CSE2046
Assignment 2

Half Traveling Salesman Problem (H-TSP)

Team Members

1) 150119683-Busenur Yılmaz

2) 150119858-Yasin Çörekci

3) 150119678-Asaf Talha Gültekin

4) 150119066-Ertan Karaoğlu

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? In this assignment, we will focus on a modified version of TSP, Half TSP, where the salesman should visit exactly half of the cities and return back to the origin city.

To solve this problem, we have selected half of the cities with the origin city, then using genetic algorithm we have tried to find a minimum route.

Selection of the cities:

First we have created adjacency matrix (map) and also another matrix which contains cities and distances to other cities in increasing order. We have ordered the cities according to the total distance to the other cities. We have selected cities which have the least total distance to the other cities as the origin city.

After selecting the origin city, we select first half of the cities which are nearest to this origin city.

```cpp
std::vector<int> TSP::calculateBestInitials()
{
    std::multimap<double, int> distToPoints;
    std::vector<int> bestInitials;

    int i;
    int limit = (int) adjDistanceToPoint.size() / 2;

    for (auto& kv : adjDistanceToPoint) {
        double total = 0.0;

        i = 1;
        limit = (int) kv.second.size() / 2;

        for (auto& e : kv.second) {
            if (i++ == limit)
                break;

            total += e.first;
        }

        distToPoints.insert({ total, kv.first });
    }

    for (auto& e : distToPoints) {
        bestInitials.push_back(e.second);
    }

    return bestInitials;
}
```

```cpp
std::vector<int> TSP::extractHalf(int index)
{
    std::vector<int> vertexes;

    vertexes.push_back(index);

    int i = 1;
    int limit = (int) adjDistanceToPoint.size() / 2;

    for (auto& e : adjDistanceToPoint[index]) {
        if (i++ == limit)
            break;

        vertexes.push_back(e.second);
    }

    return vertexes;
}
```

After selecting origin city and half of the cities to travel, we have used genetic algorithms to solve Half-TSP problem.

Solving TSP:

We have used genetic algorithms. The steps are as follows.

1) Create a random population with size n, we have chosen as 100.
2) Iterate feasible times and do the following:
   a) Select members from old generation who will survive in the new generation.
   b) Create a new routes using Crossover and add them to the new generation.
   c) Mutate old generation members in the new generation with mutation ratio.

Selection:

To select a new generation from old generation we are using rank which is 1/route. Higher rank routes are more chance to be selected, but the selection algorithm is random by allowing good routes to be selected with a higher chance.

```
void TSP::selectPopulation(std::vector<Path>& paths, std::vector<Path>& newPaths, int selectSize)
{
    if (selectSize == 1) {
        newPaths.push_back(paths[0]);
        paths.erase(paths.begin());
        return;
    }

    double totalRank = 0.0;

    for (int i = 0; i < (int) paths.size(); ++i)
        totalRank += paths[i].rank;

    int selected = 0;
    double total = 0.0;

    double selectRandom = Util::rand() * totalRank;

    for (auto& path : paths) {
        total += path.rank;

        if (total > selectRandom)
            break;

        ++selected;
    }

    if (selected >= (int) paths.size())
        selected = (int) paths.size() - 1;

    newPaths.push_back(paths[selected]);
    paths.erase(paths.begin() + selected);

    selectPopulation(paths, newPaths, selectSize - 1);
```

Crossover:

To crossover two routes, we first select a range in the route. From the first route we take the subroute and add that route to the second route in the same range. We also are careful about the cities which are not repeat twice in the crossover so we have written a code like that:

```cpp
std::vector<int> TSP::crossover(std::vector<int>& vertexes1, std::vector<int>& vertexes2)
{
    std::vector<int> newVertexes(vertexes1.size());

    int start = Util::randi(1, (int) vertexes1.size() - 1);
    int end = Util::randi(1, (int) vertexes1.size() - 1);

    if (start > end)
        std::swap(start, end);

    std::vector<int> subVertexes;

    for (int i = start; i <= end; ++i)
        subVertexes.push_back(vertexes1[i]);

    int i = 1;
    int current = 1;

    newVertexes[0] = vertexes1[0];

    while (i < start) {
        int item = vertexes2[current];

        if (std::find(subVertexes.begin(), subVertexes.end(), item) == subVertexes.end()) {
            newVertexes[i] = item;
            ++i;
        }

        ++current;
    }

    for (int i = start; i <= end; ++i)
        newVertexes[i] = subVertexes[i-start];

    i = end + 1;

    while (i < (int) vertexes1.size()) {
        int item = vertexes2[current];

        if (std::find(subVertexes.begin(), subVertexes.end(), item) == subVertexes.end()) {
            newVertexes[i] = item;
            ++i;
        }

        ++current;
    }

    return newVertexes;
}
```

New route created after crossover is added to the new generation by calculating its total distance and rank.

Mutation:

To mutate a route, with a mutation rate we decide a city index will be changed with a random city in the route or not. If a city will be mutated in the route, we select a random city in the route and we swap these two cities. The mutation code is as follows:

```
void TSP::mutate(std::vector<int>& vertexes, double mutationRate)
{
    for (int i = 1; i < (int) vertexes.size(); ++i) {
        if (Util::rand() < mutationRate) {
            int j = Util::randi(1, (int) vertexes.size() - 1);

            int item = vertexes[i];
            vertexes[i] = vertexes[j];
            vertexes[j] = item;
        }
    }
}
```

The team had regular meetings throughout the project to distribute the workload and collaborate effectively. Each team member contributed equally when implementing codes and writing the report. For more detailed information, please refer to the provided table.

WORK SHARING CHART	
Team Member	**Task**
Yasin Çörekci	**Research approximation algorithms for TSP and Half TSP**
	Implement a local search heuristic for the problem
	Write code for parsing input and generating output files
Busenur Yılmaz	**Investigate different heuristics for selecting cities**
	Design and implement a genetic algorithm approach
	Optimize the code for efficiency and scalability
Asaf Talha Gültekin	**Develop a simulated annealing algorithm for Half TSP**
	Test and debug the code for correctness
	Assist in writing the project report
Ertan Karaoğlu	**Experiment with different neighborhood search techniques**
	Evaluate and compare the performance of different algorithms
	Collaborate on the project report