

Heuristički algoritmi za problem trgovačkog putnika

Perušić, Filip

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: <https://um.nsk.hr/um:nbn:hr:168:771260>

Rights / Prava: [In copyright](#)/[Zaštićeno autorskim pravom.](#)

Download date / Datum preuzimanja: **2025-03-29**



Repository / Repozitorij:

[FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repository](#)



SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 8

**HEURISTIČKI ALGORITMI ZA PROBLEM TRGOVAČKOG
PUTNIKA**

Filip Perušić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 8

**HEURISTIČKI ALGORITMI ZA PROBLEM TRGOVAČKOG
PUTNIKA**

Filip Perušić

Zagreb, lipanj 2024.

ZAVRŠNI ZADATAK br. 8

Pristupnik: **Filip Perušić (0112082895)**
Studij: Computing
Modul: Computing
Mentorica: izv. prof. dr. sc. Anamari Nakić

Zadatak: **Heuristički algoritmi za problem trgovačkog putnika**

Opis zadatka:

Problem trgovačkog putnika, vezan za područja računalnih znanosti i operacijskih istraživanja, odnosi se na problem određivanja najkraće rute kroz sve gradove na ruti, pri čemu se svaki grad posjećuje točno jednom prije povratka na početak. Ovaj poznati problem se u matematici proučava u okviru diskretne matematike i modelira se s pomoću grafova. U ovoj terminologiji problem se svodi na pronalaženje najkraćeg Hamiltonovog ciklusa u potpunom težinskom grafu. Za ovaj zanimljiv, ali računalno zahtjevan problem, jedini poznati egzaktni algoritmi su oni eksponencijalne složenosti. Za grafove s velikim brojem vrhova takvi su algoritmi neupotrebljivi, stoga se u primjenama često zamjenjuju aproksimacijskim i heurističkim algoritmima koji daju dobre približne rezultate. U ovom će se radu detaljno predstaviti problem trgovačkog putnika. Implementirat će se tri heuristička algoritma: najbliži susjed, pohlepni heuristički i genetski algoritam. Izradit će se testni primjeri na kojima će se uspoređivati učinkovitost implementiranih programa. Dobiveni rezultati također će se usporediti s rezultatima egzaktnog algoritma. Izradit će se interaktivna aplikacija s grafičkim sučeljem koja će korisnicima ponuditi sveobuhvatnu analizu rješenja, balansirajući učinkovitost i točnost.

Rok za predaju rada: 14. lipnja 2024.

Table of contents

INTRODUCTION	1
1. MOTIVATION	3
1.1. BACKGROUND	3
1.2. REAL-WORLD APPLICATIONS	4
2. METHODOLOGY	5
2.1. GRAPH THEORY	5
2.2. PROBLEM FORMULATION	8
2.3. ALGORITHM IMPLEMENTATION.....	10
3. TSP EXPLORER	16
3.1. USER INTERFACE DESIGN AND FEATURES	16
3.2. TECHNICAL STACK	18
3.3. PERFORMANCE METRICS AND RESULTS.....	19
4. EXPERIMENTAL ANALYSIS	21
4.1. TESTING INSTANCES	21
4.2. RESULTS AND DISCUSSION	21
CONCLUSION	28
SUMMARY	29
LITERATURE	30

Introduction

The Traveling Salesman Problem (TSP) is a classic problem in the fields of computer science, discrete mathematics and operations research, and it has taken the interest of researchers and practitioners for decades. The essence of TSP lies in finding the shortest possible route that visits a set of cities exactly once and returns to the starting city. Despite its clear formulation at first, TSP is known for its computational complexity. It is an NP-hard problem, meaning that no known algorithm can solve all combinations of instances of TSP in polynomial time. [5] As a result, exact algorithms, which guarantee finding the optimal solution, often become impractical for large instances due to their time complexity.

In the field of discrete mathematics, TSP is modeled using graphs. Specifically, the problem can be described as finding the shortest Hamiltonian cycle in a complete weighted graph. [7] In this thesis, the vertices represent the cities, and the edges represent the paths between them, weighted by the distances associated with traveling those paths. The challenge is to identify a cycle that visits each city exactly once and returns to the starting city, with the minimum possible total distance.

Given the computational problematic of exact algorithms for large instances, approximation and heuristic algorithms have become important tools for tackling TSP in practice. These algorithms do not guarantee an optimal solution but aim to provide good approximate solutions within a reasonable time frame. This thesis focuses on three heuristic algorithms: the Nearest Neighbor Algorithm, the Greedy Heuristic Algorithm, and the Genetic Algorithm.

To evaluate and compare the performance of these heuristic algorithms, this thesis will include a series of test examples. These examples will vary in size and complexity, providing an insight of how each algorithm performs under different conditions. The results obtained from these heuristic methods will be compared against those from an exact algorithm where applicable, to evaluate their accuracy and efficiency. The exact algorithm, while limited to

smaller instances due to its computational demands, will serve as a benchmark to show the trade-offs between the quality of solutions for those smaller instances.

In addition to the theoretical analysis, an interactive application called TSP Explorer with a graphical user interface (GUI) will be developed. This application aims to offer users a simple and engaging way to explore TSP solutions. Users will be able to input their own sets of cities, visualize the resulting tours, and compare the performance of different algorithms. This tool will not only serve as an educational resource but also as a practical demonstration of the concepts discussed in the thesis.

1. Motivation

1.1. Background

Travelling salesman problem has been around since the 19th century. The first mention of this problem goes back to the year 1832 where it first appeared in the German handbook “Der Handlungsreisende - Von einem alten Commis – Voyageur” for salesman traveling through Germany and Switzerland. [6] The problem started being formally studied around the 1930s and 1940s by the Irish mathematician William Rowan Hamilton and by the British mathematician Thomas Kirkman.

In the early 20th century, the formal mathematical treatment of the problem started to take shape. A significant milestone was achieved in the 1950s when the first exact algorithms designed to solve the TSP were founded. [3] One of the first methods introduced was the cutting-plane method, which demonstrated the potential of linear programming techniques in addressing combinatorial optimization problems.

The theoretical significance of TSP was further shown by Richard Karp in the 1972. Richard Karp classified TSP as an NP-hard problem, a class of problems for which there does not exist an algorithm which can solve it in a polynomial-time frame. [5]

Later, as the limitations of exact algorithms became clearer, scientists turned to heuristic and approximation algorithms. In the 1970s and 1980s various methods, such as Nearest Neighbour, Greedy Heuristic, and Genetic algorithms were developed to provide better approximate solutions within a reasonable time frame. [1]

In the past few decades, advancements in computational power have helped researchers to solve much larger instances of TSP. Modern applications extend to various areas such as autonomous vehicle routing, drone delivery systems, mapping routes and much more, where TSP algorithms have made a significant impact. These advancements continue to push the boundaries, making TSP still an evolving and relevant problem in both theoretical and practical fields.

1.2. Real-World applications

TSP is widely used in various practical applications across numerous different fields. Some of the real-world applications:

- Warehouse picking: TSP provides help in finding the shortest path for picking items from different locations, reducing effort and time.
- Robotics: industrial robots need to visit multiple points to perform tasks like assembly, painting etc. TSP helps in planning the path of the robot to reduce energy consumption.
- Tourism: travel agencies use the TSP to find the optimal route of travel which would encompass visiting many locations and reducing fuel usage.
- Vehicle routing: TSP is used to optimize the routes of delivery vehicles to reduce the travel time and fuel costs.
- Agriculture: farmers use the concept of TSP to plan the routes of the machines, ensuring efficient coverage of fields.
- Satellite Coverage: TSP helps in determining the optimal paths for satellites to ensure coverage of the Earth's surface for imaging and communication purposes.

Above are some of the listed applications, but there are still many more, highlighting the importance of finding solutions for the TSP.

2. Methodology

2.1. Graph theory

Graph theory is one of the fundamental areas of mathematics and computer science that studies graphs, which are structures that model relationships between objects. A graph structure is defined with vertices (nodes) and edges (also called arcs) that connect pairs of vertices. Graph theory is widely used in numerous fields of computer science.

Some of the basic concepts of graph theory:

Graph G is defined as a pair $G = (V, E)$ where V is a set of vertices and E is a set of edges that connects the pair of vertices.

Different types of graphs:

- Undirected Graph: edges have no direction, edge $(u, v) = (v, u)$

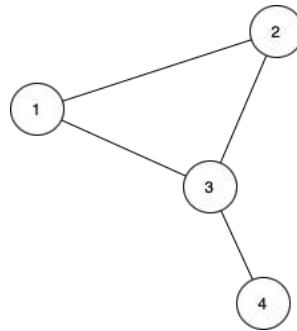


Figure 1: Undirected graph.

- Directed Graph (Digraph): edges have a direction $(u, v) \neq (v, u)$

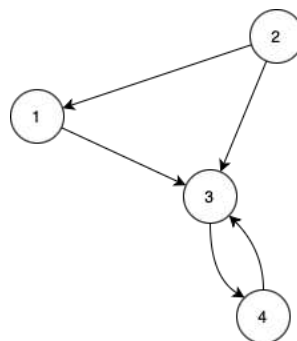


Figure 2: Directed graph.

- **Weighted Graph:** each edge has a weight which can represent distance, cost or other metrics.

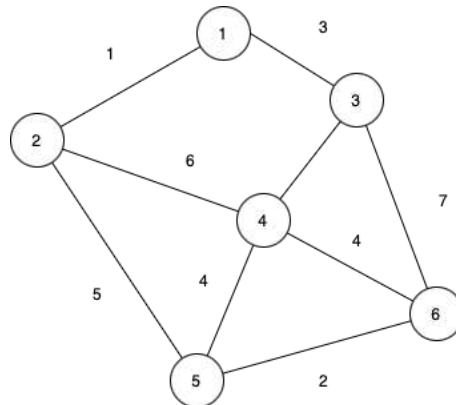


Figure 3: Weighted graph.

- **Complete Graph:** graph in which each vertex is connected to every other vertex, in other words an undirected graph where every pair of distinct vertices is connected by a unique edge.

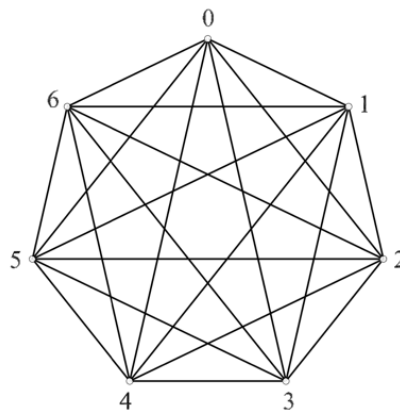


Figure 4: Complete graph K_7 .

- **Path:** A sequence of edges that connects to a sequence of vertices.
- **Cycle:** A path that start and ends at the same vertex with no other repetitions of vertices and edges.
- **Hamiltonian Graph:** Cycle which passes through all vertices of the given graph is a Hamiltonian cycle. A graph which has a Hamiltonian cycle is called a Hamiltonian Graph.

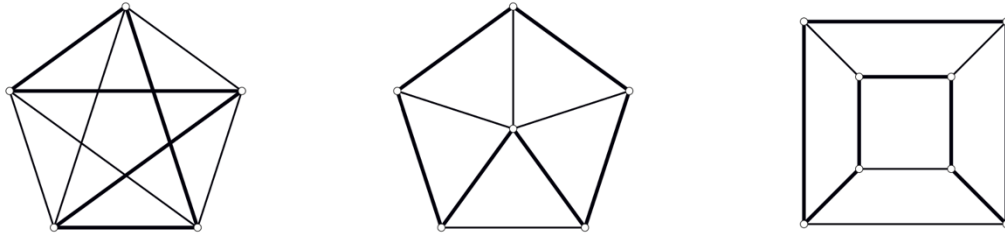


Figure 5: Three Hamiltonian graphs along with their Hamiltonian cycles.

Theorem – Ore, 1960. [13]

If G is a simple graph with n vertices, where $n \geq 3$, and if the condition:

$$\deg(v) + \deg(w) \geq n$$

holds for every pair of non-adjacent vertices v and w of graph G , then G is Hamiltonian Graph.

Proof:

Suppose the contrary, i.e., let G be a non-Hamiltonian graph with n vertices that satisfies the given degree condition. By adding edges to the given graph G edge by edge, we can achieve that the graph becomes Hamiltonian. Let us stop at the step of adding edges just before the graph becomes Hamiltonian, meaning we obtain a graph by adding just one more edge that would become Hamiltonian. Note that by adding edges we do not disrupt the degree condition, indeed, the vertex degrees can only increase. Since we are now one step away from Hamiltonicity, this means we can find a (not necessarily closed) path $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n$ that traverses every vertex. Now, since G is non-Hamiltonian, the vertices v_1 and v_n are not adjacent, so for them it holds that $\deg(v_1) + \deg(v_n) \geq n$.

This means that v_1 has at least one other neighbor besides v_2 , just as v_n has at least one other neighbor besides v_{n-1} . We define the sets:

$$A = \{ i \mid 2 \leq i \leq n, v_i \text{ is adjacent to } v_1 \}$$

$$B = \{ i \mid 2 \leq i \leq n, v_{i-1} \text{ is adjacent to } v_n \}$$

Then, $|A| = \deg(v_1)$ and $|B| = \deg(v_n)$, and it holds that $|A| + |B| \geq n$. It follows that the intersection $A \cap B$ is non-empty. Therefore, there necessarily exists some v_i adjacent to v_1 such that v_{i-1} is adjacent to v_n .

Now the path

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{i-1} \rightarrow v_n \rightarrow v_{n-1} \rightarrow \cdots \rightarrow v_i \rightarrow v_1$$

is a Hamiltonian cycle, contrary to the assumption that such does not exist.

The famous Dirac's theorem, with a slightly stronger condition on the degrees of the vertices, is now a direct consequence of the just-proven Ore's theorem.

Theorem – Dirac, 1952. [13]

If G is a simple graph with n vertices ($n \geq 3$) and if $\deg(v) \geq \frac{n}{2}$ for every vertex v in G , then G is Hamiltonian Graph.

Proof:

We can directly apply Ore's theorem, considering that the inequality from Ore's theorem is certainly satisfied: $\deg(v) + \deg(w) \geq \frac{n}{2} + \frac{n}{2} = n$.

2.2. Problem formulation

Given the theory introduction, the Traveling salesman problem can be reduced to finding a shortest Hamiltonian cycle in the complete weighted graph.

Formal definition: [14]

1. Set of nodes (cities): $N = \{1, 2, \dots, n\}$ – a set of cities that need to be visited.
2. Distance matrix: $D = [d_{ij}]_{n \times n}$ – an $n \times n$ matrix, where $[d_{ij}]$ represents the distance between city i and city j .
3. Decision variables: x_{ij} – binary values which equals to 1 if the path between cities i and j is included in the final solution and 0 otherwise.

The main objective for the TSP solution is to minimize the total distance travelled which can be represented as:

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

With additional constraints:

- Each city is visited exactly once.

- The cycle must finish in the same city from which it started.
- There are no routes that do not include all cities.

TSP Explorer employs the following mathematical formulations to address the Traveling salesman problem:

- Euclidian distance for measurements of distances between cities.
- Distance matrices to represent the problem space.
- Greedy and heuristic methods for approximations.
- Genetic algorithm for evolutionary optimization.
- Brute force for benchmarking.

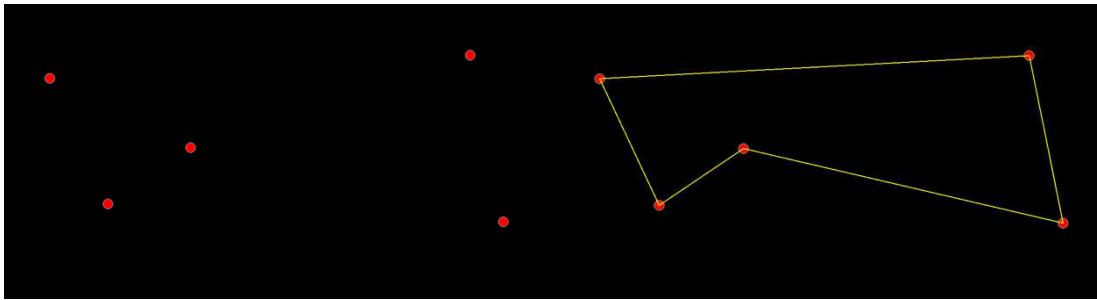


Figure 6: Example of a graph created inside the TSP Explorer application along with its optimal solution.

2.3. Algorithm implementation

2.3.1. Brute Force Algorithm

The Brute Force Algorithm also known as the *Exhaustive Search Algorithm* is a straightforward approach for solving the TSP. This algorithm guarantees finding the optimal solution because it evaluates every route. It enumerates all possible permutations of the cities and calculates the total distance travelled for each permutation to find the shortest route which makes its time complexity factorial. In this thesis and inside the TSP explorer application this algorithm is used as a benchmark for all the other algorithms on several cities where the computation is feasible. The Figure 7 depicts factorial time complexity $O(n!)$ of the algorithm. [3]

n	$(n-1)!/2$
3	1
4	3
5	12
6	60
7	360
8	2,520
9	20,160
10	181,440
20	6.08226E+16
30	4.42088E+30
40	1.01989E+46

Figure 7: Maximum number of Hamiltonian cycles. [10]

The table represents the maximum number of Hamiltonian cycles in a complete graph K_n .

The first column, labeled n represents the number of vertices in the complete graph.

The second column, labeled $\frac{(n-1)!}{2}$, represents the maximum number of Hamiltonian cycles in the complete graph K_n .

Below is the pseudocode of the implemented algorithm inside the TSP explorer application.

```
Initialize: List all possible permutations of cities
Set best_route = None and min_distance = ∞

Iteration:
1. For each permutation (route) of cities:
    a. Calculate the total distance of the route
    b. If the total distance < min_distance:
        i. Set min_distance = total distance
        ii. Set best_route = route

Output: The best_route with the minimum total distance.
```

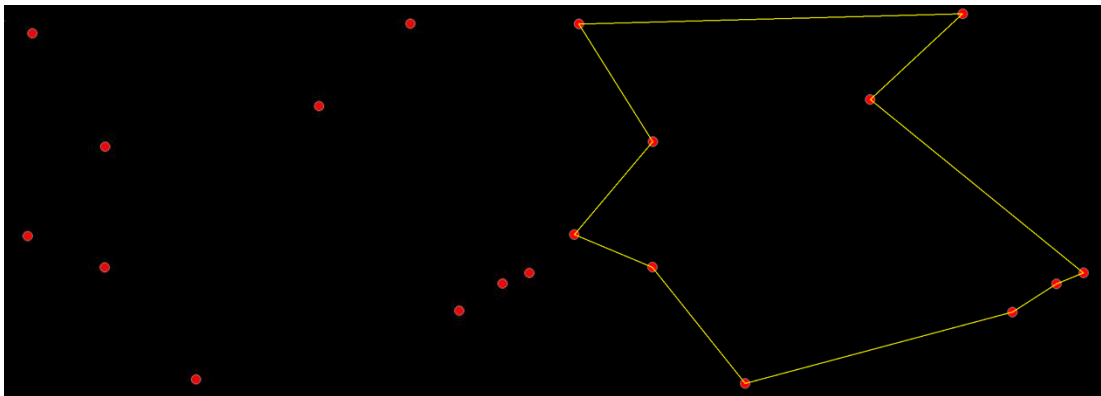


Figure 8: Brute Force Algorithm performed on a generated city map with 10 cities and the optimal solution found.

2.3.2. Nearest Neighbour Algorithm

Nearest neighbour algorithm constructs a tour starting from arbitrary city and continuously visiting the nearest city until all the cities have been visited. Once all the cities are visited and inside the tour, the algorithm returns to the starting city to close the cycle.

This algorithm is one of the most straightforward approaches and provides a quick solution that is often not optimal. Although it lacks optimality it is very fast and simple making it suitable for obtaining a fast approximation of the solution. It is important to note that due to its greedy nature it can get trapped in local optima and may not produce the shortest possible routes often. Time complexity of the algorithm is $O(n^2)$.

Below is the pseudocode used for the implementation of the Nearest Neighbour algorithm inside the TSP explorer implementation, which provides a high-level overview.

```
Initialize: Pick a starting city  $v_0$ 
Form a path  $P = \{v_0\}$  and mark  $v_0$  as visited.

Iteration:
1. Choose  $v_i \in V(G) \setminus T$  such that it is the minimum distance from  $v_r$ .
2. Put  $v_{r+1} = v_i$  and let  $P = \{v_r \rightarrow v_i\}$ 
3. Put  $r = r + 1$  and repeat until  $T = V(G)$ 

Output: A Hamiltonian Cycle -> the path with the last vertex of  $P$  made adjacent to the first.
```

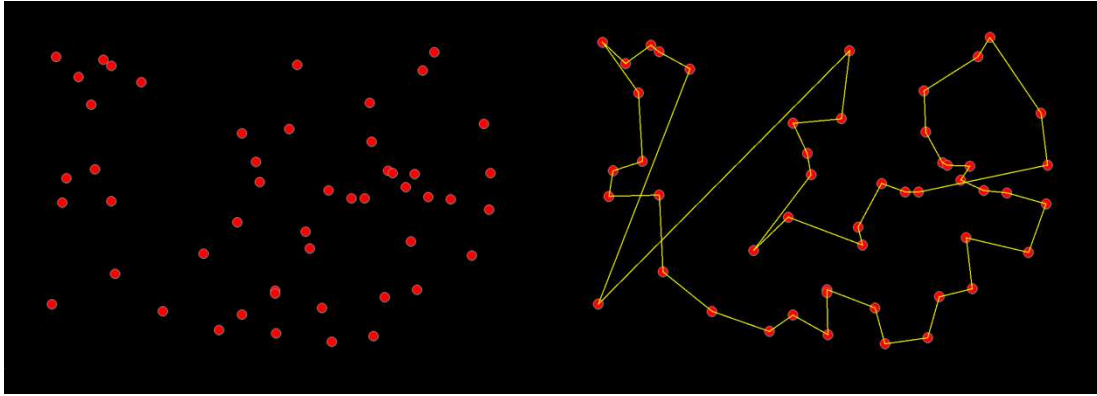


Figure 9: Nearest Neighbour algorithm performed on a generated city map with 50 cities and the shortest path found.

2.3.3. Greedy Heuristic Algorithm

The Greedy Heuristic algorithm is another simple and intuitive approach for solving the TSP. This algorithm builds a route by iteratively selecting the shortest available edge that connects the current city to an unvisited city. This algorithm also ensures that each city is visited exactly once. Algorithm returns to the starting city once all cities are included in the tour and completes the cycle.

This algorithm is operating in a way that it is making a locally optimal choice at each step. While this approach can provide quick solution it is often suboptimal because it does not consider the whole structure of the problem. In this way it is also quite like the Nearest Neighbour algorithm and the time complexity is $O(n^2)$ in average, but in some cases the worst-case scenario is $O(n^3)$.

Below is the high-level overview of the pseudocode for the algorithm implementation.

```

Initialize: Pick a starting city  $v_0$ 

Form a path  $P = \{v_0\}$  and mark  $v_0$  as visited.

Iteration:

1. Choose the next city  $v_i \in V(G) \setminus T$  that has the minimum distance from
any city in the current path  $P$ .

2. Add  $v_i$  to the path  $P$  and mark  $v_i$  as visited.

3. Repeat until all cities are visited.

Output: A Hamiltonian Cycle  $\rightarrow$  the path with the last vertex of  $P$  made
adjacent to the first.

```

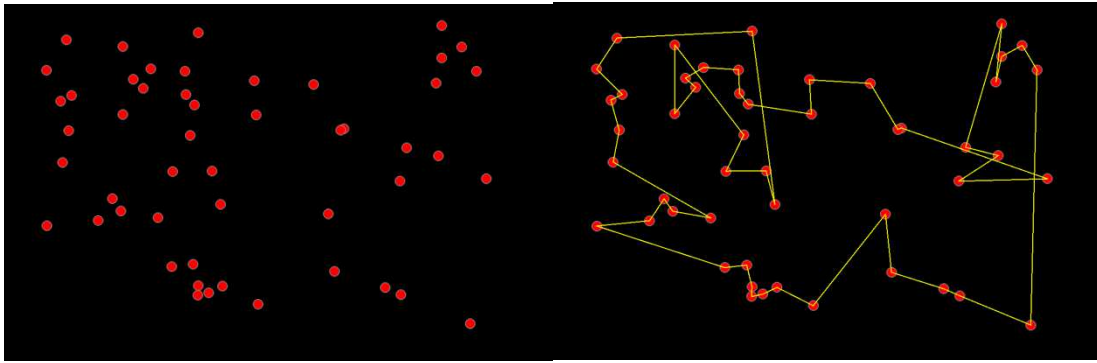


Figure 10: Greedy Heuristic algorithm performed on a generated city map with 50 cities and the shortest path found.

2.3.4. Genetic Algorithm

Genetic Algorithm (GA) is inspired by the principles of natural selection and genetics, which makes it suitable for solving the TSP with its optimization and search heuristic. GA operates by evolving a population of candidate solutions over multiple generations, using different operators like selection, crossover and mutation. GA offers ways to explore the search space and improve the quality of the solutions. [9]

GA is a powerful tool for tackling NP-hard problems like the TSP. It uses the concept of survival of the fittest and genetic inheritance to iteratively improve the population of solutions. Which makes it useful for solving complex optimization problems where the search space is large and other methods are computationally infeasible.

Total complexity of the implemented algorithm is $O(G \times P \times n)$. Where G stands for number of generations, larger instances increase a chance of finding a better solution but also increase the computational cost. P represents the population size, where the same rules apply as with the number of generations. The n stands for the number of cities in each TSP instance. Genetic algorithm implementation inside the TSP Explorer is well-fitted for smaller instances, which means the execution time is close to the greedy algorithms. In case of larger instances due to the time complexity, the implementation may not be efficient enough which will be studied and analysed in the next chapter.

Below is the high-level overview of the pseudocode which represents the algorithm used in implementation inside the TSP application.

Initialize: Generate an initial population of routes

Define population size, number of generations and mutation rate

Iteration:

1. Evaluate fitness of each route in the population (total route distance)
2. Select parents from the population using tournament selection
3. Create new population:
 - a. Perform ordered crossover between selected parents to generate children
 - b. Mutate children with a given mutation rate
4. Replace the old population with the new population
5. Repeat for the defined number of generations

Output: The best route in the final population with the shortest total distance.

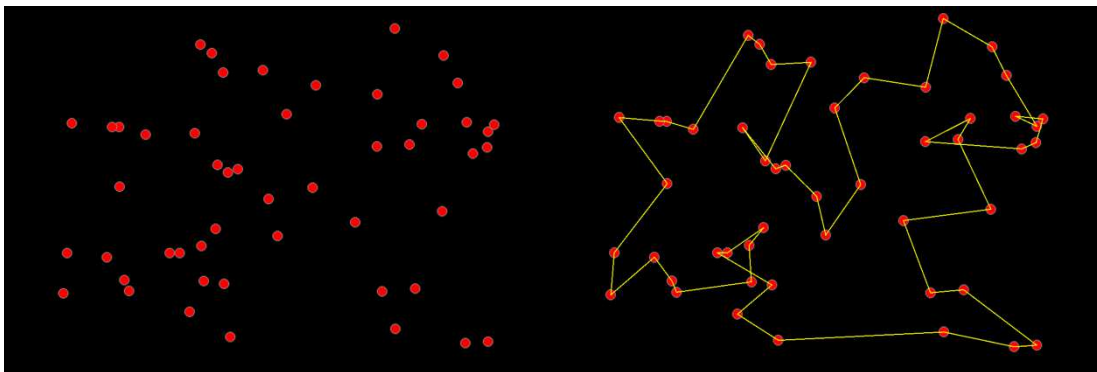


Figure 11: Genetic algorithm performed on a generated city map with 50 cities and the shortest path found.

3. TSP Explorer

3.1. User interface design and features

The practical part of this thesis was constructing and designing an application which would encompass theoretical foundation of the TSP along with the user-friendly way of seeing the whole concept of the problem. The application is built with a modular architecture to separate concerns and improve maintainability. Each algorithm is designed in its own class, allowing easy addition of new algorithms for future purposes.

One of the most important features of TSP Explorer is its ability to visualize the problem to the user and see how each of the algorithms perform. User can fully interact with the application by selecting the number of randomly generated cities along with the option to choose the algorithm which will perform the route finding. The cities are generated within the range of 1000km in the x-axis and the y-axis forming a square of 1000km². The user can also try different algorithms on the same city map or reset the generated cities and try new configurations.

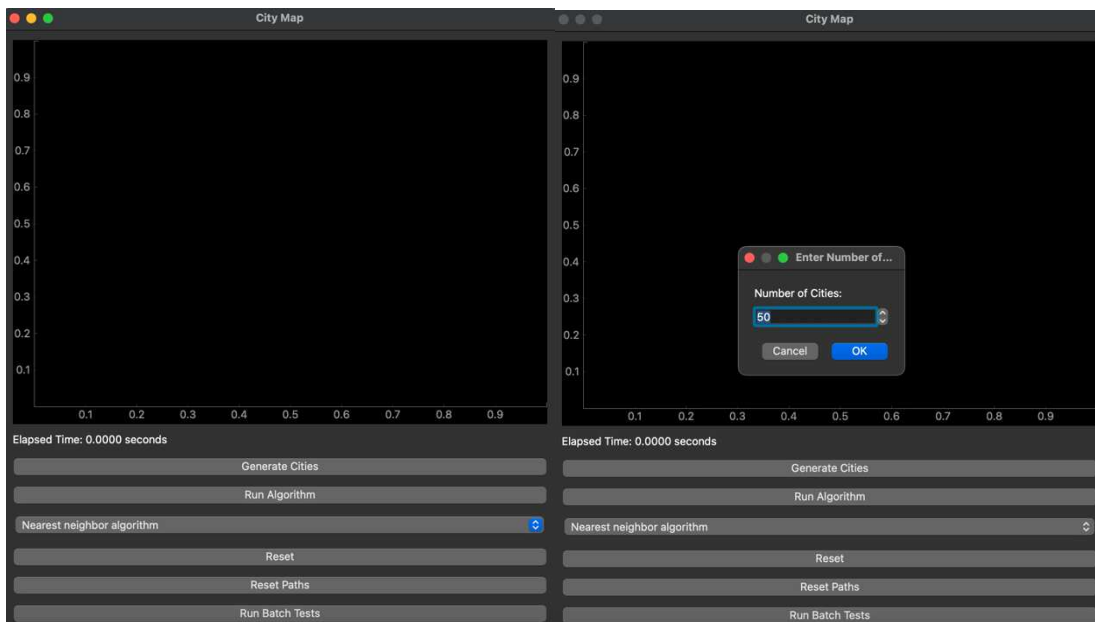


Figure 12: TSP Explorer GUI view and city generation option.

After running the application the user is prompted with the following view, after clicking the generate cities option the user can choose an arbitrary number of cities to be randomly generated.

After generating the cities, the user can see them represented as red dots on a city map. The user can now select from the dropdown menu which algorithm he wants to perform on the generated map.

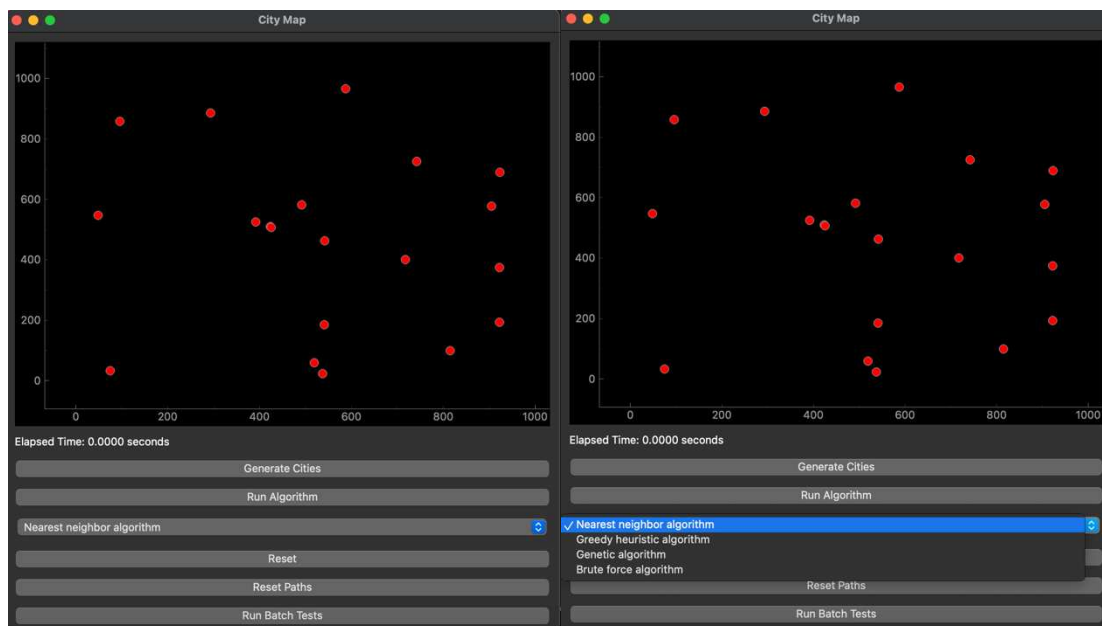


Figure 13: Plotted cities inside a city map and algorithm dropdown menu.

After the algorithm selection, the user can run the chosen algorithm on the city map. The application also offers multiple runs on the same map with the reset paths button. The generation of a new map is offered after clicking on the reset button and clicking on the generate cities option after which the process is repeated as stated before.

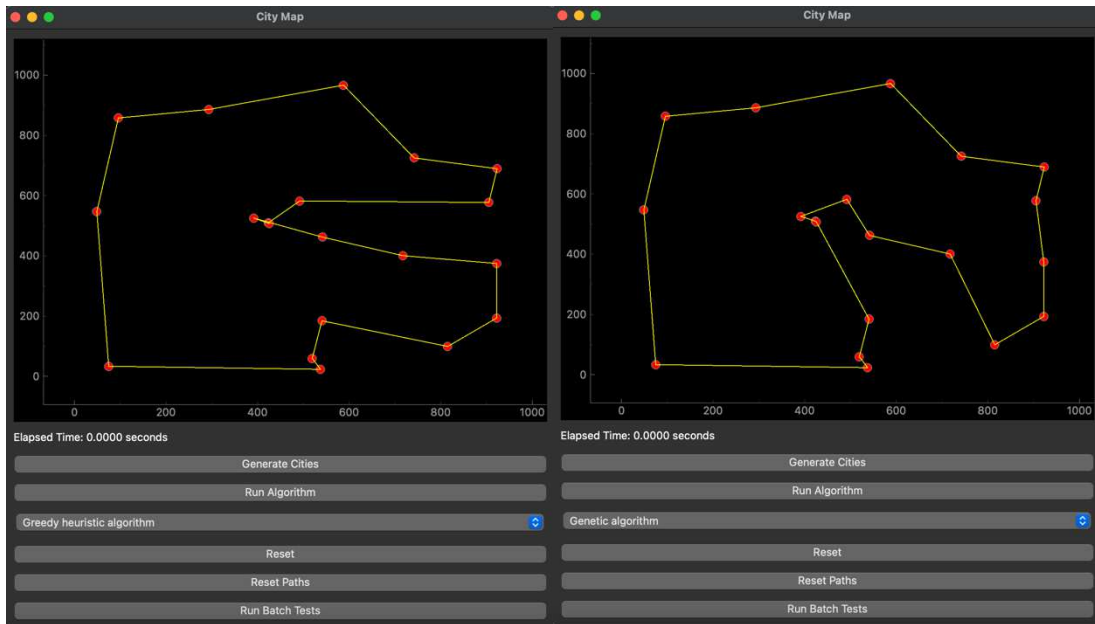


Figure 14: Greedy Heuristic and Genetic algorithm performed on a generated map.

We can notice that already some interesting solutions have been found, further analysis will be done in the next chapter.

3.2. Technical stack

Programming language used for creating TSP Explorer is Python chosen because of its extensive libraries and easy usage.

Libraries and frameworks:

- PyQt6: Used for building graphical user interface with its wide set of tools for creating a well formatted and clear interface.
- PyQtGraph: Used for plotting and visualizing the cities and their created paths, known for its fast performance and easy integration with PyQt6.
- NumPy: For numerical operations and efficient handling of matrices useful for calculating the distance matrix computations.
- Itertools and random: standart python modules used for randomness of generated cities and generating permutations for brute force algorithms.
- Pandas, seaborn and matplotlib: Used for creating visualizations of the results and creating plots.

3.3. Performance metrics and results

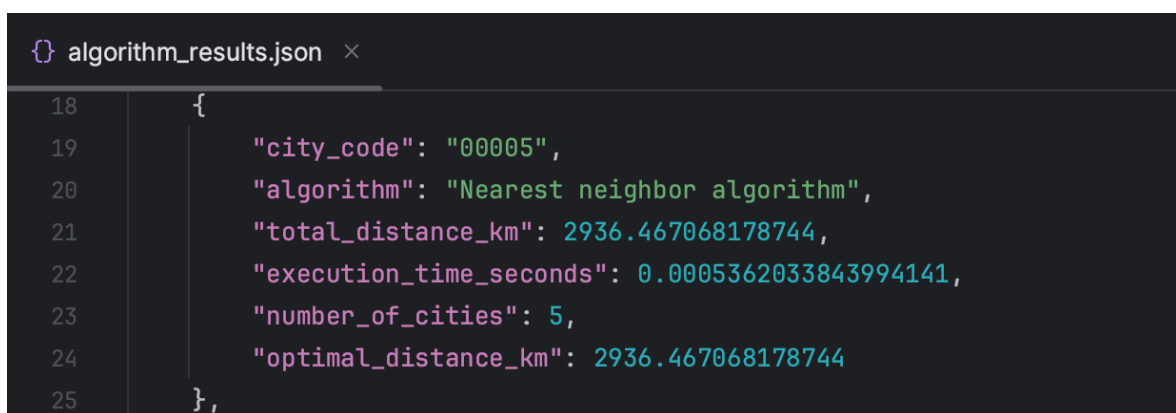
Evaluating the performance of all implemented algorithms is crucial for better understanding of their nature. The purpose of formulating the metrics is to compare and evaluate efficiency and effectiveness of each algorithm. In the TSP Explorer application, the following metrics are used:

Total distance traveled is the sum of distances between all the visited cities inside the route including the return to the starting city. This metric shows the quality of the solution, the goal of the algorithms is to minimize the total distance, so if the total distance covered is shorter it indicates a better solution.

Execution time is the time taken by an algorithm to find a solution, or in other words the shortest path from the starting city passing through all the cities and returning back to the starting point. Just like the total distance travelled this metric indicates the efficiency of an algorithm and its implementation.

Optimal distance is the shortest possible distance for a generated map, this metric is calculated using the Brute Force algorithm. This metric serves as a benchmark to compare the quality of other solutions but due to the limitations of the algorithm this is applicable only to smaller instances.

Results of the performed runs on different sets of cities are stored inside a JSON file.



```
18 {
19   "city_code": "00005",
20   "algorithm": "Nearest neighbor algorithm",
21   "total_distance_km": 2936.467068178744,
22   "execution_time_seconds": 0.0005362033843994141,
23   "number_of_cities": 5,
24   "optimal_distance_km": 2936.467068178744
25 },
```

Figure 15: Example of a JSON file for a performed run.

City code represents the instance of generated cities so that each instance can be differentiated. The JSON file also stores the information about the algorithm used, number of cities inside the map, execution time and an optimal distance in kilometres.

TSP Explorer also offers the batch input option which is used for automating the process of generating different instances and running different algorithms.

Example of a batch run:

```
def run_batch_tests(self):  
    city_sizes = [5,6,7,8,9]  
    num_runs = 5  
    algorithms = [  
        ("Nearest neighbor algorithm", NearestNeighborAlgorithm),  
        ("Greedy heuristic algorithm", GreedyHeuristicAlgorithm),  
        ("Genetic algorithm", GeneticAlgorithm),  
    ]
```

This implementation is very important for later analysis of solutions which the algorithms provided during the runs. Inside the `city_sizes` array, the number of cities generated is entered and the `num_runs` variable states the number of generations for those sizes.

4. Experimental analysis

The experimental analysis of this thesis will be formulated by generating and creating numerous instances of cities and performing algorithm runs on them. The results will be analyzed to conclude the efficiency and various metrics of those solutions. The results of the mentioned analysis are only considered within the scope of the TSP Explorer application and do not describe the general performance of those algorithms.

4.1. Testing instances

First part of testing will be performed as a batch input. We will study the behaviour of the algorithms on a small instance of cities (5-10). The Brute Force Algorithm will serve as a benchmark to study the optimality of the algorithms. Each algorithm will be run 20 times on a given number of cities.

The second part of testing will be performed only on the Greedy Heuristic, Nearest Neighbour and Genetic Algorithm on a much larger number of cities.

4.2. Results and discussion

The first test set consists of running a batch test input on 5, 6, 7, 8, 9 and 10 generated cities within the city map. Each number of cities is generated 20 times, and the algorithms are run on each created instance separately.

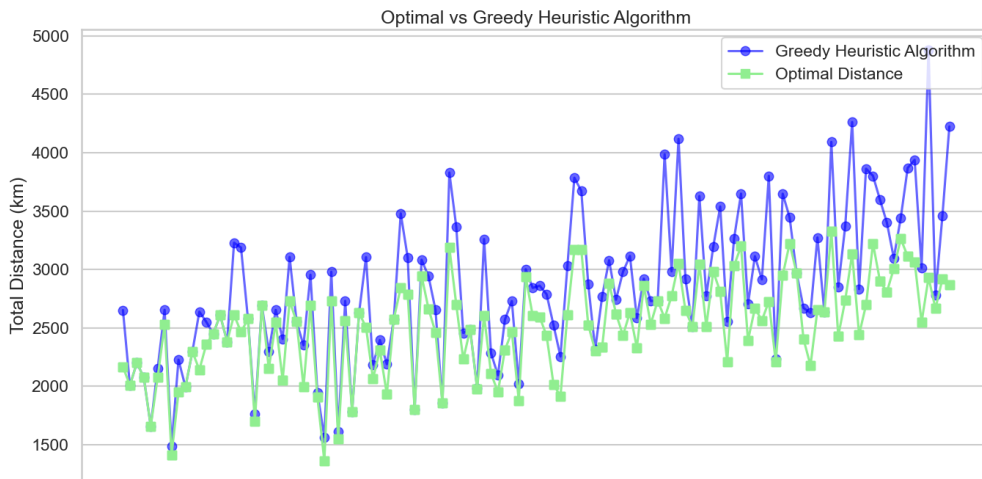


Figure 16: Optimal solution vs. Greedy Heuristic Algorithm.

Figure 16 shows the performance of the Greedy Heuristic Algorithm on a given test set. We can notice that as the number of cities increases, the route that the algorithm finds tends to deviate more from the optimal solution. For this test set, the algorithm found the optimal route 22 times or to be exact 18.3% of the time. The algorithm finds a solution with a high speed due to its greedy nature, so for cases where speed is important these results are somewhat acceptable.

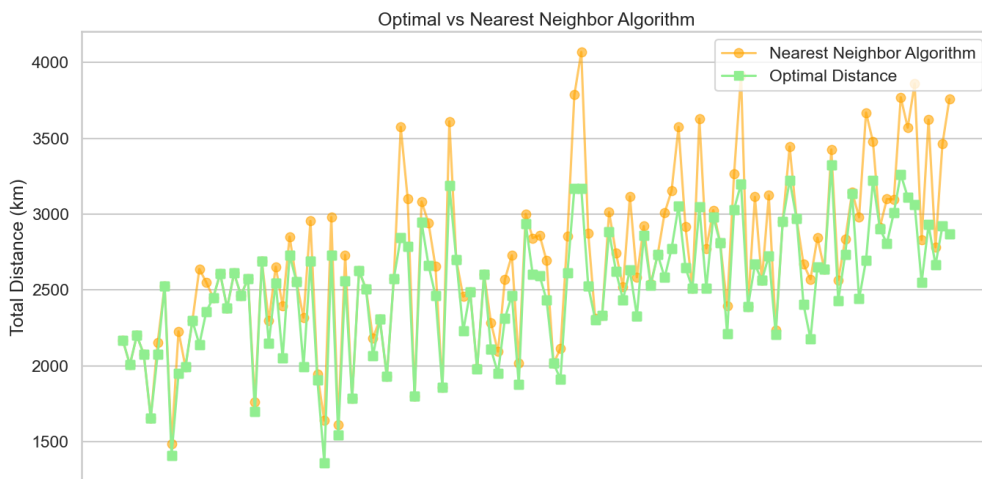


Figure 17: Optimal solution vs. Nearest Neighbour Algorithm.

Nearest Neighbour Algorithm performance is shown in **Figure 17**. We can see that algorithm has a similar behavior to the Greedy Heuristic. Due to their greedy nature, when the distances

and number of cities increase, they tend to sway further from the optimal solution. For this test set, the Nearest Neighbour Algorithm found the optimal solution 37 times out of 120, which is around 30%. What seems to be a small increase in terms of the Greedy Heuristic algorithm, so in the scope of this analysis if the number of cities is small the better option would be the Nearest Neighbour algorithm.

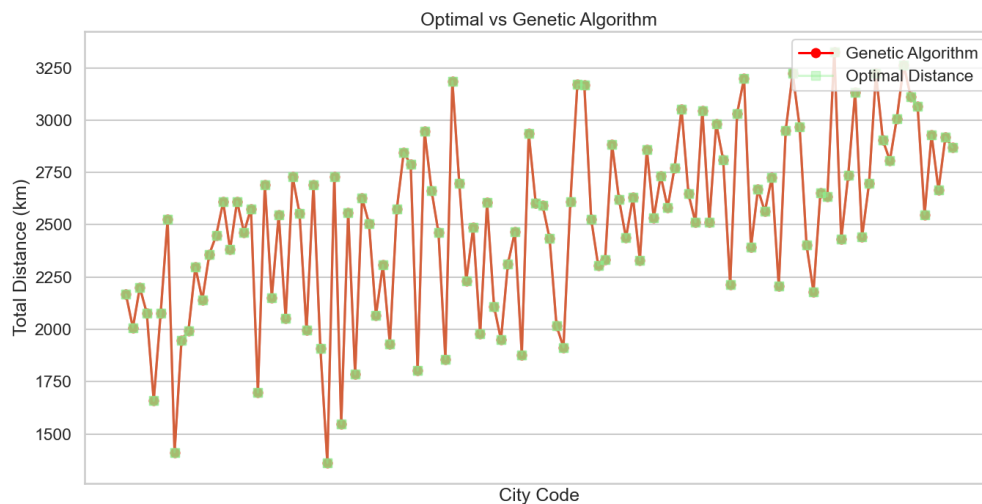


Figure 18: Optimal solution vs. Genetic Algorithm.

In **Figure 18**, the performance of the Genetic Algorithm on the given test set is shown. The algorithm found the optimal solution 120 times, making its approximation correct 100% of the time. It is important to note that this is the case for smaller instances. The algorithm is well-tuned for smaller number of cities, which explains its high effectiveness. The time taken for Genetic Algorithm to find the optimal solution is higher than in greedy algorithms. The difference is not significant enough for the trade off to sway in the other way. So, for smaller instances the better choice would be using the Genetic algorithm for finding the shortest route which always results in the optimal solution.

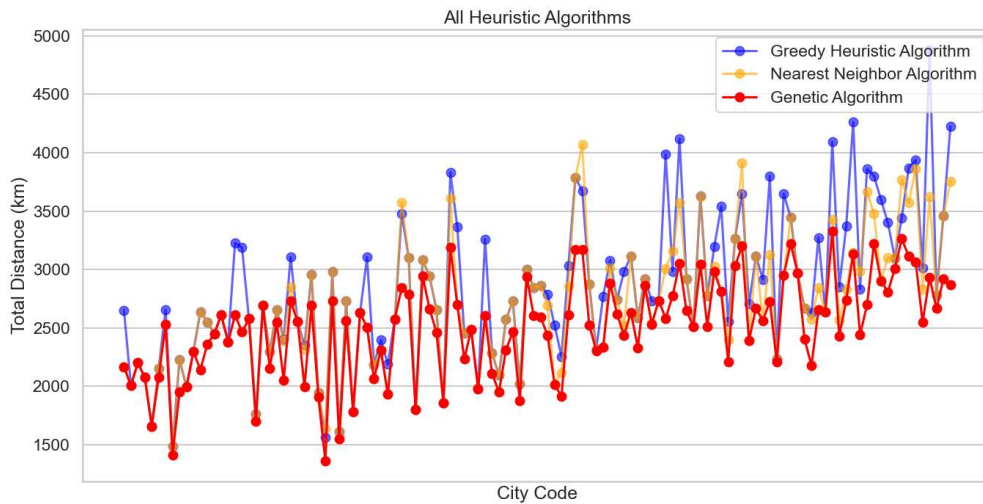


Figure 19: All heuristic algorithms performed on a test set.

In **Figure 19**, we can see a direct comparison of the given algorithms. On smaller instances, they all tend to be close, but as the distance increases along with the number of cities, the greedy algorithms do not perform as well as the Genetic Algorithm.

For the second part of the testing, I created a different test set. The test set consists of running a batch test input on 20, 40, 60, 80, and 100 generated cities within the city map. Each number of cities is generated 20 times, and the algorithms are run. Due to the factorial time complexity of the Brute Force Algorithm, I will not have the correct optimal path shown for this test set. This testing serves merely as a comparison of how the algorithms behave on larger instances.

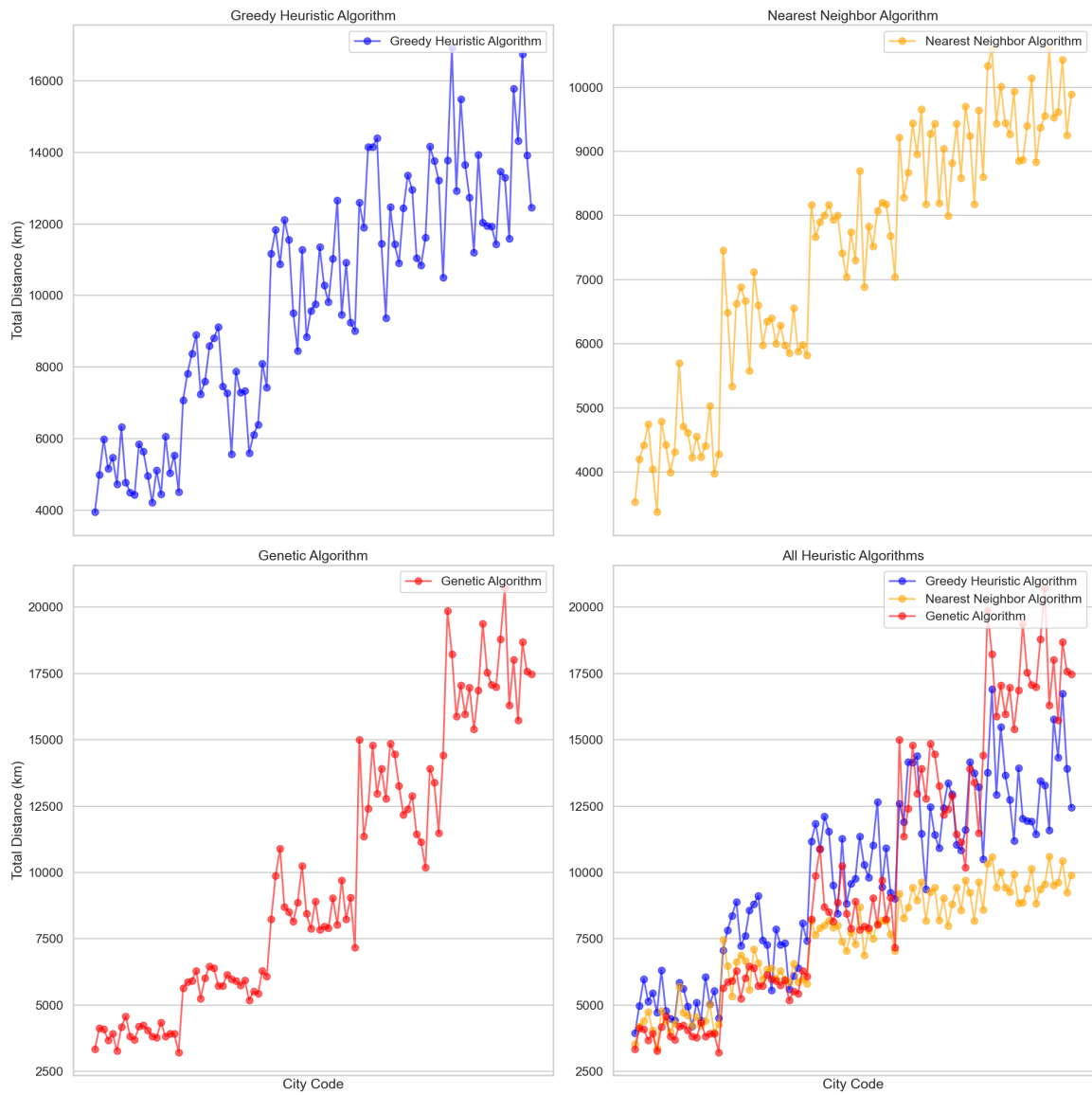


Figure 20: Second testing set results.

Figure 20 shows the performance of algorithms on the given test set. The results show the number of times the algorithm found the shortest route are following:

	20	40	60	80	100
Genetic algorithm	18/20	15/20	2/20	0/20	0/20
Greedy heuristic algorithm	0/20	1/20	0/20	0/20	0/20
Nearest Neighbour algorithm	2/20	4/20	18/20	20/20	20/20

Table 1: Second testing results.

From these results, we can see that the Genetic Algorithm is outperforming the other two algorithms on the smaller instances with 20 and 40 cities. On the other hand, as the number of cities increase the Genetic Algorithm starts to lose its effectiveness. For 60 or more cities the Nearest Neighbour algorithm gives the best performance. We can conclude that as the number of cities increases, this trend will likely remain the same. Given the nature of the Genetic Algorithm, there is still a possibility to increase its effectiveness by enhancing its structure and formulation, as it really depends on the specific implementation.

The additional test depicts where is the bound where the Genetic Algorithm starts to be less effective than the Nearest Neighbour Algorithm.

```
def run_batch_tests(self):
    city_sizes = [40, 42, 44, 46, 48, 50, 52, 54]
    num_runs = 100
    algorithms = [
        ("Genetic algorithm", GeneticAlgorithm),
        ("Nearest Neighbour algorithm", NearestNeighborAlgorithm)
    ]
```

Figure 21: The metric of the additional batch input testing.

So, for each city size ranging from [40 – 54] the algorithms will perform 100 distinctive runs.

Number of cities	40	42	44	46	48	50	52	54
Genetic algorithm	76%	73%	66%	64%	55%	40%	36%	34%
Nearest Neighbour algorithm	24%	27%	34%	34%	45%	60%	64%	66%

Table 2: Genetic algorithm vs. Nearest Neighbour algorithm

From the results, we can conclude that as the number of cities generated on the map increases, the Genetic Algorithm becomes proportionally less optimal. The turning point is around 50 cities, where the Nearest Neighbour Algorithm starts providing a better route.

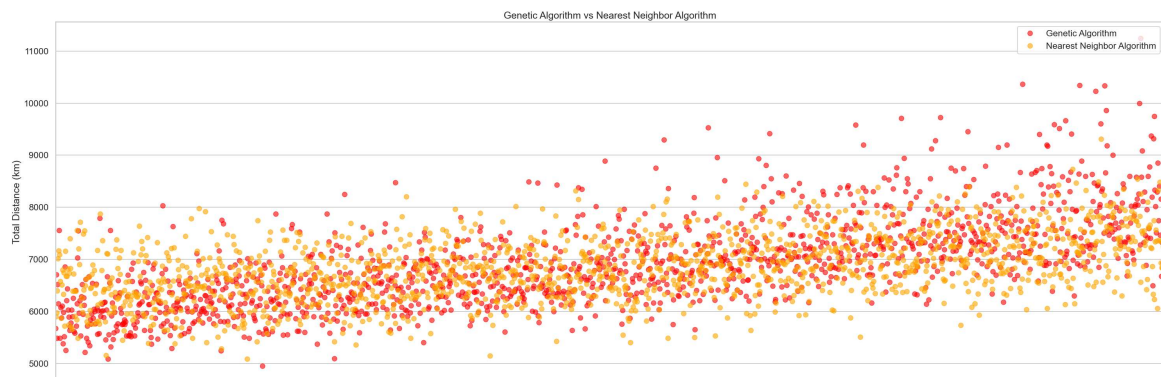


Figure 22: Total distance covered by Genetic Algorithm and Nearest Neighbour algorithm

Conclusion

Traveling Salesman Problem remains a challenging problem in the fields of mathematics and computer science. This thesis provided a brief theory introduction and various approaches to solving the TSP.

My analysis has shown that exact algorithms do provide optimal solutions but are infeasible for larger instances. Heuristic algorithms on the other hand, offer practical solutions within a reasonable time span making them suitable for real-world implementations on larger instances.

The Genetic Algorithm demonstrated better performance on smaller instances but in case when the number of cities increases the Nearest Neighbour algorithm performed more optimal. The reason for that is that the Genetic Algorithm is well-fitted for smaller number of cities.

The development and use of the built TSP Explorer application provided an interactive and fun platform to visualize and compare these algorithms. This tool serves as an educational resource and a practical demonstration of TSP solutions.

In conclusion, while no single algorithm provides a perfect solution for all instances of the TSP, the choice of the algorithm depends on the specific requirements of the problem at hand. Future work can focus on implementing more algorithms, and providing a hybrid method that will combine the strengths of multiple algorithms for further improvement on all instances of TSP.

Summary

Keywords: Traveling Salesman Problem, graph theory, heuristic algorithms, Genetic Algorithm, Nearest Neighbour Algorithm, Greedy Heuristic Algorithm, Brute Force Algorithm

The Traveling Salesman Problem is a well-known optimization challenge in computer science, seeking the shortest route to visit a set of cities and returning to the starting city leaving no city unvisited. This thesis provided an implementation of three heuristic algorithms, Greedy Heuristic, Genetic Algorithm and Nearest Neighbour Algorithm inside a TSP Explorer application. The application provided the visualization and performance metrics used for analyzing the results of the implemented algorithms. Heuristic methods, especially the Genetic Algorithm, offer effective solutions for smaller instances, while the Nearest Neighbour Algorithm performs better on larger instances due to the implementation of the Genetic Algorithm.

Literature

- [1] Flood, M. M. (1956). The Traveling-Salesman Problem. *Operations Research*, 4(1), 61-75.
- [2] Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., & Shmoys, D. B. (1985). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons. Used as a study material.
- [3] Dantzig, G. B., Fulkerson, D. R., & Johnson, S. M. (1954). Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America*, 2(4), 393-410.
- [4] Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press. Used as a study material.
- [5] Karp, R. M. (1972). Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations* (pp. 85-103).
- [6] Gutin, G., & Punnen, A. P. (2002). *The Traveling Salesman Problem and Its Variations*.
- [7] Reinelt, G. (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*.
- [8] Shabnam Sangwan. Literature Review on Travelling Salesman Problem: https://www.researchgate.net/publication/341371861_Literature_Review_on_Travelling_Salesman_Problem. Used as a study material, date accessed: 01.06.2024.
- [9] Vijay V. Vazirani. Approximation Algorithms: https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/Approximation%20Algorithms%20%5BVazirani%202010-12-01%5D.pdf. Used as a study material. Date accessed: 27.05.2024.
- [10] Corinne Brucato, M.S. University of Pittsburgh, 2013: THE TRAVELING SALESMAN PROBLEM master thesis.
- [11] Tpoint Tech. Discrete Mathematics - Travelling Salesman Problem: <https://www.javatpoint.com/discrete-mathematics-travelling-salesman-problem>. Used as a study material. Date accessed: 27.05.2024.
- [12] Kartik Rai, Lokesh Madan, Kislay Anand. Research Paper on Travelling Salesman Problem And its Solution Using Genetic Algorithm:. Student (B.tech VIII sem) Department of Computer science, Dronacharya College Of Engineering, Gurgaon-122506. © 2014 IJIRT | Volume 1 Issue 11 | ISSN: 2349-6002. IJIRT 101672 INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH IN TECHNOLOGY 103.
- [13] Domagoj Kovačević, Mario Krnić, Anamari Nakić, Mario Osvin Pavčević: *Diskretna matematika 1. Discrete mathemathics 1 textbook*.
- [14] 2024 StudySmarter GmbH: <https://www.studysmarter.co.uk/explanations/math/decision-maths/the-travelling-salesman-problem/>, date accessed: 04.06.2024.

- [15] Reinelt, G. (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*.