Multiple Algorithmic Approaches to the Traveling Salesman Problem

Arian Ahmadi

arian82ahmadi14@gmail.com LinkedIn GitHub

Contents

1 BruteForce **2**
1.1 Overview . . . 2
1.2 Key Components . . . 2
1.3 Input Format . . . 2
1.4 Limitations . . . 2
1.5 Usage . . . 2

2 Dynamic Programming **3**
2.1 Overview . . . 3
2.2 Key Components . . . 3
2.3 Input Format . . . 3
2.4 Limitations . . . 3
2.5 Usage . . . 3

3 Christofides **3**
3.1 Overview . . . 3
3.2 Key Components . . . 3
3.3 Input Format . . . 4
3.4 Limitations . . . 4
3.5 Usage . . . 4

4 DisjointCycle **4**
4.1 Overview . . . 4
4.2 Key Components . . . 4
4.3 Input Format . . . 5
4.4 Limitations . . . 5
4.5 Usage . . . 5

5 ConvexHull **5**
5.1 Overview . . . 5
5.2 Key Components . . . 5
5.3 Input Format . . . 6
5.4 Key Features . . . 6
5.5 Limitations . . . 6
5.6 Usage . . . 6

6 BestPathFromPaths **6**
6.1 Overview . . . 6
6.2 Key Components . . . 6
6.3 Input Format . . . 7
6.4 Limitations . . . 7
6.5 Usage . . . 7

7 MakeTheTourBetter **7**
7.1 Overview . . . 7
7.2 Key Components . . . 7
7.3 Input Format . . . 8
7.4 Limitations . . . 8
7.5 Usage . . . 8

8 OptimizeShortestNext **8**
8.1 Overview . . . 8
8.2 Key Components . . . 8
8.3 Input Format . . . 8
8.4 Limitations . . . 9
8.5 Usage . . . 9

9 FederatedGreedy **9**
9.1 Overview . . . 9
9.2 Key Components . . . 9
9.3 Input Format . . . 9
9.4 Limitations . . . 9
9.5 Usage . . . 9

10 ShortestEdgeFirst **9**
10.1 Overview . . . 9
10.2 Key Components . . . 10
10.3 Input Format . . . 10
10.4 Limitations . . . 10
10.5 Usage . . . 10

11 GuiApplication **10**
11.1 Overview . . . 10
11.2 Key Components . . . 10
11.3 Algorithms . . . 11
11.4 Cost Calculation and Visualization . . . 11
11.5 Input Format . . . 11
11.6 Key Features . . . 11

Introduction

The Traveling Salesman Problem (TSP) is one of the most well-known optimization problems in computer science and operations research. It involves finding the shortest possible route that visits each city exactly once and returns to the starting city. Due to its NP-hard nature, the TSP becomes computationally challenging as the number of cities increases. Various algorithms have been proposed to tackle the problem, ranging from brute-force solutions to more sophisticated heuristics and approximation methods.

In this project, we explore multiple algorithms designed to solve the TSP, including exact methods like brute force and dynamic programming, as well as approximation techniques such as the Christofides algorithm and ant colony optimization. Each approach offers a unique balance between computational efficiency and solution accuracy, and we compare their effectiveness in solving TSP instances of varying sizes.

The following sections provide an overview of the different algorithms implemented in this project, detailing their approach to solving the TSP and their expected performance.

1 BruteForce

1.1 Overview

This code implements a brute-force approach to solving the Traveling Salesman Problem (TSP). The code works by calculating the total cost of all possible tours and then determines the minimum cost tour.

1.2 Key Components

The code includes several important methods and components designed to solve the TSP.

The `calculateTourCost` method is responsible for computing the total cost of a given tour. It does this by summing up the travel costs between consecutive cities in the tour, including the return to the starting city. This ensures that the cost of completing the tour is fully accounted for.

The `findMinimumCost` method finds the tour with the minimum cost. To do so, it iterates through all possible permutations of cities, utilizing the helper function `nextPermutation` to generate each permutation. As it examines each tour, it tracks the minimum cost and the corresponding sequence of cities.

The `nextPermutation` method is used to generate the next lexicographical permutation of an array. This function is crucial for systematically exploring all possible tours in a controlled manner. By using this method, the algorithm can exhaustively test all permutations to ensure it finds the optimal solution.

The `swap` method is a utility function that swaps two elements within an array. It is used within the `nextPermutation` method to rearrange the cities when generating new permutations.

The cost matrix, which is central to the algorithm, is constructed using multiple methods, including `buildCostMatrix`, `buildCostMatrix2`, and `buildCostMatrixFromCoordinates`. These methods allow the creation of the cost matrix from various forms of input, such as a list of edges with distances, coordinates of cities, or a set of input points and their relationships. This flexibility ensures that the code can handle different types of data formats.

For scenarios where coordinates are provided, the `calculateDistance` method is used to compute the Euclidean distance between two points. This distance is then used to populate the cost matrix, which is crucial for calculating the travel costs between cities in the TSP.

The main method of the code reads the input data from a file. It first determines the input format—whether it is a list of edges, coordinates, or point pairs—and then constructs the cost matrix accordingly. Finally, it calls the `findMinimumCost` method to calculate and output the minimum cost tour.

1.3 Input Format

The input format is flexible and can vary depending on the data being provided. There are three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by edges that describe the distances between cities. For example, the input might look like `0 1 10.0`, where 0 and 1 are city indices, and 10.0 is the distance between them.

2. **Constant + Coordinates**: In this format, a constant is followed by city coordinates. For instance, the input might be `0.0 1.0`, representing the coordinates of a city.

3. **Constant + Point Pairs**: The input consists of a constant followed by paired points. An example could be `(0.0, 1.0) (1.0, 2.0)`, where the points represent the locations of two cities.

1.4 Limitations

There are some limitations to the brute-force approach implemented in this code. The most significant limitation is its exponential time complexity, $O(n!)$, which makes it infeasible for large numbers of cities. As the number of cities increases, the number of possible permutations grows factorially, causing the algorithm to become extremely slow.

1.5 Usage

This brute-force approach is ideal for solving the TSP with small datasets, where an exact solution is required. The code guarantees the identification of the optimal solution, making it useful for smaller problems. However, for larger datasets, the brute-force approach is impractical, and alternative heuristic or optimized algorithms should be considered.

2 Dynamic Programming

2.1 Overview

This code implements the Held-Karp algorithm, which is a dynamic programming approach to solving the Traveling Salesman Problem (TSP). The Held-Karp algorithm applies memoization and bitmasking techniques to efficiently calculate the minimum cost tour. By storing intermediate results and representing subsets of cities using bitmasks, the algorithm can solve the problem in a significantly faster manner than brute force methods.

2.2 Key Components

The primary method in this implementation is `tspHeldKarp`, which solves the TSP using dynamic programming. This method leverages bitmasking to represent the cities that have been visited in a given tour. It uses memoization to store the results of subproblems, which allows it to avoid redundant calculations and improves efficiency. The algorithm tracks the minimum cost associated with visiting all cities and ending at a specific city. At the end, it returns the optimal tour and its associated cost.

The `buildCostMatrix` methods are responsible for constructing the cost matrix, which is central to the algorithm. These methods can handle different input formats. One option is to provide edges, where the input consists of city pairs and their corresponding distances. Another option is to provide city coordinates, from which distances are calculated using the Euclidean distance formula. Finally, the input can be given as paired coordinates (e.g., (x_1, y_1) and (x_2, y_2)), which are used to generate the cost matrix.

In cases where coordinates are provided, the `calculateDistance` method is used to compute the Euclidean distance between two points. This distance is then used in the cost matrix to represent the travel cost between cities.

The `notIn` method is a utility function that checks whether a specific city is part of a subset, represented by a bitmask. This method is crucial for performing efficient subset operations during the dynamic programming process, as the algorithm needs to examine various subsets of cities.

The main method of the code handles the input data. It reads the data from a file, identifies the input format (whether it's edges, coordinates, or point pairs), and constructs the cost matrix using the appropriate helper function. Once the cost matrix is ready, it calls the `tspHeldKarp` method to calculate the minimum cost and optimal tour.

2.3 Input Format

The input format can vary depending on how the data is structured. There are three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by edges that represent city pairs and the distances between them. For example, the input might look like `0 1 10.0`, where 0 and 1 are the indices of the cities, and 10.0 is the distance between them.

2. **Constant + Coordinates**: In this format, a constant is followed by the coordinates of cities. For example, the input could be `0.0 1.0`, which represents the coordinates of a city in a two-dimensional space.

3. **Constant + Point Pairs**: The input consists of a constant followed by pairs of coordinates, such as `(0.0,1.0) (1.0,2.0)`. These pairs represent the locations of two cities in a two-dimensional plane.

2.4 Limitations

Despite its efficiency compared to brute force, the Held-Karp algorithm still has certain limitations. The most notable limitation is its time complexity, which is exponential in nature: $O(2^n \cdot n^2)$. While this is much more efficient than the brute-force approach, it can still be computationally expensive for very large datasets, making the algorithm impractical for instances with a large number of cities.

Additionally, the algorithm requires significant memory usage. This is due to the need to store the dynamic programming table and the bitmasks, which can become quite large as the number of cities increases.

2.5 Usage

This implementation of the Held-Karp algorithm is well-suited for small to medium-sized TSP instances where an exact solution is required. Given its exponential time complexity, it is most effective when the number of cities is relatively small, typically up to around 20–25 cities. For larger datasets, heuristic methods such as genetic algorithms, simulated annealing, or other approximation algorithms may be more appropriate, as they can provide good solutions more quickly, albeit without guaranteed optimality.

3 Christofides

3.1 Overview

This code implements the Christofides algorithm, which is a heuristic method designed to approximate a solution for the Traveling Salesman Problem (TSP). The Christofides algorithm guarantees a solution that is within a factor of 3/2 of the optimal solution, which makes it an efficient way to find near-optimal tours for the TSP. The algorithm combines several techniques, including Minimum Spanning Tree (MST), perfect matching, Eulerian circuits, and Hamiltonian circuit conversion, to produce the approximate tour.

3.2 Key Components

The primary method for solving the TSP is the `christofidesAlgorithm` method, which implements the entire Christofides algorithm. It generates the approximate tour by first constructing a Minimum Spanning Tree (MST), then finding a perfect matching for vertices with odd degrees, followed by combining the

MST and the matching into an Eulerian circuit. The final step involves converting the Eulerian circuit into a Hamiltonian circuit by removing repeated cities, thus ensuring the tour visits each city exactly once.

To build the cost matrix, the algorithm provides several methods. The `buildCostMatrix` method constructs the cost matrix based on a list of edges. The matrix is initialized with a high value (representing infinity) and then populated with the actual edge weights between cities. The `buildCostMatrixFromCoordinates` method constructs the cost matrix using city coordinates. It computes the Euclidean distances between cities, and the resulting distances are scaled by a given constant. Similarly, the `buildCostMatrix2` method is another version that handles parsing string representations of city coordinates, calculates distances, and scales them accordingly.

In the process of solving the problem, the algorithm identifies vertices with odd degrees in the MST using the `findOddDegreeVertices` method. These odd-degree vertices require perfect matching to ensure that every vertex in the graph has an even degree. The `findMinimumWeightMatching` method computes this matching using a greedy approach, selecting the pairs of odd-degree vertices that minimize the total cost.

Once the MST and the matching have been determined, the `findEulerianCircuit` method finds an Eulerian circuit in the resulting multigraph. This is achieved using a stack-based approach, where the algorithm iteratively traverses the graph to construct the circuit.

Finally, the `convertToHamiltonianCircuit` method takes the Eulerian circuit and removes any repeated cities, resulting in a Hamiltonian circuit that represents the final approximate solution. The `calculateTourCost` method is used throughout to compute the total cost of the tour by summing the edge costs.

The `calculateDistance` method computes the Euclidean distance between two cities given their coordinates and multiplies the result by a scaling factor. The main method is responsible for reading the input data, determining the format (whether edges or coordinates are provided), constructing the cost matrix, and calling the `christofidesAlgorithm` method to generate and output the approximate TSP tour and its total cost.

3.3 Input Format

The input format for the algorithm can vary depending on how the data is provided. It supports three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by a list of edges. Each edge represents a pair of cities and the corresponding distance between them. For example, the input might be `0 1 10.0`, where 0 and 1 are city indices and 10.0 is the distance between them.

2. **Constant + Coordinates**: In this case, the input consists of a constant followed by the coordinates of each city. For example, `0.0 1.0` would represent the coordinates of a city in a two-dimensional space.

3. **Constant + Point Pairs**: The input contains a constant followed by pairs of coordinates, such as `(0.0,1.0) (1.0,2.0)`, representing the locations of two cities.

3.4 Limitations

While the Christofides algorithm provides a strong approximation for the TSP, it does not guarantee the optimal solution. It guarantees that the solution will be within 1.5 times the optimal cost, but the result may still be suboptimal for certain problem instances.

3.5 Usage

This code is well-suited for approximating solutions to the TSP for moderate-sized datasets, where the exact optimal solution is difficult to compute but an approximate solution is sufficient.

4 DisjointCycle

4.1 Overview

This code implements the Christofides algorithm with a disjoint cycle approach to solve the Traveling Salesman Problem (TSP).

The algorithm involves several steps, including constructing a Minimum Spanning Tree (MST), identifying odd-degree vertices in the MST, performing a minimum weight matching for those vertices, and combining the resulting cycles to form the final tour. By using a disjoint cycle approach, the algorithm separates the tour into distinct cycles before merging them into the complete solution.

4.2 Key Components

The core of the implementation is the `christofidesWithDisjointCycle` method, which uses the Christofides algorithm with a disjoint cycle approach. This method computes the MST, identifies the odd-degree vertices, performs the minimum weight matching, and then combines the cycles formed by the MST and matching to create the final tour.

Several methods are used to build the cost matrix, which represents the distances between cities. The `buildCostMatrix` method constructs a cost matrix from a list of edges and their associated costs, ensuring that the matrix is symmetric and initializes all entries to a large value, except for the diagonal, which is set to zero. The `buildCostMatrix2` method constructs a cost matrix from coordinate-based edges and scales the distances by a constant multiplier. Another method, `buildCostMatrixFromCoordinates`, directly computes the Euclidean distances between city coordinates and scales them similarly.

The `calculateDistance` method is used to compute the Euclidean distance between two cities given their x and y coordinates, while the `calculateTourCost`

method calculates the total cost of a given tour by summing the costs of traveling between consecutive cities and returning to the starting point.

To detect cycles in the graph, the algorithm uses the `dfs` method, which performs a Depth-First Search (DFS) to mark visited cities and identify cycles. The `combineCyclesIntoTour` method is responsible for combining multiple disjoint cycles into a single tour, ensuring that the final tour starts and ends at the same city.

The `findMST` method finds the Minimum Spanning Tree (MST) of the graph using Prim's algorithm, which uses a priority queue to add the minimum weight edges to the tree. The `findOddDegreeVertices` method identifies the vertices in the MST with odd degrees, which are the focus of the matching step. These vertices are matched using the `findMinimumWeightMatching` method, which uses a greedy approach to match pairs of odd-degree vertices with the minimum possible cost.

The `findDisjointCycles` method finds disjoint cycles in the multigraph formed by combining the MST and the matching. This method uses DFS to separate the graph into individual cycles. Once the disjoint cycles are identified, the `combineCyclesIntoTour` method merges them to form the final approximate TSP tour.

The main method reads input data from a file, parses the input, and constructs the appropriate cost matrix. It then calls the `christofidesWithDisjointCycle` method to compute the approximate TSP tour and outputs the computed tour and its corresponding cost.

4.3 Input Format

The input format for the algorithm can vary depending on how the data is provided. It supports three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by a list of edges. Each edge represents a pair of cities and the corresponding distance between them. For example, the input might be `0 1 10.0`, where 0 and 1 are city indices and 10.0 is the distance between them.

2. **Constant + Coordinates**: In this case, the input consists of a constant followed by the coordinates of each city. For example, `0.0 1.0` would represent the coordinates of a city in a two-dimensional space.

3. **Constant + Point Pairs**: The input contains a constant followed by pairs of coordinates, such as `(0.0,1.0) (1.0,2.0)`, representing the locations of two cities.

4.4 Limitations

While the Christofides algorithm provides a strong approximation to the TSP, it is not guaranteed to produce the optimal solution. The approximation factor of 1.5 means that the solution could still be suboptimal in some cases, especially when there is no perfect matching for odd-degree vertices.

4.5 Usage

This code is suitable for solving the TSP for moderate-sized datasets where an approximate solution is acceptable. The Christofides algorithm offers a guaranteed approximation within a factor of 1.5 of the optimal solution, making it a practical choice for problems where exact solutions are not required.

5 ConvexHull

5.1 Overview

This code implements a heuristic approach to solve the Traveling Salesman Problem (TSP) using the Convex Hull (Graham Scan Algorithm) and greedy insertion method. The algorithm begins by calculating the convex hull of the set of points, which forms an initial route, and then iteratively inserts the remaining points into this route in a greedy manner to minimize the overall cost. The result is an approximate solution to the TSP, where the route visits each city exactly once and returns to the starting point.

5.2 Key Components

At the heart of the implementation is the `calculateOptimalRoute` method, which computes the optimal route by first calculating the convex hull of the given points. The convex hull is used as the base of the route, and the remaining points are inserted into this route one by one using a greedy strategy. For each point, the algorithm evaluates all possible positions in the current route and inserts the point in the position that minimizes the increase in the total cost.

The `findConvexHull` method is responsible for computing the convex hull of the points using Graham's scan algorithm. This algorithm starts with the point that has the smallest y-coordinate (the anchor point) and sorts the other points based on their polar angle relative to the anchor. To maintain the convexity of the hull, the algorithm removes any points that do not form a counterclockwise turn.

The `computeCrossProduct` method plays an essential role in the convex hull computation by calculating the cross product of three points. This helps determine the orientation of the points—whether they form a clockwise, counterclockwise, or collinear turn—ensuring that the points added to the hull maintain a convex shape.

To compute the total cost of the route, the `computeRouteCost` method is used. This method sums the Euclidean distances between consecutive points in the route, including the cost of returning to the starting point to complete the cycle.

The `insertPointGreedily` method is used during the route optimization process. It evaluates all possible insertion points for each remaining city and greedily selects the position that minimizes the increase in the overall cost. This greedy insertion step ensures that the route is progressively optimized as new cities are added.

Finally, the `main` method reads input data, which consists of a scaling factor and a list of 2D coordinates for

the cities. It calls the `calculateOptimalRoute` method to compute the optimized TSP route and then prints the final route along with the scaled total cost.

5.3 Input Format

The input data is provided in a file, where the first line contains a constant scaling factor, and the subsequent lines contain the coordinates of the cities in 2D space. Each city's coordinates are represented as a pair of x- and y-coordinates, separated by spaces. For example, the input could look like this:

$$\text{Scaling Factor}$$
$$(x_1, y_1)$$
$$(x_2, y_2)$$
$$\vdots$$

5.4 Key Features

One of the key features of this algorithm is the convex hull initialization. By first calculating the convex hull, the algorithm ensures that the initial route includes the outermost points, which helps provide a good starting point for the subsequent optimization. This step reduces the complexity of the problem by focusing on the boundary points first.

Another important feature is the greedy insertion method. As the algorithm progresses, each remaining city is inserted into the route in the position that minimizes the increase in total cost. This strategy dynamically improves the route, providing a reasonable approximation to the optimal solution while maintaining computational efficiency.

The use of Euclidean distance as the cost metric ensures that the algorithm computes the true distance between cities in 2D space, providing an accurate measure of the cost for each route.

5.5 Limitations

While the algorithm provides an efficient heuristic for solving the TSP, it does not guarantee the optimal solution. The greedy nature of the insertion step means that the solution may not be optimal, though it is usually a reasonable approximation for many practical cases.

5.6 Usage

It is particularly useful in scenarios where exact solutions are not critical, and a heuristic approach that provides a good balance between accuracy and efficiency is desired.

6 BestPathFromPaths

6.1 Overview

This code implements an ant colony optimization-inspired algorithm to solve the Traveling Salesman Problem (TSP). The algorithm leverages the concept of pheromone-based optimization, inspired by the foraging behavior of ants. Through multiple iterations, the algorithm simulates several agents (or "ants") that build paths and update pheromone levels, gradually improving the solution to find a near-optimal route.

6.2 Key Components

The core of the algorithm is the `FindTheBestPath` method, which simulates the main loop of ant colony optimization. This method involves multiple agents that construct paths from city to city. As the agents traverse the cities, they probabilistically choose the next city to visit, with their choices influenced by the pheromone levels (indicating the quality of past paths) and the inverse of the distances (encouraging shorter paths). After each iteration, the pheromone levels are updated, with stronger pheromones applied to the better paths (those with lower costs). The best path discovered during these iterations is tracked and returned as the final solution.

To construct the paths, the algorithm uses the `constructPath` method, which simulates an agent's journey through the cities. The agent builds a complete path by visiting one city at a time, and the next city is chosen probabilistically based on pheromone levels and distances, ensuring a balance between exploration and exploitation of good paths.

The quality of the paths is assessed by the `calculatePathCost` method, which calculates the total cost of a given path by summing the distances between consecutive cities. This cost also includes the distance required to return to the starting city, completing the tour.

To refine the paths and improve the solution, the `updateAgentStrengths` method updates the pheromone levels. After each iteration, pheromone levels are stronger on paths that resulted in lower-cost tours, thereby increasing the likelihood that future agents will follow these successful paths. However, the algorithm also incorporates pheromone evaporation through the `evaporateAgentStrengths` method, simulating the natural evaporation process. This prevents over-concentration on suboptimal paths and encourages exploration of new paths.

The `chooseNextCity` method is responsible for determining the next city that an agent will visit. Using a probabilistic approach, the agent chooses a city based on pheromone strength and inverse distance, ensuring that better paths (with stronger pheromones and shorter distances) are more likely to be chosen.

The main method of the algorithm handles the input and output. It reads data from a file, which could contain either a list of edges, city coordinates, or point pairs. Based on the input format, the distance matrix is constructed either from a list of edges or by calculating the Euclidean distance between cities. The `FindTheBestPath` method is then invoked to perform the ant colony optimization and find the best tour. Finally, the resulting best tour and its corresponding cost are printed.

6.3 Input Format

The input format is flexible and can vary depending on the data being provided. There are three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by edges that describe the distances between cities. For example, the input might look like `0 1 10.0`, where 0 and 1 are city indices, and 10.0 is the distance between them.

2. **Constant + Coordinates**: In this format, a constant is followed by city coordinates. For instance, the input might be `0.0 1.0`, representing the coordinates of a city.

3. **Constant + Point Pairs**: The input consists of a constant followed by paired points. An example could be `(0.0, 1.0) (1.0, 2.0)`, where the points represent the locations of two cities.

6.4 Limitations

Despite its strengths, the algorithm has some limitations. As a heuristic approach, it does not guarantee that the global optimum will be found. The quality of the solution depends on factors like the initial pheromone levels, the number of agents, and the number of iterations.

Another limitation is the sensitivity of the algorithm's performance to parameter tuning. Parameters such as `PARAM_A`, `PARAM_B`, and the evaporation rate must be carefully adjusted to achieve optimal performance. Improper tuning can lead to poor results, either by overexploring suboptimal paths or converging too early to a local optimum.

Finally, although the algorithm can handle medium-sized datasets effectively, it may become computationally expensive for very large TSP instances. The number of iterations, combined with the number of agents, can make the algorithm slow for large inputs, and more efficient methods may be required for such cases.

6.5 Usage

This code is ideal for solving medium-sized TSP instances where an exact solution is not necessary and a reasonably good solution is acceptable within a reasonable amount of time. It is particularly suitable for problems where the trade-off between computational cost and solution quality is important. However, for very large datasets or instances where an exact solution is required, more advanced or specialized algorithms may be more appropriate.

7 MakeTheTourBetter

7.1 Overview

This code implements a 2-opt heuristic algorithm to solve the Traveling Salesman Problem (TSP). The goal of the algorithm is to improve an initial tour by iteratively swapping segments of the tour in order to minimize the total travel cost. The 2-opt algorithm achieves this by identifying two edges in the tour and swapping their connecting segments, which can reduce the overall distance. The process is repeated until no further improvements can be made, resulting in a locally optimal tour. The algorithm is capable of working with various input formats and dynamically constructs the cost matrix based on the provided input.

7.2 Key Components

The code's functionality revolves around a set of methods that are designed to construct cost matrices from different input types, calculate distances between cities, and iteratively improve a given tour using the 2-opt heuristic.

The `buildCostMatrix` method is responsible for constructing a cost matrix from a list of edges representing the distances between cities. It initializes the matrix with high values to represent unconnected cities and populates it with the provided edge costs, ensuring symmetry for undirected graphs. Similarly, the `buildCostMatrix2` method generates a cost matrix from input pairs of city coordinates, calculating the Euclidean distances between the cities and then scaling the distances using a constant multiplier.

The `buildCostMatrixFromCoordinates` method directly constructs the cost matrix from a list of city coordinates, calculating pairwise Euclidean distances between all cities and scaling them by a given constant to derive the cost. The `calculateDistance` method is used to compute the Euclidean distance between two points in 2D space, returning the scaled distance as an integer for inclusion in the cost matrix.

The `calculateTourCost` method calculates the total cost of a given tour. It sums the costs of traveling between consecutive cities and also includes the cost of returning to the starting city, thus computing the full tour cost.

The core of the algorithm is the `MakeTheTourBetterAlgorithm` method, which implements the 2-opt heuristic. This method iteratively identifies and swaps segments of the current tour to reduce its total cost. A segment swap occurs between two specified indices in the tour, and the algorithm continues making swaps until no further improvement can be achieved. The `twoOptSwap` method is responsible for reversing a segment of the tour between two indices, which is used within the 2-opt algorithm to test new configurations of the tour.

The main method of the code handles input reading from a file in various formats. It supports input data that represents city connections via edges with distances, city coordinates, or coordinate pairs. Based on the input type, the method constructs the cost matrix dynamically. After constructing the cost matrix, the `MakeTheTourBetterAlgorithm` method is invoked to compute an improved tour, which is then outputted along with its total cost.

7.3 Input Format

The input format is flexible and can vary depending on the data being provided. There are three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by edges that describe the distances between cities. For example, the input might look like `0 1 10.0`, where 0 and 1 are city indices, and 10.0 is the distance between them.

2. **Constant + Coordinates**: In this format, a constant is followed by city coordinates. For instance, the input might be `0.0 1.0`, representing the coordinates of a city.

3. **Constant + Point Pairs**: The input consists of a constant followed by paired points. An example could be `(0.0, 1.0) (1.0, 2.0)`, where the points represent the locations of two cities.

7.4 Limitations

While the 2-opt heuristic is an efficient method, it is important to note that it is an approximation algorithm. It does not guarantee the optimal solution but instead finds a locally optimal solution based on the given input. The quality of the solution depends on the initial tour, and the algorithm may not explore the global optimum.

Additionally, the performance of the algorithm depends on the accuracy of the input data. The chosen constant multiplier for scaling distances can affect the final result, and the algorithm's effectiveness may vary depending on the quality of the input.

Another limitation is that the 2-opt algorithm is susceptible to getting stuck in local optima. If the algorithm converges to a local minimum, it may not explore other potential solutions, resulting in suboptimal tours.

7.5 Usage

The algorithm provides a good balance between computational efficiency and solution quality, making it an excellent choice for cases where finding an exact solution is not crucial but where a reasonable tour is needed in a practical amount of time.

8 OptimizeShortestNext

8.1 Overview

This code implements a heuristic approach to solve the Traveling Salesman Problem (TSP) by combining the Nearest Neighbor algorithm with 2-opt optimization. The approach begins by constructing an initial tour using the Nearest Neighbor algorithm, which is a greedy heuristic that iteratively selects the nearest unvisited city. Once the initial tour is generated, the algorithm refines the tour by applying the 2-opt method, an optimization technique that iteratively reverses segments of the tour to reduce the total cost. By combining these two methods, the code aims to provide a good solution to the TSP in a relatively short amount of time.

8.2 Key Components

The core functionality of this solution lies in several key methods that work together to build the cost matrix, construct the initial tour, and optimize the tour.

The `buildCostMatrix` method is used to build a cost matrix from a list of edges, where each edge connects two cities and has an associated cost (distance). Initially, the cost matrix is filled with large values, and the distances from the provided edges are used to update the matrix with the correct values. The `buildCostMatrix2` method builds the cost matrix from a list of city coordinates. This method calculates the Euclidean distance between each pair of cities and stores the scaled value in the cost matrix. Similarly, the `buildCostMatrixFromCoordinates` method directly constructs the cost matrix based on city coordinates. It computes the Euclidean distances between all pairs of cities and ensures the diagonal elements are set to zero, as a city does not need to travel to itself.

The `calculateTourCost` method is used to compute the total cost of a given tour by summing the costs of traveling between consecutive cities. Since the tour is circular, the cost of returning from the last city to the starting city is also included. The `calculateDistance` method is used to compute the Euclidean distance between two cities, which is then scaled for accuracy.

The `twoOptOptimization` method applies the 2-opt optimization technique to the current tour. This method iteratively examines segments of the tour and reverses segments that reduce the total travel cost, improving the tour in each iteration. The `nearestNeighborAlgorithm` method is responsible for constructing the initial tour using the Nearest Neighbor heuristic. Starting from an arbitrary city, the algorithm selects the nearest unvisited city at each step, continuing until all cities are visited.

Finally, the main method reads input data from a file, determining whether the cost matrix format is based on edges or coordinates. It then constructs the cost matrix using the appropriate method and computes the initial tour using the Nearest Neighbor algorithm. The tour is then optimized using the 2-opt method. The code outputs the tour before and after optimization, along with the corresponding total costs.

8.3 Input Format

The input format is flexible and can vary depending on the data being provided. There are three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by edges that describe the distances between cities. For example, the input might look like `0 1 10.0`, where 0 and 1 are city indices, and 10.0 is the distance between them.

2. **Constant + Coordinates**: In this format, a constant is followed by city coordinates. For instance, the input might be `0.0 1.0`, representing the coordinates of a city.

3. **Constant + Point Pairs**: The input consists of a constant followed by paired points. An example could be `(0.0, 1.0) (1.0, 2.0)`, where the points represent

the locations of two cities.

8.4 Limitations

While the Nearest Neighbor algorithm is a fast and simple heuristic, it may not always provide the globally optimal solution. As a greedy algorithm, it always selects the nearest unvisited city at each step, which can lead to suboptimal solutions, particularly in cases where a more distant city may have led to a better overall tour. The 2-opt optimization improves the tour, but it is not guaranteed to find the global optimum, as it is based on local improvements.

8.5 Usage

This code offers a good balance between computational efficiency and the quality of the solution, making it ideal for practical, real-world applications of the TSP where an exact solution is not required.

9 FederatedGreedy

9.1 Overview

This code implements a greedy heuristic approach to solve the Traveling Salesman Problem (TSP) using a federated model, where the cities involved are distributed across multiple federated servers. This solution constructs a tour by iteratively selecting the nearest unvisited city, continuing until all cities are visited. Once the tour is complete, the total cost is calculated.

9.2 Key Components

The solution consists of several key components, each responsible for building the cities, constructing the distance matrix, and solving the TSP using a greedy heuristic approach.

The `City` class represents an individual city, with each city having a unique ID and a map of distances to other cities. It provides methods for adding distances to other cities and for finding the nearest unvisited city. This class is essential in managing the cities and facilitating the route construction.

The `FederatedServer` class manages a list of cities and the global route. It implements the core TSP-solving logic by using the greedy heuristic. This method iteratively selects the nearest unvisited city from the current city and adds it to the route until all cities are visited. The class also includes a method to calculate the total cost of a given route, summing the distances between consecutive cities.

The `solveTSP` method implements the greedy algorithm by starting from a given city and selecting the nearest unvisited city at each step. This continues until all cities are visited, forming a complete route. The method then returns the route.

The `calculateRouteCost` method computes the total cost of a route by summing the distances between consecutive cities. This total cost is essential for evaluating the efficiency of the constructed tour.

To build the cities and distances, the `createCities` method takes a distance matrix and creates a list of `City` objects. It then uses this list to build the tour. The `createDistanceMatrixFromEdges` method constructs the distance matrix from a list of edges, where each edge specifies a pair of cities and the distance between them. Alternatively, the `createDistanceMatrixFromCoordinates` method generates a distance matrix based on city coordinates. This method calculates the Euclidean distance between each pair of cities and stores the results in the matrix.

The `calculateEuclideanDistance` method computes the Euclidean distance between two cities, given their coordinates. The distance is scaled by a factor of 10,000 to ensure accuracy when dealing with real-world distances.

Finally, the main method reads input data from a file. This file may either contain edges with costs or city coordinates. After reading the data, the program constructs the distance matrix and creates the city objects. It then solves the TSP by invoking the `solveTSP` method and calculates the total cost of the route. The program outputs the final route and its cost.

9.3 Input Format

The input format is flexible and can vary depending on the data being provided. There are three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by edges that describe the distances between cities. For example, the input might look like `0 1 10.0`, where 0 and 1 are city indices, and 10.0 is the distance between them.

2. **Constant + Coordinates**: In this format, a constant is followed by city coordinates. For instance, the input might be `0.0 1.0`, representing the coordinates of a city.

3. **Constant + Point Pairs**: The input consists of a constant followed by paired points. An example could be `(0.0, 1.0) (1.0, 2.0)`, where the points represent the locations of two cities.

9.4 Limitations

Although the greedy approach is efficient, it is not guaranteed to find the globally optimal solution. The algorithm's greedy nature means it may get stuck in local minima, resulting in suboptimal routes.

9.5 Usage

It is especially useful in scenarios where cities are distributed across federated systems, making the approach modular and scalable.

10 ShortestEdgeFirst

10.1 Overview

This code implements a solution to the Traveling Salesman Problem (TSP) using a combination of a greedy

approach and 2-opt optimization.

10.2 Key Components

The core of the solution lies in several important methods. The `buildCostMatrix` methods are responsible for constructing the cost matrix. These methods handle two different types of input: a list of edges with their respective costs or the coordinates of cities. When city coordinates are provided, distances are computed using the Euclidean distance formula. The cost matrix is essential as it stores the distances between every pair of cities, and it is used to calculate the cost of a given tour.

The `greedyTSP` method implements the greedy algorithm, which is used to construct an initial tour. The algorithm selects the shortest available edge at each step, ensuring that no cycles are formed during the construction of the tour. To efficiently find the shortest edges, the method uses a priority queue to sort edges by their cost.

To prevent cycles from forming, the `formsCycle` and `hasCycle` methods are employed. These methods detect whether adding a new edge would create a cycle in the tour. They achieve this by using a Depth-First Search (DFS) algorithm to check the validity of the tour as edges are added. This is crucial to ensure that the final tour is valid and does not contain any redundant paths.

Once the greedy tour is constructed, the `twoOpt` method is used to optimize it. The 2-opt optimization technique works by iteratively examining pairs of edges in the tour and reversing sections of the route if such reversals reduce the overall cost. This method helps to eliminate unnecessary crossings in the tour, leading to a more efficient route.

The `calculateTourCost` method computes the total cost of the constructed tour. This is done by summing the distances between consecutive cities in the tour, ensuring that the return trip to the starting city is also included. The total cost is then used to evaluate the effectiveness of the algorithm and the optimization applied.

The `calculateDistance` method is employed when the input consists of city coordinates. This method computes the Euclidean distance between two cities, which is essential for populating the cost matrix.

The `dfs` method is a helper function that performs a depth-first traversal of the cities in the tour. It helps in constructing the final tour from the edges selected by the greedy algorithm and ensures that the route is valid.

The main method orchestrates the entire process. It begins by reading the input data from a file, determining whether the input consists of edges or coordinates. Based on the input format, it constructs the cost matrix and then computes the initial tour using the greedy algorithm. Afterward, it applies the 2-opt optimization to improve the tour and outputs the tour before and after the optimization, along with their respective costs.

10.3 Input Format

The input format is flexible and can vary depending on the data being provided. There are three possible formats:

1. **Constant + Edges**: The input starts with a constant, followed by edges that describe the distances between cities. For example, the input might look like `0 1 10.0`, where 0 and 1 are city indices, and 10.0 is the distance between them.
2. **Constant + Coordinates**: In this format, a constant is followed by city coordinates. For instance, the input might be `0.0 1.0`, representing the coordinates of a city.
3. **Constant + Point Pairs**: The input consists of a constant followed by paired points. An example could be `(0.0, 1.0) (1.0, 2.0)`, where the points represent the locations of two cities.

10.4 Limitations

Despite its efficiency, the algorithm has several limitations. The primary limitation is that the initial solution is based on a **Greedy Approach**, which does not always produce the optimal tour. The greedy algorithm may get stuck in local minima and may not find the globally optimal solution for the TSP.

10.5 Usage

This code is optimized for fast performance. The greedy algorithm quickly constructs an initial tour, which is then efficiently refined using the 2-opt optimization technique.

11 GuiApplication

11.1 Overview

The `TSPGuiApplication` is a graphical user interface (GUI) built with JavaFX that allows users to interactively solve the Traveling Salesman Problem (TSP) using various algorithms. The primary feature of this application is its canvas, where users can click to add cities, which are displayed as blue dots. Once the cities are placed, the application allows users to run different TSP algorithms, including Dynamic Programming, Christofides' Algorithm, Brute Force, Shortest Next, Federated Greedy, and others. Each algorithm attempts to find the optimal or near-optimal route for visiting all cities, and the resulting tour, along with its associated cost, is displayed on the canvas.

11.2 Key Components

The application is designed to provide an intuitive and interactive experience for solving the TSP. At the core of the interface is the canvas, where users can click to add cities. These cities are visually represented as blue dots on the canvas. Once the cities are added, the selected TSP algorithm is run, and the computed tour is drawn on the canvas in red, connecting the cities in the order

determined by the algorithm. The total cost of the tour is dynamically updated and displayed in a label.

The application includes a variety of buttons to allow the user to choose different TSP algorithms.

In addition to the buttons for running algorithms, there is also a "Clear Points" button, which allows the user to remove all cities and the computed tour, providing a fresh starting point. The cities are stored as a list, and the coordinates of the cities are used to construct the cost matrix. This matrix contains the pairwise distances between all cities, which are crucial for calculating the tour cost. Each time a new city is added to the canvas, the cost matrix is updated.

Once the cities and cost matrix are ready, the algorithms calculate a tour based on the cities' positions. The computed tour is drawn on the canvas with red lines connecting the cities in the order of the tour. The cost of the tour is calculated by summing the distances between consecutive cities and is displayed in the tour cost label, providing feedback to the user on the efficiency of the algorithm.

11.3 Algorithms

Several algorithms are implemented to solve the TSP, each with its own approach and complexity.

The `TSPDynamicProgramming` class implements the Dynamic Programming method, specifically the Held-Karp algorithm, which has a time complexity of $O(n^2 \cdot 2^n)$.

The `TSPChristofides` class implements Christofides' Algorithm, a heuristic approach that guarantees a solution within a factor of 3/2 of the optimal solution. This algorithm uses a combination of minimum spanning tree, minimum-weight matching, and Eulerian circuits to construct an approximate solution.

The `TSPBruteForce` class implements the brute force approach, which evaluates all possible permutations of the cities and calculates the cost of each tour to determine the minimum cost.

For faster, heuristic solutions, the `TSPOptimizeShortestNext` class implements the nearest neighbor (Shortest Next) algorithm. This greedy approach selects the nearest unvisited city at each step to build the tour. While the solution is not guaranteed to be optimal, it is computationally efficient and provides a good approximation in many cases.

Additionally, the `TSPFederatedGreedy` class implements the federated greedy algorithm, which uses a distributed approach to solve the problem in parallel across multiple servers. The `TSPDisjointCycle` class applies a variant of Christofides' Algorithm using disjoint cycles to further improve the solution.

Optimization algorithms, such as the 2-opt algorithm, are also integrated into the application. These algorithms are designed to improve the initial tour generated by heuristics like Shortest Next or Shortest Edge First by making local changes to the tour structure that reduce the overall cost.

11.4 Cost Calculation and Visualization

The cost of the tour is calculated by summing the distances between consecutive cities. These distances are computed using the Euclidean distance formula, based on the coordinates of the cities. The total cost is then displayed in the `tourCostLabel`, providing real-time feedback on the quality of the solution.

The cost matrix, which stores the pairwise distances between all cities, is constructed each time the user adds or removes cities from the canvas. This matrix is essential for the algorithms, as it allows them to quickly look up the distances between cities when calculating the cost of different tours. As new cities are added to the canvas, the cost matrix is updated, ensuring that the calculations remain accurate.

11.5 Input Format

Cities are added to the canvas by clicking on the screen, and their coordinates are stored as integer pairs. The algorithms use these coordinates to build the cost matrix, which represents the distances between every pair of cities. This format allows for flexibility, as users can freely place cities wherever they choose on the canvas.

11.6 Key Features

One of the main features of the `TSPGuiApplication` is its interactive GUI. The application allows users to easily add cities, choose from a variety of TSP algorithms, and visualize the results. This hands-on approach makes it ideal for demonstrating the differences between algorithms and for understanding how each one tackles the problem. The real-time calculation and display of the tour cost provide immediate feedback on the effectiveness of the chosen algorithm.

The application supports multiple algorithms, from exact methods like Dynamic Programming to heuristic methods like Christofides' Algorithm and Greedy approaches. This variety gives users the ability to experiment with different solutions and observe their effectiveness in real time. The ability to visualize the computed tour on the canvas and track the cost of the tour enhances the user experience and provides a clear understanding of the results.

Algorithm Name	**Time Complexity**	**Space Complexity**
Brute Force	$O(n!)$	matrix: $O(n^2)$
Dynamic Programming	$O(n^2 \cdot 2^n)$	matrix: $O(n^2)$ Mem: $O(n \cdot 2^n)$
Christofides	$O(n^2 \log n)$	matrix: $O(n^2)$ List: $O(n + E)$
Optimize Shortest Next	$O(n^3)$	matrix: $O(n^2)$
Shortest Edge First	$O(n^3)$	matrix: $O(n^2)$ List: $O(n + E)$
Best Path From Paths	$O(k \cdot a \cdot n^2)$	matrix: $O(n^2)$
Make The Tour Better	$O(n^3)$	matrix: $O(n^2)$
Disjoint Cycle	$O(n^2 \log n)$	matrix: $O(n^2)$ List: $O(n + E)$
Federated Greedy	$O(n^2)$	matrix: $O(n^2)$

Table 1: Algorithms with their Time and Space Complexity