), initializes the traffic light schedules that we optimize, and runs the optimization algorithm. After the optimization is completed, the results and logs are saved to the specified log directory.

The fitness evaluation for schedules is implemented in the `_evaluate` method. The schedules initialization is implemented in the `_create_individual` method. All statistics and logs are handled together by the `_save_statistics` method.

The `operators.py` file contains the tweaked DEAP functions together with other custom functions such as `crossover`, `mutation`,
and `tournament_selection_with_elitism`.
All three algorithms—`genetic_algorithm`, `hill_climbing`,
and `simulated_annealing` functions—are modified versions of the original `eaSimple` function from DEAP, which implements the simple genetic algorithm.

The file contains Cython type hints to enable compilation for faster performance during the optimization.

[3]`https://github.com/deap/deap`