Murat Albayrak 150120025
Kadir Pekdemir 150121069

CSE2046 Assignment 2
Half Traveling Salesman Problem (H-TSP)

This code first reads the data in the input file in the appropriate format and after reading it includes two different algorithms to solve the Half Traveling Salesman Problem: the initial tour generation and the 2-Opt local search algorithm. The Initial Tour generation function finds the best starting result according to the average of the coordinates of all cities and creates the tour array with half of it. Then, this tour array comes to the best result with the developed 2-opt local search algorithm. Below you can find a more detailed explanation of the 2-Opt local search algorithm we developed.

The **findPath** function is used to find a path for the half-traveling salesman problem based on the given city list (**cities**), initial tour (**tour**), and the number of cities (**numCities**). Here is a summary of its operation in five paragraphs:

1. In the first step, a matrix called **tourDistances** is created to cache the tour distances. This matrix is used to store the distances between each pair of cities.
2. The distance of the initial tour is saved in the variable **bestDistance**. This value represents the total distance of the initially found tour.
3. The **improved** variable is set to 1, indicating that the best tour can potentially be further improved.
4. The loop continues as long as improvements can be made to the tour. It terminates when no further improvements are achieved.
5. Within the loop, by swapping the positions of cities in the tour using the **swapCities** function, a new tour is created, and the distance of this new tour is saved in the **newDistance** variable. If the new distance is shorter than the current best distance (**bestDistance**), the best distance is updated, and the improvement flag (**improved**) is set to 1 again. Otherwise, the change is reversed, and the original tour is preserved.
6. The **iterations** variable is incremented by one in each iteration, and if it exceeds a certain limit (**numCities/10**), the loop is terminated.
7. In the final step, the used memory is freed, and the function completes.

This algorithm employs the 2-opt technique to improve the initial tour. 2-opt aims to reduce the tour distance by swapping the positions of cities in the tour. The improvement process is repeated until a better tour with a shorter distance is found.

The expression **numCities/10** is a limit used to control the value of **iterations** in each iteration of the loop. The purpose of this limit is to prevent the algorithm from entering infinite loops and to improve performance.

This limitation is particularly beneficial for larger numbers of cities. By reducing the number of iterations, the algorithm can run faster and achieve better results in a shorter amount of time. However, the exact value of **numCities/10** may vary depending on the specific problem and the performance of the algorithm, and it may need to be adjusted accordingly. In our performance tests, we discovered that this method was the most efficient.

Another optimization in the algorithm is the use of caching to calculate the tour distance quickly.

The function **calculateTourDistanceCache** is used to calculate the total distance of a tour by utilizing a precomputed distance matrix (**tourDistances**) instead of directly calculating distances between cities using the **calculateTourDistance** function. This caching approach offers faster computation compared to **calculateTourDistance**.

The reason why **calculateTourDistanceCache** is faster than **calculateTourDistance** is the utilization of the precomputed **tourDistances** matrix. Instead of recalculating distances for each pair of cities, the distances are stored in the **tourDistances** matrix beforehand. This allows for quick access to distance values during the computation of the tour distance.

By avoiding redundant calculations and leveraging precomputed distances, the **calculateTourDistanceCache** function significantly reduces the computation time by eliminating the need for repetitive calculations.

This function holds a crucial role in our code because if we had used a function that recalculates the tour distance every time instead of this function, our code would have run 20 to 25 times slower. This situation could have led to days of computation for very large inputs.

Division of Labor

The findPath algorithm was implemented collaboratively by both Kadir and Murat.

Kadir was responsible for tasks such as creating the initial tour array, determining half of the cities, reading the input in the appropriate format, and writing the output file in the required format.

Murat's contributions included developing the calculateTourDistanceCache function, which served as a helper function for the main algorithm. Additionally, Murat performed the verification of the output file using the half_tsp_verifier.py script.

	Test-input-1	Test-input-2	Test-input-3	Test-input-4
Tour Distance	1,606	142,727	35,232,473	5,868
Time	0sec	0.06sec	312sec	1.18sec