# Greedy Heuristics for Hamiltonian Cycle Problem

Evolutionary Computation Laboratory: Assignment 1

**151927** Samuel Janas, samuel.janas@student.put.poznan.pl
**151955** Bruno Urbaniak, bruno.urbaniak@student.put.poznan.pl
**151960** Patryk Maciejewski, patryk.maciejewski.1@student.put.poznan.pl

## 1 Problem Description and Implementation

In this project, we aim to solve an optimization problem involving a set of nodes located in a two-dimensional plane. Each node is defined by its $x$ and $y$ coordinates, along with an associated cost, representing the expense of including that node in our solution. The main goal is to form a Hamiltonian cycle - a closed loop that visits a subset of the nodes exactly once - while minimizing an objective function. This objective function consists of two key components:

1. The total cost of the selected nodes.
2. The total distance of the cycle, calculated as the sum of Euclidean distances between the selected nodes, rounded to the nearest integer.

We need to select exactly 50% of the available nodes. If the number of nodes is odd, the selection is rounded up. After selecting the nodes, the task is to form a Hamiltonian cycle that minimizes both the total path length and the total cost of the selected nodes.

### 1.1 Instance Data

Each instance of this problem is characterized by three key attributes for every node:

- $x$ and $y$ coordinates: These represent the node's location in the plane.
- Node cost: A numerical value representing the cost associated with including the node in the cycle.

Here is an example of the first five rows of two data sets (Instance A and Instance B):

| x | y | Cost |
|---|---|---|
| 1355 | 1796 | 496 |
| 2524 | 387 | 414 |
| 2769 | 430 | 500 |
| 3131 | 1199 | 1133 |
| 661 | 87 | 903 |

**Table 1:** First five rows of A instance data.

| **x** | **y** | **Cost** |
|---|---|---|
| 2249 | 1105 | 40 |
| 278 | 832 | 247 |
| 1459 | 1404 | 932 |
| 2116 | 1873 | 105 |
| 3337 | 1928 | 515 |

**Table 2:** First five rows of B instance data.

## 1.2 Implementation Details

We implemented the solution in two parts: the core algorithms are written in **Go** for efficient computation, and **Python** is used for visualizing the results. Go was chosen because of its concurrency support and fast execution time, which is crucial for handling larger datasets and running multiple iterations efficiently. Python, on the other hand, was selected for its rich visualization libraries, such as Matplotlib, which enable us to easily generate visual insights from the computed solutions.

The system is divided into the following key components:

### 1.2.1 Data Loading and Processing

The first step in our system is to load the instance data from a CSV file, which contains the node positions (x, y coordinates) and the associated costs. This is implemented in Go with the `LoadNodes` function. Each row in the CSV file corresponds to a node, and we store these nodes in a slice of `Node` structs, where each struct holds:

- The node's ID.
- The $x$ and $y$ coordinates of the node.
- The cost associated with that node.

The `LoadNodes` function reads the CSV file using the `encoding/csv` package, parses the coordinates and costs, and returns the list of nodes. This list is later used to generate the distance matrix and compute the total cost of each solution.

### 1.2.2 Distance Matrix Calculation

Once the nodes are loaded, we calculate the **distance matrix** using the `CalculateCostMatrix` function. The distance between any two nodes is calculated using the **Euclidean distance formula**:

$$d(i,j) = \left\lceil \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right\rceil$$

The distance is rounded to the nearest integer, and the cost of the destination node is added to the computed distance. This matrix is essential for all of our optimization algorithms, as it allows us to compute the total travel cost between selected nodes.

The matrix calculation process is structured as follows:

1. Iterate over all pairs of nodes.
2. For each pair, calculate the Euclidean distance.
3. Add the cost of the destination node.
4. Store the computed value in the distance matrix.

### 1.2.3 Optimization Algorithms

The optimization methods are implemented as heuristic-based algorithms in Go. Each algorithm selects a subset of nodes (50% of the total) and tries to form a **Hamiltonian cycle** that minimizes the objective function. The following heuristics are used:

- **Random Solution**: This method randomly selects 50% of the nodes and arranges them in a cycle.
- **Nearest Neighbor (End Insertion)**: This heuristic adds the nearest unvisited neighbor to the end of the current path, forming a cycle.
- **Nearest Neighbor (Flexible Insertion)**: Here, the nearest unvisited neighbor is inserted at the position that minimizes the overall cycle length.
- **Greedy Cycle**: This method starts with two nodes and iteratively inserts nodes at the position that causes the least increase in the total cycle length.

The core logic for each method follows a similar structure:

1. Select a random starting node.
2. Iteratively add nodes based on the specific heuristic.
3. Form a cycle by connecting the last node back to the first.

Each method runs multiple times (200 iterations per instance) to explore a wide range of possible solutions.

### 1.2.4 Fitness Evaluation

After forming a cycle, the **Fitness function** evaluates the solution's quality by calculating the total cost, which includes the distances between the selected nodes and their individual costs. The formula for the total cost is:

$$C = \sum_{i=1}^{n} d(i, i+1) + \sum_{i=1}^{n} c(i)$$

Where:

- $d(i, i+1)$ is the distance between node $i$ and node $i+1$, with the final node connected back to the first to form a closed cycle.
- $c(i)$ is the cost of node $i$.

### 1.2.5 Logging and Output

The Go implementation generates a log of the results, which includes:

- The best solution found.
- The worst solution.
- The average fitness across all iterations.

Each solution is stored as a list of node indices, and the fitness values are logged for analysis.

### 1.2.6 Python for Visualization

After computing the best and worst solutions in Go, the results are passed to Python for visualization. The `plot_solution` function in Python generates 2D scatter plots of the nodes, highlighting the selected nodes in the Hamiltonian cycle. Each node's size is proportional to its cost, and the edges between the nodes are color-coded based on the Euclidean distance between them. The color scale helps visualize which parts of the cycle contribute the most to the total cost.

### 1.2.7 Visualization Steps:

1. **Scatter Plot**: All nodes are plotted on the 2D plane, with unused nodes shown with reduced opacity.
2. **Cycle Path**: The edges between nodes in the selected cycle are plotted, with colors representing the distance (cost) of each edge.
3. **Cost Normalization**: The node sizes and edge colors are normalized based on their respective costs to improve visual clarity.

### 1.2.8 Integration between Go and Python

The integration between Go and Python is seamless due to the use of intermediate result files (in JSON format). After each run in Go, the results are serialized into a JSON file, which is then processed by Python. This allows for easy handoff between the computation (Go) and visualization (Python) stages, making it efficient to manage large-scale problem instances.

### 1.2.9 Command-Line Execution

The Go code can be executed from the command line by specifying the instance file and the optimization method. The system runs all methods on multiple instances and generates logs and visualizations for each method. For example, the following command runs the "nearest neighbor flexible" algorithm on the A instance:

```
go run main.go data/tspA.csv nearest_neighbor_flexible
```

This command triggers the execution of the specified method on the given dataset and logs the results. Each method runs multiple times (200 iterations) to ensure a diverse set of solutions is explored, and the best, worst, and average solutions are saved for analysis. The logs include the node indices of the best and worst Hamiltonian cycles found, along with the respective total costs. Python scripts are then called to generate visualizations of these cycles, providing insights into the performance of each heuristic.

#### 1.2.10 Scripts for Reporting and Validation

Two Python scripts are used to streamline the process of generating reports and validating solutions:

1. **Results Processing Script**: This script reads the 'results.json' files generated by the optimization algorithms, extracting best and worst fitness values, as well as the node indices for each solution. It then automatically generates LaTeX files for each method and instance, formatted for direct inclusion in the final report. The script also logs the process for completeness.

2. **Solution Checker Automation Script**: This script automates the verification of the best solutions by integrating Python with an Excel-based solution checker using the 'xlwings' library. It inserts the solutions into Excel, triggers recalculations, and retrieves the recalculated fitness values. The results are compared to the fitness values from the optimization algorithms, with the comparison output saved as a LaTeX table for inclusion in the report.

## 2 Pseudocode for Heuristic Methods

**Problem 1: Random Solution Algorithm**

This algorithm randomly selects 50% of the nodes and creates a Hamiltonian cycle with them.

1. **Input**: A distance matrix distanceMatrix representing distances between nodes.
2. **Output**: A list of node indices representing a Hamiltonian cycle.
3. **Procedure**:
   (i) Calculate the number of nodes to select $\text{numToSelect} = \left\lceil \frac{\text{numNodes}}{2} \right\rceil$.
   (ii) Randomly shuffle the node indices.
   (iii) Select the first numToSelect nodes.
   (iv) Return the selected nodes as a cycle.

**Problem 2: Greedy Cycle Algorithm**

This algorithm builds a Hamiltonian cycle by iteratively adding the node that causes the smallest increase in the total cycle length.

1. **Input**: A distance matrix distanceMatrix representing distances between nodes.
2. **Output**: A list of node indices representing a Hamiltonian cycle.

3. **Procedure**:
   - (i) Select a random starting node and add it to the cycle.
   - (ii) Mark the node as visited.
   - (iii) While the number of nodes in the cycle is less than numToSelect:
     - (a) For each unvisited node, calculate the increase in cycle length if it were inserted between every pair of consecutive nodes in the current cycle.
     - (b) Insert the node that causes the smallest increase in the total cycle length at the optimal position.
     - (c) Mark the node as visited.
   - (iv) Return the list of selected nodes as the Hamiltonian cycle.

**Problem 3: Nearest Neighbor (End Insertion)**

This algorithm builds a Hamiltonian cycle by adding the nearest unvisited neighbor to the end of the current path.

1. **Input**: A distance matrix distanceMatrix representing distances between nodes.
2. **Output**: A list of node indices representing a Hamiltonian cycle.
3. **Procedure**:
   - (i) Select a random starting node and add it to the path.
   - (ii) Mark the node as visited.
   - (iii) While the number of nodes in the path is less than numToSelect:
     - (a) Identify the nearest unvisited neighbor of the last node in the current path.
     - (b) Add the nearest neighbor to the end of the path.
     - (c) Mark the node as visited.
   - (iv) Return the list of selected nodes as a Hamiltonian cycle.

**Problem 4: Nearest Neighbor (Flexible Insertion)**

This algorithm builds a Hamiltonian cycle by adding the nearest unvisited neighbor at the position that causes the least increase in the total cycle length.

1. **Input**: A distance matrix distanceMatrix representing distances between nodes.
2. **Output**: A list of node indices representing a Hamiltonian cycle.
3. **Procedure**:
   - (i) Select a random starting node and add it to the path.
   - (ii) Mark the node as visited.
   - (iii) While the number of nodes in the path is less than numToSelect:

   (a) For each unvisited node, calculate the increase in total cycle length for every possible insertion position.
   (b) Add the nearest unvisited node at the position that minimizes the total distance increase.
   (c) Mark the node as visited.

(iv) Return the list of selected nodes as a Hamiltonian cycle.

## 3 Results of Computational Experiment

We conducted a series of experiments for each instance (A and B) using four heuristic methods: **Greedy Cycle**, **Nearest Neighbor (End Insertion)**, **Nearest Neighbor (Flexible Insertion)**, and **Random Solution**. For each method, we ran 200 iterations and recorded the best, worst, and average fitness values (objective function values).

Below are the results of the experiment, showing the minimum (best), maximum (worst), and average values of the objective function for each method and instance.

### 3.1 Instance A Results

| **Method** | **Best Fitness** | **Worst Fitness** | **Average Fitness** |
|---|---|---|---|
| Greedy Cycle | 71488 | 74410 | 72634.87 |
| Nearest Neighbor (End Insertion) | 83182 | 89433 | 85108.51 |
| Nearest Neighbor (Flexible) | 71179 | 75450 | 73178.55 |
| Random Solution | 225047 | 290346 | 264724.25 |

**Table 3:** Updated results for Instance A showing best, worst, and average fitness values across 200 iterations for each heuristic method.

The updated results for Instance A indicate that the **Greedy Cycle** and **Nearest Neighbor (Flexible)** methods still performed better than the other heuristics. The **Random Solution** method continues to perform the worst in terms of both the best and average fitness values.

### 3.2 Instance B Results

| **Method** | **Best Fitness** | **Worst Fitness** | **Average Fitness** |
|---|---|---|---|
| Greedy Cycle | 49001 | 57262 | 51397.91 |
| Nearest Neighbor (End Insertion) | 52319 | 59030 | 54390.43 |
| Nearest Neighbor (Flexible) | 44417 | 53438 | 45870.25 |
| Random Solution | 190200 | 235839 | 213180.80 |

**Table 4:** Updated results for Instance B showing best, worst, and average fitness values across 200 iterations for each heuristic method.

For Instance B, the updated results reveal a similar pattern. The **Nearest Neighbor (Flexible)** method achieved the best fitness values overall, while the **Random Solution** again showed the poorest performance in both best and average fitness values.

## 4 2D Visualization of the Best Solutions

For each instance and method, we generated 2D visualizations of the best solutions. The selected nodes are plotted in a scatter plot where:

- Node size is proportional to the cost of the node.
- The color of the connecting edges represents the Euclidean distance between the nodes, with a color scale from light (short distance) to dark (long distance).

The figures below display the best solution for each heuristic and instance.

### 4.1 Instance A Best Solutions

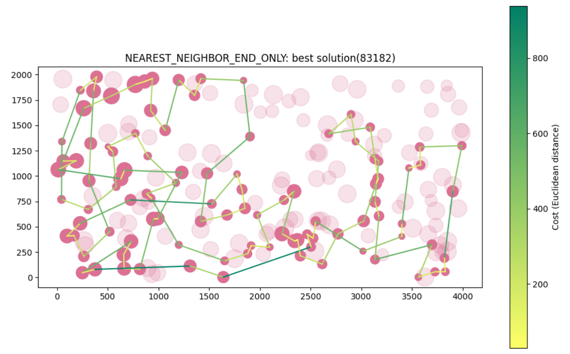

**Figure 1:** Best solution for Nearest Neighbor End Only method on A instance (Fitness: 83182).

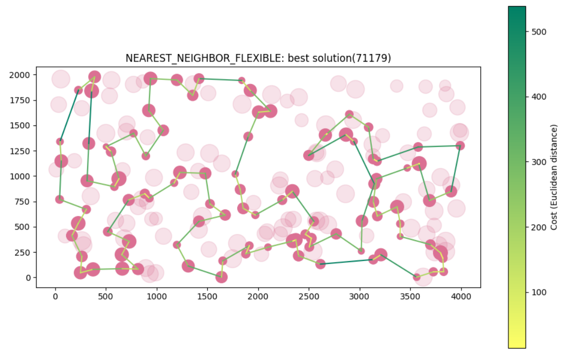

**Figure 2:** Best solution for Nearest Neighbor Flexible method on A instance (Fitness: 71179).

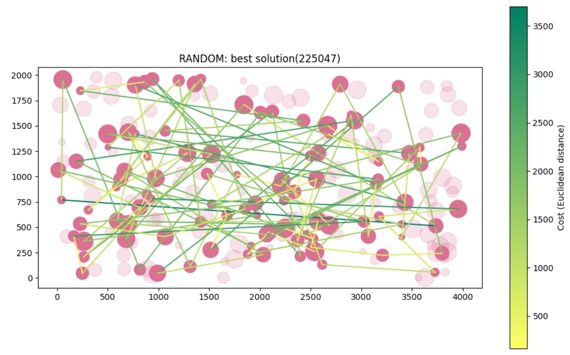

**Figure 3:** Best solution for Random Solution method on A instance (Fitness: 225047).

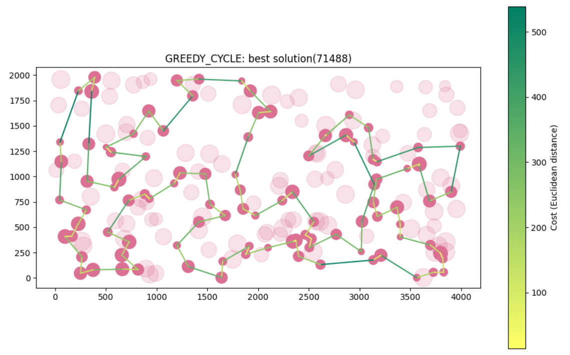

**Figure 4:** Best solution for Greedy Cycle method on A instance (Fitness: 71488).

### 4.2 Instance B Best Solutions

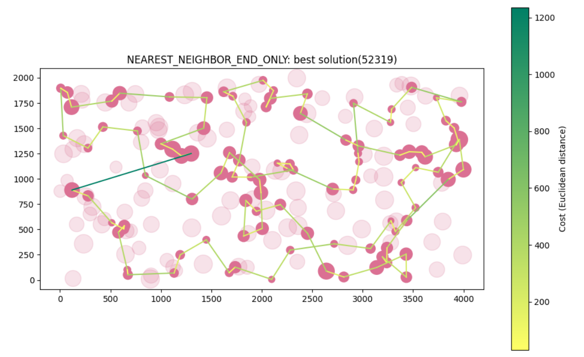

**Figure 5:** Best solution for Nearest Neighbor End Only method on B instance (Fitness: 52319).

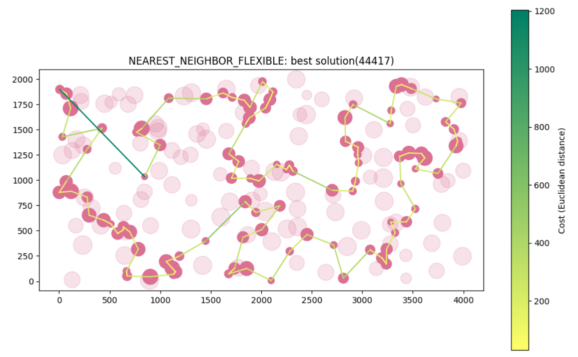

**Figure 6:** Best solution for Nearest Neighbor Flexible method on B instance (Fitness: 44417).

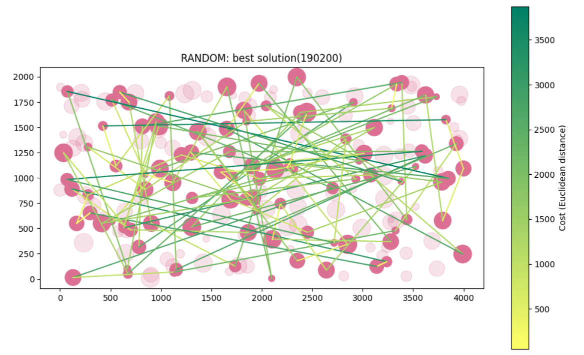

**Figure 7:** Best solution for Random Solution method on B instance (Fitness: 190200).

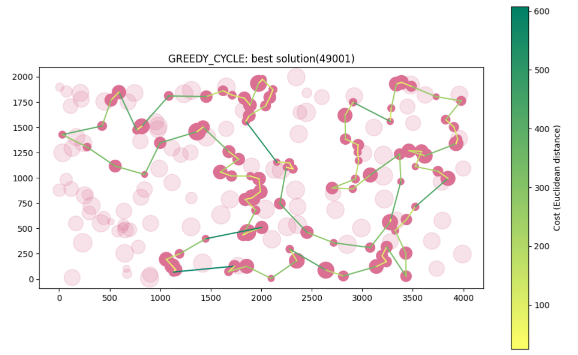

**Figure 8:** Best solution for Greedy Cycle method on B instance (Fitness: 49001).

## 5 Best Solutions for Each Instance and Method

Below is a list of the best solutions for each method and instance, presented as a list of node indices (starting from 0). The data is imported directly from the ‘results.json‘ files.

### 5.1 Instance A Best Solutions

#### 5.1.1 Results for Greedy Cycle

**Best Fitness: 71488**

117, 143, 183, 89, 186, 23, 137, 176, 80, 79, 63, 94, 124, 152, 97, 1, 101, 2, 120, 129, 55, 49, 102, 148, 9, 62, 144, 14, 178, 106, 165, 90, 81, 196, 40, 119, 185, 52, 57, 92, 179, 145, 78, 31, 56, 113, 175, 171, 16, 25, 44, 75, 86, 26, 100, 53, 154, 180, 135, 70, 127, 123, 162, 133, 151, 51, 118, 59, 65, 116, 43, 184, 35, 84, 112, 4, 190, 10, 177, 30, 54, 48, 160, 34, 146, 22, 18, 108, 69, 159, 181, 42, 5, 115, 41, 193, 139, 68, 46, 0

**Worst Fitness: 74410**

56, 113, 175, 171, 16, 25, 44, 120, 2, 75, 86, 26, 94, 63, 79, 80, 137, 23, 89, 183, 143, 176, 133, 151, 51, 118, 59, 115, 46, 68, 139, 193, 41, 5, 42, 181, 159, 69, 108, 18, 22, 146, 34, 160, 48, 54, 30, 177, 10, 190, 4, 112, 84, 35, 184, 43, 116, 65, 131, 149, 162, 123, 127, 70, 135, 154, 180, 53, 100, 101, 1, 97, 152, 129, 55, 49, 102, 148, 9, 62, 144, 14, 3, 178, 106, 165, 90, 81, 196, 40, 119, 185, 52, 57, 92, 179, 78, 145, 31, 188

**Average Fitness: 72634.87**

### 5.1.2 Results for Nearest Neighbor End Only

**Best Fitness: 83182**

124, 94, 63, 53, 180, 154, 135, 123, 65, 116, 59, 115, 139, 193, 41, 42, 160, 34, 22, 18, 108, 69, 159, 181, 184, 177, 54, 30, 48, 43, 151, 176, 80, 79, 133, 162, 51, 137, 183, 143, 0, 117, 46, 68, 93, 140, 36, 163, 199, 146, 195, 103, 5, 96, 118, 149, 131, 112, 4, 84, 35, 10, 190, 127, 70, 101, 97, 1, 152, 120, 78, 145, 185, 40, 165, 90, 81, 113, 175, 171, 16, 31, 44, 92, 57, 106, 49, 144, 62, 14, 178, 52, 55, 129, 2, 75, 86, 26, 100, 121

**Worst Fitness: 89433**

30, 34, 160, 42, 193, 41, 139, 115, 59, 65, 116, 123, 135, 180, 53, 63, 176, 80, 151, 133, 154, 101, 97, 1, 152, 120, 78, 145, 185, 40, 165, 90, 81, 113, 175, 171, 16, 31, 44, 92, 57, 106, 49, 144, 62, 14, 178, 52, 55, 129, 2, 75, 86, 94, 79, 137, 183, 143, 0, 117, 46, 68, 18, 22, 159, 181, 184, 177, 54, 48, 43, 149, 162, 51, 118, 72, 127, 70, 112, 4, 84, 35, 131, 47, 5, 96, 146, 195, 103, 108, 69, 93, 140, 36, 163, 199, 20, 134, 67, 198

**Average Fitness: 85108.51**

### 5.1.3 Results for Nearest Neighbor Flexible

**Best Fitness: 71179**

68, 46, 115, 139, 193, 41, 5, 42, 181, 159, 69, 108, 18, 22, 146, 34, 160, 48, 54, 177, 10, 190, 4, 112, 84, 35, 184, 43, 116, 65, 59, 118, 51, 151, 133, 162, 123, 127, 70, 135, 180, 154, 53, 100, 26, 86, 75, 44, 25, 16, 171, 175, 113, 56, 31, 78, 145, 179, 92, 57, 52, 185, 119, 40, 196, 81, 90, 165, 106, 178, 14, 144, 62, 9, 148, 102, 49, 55, 129, 120, 2, 101, 1, 97, 152, 124, 94, 63, 79, 80, 176, 137, 23, 186, 89, 183, 143, 0, 117, 93

**Worst Fitness: 75450**

16, 171, 175, 113, 56, 13, 31, 78, 145, 179, 92, 57, 52, 185, 119, 40, 196, 81, 90, 165, 106, 178, 14, 144, 62, 9, 148, 102, 49, 55, 129, 25, 44, 120, 2, 75, 101, 1, 152, 97, 26, 100, 86, 53, 94, 63, 79, 80, 176, 137, 23, 186, 89, 183, 143, 117, 0, 51, 151, 162, 133, 180, 154, 135, 70, 127, 123, 149, 131, 65, 116, 43, 184, 35, 84, 112, 4, 190, 10, 177, 54, 48, 160, 34, 181, 42, 59, 118, 115, 46, 68, 139, 41, 193, 159, 146, 22, 18, 69, 108

**Average Fitness: 73178.55**

### 5.1.4 Results for Random

**Best Fitness: 225047**

5, 170, 116, 11, 76, 37, 157, 34, 134, 112, 23, 40, 97, 93, 86, 186, 177, 77, 103, 63, 147, 60, 42, 160, 115, 191, 127, 176, 130, 73, 91, 139, 28, 87, 12, 107, 78, 8, 46, 15, 18, 140, 94, 145, 154, 182, 150, 165, 119, 180, 195, 155, 29, 158, 96, 1, 99, 90, 136, 166, 141, 142, 148, 124, 75, 171, 92, 98, 65, 10, 54, 129, 51, 133, 194, 21, 56, 25, 101, 74, 162, 117, 193, 52, 151, 62, 106, 61, 197, 36, 55, 82, 144, 26, 111, 122, 48, 24, 79, 143

**Worst Fitness: 290346**

61, 196, 104, 68, 35, 147, 15, 85, 93, 152, 142, 185, 108, 114, 161, 45, 62, 144, 98, 84, 164, 179, 170, 26, 174, 109, 124, 77, 70, 79, 92, 121, 177, 140, 83, 123, 181, 194, 44, 21, 18, 127, 132, 75, 145, 28, 169, 136, 148, 184, 56, 193, 134, 17, 102, 97, 198, 126, 74, 187, 141, 106, 43, 95, 52, 99, 120, 130, 36, 72, 49, 146, 10, 125, 153, 115, 159, 171, 76, 103, 38, 188, 199, 7, 86, 12, 160, 117, 168, 69, 73, 33, 175, 162, 110, 156, 47, 2, 22, 119

**Average Fitness: 264724.25**

## 5.2 Instance B Best Solutions

### 5.2.1 Results for Greedy Cycle

**Best Fitness: 49001**

134, 139, 11, 182, 138, 33, 160, 144, 56, 104, 8, 21, 87, 82, 177, 5, 45, 142, 78, 175, 61, 36, 91, 141, 97, 77, 146, 187, 165, 127, 89, 103, 137, 114, 113, 194, 166, 172, 179, 185, 99, 130, 22, 66, 94, 47, 148, 60, 20, 28, 149, 4, 140, 183, 152, 170, 34, 55, 18, 62, 124, 143, 106, 128, 95, 86, 176, 180, 163, 153, 81, 111, 0, 35, 109, 29, 168, 195, 145, 15, 3, 70, 161, 13, 132, 169, 188, 6, 147, 191, 90, 10, 133, 122, 63, 135, 131, 121, 51, 85

**Worst Fitness: 57262**

100, 72, 44, 133, 10, 178, 115, 147, 71, 120, 51, 98, 74, 118, 116, 121, 112, 19, 151, 73, 164, 54, 31, 136, 45, 5, 177, 25, 104, 138, 182, 139, 168, 195, 145, 15, 3, 70, 161, 13, 132, 169, 188, 6, 134, 43, 11, 49, 33, 160, 29, 109, 35, 0, 111, 144, 56, 8, 82, 87, 21, 77, 81, 153, 163, 103, 89, 127, 187, 97, 141, 91, 61, 36, 142, 78, 175, 162, 80, 190, 193, 117, 198, 156, 1, 16, 27, 38, 131, 125, 191, 90, 122, 135, 32, 102, 63, 40, 107, 17

**Average Fitness: 51397.906**

### 5.2.2 Results for Nearest Neighbor End Only

**Best Fitness: 52319**

16, 1, 117, 31, 54, 193, 190, 80, 175, 5, 177, 36, 61, 141, 77, 153, 163, 176, 113, 166, 86, 185, 179, 94, 47, 148, 20, 60, 28, 140, 183, 152, 18, 62, 124, 106, 143, 0, 29, 109, 35, 33, 138, 11, 168, 169, 188, 70, 3, 145, 15, 155, 189, 34, 55, 95, 130, 99, 22, 66, 154, 57, 172, 194, 103, 127, 89, 137, 114, 165, 187, 146, 81, 111, 8, 104, 21, 82, 144, 160, 139, 182, 25, 121, 90, 122, 135, 63, 40, 107, 100, 133, 10, 147, 6, 134, 51, 98, 118, 74

**Worst Fitness: 59030**

34, 18, 62, 124, 106, 176, 113, 166, 86, 185, 179, 94, 47, 148, 20, 60, 28, 140, 183, 152, 155, 3, 70, 169, 188, 168, 195, 145, 15, 13, 132, 6, 147, 90, 121, 117, 31, 54, 193, 190, 80, 175, 5, 177, 36, 61, 141, 77, 153, 163, 89, 127, 103, 114, 137, 194, 180, 26, 165, 187, 146, 81, 111, 8, 33, 29, 0, 109, 35, 143, 159, 55, 95, 130, 99, 22, 66, 154, 57, 172, 52, 48, 76, 75, 93, 88, 128, 83, 170, 184, 189, 167, 69, 11, 138, 182, 139, 134, 51, 122

**Average Fitness: 54390.43**

#### 5.2.3 Results for Nearest Neighbor Flexible

**Best Fitness: 44417**

40, 107, 100, 63, 122, 135, 38, 27, 16, 1, 156, 198, 117, 193, 31, 54, 73, 136, 190, 80, 162, 175, 78, 142, 45, 5, 177, 104, 8, 111, 82, 21, 61, 36, 91, 141, 77, 81, 153, 187, 163, 89, 127, 103, 113, 176, 194, 166, 86, 95, 130, 99, 22, 185, 179, 66, 94, 47, 148, 60, 20, 28, 149, 4, 140, 183, 152, 170, 34, 55, 18, 62, 124, 106, 143, 35, 109, 0, 29, 160, 33, 138, 11, 139, 168, 195, 145, 15, 3, 70, 13, 132, 169, 188, 6, 147, 191, 90, 51, 121

**Worst Fitness: 53438**

40, 107, 100, 63, 122, 133, 10, 178, 147, 90, 191, 125, 51, 121, 131, 135, 38, 27, 16, 1, 156, 198, 117, 54, 31, 193, 164, 73, 136, 45, 5, 177, 25, 104, 138, 182, 139, 168, 195, 145, 15, 3, 70, 161, 13, 132, 169, 188, 6, 134, 43, 11, 49, 33, 160, 29, 109, 35, 0, 111, 144, 56, 8, 82, 87, 21, 77, 81, 153, 163, 176, 86, 95, 18, 183, 140, 130, 99, 185, 179, 166, 194, 113, 103, 89, 127, 165, 187, 97, 141, 91, 61, 36, 142, 78, 175, 162, 80, 190, 108

**Average Fitness: 45870.254**

#### 5.2.4 Results for Random

**Best Fitness: 190200**

178, 173, 61, 84, 194, 149, 191, 136, 152, 116, 117, 35, 126, 141, 65, 158, 121, 133, 143, 108, 175, 113, 66, 87, 73, 16, 127, 165, 156, 42, 1, 118, 160, 128, 93, 43, 161, 26, 58, 111, 146, 185, 123, 189, 20, 138, 59, 48, 57, 148, 122, 131, 147, 78, 22, 31, 71, 190, 107, 172, 74, 157, 81, 52, 29, 83, 39, 182, 56, 174, 49, 4, 140, 168, 112, 135, 104, 30, 139, 99, 38, 153, 34, 8, 85, 97, 186, 0, 109, 167, 80, 92, 54, 67, 62, 25, 124, 145, 86, 10

**Worst Fitness: 235839**

33, 57, 73, 48, 152, 164, 186, 112, 86, 4, 81, 14, 54, 22, 26, 36, 18, 17, 134, 154, 199, 104, 101, 32, 107, 126, 29, 180, 68, 148, 25, 102, 176, 80, 155, 192, 170, 111, 3, 7, 47, 27, 139, 162, 122, 37, 106, 173, 116, 55, 135, 51, 109, 23, 123, 8, 44, 157, 79, 2, 172, 24, 147, 160, 89, 42, 82, 59, 120, 60, 138, 184, 10, 119, 11, 16, 66, 40, 100, 159, 187, 121, 183, 72, 156, 91, 188, 110, 53, 64, 76, 74, 129, 196, 150, 77, 56, 142, 178, 21

**Average Fitness: 213180.8**

## 6 Solution Verification with the Solution Checker

In this section, we verify the correctness of the best solutions for each instance and method by comparing the calculated fitness values with the values computed using the solution checker. The verification was performed using an Excel-based solution checker tool, where the best solutions were inserted, and the corresponding objective function (fitness) was recalculated. Below is a summary of the results, showing whether the calculated fitness matches the value obtained from the solution checker.

**Table 5:** Fitness Verification for both instances

| method | instance | calculated fitness | excel fitness | match |
|---|---|---|---|---|
| Greedy Cycle | tspA | 71488 | 71488.00 | True |
| Greedy Cycle | tspB | 49001 | 49001.00 | True |
| Nearest Neighbor End Only | tspA | 83182 | 83182.00 | True |
| Nearest Neighbor End Only | tspB | 52319 | 52319.00 | True |
| Nearest Neighbor Flexible | tspA | 71179 | 71179.00 | True |
| Nearest Neighbor Flexible | tspB | 44417 | 44417.00 | True |
| Random | tspA | 225047 | 225047.00 | True |
| Random | tspB | 190200 | 190200.00 | True |

The results in the table show that for every method and instance, the fitness values calculated by our algorithm match exactly with those verified by the solution checker. This confirms the correctness of the best solutions generated by the algorithm for both instances.

## 7 Source Code

The complete source code for this project, including all implemented algorithms, the solution checker, and the Python scripts for data processing and LaTeX generation, is available on GitHub. You can access it using the following link:

https://github.com/notbulubula/EvolutionaryComputation

The repository contains:

- Go code for the heuristic algorithms.
- Python scripts for handling solution checking and data visualization.

## 8 Conclusion

In this project, we explored several heuristic methods for solving the Hamiltonian Cycle problem on two distinct instances: A and B. The goal was to minimize both the total path length and the node costs while visiting exactly 50% of the available nodes. Through our experiments, we tested four heuristic methods: **Greedy Cycle**, **Nearest Neighbor (End Insertion)**, **Nearest Neighbor (Flexible Insertion)**, and **Random Solution**.

The experimental results showed that:

- The **Greedy Cycle** and **Nearest Neighbor (Flexible Insertion)** methods consistently performed better than the other approaches, producing lower fitness values in both instances.
- The **Random Solution** method produced significantly worse results, indicating that random selection of nodes is not an effective approach for this problem.

- Both **A** and **B** instances followed a similar performance trend, with the same methods outperforming the others.

We also verified the correctness of the best solutions using an automated solution checker, and the fitness values matched exactly with those computed by the Excel-based solution checker. This confirms that our implementation is correct and that the solutions generated by our algorithms are valid and optimal based on the defined objective function.