Technical University of Košice
Faculty of Electrical Engineering and Informatics

Visualization of Optimization Algorithms

Bachelor thesis

2024 **Olha Shvenda**

Technical University of Košice
Faculty of Electrical Engineering and Informatics

Visualization of Optimization Algorithms

Bachelor thesis

Study Programme:	Intelligent systems
Field of Study:	Informatics
Department:	Department of Cybernetics and Artificial Intelligence (DCAI)
Supervisor:	Ing. Ján Magyar, PhD.

Košice 2024 **Olha Shvenda**

Abstract in English

This thesis presents a comprehensive study and implementation of optimization algorithms, with a focus on Hill Climbing and its variants. The objective of this work is to provide a detailed visualization of the operational principles underlying these algorithms. The focus of this thesis is on the solution of the Traveling Salesman Problem. We delve into the problem analysis, propose a solution, and detail the implementation of a web application that visualizes the workings of these algorithms. At the conclusion of the thesis, the algorithms were evaluated and compared. The results of this work include a detailed study of the algorithms, an interactive visualization of their operational principles, and a comparative evaluation of their performance. The benefits of this work lie in its potential to serve as an educational tool, enhancing the understanding of optimization algorithms and their application in solving complex problems.

Keywords in English

Hill Climbing, Hill Climbing with larger search radii, Hill Climbing with restarts, Optimization algorithms, Visualization

Abstract in Slovak

Táto práca predstavuje komplexné štúdium a implementáciu optimalizačných algoritmov so zameraním na Hill Climbing a jeho varianty. Cieľom tejto práce je poskytnúť podrobnú vizualizáciu princípov fungovania, na ktorých sú tieto algoritmy založené. Táto práca sa zameriava na riešenie problému obchodného cestujúceho (TSP). Venujeme sa analýze problému, navrhujeme riešenie a podrobne popisujeme implementáciu webovej aplikácie, ktorá vizualizuje fungovanie týchto algoritmov. V závere práce boli algoritmy vyhodnotené a porovnané. Výsledky tejto práce zahŕňajú podrobnú štúdiu algoritmov, interaktívnu vizualizáciu ich princípov fungovania a porovnávacie hodnotenie ich výkonnosti. Prínos tejto práce spočíva v jej potenciáli slúžiť ako výučbový nástroj, zlepšujúci pochopenie optimalizačných algoritmov a ich aplikáciu pri riešení zložitých problémov.

Keywords in Slovak

Hill Climbing, Hill Climbing s opakovanými reštartmi, Hill Climbing s väčšími okoliami, Optimalizačné algoritmy, Vizualizácia

Bibliographic Citation

SHVENDA, Olha. *Visualization of Optimization Algorithms*. Košice: Technical University of Košice, Faculty of Electrical Engineering and Informatics, 2024. 41s. Supervisor: Ing. Ján Magyar, PhD.

73196

TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Department of Cybernetics and Artificial Intelligence

BACHELOR THESIS ASSIGNMENT

Field of study: **Computer Science**

Study programme: **Intelligent Systems**

Thesis title:

Visualization of Optimization Algorithms

Vizualizácia optimalizačných algoritmov

Student: **Olha Shvenda**

Supervisor: **Ing. Ján Magyar, PhD.**

Supervising department: **Department of Cybernetics and Artificial Intelligence**

Consultant:

Consultant`s affiliation:

Thesis preparation instructions:

1. Present an overview of visualizing optimization algorithms
2. Present an overview of existing solutions in the application domain
3. Design and implement an educational tool for optimization algorithms
4. Test the implemented solution and evaluate the results
5. Write documentation according to the instructions of the thesis supervisor

Language of the thesis: English

Thesis submission deadline: 24.05.2024

Assigned on: 31.10.2023

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -11-

N.z. Pribilová

prof. Ing. Liberios Vokorokos, PhD.

Dean of the Faculty

Declaration

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged.

Košice, 24.5.2024 . . . . . .

Signature

Acknowledgement

At this point, I would like to thank my thesis advisor for his time and expert guidance during the course of my thesis.

I would also like to thank my friends for their support and encouragement throughout the writing of this thesis.

Contents

List of Figures

List of Tables

Introduction

In this thesis, we are conducting an exhaustive exploration and execution of optimization algorithms, specifically concentrating on Hill Climbing and its variations. Our goal is to create a comprehensive visualization of the fundamental principles that govern these algorithms, using the Traveling Salesman Problem as our main case study.

Our work begins with a thorough analysis of the problem, forming the foundation for our subsequent solution proposal. This proposed solution includes the development of a web application that illustrates the functioning of the optimization algorithms. The application is constructed using Python and the Django framework, integrating various libraries for data management and visualization.

Our implementation includes three variations of the Hill Climbing algorithm: the basic version, a version with restarts, and a version with larger search radii. Each variation is rigorously tested and evaluated.

The visualization aspect of our work is a major contribution, providing an interactive and intuitive means to understand the operation of the optimization algorithms. We believe this is a valuable educational resource, enhancing the understanding of these algorithms and their role in solving complex problems.

In terms of achieving our objectives, we aim to successfully execute and visualize the selected optimization algorithms, and our web application is designed to effectively illustrate their operations.

By the end of this thesis, we hope to have made a substantial contribution to the understanding and visualization of optimization algorithms. We eagerly anticipate the potential influence of our work on both the academic sphere and practical applications.

1 Task formulation

In this thesis, we are undertaking the following tasks:

Present an overview of visualizing optimizing algorithms: In 2.3, we delve into the concept of algorithm visualization, which uses visual representations to convey information about the structure and operation of data and algorithms. Our focus is on visualizing optimization algorithms, particularly Hill Climbing and its variants. We are using libraries such as matplotlib and NetworkX for creating interactive visualizations.

Present an overview of existing solutions in the application domain: We provide a comparative analysis of existing solutions that visualize optimization algorithms in 2.4. This includes a discussion on the use of different programming languages and UI technologies in these solutions.

Design and implement an educational tool for optimizing algorithms: We design and implement a web application that serves as an educational tool for understanding optimization algorithms. We describe the implementation part in 4. The application allows users to visualize the workings of the Hill Climbing algorithm and its variants, thereby enhancing their understanding of these algorithms.

Test the implemented solution and evaluate the results: We thoroughly test and evaluate our implemented solution in 5. This involves comparing the solutions of the three algorithms with the optimal solution, and analyzing the performance of each algorithm. The results of these tests provide insights into the efficiency and robustness of the algorithms.

Create documentation according to the instructions of the thesis supervisor: In addition to our primary work, we are also creating a comprehensive user manual as an integral part of our documentation. This manual serves as a guide for users to understand and navigate through the application effectively. It details the functionality of the application and provides step-by-step instructions on how to use the various features.

2 Problem analysis

In mathematics and computer science, optimization is a fundamental concept that seeks to find the best solution from a set of possible options. Optimization is often necessary when we need to make the best choice given a set of constraints.

This chapter comprehensively overviews optimization techniques, from theoretical foundations to practical implementations. The chapter begins with fundamental concepts such as problem formulation, single-objective and multi-objective optimization, and constraints. It also emphasizes the importance of objective functions in evaluating solution quality. In Section 2.2, we look at local search and its subset algorithms. We explain the process of finding a suboptimal solution using a visual example and the problem of getting stuck in a local optimum. Next, we look at various techniques for escaping the local optimum. Section 2.3 introduces and compares the most common tools for visualizing algorithms. At the end of this chapter, we analyze previous solutions for visualizing optimization algorithms.

2.1 Optimization

At its core, optimization is about finding the maximum or minimum of a function. In other words, we are looking for the point at which a particular function reaches its highest or lowest value. The objective of optimization in real-life scenarios can vary widely. In the case of maximization, it can be maximizing profit in financial investments, increasing output in manufacturing processes, improving performance in sports training, and increasing efficiency in logistics operations. The opposite objective may be to minimize costs in supply chain management, reduce energy consumption in building design, and decrease overall weight in aerospace engineering [1].

Figure 2.1 illustrates the most fundamental optimization procedure. Initially, we are given a physical problem that needs to be optimized. It involves inputs (relevant data) and certain resources (materials, human capital, etc.). Then we

translate the problem into a mathematical model. After that, objective functions and constraints help to create a mathematical formulation. Using optimization techniques, we derive potential solutions based on mathematical formulations. The solutions are then evaluated against criteria (efficiency, profitability, etc.). This process ends with selecting the optimal solution that maximizes benefits while meeting constraints.

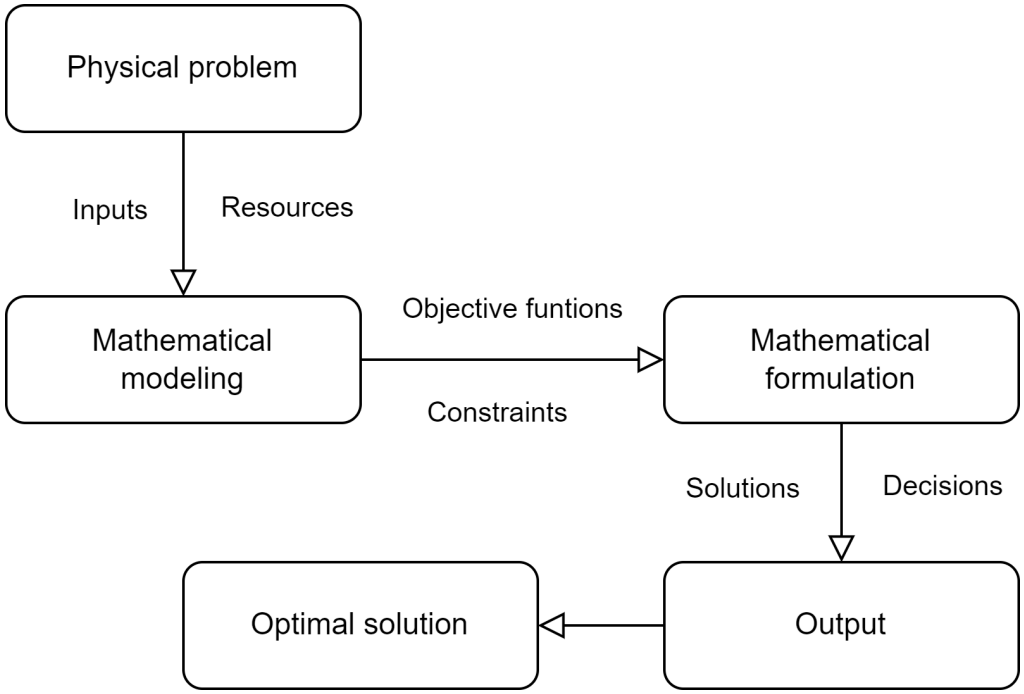

Figure 2.1: Flow chart of modeling physical problem to get an optimal solution[2]

A real-life example of optimization can be seen in logistics. Delivery companies often use optimization algorithms to determine the most efficient routes for their delivery trucks. These companies can save on fuel costs and reduce delivery times by minimizing the total distance traveled. This is a complex problem as it involves multiple variables such as the number and location of deliveries, the capacity of the truck, and the time window for each delivery. Despite its complexity, optimization techniques can solve this problem efficiently, providing significant benefits to the company. The example of the goods delivery described above demonstrates the power of optimization in many fields. For example, in civil engineering, it helps design structures like bridges and towers while keeping costs low. Aerospace engineers use it to design lightweight aircraft for better performance. In mechanical engineering, it is used to make gears and machines work more efficiently. Water distribution systems benefit from optimization by ensuring water is delivered where it is needed most. In various industries, it helps

to schedule tasks and allocate resources effectively [2]. In [3], Xin-She Yang describes an extended mathematical form of optimization problems:

$$\text{minimize} \quad f_i(\mathbf{x}), \quad (i = 1, 2, ..., M), \quad \mathbf{x} \in \mathbb{R}^D \tag{2.1}$$

subject to

$$h_j(\mathbf{x}) = 0, \quad (j = 1, 2, ..., J), \tag{2.2}$$

$$g_k(\mathbf{x}) \leq 0, \quad (k = 1, 2, ..., K), \tag{2.3}$$

where $f_i(\mathbf{x}), h_j(\mathbf{x})$ and $g_k(\mathbf{x})$ are functions of the design vector

$$\mathbf{x} = (x_1, x_2, ..., x_D)^T. \tag{2.4}$$

Functions $f_i(\mathbf{x})$ are crucial. We call them *objective functions* or *cost functions*. When $M = 1$, we are dealing with a *single objective*. If $M > 1$, we have a *multiobjective* or a *multicriteria problem*. The objective functions can be *linear* or *nonlinear*. These functions quantify what we want to maximize or minimize. For instance, considering the earlier example of optimization in logistics, the objective function might be to minimize the total distance traveled by all trucks. It is important to note that multiple functions can exist within a single problem.

Next, we have $h_j(\mathbf{x})$ and $g_k(\mathbf{x})$, which are *equality* and *inequality constraints*, respectively. Equality constraints ensure that certain relationships hold true. This could refer to the number of trucks required to deliver batches of goods. Conversely, inequality constraints set boundaries. They might limit resource usage, time, or other factors. The problem may have no constraints. In this case, we call it an *unconstrained optimization problem*. A problem with only one type of constraint is classified as either an *equality-constrained problem* or an *inequality-constrained problem*.

Components x_i within vector $\mathbf{x}$ serve as our *design* or *decision variables*. In practice, these can be the routes, the amount of cargo each truck can take, and the window times for each delivery. The *design space* or *search space*, denoted as $\mathbb{R}^D$, encompasses all possible combinations of our decision variables. On the other hand, the *solution space* or *response space* captures the outcomes of our objective functions.

Optimization problems can be broadly classified into two categories: continuous optimization problems and combinatorial optimization problems (COPs).

COPs involve continuous decision variables and can yield infinite potential solutions. On the other hand, COPs, also called discrete optimization problems, have discrete decision variables and a finite number of solutions [4].

Traveling salesman problem

The Travelling Salesman Problem (TSP) is a classic example of a COP. It has numerous real-world applications, including transportation, robot routing, biology, and circuit design [5]. The TSP is considered NP-hard, making it extremely challenging from a theoretical computer science perspective. The TSP involves a set of cities and their distances. The objective is to find the shortest possible tour that visits all the cities once and returns to the starting point [6].

In this paper, we solve a symmetric TSP, i.e. the distance between any two cities from opposite directions is identical. A complete undirected graph $G = (V, E)$ has a set of nodes $V = \{1, 2, ..., n\}$ representing the cities. Each edge $\{i, j\} \in E$ has a cost d_{ij} ($d_{ij} = d_{ji}$ for symmetric TSP).The mathematical model of the symmetric TSP is in Equations 2.5 - 2.8.

For weighted graphs $G = (V, E)$, objective function is

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij} \tag{2.5}$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1, i \in V \tag{2.6}$$

$$\sum_{i=1}^{n} x_{ij} = 1, j \in V \tag{2.7}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \quad \forall S \subset V,\ 2 \leq |S| \leq n - 1,\ x_{ij} = \{0, 1\},\ i, j \in E \tag{2.8}$$

Here, Equations 2.6 and 2.7 guarantee that each city is visited only once. The constraint given in Equation 2.8 guarantees that no subloop solutions will be generated [7].

2.2 Optimization algorithms

This section examines the advanced concepts of Local search (LS) algorithms, including the neighborhood relation and evaluation function. It presents practical examples of their application and discusses the challenges posed by local optima, outlining strategies to overcome these, such as restarts and larger search radii.

2.2.1 Iterative improvement

LS algorithms are a broad category of optimization techniques that iteratively explore neighboring solutions to improve the current one. This exploration typically focuses on a local region of the solution space, rather than exhaustively searching the entire space. The primary goal is to find a solution that optimizes the objective function within that local neighborhood.

The algorithm includes two new concepts: neighborhood relation $\mathcal{N}$ and an evaluation function $\mathfrak{f}$. "$\mathcal{N}$ is commonly described by the possible outcomes of a particular transformation which links each solution to a set of neighbors with a similar structure. The evaluation function associates a scalar value to each solution x and implies a preference relation between couples of solutions" [8].

A clear illustration of an LS was presented in [9] using an example of an optimization problem where we have to minimize the value of the evaluation function $\mathfrak{f}$. Figure 2.2 shows us the fitness landscape for this problem. The fitness landscape can be described as a triplet (search space, neighborhood relation, evaluation function).

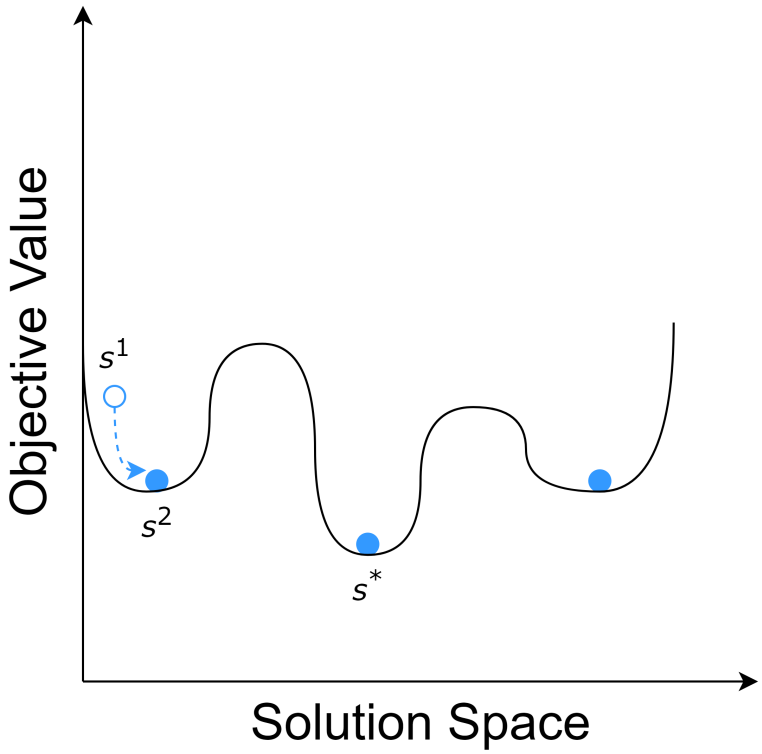

Figure 2.2: A local search algorithm [9]

Selecting s^1 as our starting point, LS proceeds to identify the local optimum s^2 situated at the initial valley within the solution space, given a sufficient number

of iterations. This characteristic enables LS to find a solution superior to s^1 in the neighborhood relation $\mathcal{N}$. This process will be repeated until it cannot identify a superior solution from the neighbors, resulting in a local optimum, as exemplified by s^2. This example illustrates that although LS may rapidly converge on a local optimum, it can still approximate the optimal solution to the optimization problem if a search algorithm can be integrated with other global search algorithms. Algorithm 1 is a generalized algorithm for LS.

Algorithm 1 Local Search

1: Randomly create the initial solution s
2: $f_s = Evaluate(s)$
3: **while** the termination criterion is not met **do**
4: $\quad v = DeterministicNeighborSelection(s)$
5: $\quad f_v = Evaluate(v)$
6: $\quad s, f_s = Improve(s, v, f_s, f_v)$
7: **end while**
8: **Output** s

Iterative improvement algorithms, specifically Hill Climbing (HC) or *climber* (*descent*, for minimization problems) algorithms, can be seen as a subset of LS. The search strategy of HC is to accept only the solution that improves the evaluation function ƒ. Consequently, if local optimum exists between the initial and optimal solutions, HC risks becoming trapped at a local optimum.

Figure 2.3 shows an extended version of the previous minimization problem. The example of this solution space shows that the HC can fall into 3 local optima. It is worth noting that we are discussing a version of HC that selects only the best neighboring solution, i.e. $f(v) > f(x)$. Therefore, if our initial state is in s^3, we have a 50/50 chance of moving to s^2 or s^4, and in either case, the algorithm will end up in a local optimum. If the initial state is s^6, the algorithm finds no better neighbor and stays in s^6.

The rule for comparing the current solution x with neighboring solutions v in $\mathcal{N}$ can be modified to select the best neighboring solution and a solution that is as good as the current one, i.e., $f(v) \geq f(x)$. In some cases, such a change can help to avoid getting into a local optimum.

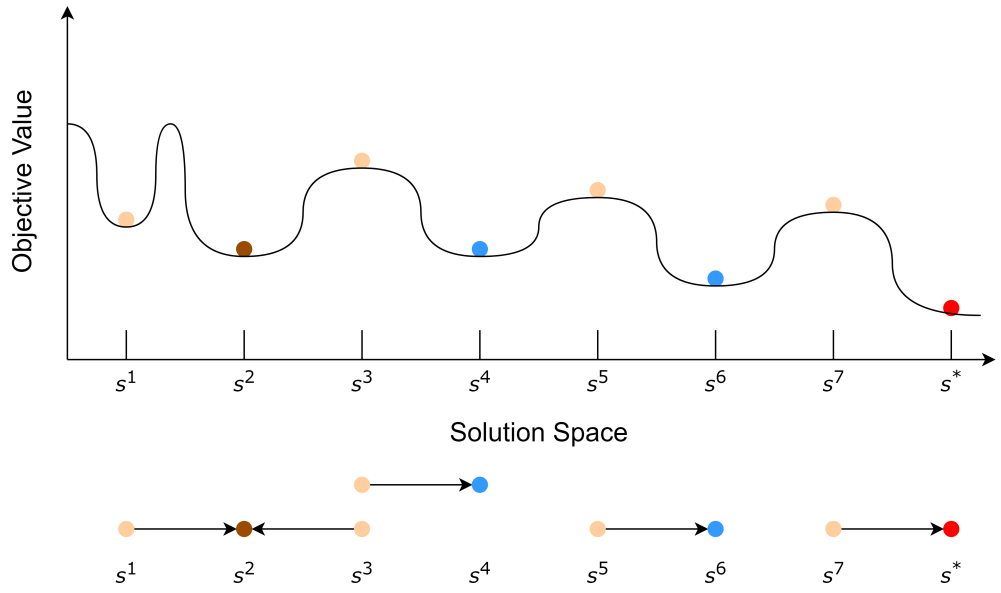

Figure 2.3: The search strategy of Hill Climbing

Algorithm 2 illustrates the HC algorithm. Compared to Algorithm 1, the modifications can be identified in 4th and 6th lines. LS employs deterministic neighbor selection and has multiple mechanisms to improve the solution, whereas HC consistently selects the better solution.

Algorithm 2 Hill Climbing

hbo

1: Randomly create the initial solution s
2: $f_s = Evaluate(s)$
3: **while** the termination criterion is not met **do**
4: $v = NeighborSelection(s)$
5: $f_v = Evaluate(v)$
6: **if** f_v is better than f_s **then**
7: $s = v$
8: $f_s = f_v$
9: **end if**
10: **end while**
11: **Output** s

2.2.2 Escaping local optima

One of the main challenges in LS and HC algorithms is the tendency to get trapped in local optima. As we have seen, these algorithms can rapidly converge to a so-

lution that is the best within a local neighborhood, but not necessarily the best overall solution. Various strategies have been developed to overcome this limitation, to help these algorithms escape from local optima and explore more of the solution space.

- Restarts:

  Restart strategies involve periodically resetting the search process to explore different regions of the solution space. When the algorithm converges to a local optimum, a restart is triggered, and the search begins again from a new starting point. By introducing randomness through restarts, the algorithm can explore alternative paths in the solution space, increasing the likelihood of finding a better solution. However, the effectiveness of restarts heavily depends on the distribution of local optima in the solution space and the proximity of the global optimum to the initial points.

- Larger Search Radii:

  Larger search radii strategy considers solutions that are not direct neighbors of the current solution [10]. This can be achieved by modifying the neighborhood relation $\mathcal{N}$ to include solutions further away from the current solution. Nonetheless, increasing the search radius also increases the algorithm's computational complexity, as more solutions need to be evaluated at each step.

- Changing Neighborhoods:

  The concept of escaping local optima by changing neighborhoods is rooted in a straightforward principle: the systematic alteration of neighborhood structures during the search process. The Variable Neighborhood Search methodology proposes three exploration strategies: random exploration with randomly chosen neighborhoods, deterministic exploration following a specific sequence or pattern, and mixed exploration combining both random and deterministic approaches. A specific variant of Variable Neighborhood Search, known as Variable Neighborhood Descent, employs a deterministic exploration strategy. Variable Neighborhood Descent typically begins by investigating smaller neighborhoods. Once a local optimum is found within a given neighborhood, the search process transitions to a different, usually larger, neighborhood that allows further progress [11].

2.3 Visualization of optimization algorithms

Algorithm visualization (AV) uses visual representations, commonly in the form of graphs or diagrams, to convey information about the structure and operation of data and algorithms [12]. In the field of education, AV can be employed in a variety of ways. For instance, a visualization may illustrate a specific example or be the foundation for an interactive learning activity. The most common AV techniques include the following:

- Static Visualization: This is the simplest form of AV, yet it is frequently employed. It involves using diagrams or other static images to represent the state of an algorithm at different stages. Figure 2.3 explains the HC algorithm using this technique. In that example, we used a graph to show the fitness landscape and a local optima.

- Animated Visualization: This technique involves creating an animation to show the algorithm in action. This can make it easier to understand how the algorithm works over time. In the context of the HC algorithm, an animated visualization could show the algorithm moving from one state to another in the search space, visually representing how the algorithm selects solutions.

- Interactive Visualization: This is a more advanced form of visualization that allows the user to control the input to the algorithm and see how it affects the output. For the HC algorithm, an interactive visualization might allow the user to change the starting point of the algorithm or enable them to adjust parameters relevant to the algorithm. It can be a powerful tool for understanding how different inputs can affect the behavior of an algorithm. However, previous research in this area has produced mixed results. Some studies have found that interrupting the animation process can have negative effects, while others have emphasized the benefits of providing user control [13, 14].

The next step is data representation. This is a critical aspect of AV. It involves choosing the most effective way to represent the data the algorithm is working with visually. This could be in the form of a pie chart, line chart, bar chart, area chart, graph, map, heat map, etc. The choice of data representation can significantly impact how well the algorithm's operation is understood. AV involves choosing the type of application for visualization, the programming language for logic implementation, tools for data representation, and creating a User interface

(UI). Table 2.1 presents a comparative analysis of the three most common types of applications: software applications, web applications, and mobile applications.

Table 2.1: Comparison of application types

Property	Software app	Web app	Mobile app
Platform-independent	No	Yes	Yes
Complexity	High	Moderate	Moderate
Performance	High	Moderate	Moderate
Scalability	Limited	High	High
Internet Access	Not required	Required	Required

The choice of programming language for creating software or a web application for AV depends largely on the programmer's skills and the specific problem. However, knowing the differences between programming languages is still important to make a more effective choice.

Reference [15] analyzed and compared the most popular programming languages. The following is a summary of the main advantages and disadvantages listed in it.

Python's code is often more concise than other languages, including C, Java, C++, Visual Basic, and .NET. Its syntax is designed for readability, allowing programmers to express concepts in fewer lines of code. Python is portable and can run on various operating systems, including Windows, Linux, Amigo, macOS, and UNIX. Furthermore, it is compatible with other technologies like COM and .NET. Nevertheless, because the computer has to do a lot of referencing to understand the definitions, Python's flexibility can lead to slower performance.

C# is a frequently used language for developing sophisticated, dynamic websites on the .NET platform or in open-source technologies. Additionally, it is commonly used for Windows applications, given its development by Microsoft and the Windows .NET framework. C#'s syntax is easy to read, write, and understand, making it beginner-friendly. It integrates easily with visual programming. However, it has less flexibility than other languages and relies heavily on Microsoft's .NET framework. Its load times can be slow, and compilation can be resource-intensive.

Java is a dynamic language designed to handle a significant amount of runtime. It is platform-independent, supports multiple inheritance through interfaces, and its code is simple, safe, and reliable. It is often observed that Java programs run faster than Python programs.

MATLAB's high-level language and interactive environment allow us to perform computationally intensive tasks quickly and easily. This capability is enhanced by MATLAB's extensive graphics capabilities, which provide a robust platform for data visualization and exploration. In addition to these features, MATLAB has a large library of mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, and numerical integration [16]. However, it's important to note that MATLAB might exhibit slower performance for certain types of computations compared to Python and Java.

JavaScript is a high-level, just-in-time compiled language. JavaScript is used to make web pages interactive, providing structure and animation to pages. Additionally, it can be used for server-side programming and helps pages connect to the server side [17].

2.4 Overview of existing solutions

Table 2.2 presents a comparative analysis of the works of students from previous years who also solved the problem of visualizing optimization algorithms [18, 19, 20, 21, 22, 23]. Regarding application type, five out of six papers (83%) use software instead of web applications.

- Python 3 is used for logic implementation in two of the six papers (33%). The UI in these papers is implemented using CustomTkinter[1] or MATLAB.
- MATLAB is employed in two of the six papers (33%) for both logic and UI implementations. One of these papers uses MATLAB App Designer[2] for the UI.
- Other works use Java, C#, or JavaScript for logic implementation, paired with their respective UI technologies: JavaFX[3] for Java, Win UI 3[4] for C#, and React[5] for JavaScript.

The program developed within [18] visualizes the three algorithms: Particle Swarm Optimization, Bat Algorithm, and Cuckoo Search. These algorithms are used to solve the symmetric 2-dimensional version of the TSP.

[1] https://customtkinter.tomschimansky.com/
[2] https://www.mathworks.com/products/matlab/app-designer.html
[3] https://openjfx.io/
[4] https://learn.microsoft.com/en-us/windows/apps/winui/winui3/
[5] https://react.dev/

Table 2.2: Types of technologies used for logic and UI development

REF	Application type	Logic implementation	UI implementation
[18]	Software application	Python 3	CustomTkinter
[19]	Software application	Python 3	MATLAB
[20]	Software application	MATLAB	MATLAB App Designer
[21]	Software application	Java	JavaFX
[22]	Software application	C#	Win UI 3
[23]	Web application	JavaScript	React

The program was written in Python 3, and the libraries customtkinter and Matplotlib[6] were used to create the Graphical user interface (GUI). The CustomTkinter library is a visually enhanced version of Tkinter[7], the standard library for creating GUIs in Python. The Matplotlib library was employed to generate graphical representations of the algorithms' progress.

In [19], the Firefly Algorithm was employed to address three problems. These include the TSP, the Knapsack Problem (KNP), and the Partition Problem. A non-oriented complete graph served as the AV for the TSP, while a dot plot was used for the KNP and the Partition Problem. Python was the language of choice for implementing the logic, with Matlab creating the Graphical user interface (GUI).

Reference [20] visualizes an algorithm inspired by the behavior of crows, known as Crow Search Optimization. This algorithm solves three problems: the TSP, the KNP, and the N-Queen Problem. The TSP is represented as a graph with coordinates representing cities. The KNP is depicted with a bar chart, and in the N-Queen Problem, a chessboard of a given size is constructed. The program was written in MATLAB, and the GUI was created using the MATLAB App Designer development environment.

In [21], the Genetic Algorithm, Artificial Bee Colony Algorithm, and Ant Colony System are employed to address the Vehicle Routing Problem, KNP, and Weapon Target Assignment Problem. Visualizing the Vehicle Routing Problem involves plotting all locations on a dedicated space. The AV for the KNP creates a rectangle representing a backpack with different items. The Weapon Target Assignment Problem is visualized in the form of a bipartite graph. The Java programming language was used to implement the logic. The AV was implemented using the

[6] https://matplotlib.org/

[7] https://docs.python.org/3/library/tkinter.html

JavaFX library.

The implementation of Black Widow Optimization, Rat Swarm Optimization, and Butterfly Optimization algorithms for the resolution of the given functions is presented in [22]. The AV is presented in the form of a graph, with the x-axis representing generation and the y-axis representing fitness. The three algorithms were implemented in C#, and WinUI 3 was employed for the UI.

In [23], the Genetic Algorithm and Grey Wolf Optimizer were implemented. The TSP, Vehicle Routing Problem, and N-Queen Problem were solved. The TSP visualization used icons to represent cities, connected by lines to form a tour. For the Vehicle Routing Problem, different colors distinguished customers and depots, with lines indicating routes. The N-Queen Problem visualization presented a chessboard with checkerboard icons. The implementation of both the logic and UI utilized JavaScript and React, respectively. Several issues have been identified in existing works that can hinder the effectiveness of these visualizations.

- Stability and robustness issues: The application crashes under certain conditions, such as when the input exceeds a limit or when an incorrect data type is entered. This not only disrupts the user experience but also hinders the learning process.

- UI design flaws: Several UI design flaws have been identified in existing works. These include the use of buttons or manual input instead of sliders, which can make the interface less intuitive and harder to control. Additionally, sliders often do not indicate the exact number (such as the number of iterations or parameter values) that will be used, leading to ambiguity and potential misuse.

- Lack of progress indicators: Another common issue is the lack of information about the progress of algorithms, such as the number of iterations. This can make it difficult for users to track the algorithm's progress and understand how changes in parameters affect the outcome.

- Absence of essential navigation options: Finally, many applications lack essential navigation options, such as a back button. This can make the interface less user-friendly, particularly for users who are accustomed to these features in other applications.

3 Proposal for a solution

This chapter presents a comprehensive proposal for the development of an AV tool. It begins with a general overview of the proposal. This includes a discussion of the programming environment choice. The libraries chosen for our implementation, such as matplotlib[1], NetworkX[2], SciPy[3], and jQuery[4], are also discussed in this chapter. Furthermore, it explains the selection of the algorithm and the problem. The next section outlines the system requirements, both functional and non-functional, to set clear expectations for what our system should do and how it should perform. This includes the specific functionality our system should provide and the performance and usability aspects of our system. Finally, the chapter concludes with a discussion of UI design.

3.1 General proposal information

This section presents an overview of the key decisions and selections made during the development of our AV tool. These include the programming environment, libraries, and the specific algorithm and problem chosen for visualization.

3.1.1 Environment selection

When selecting a programming language, we were guided by our experience, a previous comparison of general characteristics in 2.3, and our problem. We chose Python for logic implementation. The readability and conciseness of Python are advantageous, even when expressing simple optimization algorithms such as HC. We opted for a web application over a software application because it offers platform independence and can be accessed online. This aligns with our goal of making our visualization tool widely accessible.

[1] https://matplotlib.org/
[2] https://networkx.org/
[3] https://scipy.org/
[4] https://jquery.com/

Django[5], a Python web framework, was selected to implement our web application. It supports rapid development and provides a clean design, which is beneficial for maintaining code quality as our project evolves.

For crafting the UI, we chose HyperText Markup Language (HTML), JavaScript (JS), and Cascading Style Sheets (CSS) because they are standard in web development. They allow us to create an interactive AV and user-friendly interface, which is crucial for effectively visualizing optimization algorithms.

3.1.2 Libraries

Our implementation will use matplotlib, NetworkX, and SciPy due to their specific strengths, which align with our project's requirements. We will use matplotlib for its comprehensive capabilities to create interactive visualizations in Python. It's handy in our project for plotting the steps of the optimization algorithms and visualizing the results. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. In our project, we can use it to generate complete graphs and draw them with specific properties. Additionally, the SciPy library can be employed for scientific and technical computing.

To make it easier to create a web page, we decided to use jQuery. The library offers a simplified approach to various tasks related to DOM[6] manipulation, event handling, and element selection.

3.1.3 Algorithm and problem selection

We decided to visualize the HC algorithm first due to its simplicity and effectiveness as an iterative improvement algorithm. It provides a solid foundation for understanding the basic principles of LS, making it an excellent starting point. Next, we chose two techniques described in 2.2.2 to escape the local optima. HC with restarts will demonstrate how introducing randomness can help overcome the limitation of getting trapped in local optima. Following that, we can visualize HC with larger search radii to show how considering solutions that are not direct neighbors of the current solution can further enhance the search process.

As for the problem selection, we chose the TSP because it's a classic COP. Most existing works cannot solve extremely large-scale TSP instances within a limited time [24]. We will demonstrate that in most cases, a satisfactory solution

[5] `https://www.djangoproject.com/`
[6] Document Object Model

is sufficient.

3.2 System requirements

This section outlines the general requirements for our system. It's critical because it sets expectations for what our platform should do and how it should perform. Clearly defining our system requirements ensures that it meets its users' needs and achieves its intended purpose.

3.2.1 System functional requirements

This part describes the specific functionalities that our system should provide. This includes the tasks that the system should be able to perform and the user interactions it should support. Functional requirements assist us in defining the scope of our platform and provide a clear roadmap for its development.

The system will be able to:

- Simulate the solution of the TSP.
- Visualize the progress of three selected algorithms.
- Display the progress time.
- Handle errors and empty parameters.
- Display relevant information in each step of the algorithm (current tour, current distance, and current iteration number).

The user will be able to:

- Choose between algorithms.
- Modify the parameters of the problem.
- Upload a CSV[7] file of a map of the TSP to the program.
- Navigate through the visualization in both forward and backward directions.
- Restart the visualization.

[7]Comma-separated values

3.2.2 System non-functional requirements

Finally, this subsection is devoted to our system's performance and usability aspects, including reliability, efficiency, maintainability, usability, and scalability.

- The system will be clear and intuitive to use.
- The application should provide easy control and parameter input.
- The logical part of the application must support scalability.
- To make it more accessible, the system will be written in English.

3.3 User Interface design

Prior to the implementation of the UI, a low-fidelity prototype was created to better understand the layout of the web application. To simplify this process, the online platform Eraser[8] was used. Figure 3.1 presents a comprehensive representation of the entire webpage, which has been divided into several sections.

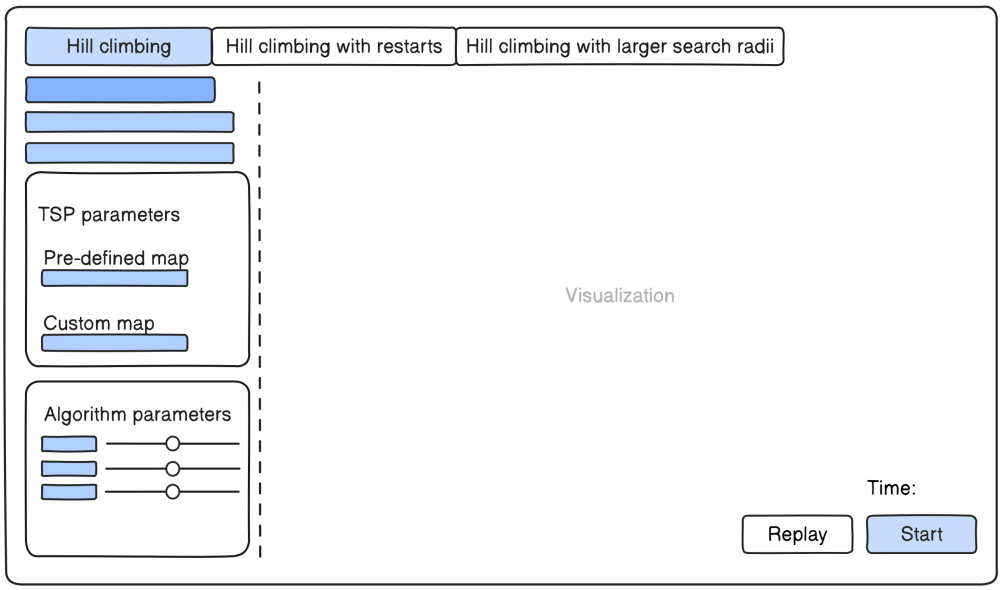

Figure 3.1: Low-fidelity prototype of user interface

Algorithm Selection Area: At the top are three tabs labeled "Hill climbing", "Hill climbing with restarts", and "Hill climbing with larger search radii". These tabs will allow users to select different variations of the HC algorithm to visualize.

[8] https://www.eraser.io/?r=0

Parameter Input Section: Below the algorithm selection area are two subsections titled "TSP parameters" and "Algorithm parameters". The "TSP parameters" menu includes options for a "Pre-defined map" and a "Custom map", allowing users to choose or upload specific maps for the TSP that the algorithm will solve. The "Algorithm parameters" menu provides sliders or input fields where users can adjust various settings for the selected HC algorithm.

Visualization Area: To the right side of these menus is a large space reserved for AV. This is where the graphical representation of the algorithm's execution will be displayed, providing visual feedback on its performance in solving TSP on either pre-defined or custom maps.

Control Panel: In the bottom right corner, there is a control panel with two buttons labeled "Replay" and "Start". These controls will allow users to initiate the visualization of an algorithm run and replay it if needed to analyze it further. In addition, buttons can be created to control the algorithm step by step.

4 Implementation

This chapter describes the practical aspects of our application, including the implementation of the algorithms and the modifications that were made during the development of our platform. The chapter begins with the implementation of the algorithms that we have chosen and explains their main stages. This is followed by the "Visualization implementation" section. This section focuses on how we visually represent the optimization problem and the algorithms used to solve it. After that, we move on to an equally important part - UI. We then proceed to discuss the changes that have been made to make our platform more user-friendly and intuitive.

4.1 Algorithm implementation

The choice of language and environment has not been changed. We implemented the logic in Python and used the Django framework to build our system. The following identical parameters and similar steps can be generalized in the implementation of all three algorithms for TSP.

Algorithm parameters:

`csv_file` - this parameter is used to select the map for the TSP. It allows the user to specify a CSV file containing the cities' positions. We use files with X and Y coordinates for each city. A valid entry must have two columns and at least five rows. If the user selects the "Pre-defined map" option, then we use pre-made maps with the required number of cities.

`num_runs`- this parameter specifies the number of iterations to be performed while searching for a solution. It is not a required parameter and is only used in HC with restarts algorithm. It's set to 1 by default.

Steps of the algorithm:

Initialization: In the `__init__` specified in listing 4.1 we initialized the number of cities, the CSV file, and the positions and distance matrix of the cities. We also set a random seed based on the current time.

Generation of distance matrix - in the `generate_distance_matrix` method, we checked if a CSV file was provided and if it existed. If so, we read the data from the file to get the cities' positions and computed the distance matrix.

Listing 4.1: Class definition for HC

```
class HillClimbingTSP:
        def __init__(self, num_runs=1, csv_file=None):
        self.num_runs = num_runs
        self.csv_file = csv_file
        self.positions, self.distance_matrix = self.
           generate_distance_matrix()
        self.random_seed = int(time.time())
        random.seed(self.random_seed)
```

Calculation of total distance - the `total_distance` method calculates the total distance of a given tour. It sums up the distances between each pair of consecutive cities in the tour and adds the distance from the last city back to the first city.

Generation of neighbors - the `get_neighbors` method generates all neighboring tours of the current tour by swapping each pair of cities. It then checks if the total distance of the new tour is less than the current distance. If this is the case, the new tour is added to the list of neighbors.

4.1.1 Hill Climbing

Figure 4.1 illustrates the flowchart of the HC algorithm. The algorithm begins by initializing the number of cities and the CSV file. It checks if a CSV file was provided and if it exists. If so, the algorithm reads the data from the file. The algorithm then generates a random initial tour of cities in the `run` function. This is accomplished by creating a list of city indices and shuffling them to obtain a random order. The algorithm enters a loop, continuously generating neighbors of the current tour. A neighbor is generated by swapping each pair of adjacent cities in the tour. For each neighbor, the algorithm calculates its total distance and compares it to the total distance of the current tour. If the neighbor's distance is less than the current distance, it is considered an improvement, and the neighbor is added to the list of potential new tours.

If there are any neighbors that are an improvement, the algorithm randomly chooses one of them as the new tour. The algorithm repeats the process of getting neighbors and checking for improvement until no improvement can be found. At this point, the algorithm terminates, and the current tour and its total distance

are returned as the best tour and best distance.

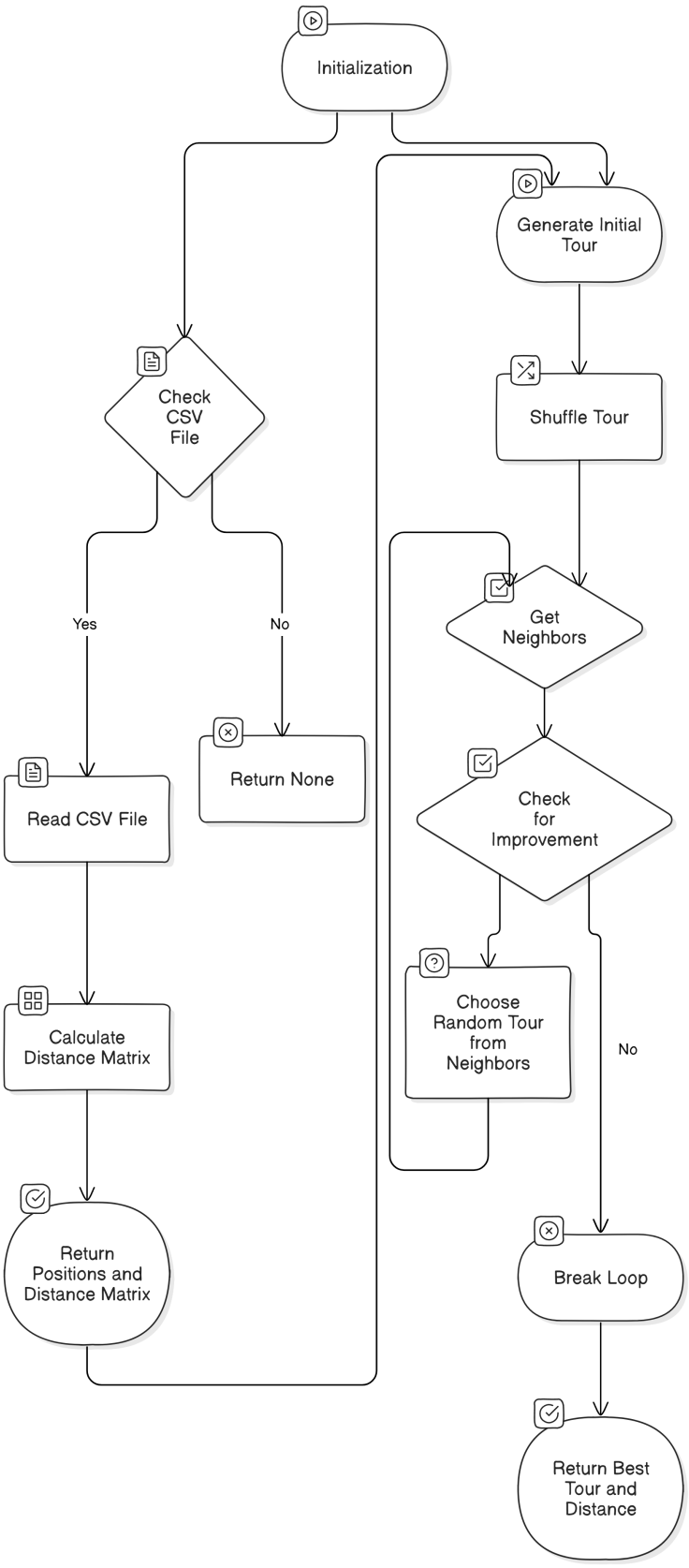

Figure 4.1: Flowchart of the Hill Climbing algorithm

4.1.2 Hill Climbing with restarts

The primary difference between the basic HC algorithm and HC with restarts lies in the `run` function shown in listing 4.2. In the basic HC algorithm, the `run` function generates a single random initial tour and continuously improves it until no better neighbors can be found. On the other hand, in HC with restarts, the `run` function includes an additional loop that runs for a specified number of times. In each run, a new random initial tour is generated. The algorithm tries to improve it as in the basic HC. However, instead of terminating after the first run, the algorithm compares the best tour and distance found in the current run with the best ones found in previous runs. If the current run yields a better tour, the algorithm updates the best tour and distance.

Listing 4.2: Run function in HC with restarts

```
def run(self):
        best_tour = None
        best_distance = float('inf')
        for _ in range(self.num_runs):
        tour = list(range(self.num_cities))
        random.shuffle(tour)
        tour.append(tour[0])

        while True:
                neighbors = self.get_neighbors(tour)
                if not neighbors:
                        break
                tour = random.choice(neighbors)
                tour_distance = self.total_distance(tour)
                if tour_distance < best_distance:
                        best_tour = tour
                        best_distance = tour_distance

        return best_tour, best_distance
```

4.1.3 Hill Climbing with larger search radii

The `get_neighbors` function demonstrated in listing 4.3 is the main difference between the basic HC algorithm and HC with larger search radii. In the basic HC algorithm, the `get_neighbors` function generates neighbors of the current tour by

swapping each pair of adjacent cities. This means that the algorithm only considers tours that can be obtained by swapping two cities that are next to each other in the tour. This is a local search strategy, as it only explores the immediate neighborhood of the current solution.

Conversely, in HC with larger search radii, the `get_neighbors` function generates neighbors by swapping any two cities in the tour, not just adjacent ones. This modification allows the algorithm to explore a larger portion of the solution space.

Listing 4.3: `get_neighbors` function in HC with larger search radii

```
def get_neighbors(self, tour):
        neighbors = []
        current_distance = self.total_distance(tour)

        for i in range(1, len(tour) - 2):
                for j in range(i + 1, len(tour) - 1):
                        new_tour = tour[:]
                        new_tour[i], new_tour[j] = new_tour[j],
                           new_tour[i]
                        new_distance = self.total_distance(
                           new_tour)
                        if new_distance < current_distance:
                                neighbors.append(new_tour)

        return neighbors
```

4.2 Visualization implementation

In the process of implementing our solution, we utilized all libraries that were mentioned in 3.1.2 to achieve our objectives. Furthermore, additional libraries were used.

We imported the random[1] library to generate random numbers, which was crucial for creating a random initial tour of cities in the run method of our `HillClimbingTSP` class. Additionally, we used the time[2] library to generate a random seed based on the current time.

The NumPy[3] library was used for its powerful N-dimensional array object,

[1] `https://docs.python.org/3/library/random.html`
[2] `https://docs.python.org/3/library/time.html`
[3] `https://numpy.org/`

with which we created the positions of the cities in the method `generate_distance_matrix`. If a CSV file was provided and existed, we used the pandas[4] library to read the data from the CSV file.

We used the NetworkX library to generate a complete graph of cities in the `generate_complete_graph` method. The matplotlib library was used extensively for visualizing the progress of the HC algorithm in the `plot_graph_step` method. It allowed us to create interactive visualizations, plot the steps of the optimization algorithms, and visualize the results.

The SciPy library was utilized in the `generate_distance_matrix` method for its `pdist` and `squareform` functions. These functions were used to calculate the Euclidean distance between cities and to convert a vector-form distance vector to a square-form distance matrix, respectively.

4.2.1 Visualization of the optimization problem

As mentioned in 2.1, we solve a symmetric TSP. We have chosen to visualize it as a complete undirected graph. We use a graphical representation of the problem for this purpose. As an illustration, Figure 4.2 presents a graph with weights marked on the selected tour.

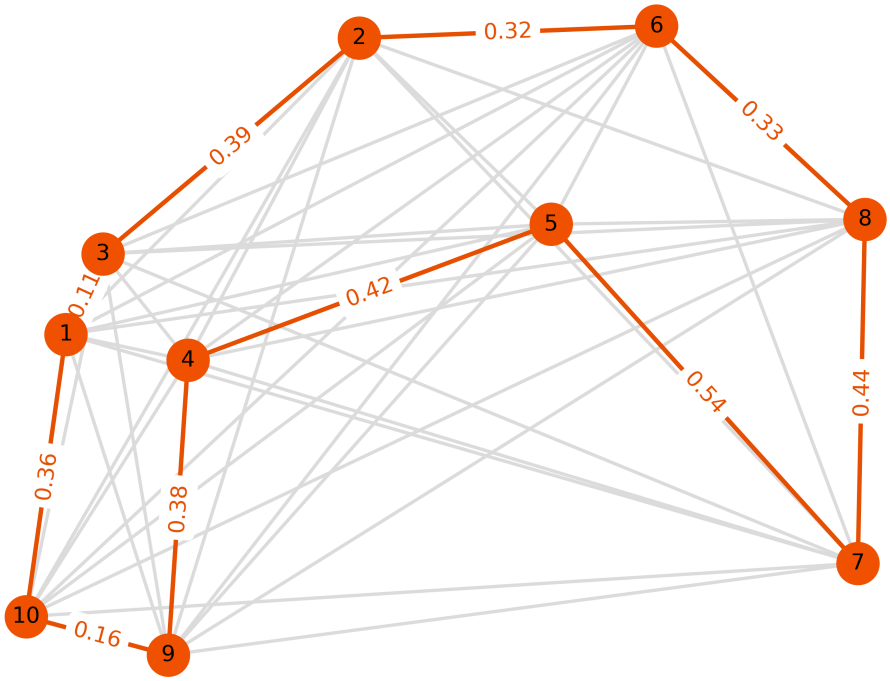

Figure 4.2: Traveling Salesman Problem visualization

[4] `https://pandas.pydata.org/`

This approach allows us to see the structure of the problem and understand the complexity involved in finding the shortest route. Cities are represented as nodes within a network. Each node symbolizes a city, and the edges connecting these nodes depict the possible paths that can be taken between cities. Since we use a complete graph, each pair of cities is connected by a unique road. In our visual representation, we have opted for clarity and simplicity; thus, each road is grayed out until it is in the selected tour. Cities are represented by points on a two-dimensional plane. To accommodate different user preferences and analysis needs, our graphs can be configured to display with or without weights.

An important aspect of our visualization is step by step updates on the current tours and distances at each step of the algorithm. This information is crucial for understanding the progress and efficiency of the algorithm. By providing these details, we aim to make the process of solving the TSP more transparent and understandable.

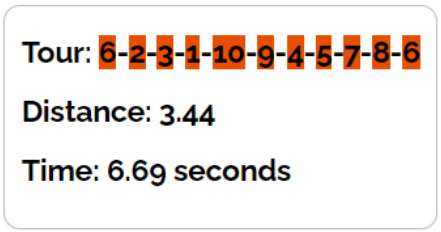

Figure 4.3: Step by step updates

4.2.2 Visualization of the algorithms

HC: At the initial stage, all cities are represented by green and all paths are represented by blue. This symbolizes a randomly selected initial tour. Subsequently, two adjacent cities are swapped, and two edges of the graph are altered. For better comprehension, we highlight four cities that were affected and whose edges changed in yellow. In addition, we mark this two edges in red. Next, the standard representation of the newly selected tour is shown again. This process is repeated until the algorithm identifies a local optimum, which is then marked in red.

HC with restarts: The visualization was simplified, and the most important parts of the algorithm were chosen for display. The visualization consists of all the optimal paths identified in each iteration. Ultimately, the best one is selected, and it is the only one marked in red.

HC with larger search radii: The fundamental principle of visualization remains consistent with that of HC, with the exception that this algorithm covers

all neighbors, thus enabling the alteration of four edges simultaneously. Consequently, it was determined that only the cities that were swapped and the two or four edges that were altered should be marked in yellow.

4.3 Implementation of a web application

Figure 4.4 shows the main architecture of our project with the main folders and files. At the top level, there is a main directory named `Visualizer`, which acts as a container for the entire project. Within this directory, there are subdirectories for different components of the project.

The `playground` repository is where the core functionality of the HC visualizations is implemented. This repository's `views.py` file defines how to handle user requests and serve appropriate responses, which is crucial for interactive visualization. It also contains the Python files `hc.py`, `hc_larger_radii.py`, and `hc_restarts.py`, which implement the HC, HC with larger search radii, and HC with restarts algorithms, respectively.

Adjacent to `playground`, there's a file named `manage.py`. This script provides command line utilities for interacting with Django projects, such as running servers or creating new apps.

Next to it lies another important directory called `visualizer`, which contains settings configurations in `settings.py` — a critical file that includes all configurations necessary for Django projects such as database settings, installed apps, middleware configurations, etc. The `urls.py` file within this same directory maps URLs to views; it's essential for defining how different paths on a website correspond to different views functions.

The data directory is divided into two subdirectories: `custom_maps` and `defined_maps`. The TSP maps that our algorithms will solve are stored in these directories.

The `media` directory contains another subdirectory named `plots`. This is where we store the graphical representations generated by our algorithms.

Within the `playground` directory, we also have the `static` and `templates` directories. The `static` folder contains static files such as CSS and JS files that style our web pages and manage dynamic content. The `templates` folder contains HTML files that define the structure of our web pages. This structure allows us to keep our project organized and maintainable, ensuring that each component of our application has a clear role and responsibility.

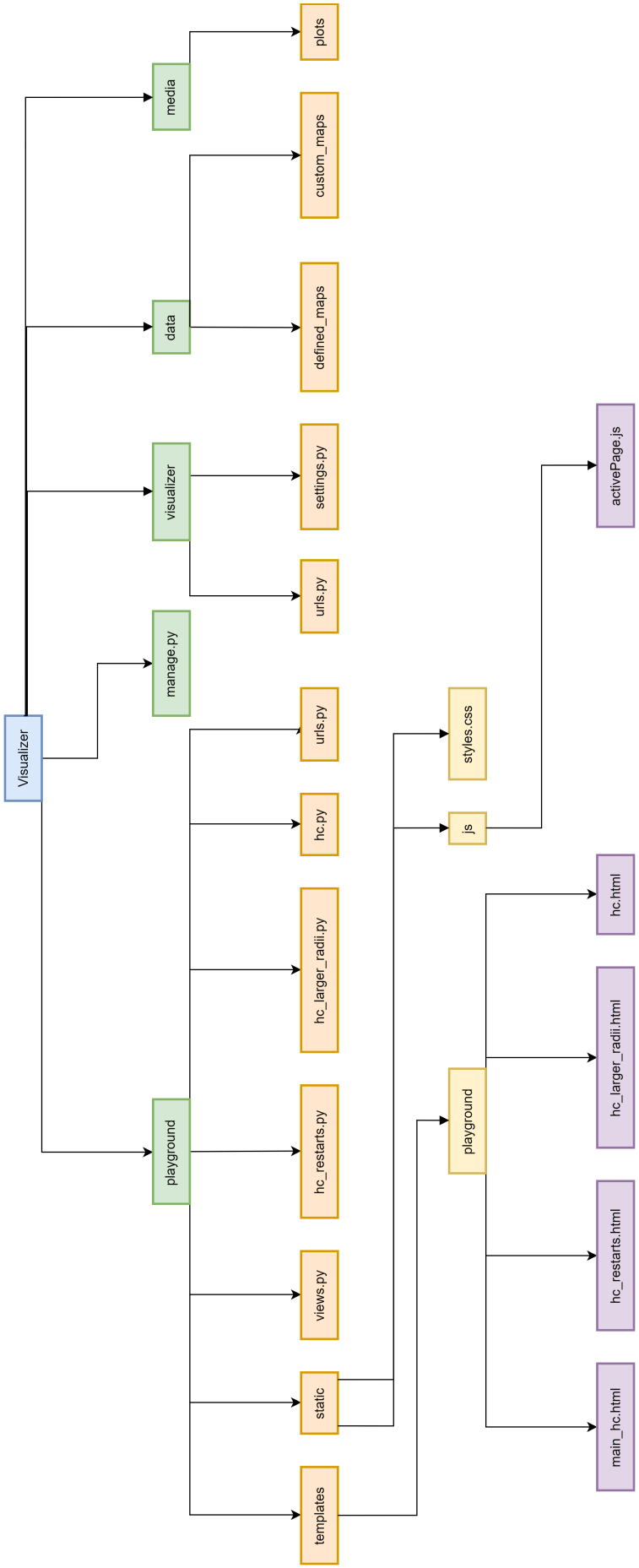

Figure 4.4: Tree diagram of the Django project architecture

4.3.1 Algorithm integration and data handling

In our Django project, we have integrated the HC algorithms as Python modules within the `playground` app. The `views.py` file in the `playground` directory is the main point of integration for these algorithms. As previously mentioned, we have created separate Python modules for each algorithm. These modules are imported at the beginning of the `views.py` file.

The `run_algorithm` function in `views.py` is responsible for executing the appropriate algorithm based on the request received. This function takes the class of the algorithm to be run and the request object as input. It first handles the file upload, cleans up old plots, and then instantiates the appropriate algorithm class with the necessary parameters. The algorithm results are returned in a JSON[5] response. It contains the following data:

`tours`: A list of tours, where each tour is a list of city indices.

`swapped_nodes_list`: A list of swapped nodes for each iteration of the algorithm. This information is used to highlight the swapped nodes in the visualization.

`distances`: A list of total distances for each tour. This provides a measure of the quality of each tour and allows us to track the progress of the algorithm.

`elapsed_time`: The total time taken to run the algorithm. This is useful for performance analysis and comparison with other algorithms.

4.3.2 HTML template design for visualization

To display the visualization of HC algorithms, we have created HTML templates. These templates are located in the `templates` directory within the `playground` app. The main structure of the HTML templates consists of a navigation bar, a sidebar for parameters and controls, and a main area for visualization. The navigation bar provides links to the different algorithm visualizations: HC, HC with restarts, and HC with larger search radii. The sidebar is divided into several sections. The `description` section provides a brief explanation of the purpose of the page. The `tsp-parameters` section allows the user to select the parameters for the TSP. The `algorithm-parameters` section, which is only present in the `hc_restarts.html` template, allows the user to specify the number of runs for the HC with restarts algorithm. The `button-container` section provides controls for navigating through the visualization.

The main visualization area displays a waiting message while the algorithm

[5]JavaScript Object Notation

is running and the results once the algorithm has finished. The results include a rotating list of plots, and information about the current iteration, tour, distance, and elapsed time.

4.3.3 Interactive visualization and error handling

We use JS extensively in our web application to handle user interactions, manage the visualization process, and dynamically update the page content. The `startVisualization` function is responsible for initiating the visualization process. The plots are added to the page with a rotation animation that cycles through each plot. The `$(document).ready` function sets up event handlers for various user interactions. For example, it changes the display of predefined and custom options based on the selected location and validates the file extension when a custom file is uploaded. The `$('#start-button').click` function is triggered when the start button is clicked. It collects the selected parameters from the user, creates a `FormData` object, and sends an AJAX[6] POST request to the server. If the request is successful, it starts the visualization with the received data. If there is an error, it displays the error message. This use of JS allows us to create a dynamic and interactive UI, where the page content is updated based on user input and server responses without requiring a full page reload.

In our Django application, we have implemented error handling in the `validate_csv` function in `views.py`. This function checks the validity of a CSV file based on two conditions:

1. The file must have at least 5 rows.

2. The file must have exactly 2 columns for X and Y coordinates.

On the client side, we use JS for error handling. Our script checks for user input and displays appropriate error messages in the following scenarios:

If the user has selected the custom location option but has not uploaded a custom map file, the script displays an error message “Please upload a custom map file.” and halts the execution.

If neither a custom file nor a city count has been selected, the script displays an error message “Please select the number of cities.” and halts the execution.

[6] Asynchronous JavaScript And XML

Tours visualization

As previously stated, the `views.py` module returns a JSON response containing all the relevant information. Subsequently, JS is employed to render this information in the visualization.

`highlightTour(tour, swappedNodes, backgroundColor)`: This function updates the tour display by highlighting the nodes in the tour. If there are swapped nodes, they are highlighted in a different color.

`displayNextTour(data, tourIndex)`: This function is responsible for displaying each tour in the visualization. It highlights the current tour, updates the distance display, and if the algorithm is HC with restarts, updates the iteration count. It then schedules the next call to itself after a delay, creating a loop that displays each tour one by one.

Navigation

`displayHistory(data, idx, hcRestarts, color)`: This function is responsible for displaying the state of the algorithm at a specific index. It hides all images, shows the image at the current index, highlights the current tour, and updates the distances text. If the algorithm is HC with restarts, it also updates the iteration count.

`$('#prev-button').click()`: This event handler is triggered when the user clicks on the previous button. It calls the `displayHistory` function to update the display based on the new index, and enables the next button.

`$('#next-button').click()`: This event handler is triggered when the user clicks on the next button. If the current plot index is less than the total number of plots, it increments the current plot index. Then, it enables the previous button. If the current plot index is equal to the total number of plots after the increment, it disables the next button.

`$('#replay-button').click()`: This event handler is triggered when the user clicks on the replay button. It calls `startVisualization` to restart the visualization with the same data.

4.4 User Interface

The overall layout has some changes compared to the initial prototype. The "Algorithm parameters" area is now only available on the HC with restarts tab. The algorithm control panel has been moved to the left side for faster user interaction with the interface. In most cases, the user will enter the necessary input data from

top to bottom and reach the control panel where they can immediately start the algorithm. Back and forward options have been added to the same panel. Also, a panel with the current tour, distance and total time has been added to the top right side, as described in 4.2.1. The UI on the "HC with restarts" tab is shown in Figure 4.5.

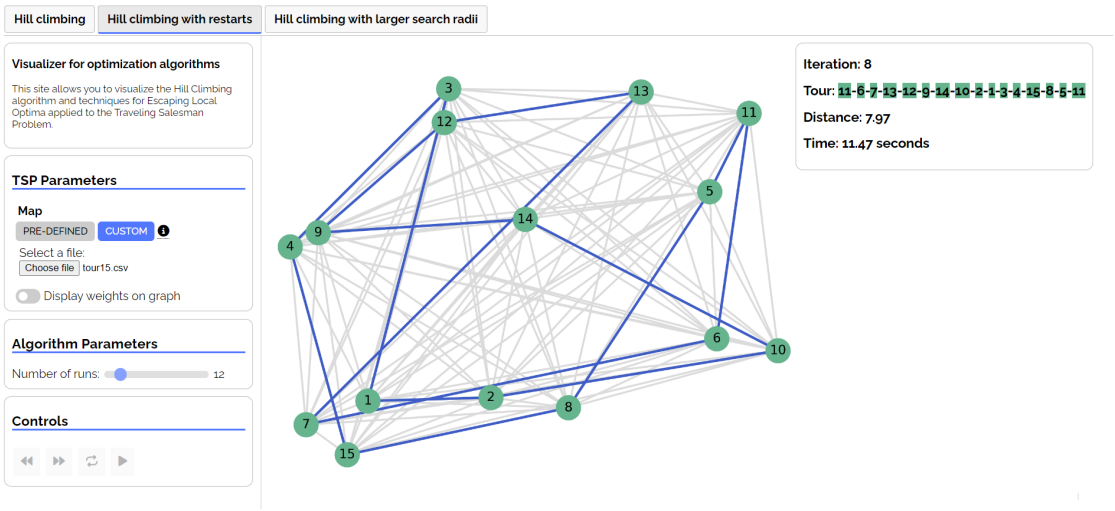

Figure 4.5: User interface

5 Evaluation

This chapter delves into the comprehensive evaluation of the algorithms used. The evaluation process is carefully designed to cover all aspects that contribute to the effectiveness and efficiency of an algorithm. We begin by examining the technical parameters of the test equipment, providing a solid foundation for understanding the environment in which the algorithms operate. Following this, we turn our attention to the testing of the algorithms. Our initial focus was on comparing the basic performance metrics of each algorithm. This comparison allowed us to gain insights into the fundamental strengths and weaknesses of each algorithm. Next, we assessed the scalability of the algorithms. Scalability, in the context of algorithm evaluation, is the ability of an algorithm to maintain or even improve its performance as the size of the input data increases. Lastly, we evaluated the quality of the solution. This was achieved by comparing the solutions generated by the three algorithms with the optimal solution, obtained using the Brute Force algorithm.

5.1 Test equipment technical parameters

Our test equipment for the evaluation process is the **Lenovo IdeaPad Slim 3 15AMN8**. Here are the key technical parameters:

Processor: **AMD Ryzen™ 5 7520U** processor, which has 4 cores, 8 threads, a base frequency of 2.8GHz, and a max frequency of 4.3GHz.

Memory: **16GB LPDDR5-5500**.

Storage: **512GB M.2 PCIe® NVMe® SSD**.

Graphics: **AMD Radeon™ 610M Graphics**.

Operating System: **Windows 11 Enterprise**.

5.2 Algorithms testing

Our first step was to compare the basic performance metrics of each algorithm. We selected a map of twenty cities and set up ten iterations for HC with restarts. Firstly, we measure the execution time, which is the time taken by each algorithm to find a solution. Secondly, we count the number of iterations each algorithm takes to converge. Lastly, we compare the final solution quality by looking at the final distances of the solutions obtained by each algorithm. The results are presented in Table 5.1. HC and HC with restarts both achieved the same best distance of 8.193. However, HC with restarts took more time (0.025s) and more iterations (95) compared to HC which took only 0.007 seconds and 17 iterations. This suggests that while HC with restarts can potentially escape local optima, it does so at the cost of increased time and iterations.

Table 5.1: Basic performance comparison

Algorithm	Best Distance	Execution Time (s)	Iterations
HC	8.192721	0.007121	17
HC with Restarts	8.192721	0.024935	95
HC with Larger Search Radii	4.781514	0.064780	30

HC with larger search radii achieved a significantly better best distance of 4.782, which is almost half of the distance achieved by the other two algorithms. However, it also took the longest execution time of 0.065 and 30 iterations. This confirms our statement in 2.2.2 that increasing the search radii can lead to better solutions, but also increases the computational cost.

The second step in the evaluation process was to assess the scalability of the algorithms. In the context of algorithm evaluation, scalability refers to the ability of an algorithm to maintain or improve its performance as the size of the input data increases. In other words, a scalable algorithm can handle larger problem sizes without significantly decreasing performance. Figure 5.1 displays the execution time in seconds for each algorithm as the number of cities increases.

HC: The blue line represents the standard HC algorithm. The line remains relatively flat as the number of cities increases, suggesting that the algorithm maintains a consistent performance regardless of the problem size. This indicates that the algorithm exhibits good scalability.

HC with restarts: The orange line represents the HC algorithm with restarts. This line indicates a slight increase in execution time as the number of cities in-

creases, suggesting that this algorithm scales relatively well with the problem size.

HC with larger search radii: The green line represents the HC algorithm with larger search radii. This line demonstrates a pronounced increase in execution time as the number of cities increases, particularly when the number of cities approaches 100. This indicates that the algorithm does not scale well with the problem size.

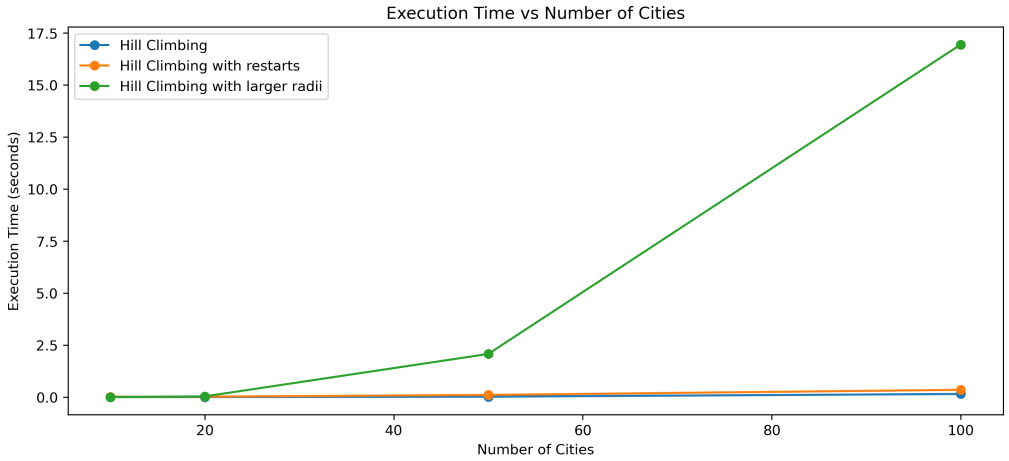

Figure 5.1: Scalability comparison: Execution Time vs Number of Cities

Figure 5.2 illustrates the number of iterations performed by each algorithm prior to convergence, relative to the number of cities.

The HC algorithm demonstrates a gradual increase in the number of iterations as the number of cities increases. This indicates that the algorithm requires more iterations to find a solution as the problem size increases.

With a small number of cities, the HC with restarts starts with a high number of iterations, peaking at 100 iterations. As the number of cities increases, the number of steps decreases significantly and stabilizes at about 40-50 steps. After 40 cities, the number of iterations remains relatively stable. This indicates a consistent performance where the algorithm efficiently navigates the search space without excessively increasing the number of iterations required as the problem size grows.

Conversely, HC with larger search radii requires more iterations as the number of cities increases, reaching a maximum of 100 iterations for 50 and 100 cities. In terms of scalability, HC with restarts might be more efficient for larger numbers of cities as it tends to require fewer iterations.

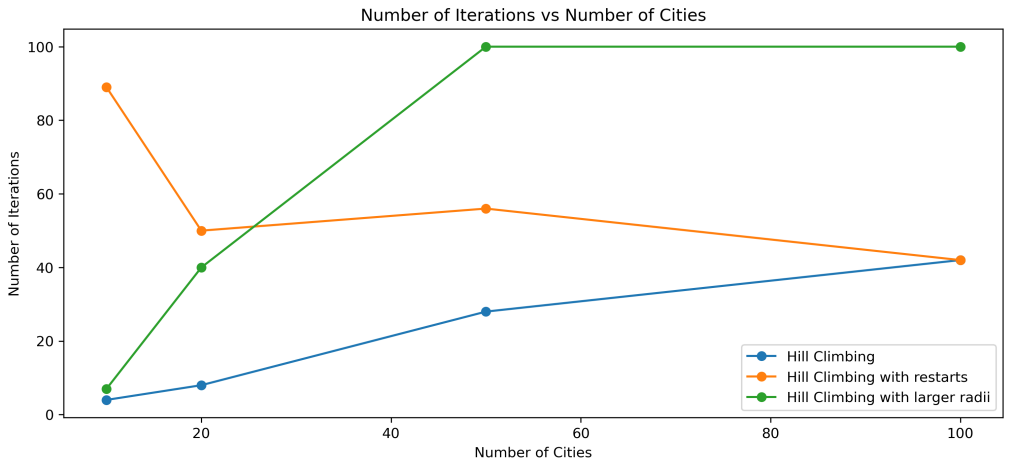

Figure 5.2: Scalability comparison: Number of Iterations vs Number of Cities

The third step in the process was to test the quality of the solution. This was accomplished by comparing the solutions of the three algorithms with the optimal solution (obtained using the Brute Force algorithm) as illustrated in Figure 5.3.

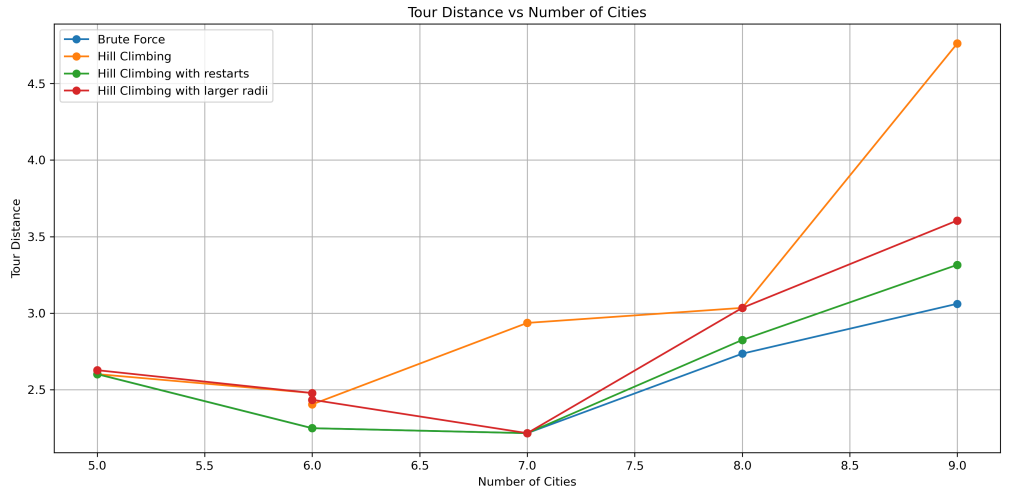

Figure 5.3: Comparison of algorithms solution quality

HC: For 5 and 6 cities, HC performs relatively close to the optimal solution, with only a slight increase in tour distance. However, from 8 to 9 cities, the deviation becomes more pronounced. It can be concluded that the HC algorithm, without restarts or larger radii, is less efficient at maintaining good performance as the number of cities increases.

HC with restarts: From 5 to 7 cities, the algorithm finds the optimal solution. From 7 to 9 cities, it performs better than basic HC but still deviates from the optimal solution, though not as drastically as basic HC. Upon examination of the data, it can be confirmed that HC with restarts improves the algorithm's robust-

ness, allowing it to escape local optima better than basic HC. Nevertheless, as the number of cities increases, the algorithm is still unable to consistently identify the global optimum; despite this, it performs better than basic HC.

HC with larger search radii: For 5 cities, it performs close to the Brute Force but slightly worse than HC with restarts. From 7 to 9 cities, it performs better than basic HC but slightly worse than HC with restarts, especially noticeable at 9 cities where it deviates more significantly. The approach of HC with larger search radii helps improve performance compared to basic HC, but it still falls short compared to HC with restarts as the problem size grows. This suggests that while larger search radii help, the randomness and multiple restarts in HC with restarts provide a better mechanism to explore the solution space.

6 Conclusion

In this thesis, we embarked on a comprehensive study and implementation of optimization algorithms, with a particular focus on HC and its variants. Our objective was to provide a detailed visualization of the operational principles underlying these algorithms, with the TSP serving as our primary use case.

We began with an in-depth problem analysis, which laid the groundwork for our subsequent proposal of a solution. This solution encompassed the implementation of a web application that visualizes the workings of the optimization algorithms. The application was built using Python and the Django framework, and it incorporated various libraries for data handling and visualization.

Our implementation covered three variants of the HC algorithm: the standard version, a version with restarts, and a version with larger search radii. Each variant was thoroughly tested and evaluated, with the results indicating that while the standard HC algorithm was efficient in terms of execution time, the versions with restarts and larger search radii demonstrated better performance in finding optimal solutions.

The visualization component of our work was a significant contribution, providing an interactive and intuitive way to understand the operation of the optimization algorithms. We believe this could serve as a valuable educational tool, enhancing the understanding of these algorithms and their applications in solving complex problems. In terms of the achievement of our objectives, we successfully implemented and visualized the chosen optimization algorithms, and our web application was able to effectively demonstrate their workings. However, there were some limitations in our work. The scalability of the algorithms, particularly the version with larger search radii, was a challenge as the problem size increased.

Looking ahead, there are several open issues and potential avenues for further work. The scalability issue could be addressed by implementing more advanced techniques that yield faster results and enable the visualization of larger problems, even though step-by-step visualization might be more challenging. One

such technique mentioned earlier is Changing Neighborhoods, a concept rooted in the systematic alteration of neighborhood structures during the search process. The user interface of the application could be enhanced to provide a more engaging user experience. Furthermore, the application could be extended to cover more optimization algorithms and problem domains.

In conclusion, this thesis represents a significant step forward in the understanding and visualization of optimization algorithms. We believe our work has not only contributed to the academic field but also has practical implications in providing a useful tool for learning and teaching optimization algorithms.

Bibliography

1. RAO, Singiresu S. Introduction to Optimization. In: *Engineering Optimization Theory and Practice*. John Wiley & Sons, Ltd, 2019, chap. 1, pp. 1–56. Available from DOI: 10.1002/9781119454816.ch1.
2. NAYAK, Sukanta. Chapter one - Introduction to optimization. In: NAYAK, Sukanta (ed.). *Fundamentals of Optimization Techniques with Algorithms*. Academic Press, 2020, chap. 1, pp. 1–7. Available from DOI: 10.1016/B978-0-12-821126-7.00001-2.
3. YANG, Xin-She. Chapter 1 - Introduction to Algorithms. In: YANG, Xin-She (ed.). *Nature-Inspired Optimization Algorithms*. 2nd ed. Academic Press, 2021, chap. 1, pp. 1–22. Available from DOI: 10.1016/B978-0-12-821986-7.00008-1.
4. PERES, Fernando; CASTELLI, Mauro. Combinatorial Optimization Problems and Metaheuristics: Review, Challenges, Design, and Development. *Applied Sciences*. 2021, vol. 11, no. 14. Available from DOI: 10.3390/app11146449.
5. FU, Zhang-Hua; QIU, Kai-Bin; ZHA, Hongyuan. Generalize a Small Pre-trained Model to Arbitrarily Large TSP Instances. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2021, vol. 35, no. 8, pp. 7474–7482. Available from DOI: 10.1609/aaai.v35i8.16916.
6. GUPTA, Shruti; RANA, Ajay; KANSAL, Vineet. Comparison of Heuristic techniques: A case of TSP. In: *2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. 2020, pp. 172–177. Available from DOI: 10.1109/Confluence47617.2020.9058211.
7. SHI, Yong; ZHANG, Yuanying. The neural network methods for solving Traveling Salesman Problem. *Procedia Computer Science*. 2022, vol. 199, pp. 681–686. Available from DOI: 10.1016/j.procs.2022.01.084. The 8th International Conference on Information Technology and Quantitative Man-

agement (ITQM 2020 & 2021): Developing Global Digital Economy after COVID-19.

8. TARI, Sara; BASSEUR, Matthieu; GOËFFON, Adrien. Expansion-based Hill-climbing. *Information Sciences*. 2023, vol. 649, p. 119635. Available from DOI: 10.1016/j.ins.2023.119635.

9. TSAI, Chun-Wei; CHIANG, Ming-Chao. Chapter Sixteen - Local search algorithm. In: TSAI, Chun-Wei; CHIANG, Ming-Chao (eds.). *Handbook of Metaheuristic Algorithms*. Academic Press, 2023, chap. 16, pp. 351–374. Uncertainty, Computational Techniques, and Decision Intelligence. Available from DOI: https://doi.org/10.1016/B978-0-44-319108-4.00030-7.

10. FRIEDRICH, Tobias; KÖTZING, Timo; KREJCA, Martin S.; RAJABI, Amirhossein. Escaping Local Optima with Local Search: A Theory-Driven Discussion. In: RUDOLPH, Günter; KONONOVA, Anna V.; AGUIRRE, Hernán; KERSCHKE, Pascal; OCHOA, Gabriela; TUŠAR, Tea (eds.). *Parallel Problem Solving from Nature – PPSN XVII*. Cham: Springer International Publishing, 2022, pp. 442–455. Available from DOI: 10.1007/978-3-031-14721-0_31.

11. DUARTE, Abraham; SÁNCHEZ-ORO, Jesús; MLADENOVIĆ, Nenad; TODOSIJEVIĆ, Raca. Variable Neighborhood Descent. In: *Handbook of Heuristics*. Ed. by MARTÍ, Rafael; PARDALOS, Panos M.; RESENDE, Mauricio G. C. Cham: Springer International Publishing, 2018, pp. 341–367. Available from DOI: 10.1007/978-3-319-07124-4_9.

12. ZAVGORODNIAIA, Albina; TILANTERÄ, Artturi; KORHONEN, Ari; SEPPÄLÄ, Otto; HELLAS, Arto; SORVA, Juha. Algorithm Visualization and the Elusive Modality Effect. In: *Proceedings of the 17th ACM Conference on International Computing Education Research*. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 368–378. ICER 2021. Available from DOI: 10.1145/3446871.3469747.

13. ROZALIA OSZTIÁN, Pálma; KÁTAI, Zoltán; OSZTIÁN, Erika. Algorithm Visualization Environments: Degree of interactivity as an influence on student-learning. In: *2020 IEEE Frontiers in Education Conference (FIE)*. 2020, pp. 1–8. Available from DOI: 10.1109/FIE44824.2020.9273892.

14. WANG, Zijie J.; TURKO, Robert; SHAIKH, Omar; PARK, Haekyu; DAS, Nilaksh; HOHMAN, Fred; KAHNG, Minsuk; POLO CHAU, Duen Horng. CNN Explainer: Learning Convolutional Neural Networks with Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*. 2021, vol. 27, no. 2, pp. 1396–1406. Available from DOI: 10.1109/TVCG.2020.3030418.

15. BAHAR, Abdullrahman Yousif; SHORMAN, Samer Mahmoud; KHDER, Moaiad Ahmad; QUADIR, Abdullah Muhammad; ALMOSAWI, Sayed Ahmed. Survey on Features and Comparisons of Programming Languages (PYTHON, JAVA, AND C#). In: *2022 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems (ICETSIS)*. 2022, pp. 154–163. Available from DOI: 10.1109/ICETSIS55481.2022.9888839.

16. VALENTINE, Daniel T.; HAHN, Brian D. Chapter 12 - Dynamical systems. In: VALENTINE, Daniel T.; HAHN, Brian D. (eds.). *Essential MATLAB for Engineers and Scientists*. 8th ed. Academic Press, 2023, chap. 12, pp. 261–280. Available from DOI: 10.1016/B978-0-32-399548-1.00019-9.

17. JAISWAL, Priyanka; HELIWAL, Sumit. Competitive Analysis of Web Development Frameworks. In: KARRUPUSAMY, P.; BALAS, Valentina Emilia; SHI, Yong (eds.). *Sustainable Communication Networks and Application*. Singapore: Springer Nature Singapore, 2022, pp. 709–717. Available from DOI: 10.1007/978-981-16-6605-6_53.

18. DEBNÁR, Peter. *Spracovanie a vizualizácia optimalizačných algoritmov inšpirovaných prírodou* [*Processing and visualization of nature-inspired optimization algorithms*]. Bratislava, SK, 2022. Available also from: https://opac.crzp.sk/?fn=detailBiblioFormChild01VQG&sid=C03CB0FBA6588971DE3120D1B6BF&seo=CRZP-detail-kniha.

19. KALOUSKOVÁ, Veronika. *Vizualizácia optimalizačných algoritmov inšpirovaných prírodou* [*Visualizing nature-inspired optimization algorithms*]. Bratislava, SK, 2021. Available also from: https://opac.crzp.sk/?fn=detailBiblioFormChildM3EJD&sid=1CB25EF4639541CD2BCDB9031481&seo=CRZP-detail-kniha.

20. DZVONÍKOVÁ, Dominika. *Spracovanie a vizualizácia optimalizačných algoritmov inšpirovaných prírodou* [*Processing and visualization of nature-inspired optimization algorithms*]. Bratislava, SK, 2023. Available also from: https://opac.crzp.sk/?fn=detailBiblioFormChildS37J1&sid=C03CB0FBA6588971DE3126D1B6BF&seo=CRZP-detail-kniha.

21. DELINČÁK, Matej. *Vizualizácia optimalizačných algoritmov* [*Visualization of optimization algorithms*]. Bratislava, SK, 2022. Available also from: https://opac.crzp.sk/?fn=detailBiblioFormChildA46E1&sid=7E432E204D1F6C329459883802D8&seo=CRZP-detail-kniha.

22. KUCHÁR, Patrik. *Spracovanie a vizualizácia optimalizačných algoritmov inšpirovaných prírodou* [*Processing and visualization of nature-inspired optimization algorithms*]. Bratislava, SK, 2023. Available also from: `https://opac.crzp.sk/?fn=detailBiblioFormChildM1T3P&sid=C03CB0FBA6588971DA3823D1B6BF&seo=CRZP-detail-kniha`.

23. DLHÝ, Ľubomír. *Vizualizácia optimalizačných algoritmov* [*Visualization of optimization algorithms*]. Bratislava, SK, 2022. Available also from: `https://opac.crzp.sk/?fn=detailBiblioFormChildM1CDJ&sid=4F55101800366CEF5AF3C8DB109A&seo=CRZP-detail-kniha`.

24. CHENG, Hanni; ZHENG, Haosi; CONG, Ya; JIANG, Weihao; PU, Shiliang. Select and Optimize: Learning to solve large-scale TSP instances. In: RUIZ, Francisco; DY, Jennifer; MEENT, Jan-Willem van de (eds.). *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*. PMLR, 2023, vol. 206, pp. 1219–1231. Proceedings of Machine Learning Research.

List of Abbreviations

AV Algorithm visualization.

COP Combinatorial optimization problem.

GUI Graphical user interface.

HC Hill Climbing.

HTML HyperText Markup Language.

JS JavaScript.

KNP Knapsack Problem.

LS Local search.

TSP Travelling Salesman Problem.

UI User interface.

List of Appendixes

Appendix A User manual

Appendix B System manual

Appendix C CD medium – final thesis in electronic form,

A User manual

A.1 Introduction

In this application, we aim to provide a comprehensive visualization of optimization algorithms, specifically focusing on Hill Climbing and its variants. The application is designed to be an educational tool, enhancing the understanding of these algorithms and their applications in solving complex problems such as the Traveling Salesman Problem.

A.2 User interface overview

The user interface consists of several sections:

Algorithm Selection Tabs: At the top of the interface, there are tabs labeled "Hill Climbing", "Hill Climbing with restarts", and "Hill Climbing with larger search radii". Users can select the algorithm they want to visualize by clicking on these tabs.

TSP Parameters Section: This section allows users to customize the parameters for the Traveling Salesman Problem (TSP). Options include selecting the map size and type, and setting algorithm weights.

Visualization Area: The central area of the interface displays a graphical representation of the TSP and the selected optimization algorithm's progress. The TSP is represented as a set of numbered points (cities) connected by lines (paths), and the algorithm's current solution is highlighted.

Control Panel: The control panel includes buttons for starting and stopping the algorithm, as well as sliders for adjusting parameters like mutation rate.

Information Panel: On the right side of the interface, the information panel displays real-time information about the algorithm's progress, including the current iteration count, tour information, total distance, and elapsed time.

A.3 Using the application

Select an Algorithm: Click on the tab corresponding to the algorithm you want to visualize.

Set TSP Parameters: In the TSP Parameters section, select your desired map size and type, and set the algorithm weights as needed.

Start the Visualization: Click the "Start" button in the control panel to begin the visualization. You will see the algorithm's progress in the visualization area and real-time information in the information panel.

Review the Results: After the algorithm has completed, review the results in the information panel. You can compare these results with different algorithms or parameter settings to gain a deeper understanding of the optimization algorithms.

B System manual

B.1 Installing the application

This section provides detailed instructions on how to install and run the Django project locally on your machine. Follow these steps carefully to ensure a successful setup.

B.1.1 Prerequisites

Before you begin, ensure you have the following installed on your machine:

1. **Python 3.x** - You can download Python from the official website.
2. **pip** - This is the package installer for Python. It usually comes with Python, but you can verify its installation by running `pip -version` in your terminal.
3. **virtualenv** - A tool to create isolated Python environments. Install it using pip if you don't have it: `pip install virtualenv`.

B.1.2 Step-by-Step Installation Guide

1. **Clone the Repository**
   First, clone the project repository from GitHub.

   ```
   git clone https://github.com/olha133/visualization.git
   cd visualization
   ```

2. **Create a Virtual Environment**
   Create a virtual environment to keep dependencies isolated.

   ```
   python -m venv env
   ```

3. **Activate the Virtual Environment**
   Activate the virtual environment. The command varies depending on your operating system.

   - **On Windows:**

     ```
     .\env\Scripts\activate
     ```

   - **On macOS and Linux:**

     ```
     source env/bin/activate
     ```

4. **Install Project Dependencies**
   Install the required dependencies using `pip` and the provided `Pipfile.lock`.

   ```
   pip install pipenv
   pipenv install
   ```

5. **Set Up the Database**
   Run the following commands to set up the SQLite database.

   ```
   python manage.py makemigrations
   python manage.py migrate
   ```

6. **Create a Superuser**
   Create an admin user to access the Django admin interface.

   ```
   python manage.py createsuperuser
   ```

   Follow the prompts to set up the admin user credentials.

7. **Run the Development Server**
   Start the Django development server to run the application locally.

   ```
   python manage.py runserver
   ```

   By default, the server will start at `http://127.0.0.1:8000/`. You can now open this URL in your web browser to see the application running.

B.2 File Descriptions

This section provides a brief description of each file and its purpose within the Django project structure.

B.2.1 Data Directory

- **data/custom_maps**
  Contains custom TSP maps uploaded by the user in CSV format.
- **data/defined_maps**
  Contains predefined TSP maps used for initial testing and demonstration purposes.

B.2.2 Media Directory

- **media/plots**
  Stores generated plots of TSP solutions for visualization purposes.

B.2.3 Playground Directory

- **playground/hc.py**
  Implements the basic Hill Climbing algorithm for solving the TSP.
- **playground/hc_larger_radii.py**
  Implements the Hill Climbing algorithm with larger search radii for solving the TSP.
- **playground/hc_restarts.py**
  Implements the Hill Climbing algorithm with restarts for solving the TSP.
- **playground/urls.py**
  Defines the URL patterns for the `playground` app, mapping URLs to their respective views.
- **playground/views.py**
  Contains the views for handling the web requests and rendering the templates.

B.2.4 Static Directory

- **playground/static/styles.css**
  Contains the CSS styles used to design the appearance of the web pages.

- **playground/static/js/activePage.js**
  Contains JavaScript code to handle the dynamic aspects of the web pages.

B.2.5 Templates Directory

- **playground/templates/playground/hc.html**
  HTML template for displaying the basic Hill Climbing algorithm visualization.

- **playground/templates/playground/hc_larger_radii.html**
  HTML template for displaying the Hill Climbing algorithm with larger search radii visualization.

- **playground/templates/playground/hc_restarts.html**
  HTML template for displaying the Hill Climbing algorithm with restarts visualization.

- **playground/templates/playground/main_hc.html**
  Main HTML template for displaying the initial view of the Hill Climbing algorithms.

B.2.6 Visualizer Directory

- **visualizer/settings.py**
  Configures the settings for the Django project, including installed apps, middleware, and database settings.

- **visualizer/urls.py**
  Defines the URL patterns for the main project, routing requests to the appropriate app and view.