Parallel Computing Lab - Assignment 4

Mohd Ubaid Shaikh
Roll no. 180001050

Under the Guidance of
Dr. Chandresh Kumar Maurya

Computer Science and Engineering

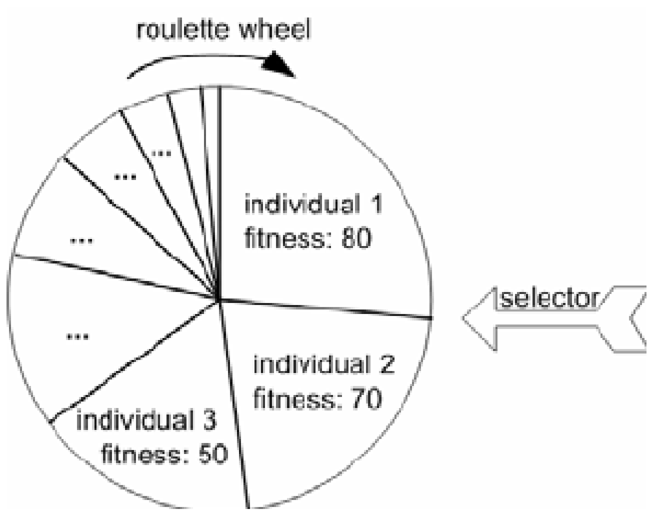
April 23, 2021

Genetic Algorithm Pseudo Code

create and initialize a population P(0);
Repeat
    Evaluate the fitness, $f(x_i)$, for all x_i belonging to P(t);
    Select fittest people from the population to become parents
    Perform cross-over to produce offspring;
    Perform mutation on offspring;
    Select population P(t+1) of new generation;
    Advance to the new generation, i.e. t = t+1;
until max generation reached;

Reference: Computational Intelligence Class Slides

Each individual is represented by a permutation of city ids
Fitness of an individual = 1 / (distance of the tour)
For selecting parents from a population, I used the roulette wheel selection which is also known as fitness proportionate selection. Its basis is that individuals having higher fitness have higher chance of getting selected.



For crossover, I am using two point crossover, a randomly chosen subarray of the first parent is copied into the child and remaining genes are taken from the second such that cities are not duplicated.

OpenMP parallel constructs used:

```
#pragma omp parallel for reduction(max:best_fitness)
    for(int i = 0; i < population.routes.size(); ++i){
        computeFitness(population.routes[i]);
        if(population.routes[i].fitness > best_fitness){ /* if fitness of
current route is better that current best_fitness, then */
            best_fitness = population.routes[i].fitness; /* update best_fitness
*/
        }
    }
```

```
#pragma omp parallel for
    for(int i = 0; i < pop_size; ++i){
        population.routes[i] = generateRandomRoute(i);
    }
```

```
#pragma omp parallel for
    for(int k = 0; k < offsprings.size(); ++k){
        for(int i = 0; i < cities_cnt; ++i){
            int random = rand() % 10000;
            float rand_val = random / 10000.0;
            if(rand_val  < mutation_rate){
                int j = rand()%cities_cnt;
                swap(offsprings[k].cities[i], offsprings[k].cities[j]);
            }
        }
    }
```

```
#pragma omp parallel for
    for(int i = elite_size; i < parents.size(); ++i){
        Route child;
        child.id = i;
        performCrossOver(parents[i], parents[parents.size() - 1 - i], child);
        new_generation[i] = child;
    }
```

```
#pragma omp parallel for reduction(+:fitness_sum)
    for(int i = 0; i < people.size(); ++i)
```

```
        fitness_sum += people[i].fitness;
```

```
#pragma omp parallel for
   for(int i = elite_size; i < people.size(); ++i){
       int num = rand()%10000;
       float rand_num = num/10000.0;
       for(auto &person:people){
           if(rand_num < person.fitness){
               fit_people[i] = person;
               break;
           }
       }
   }
```

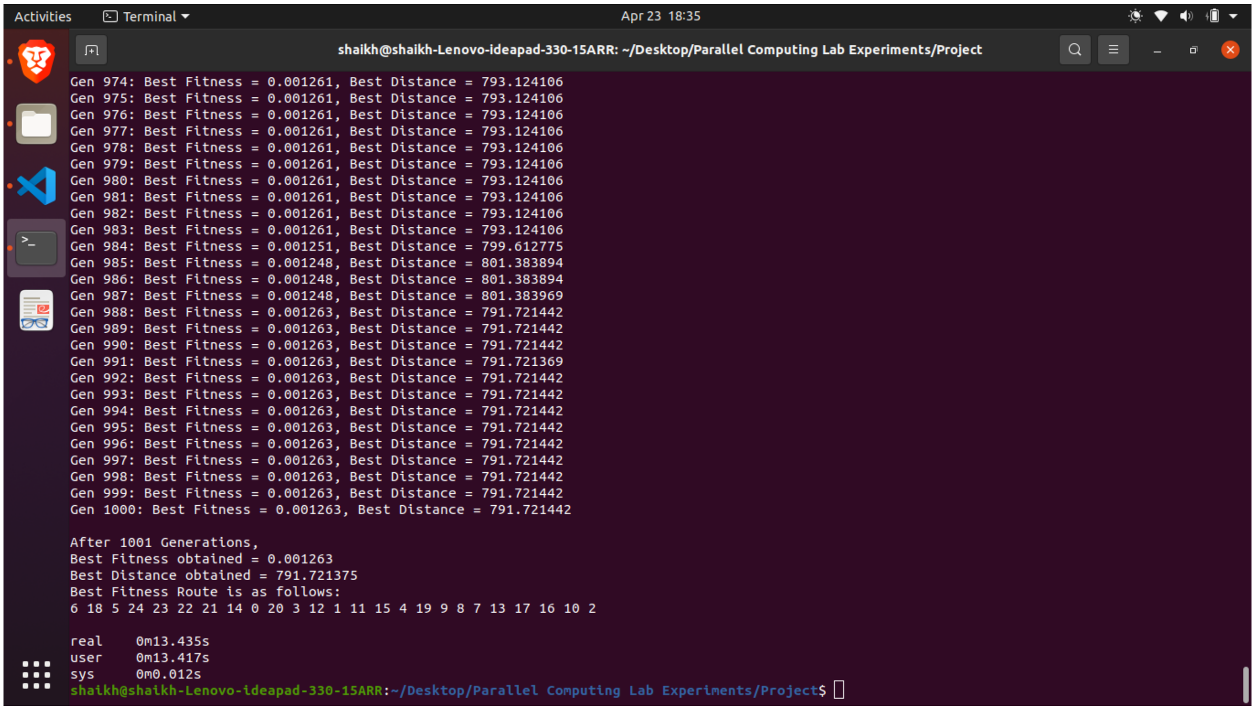
```
#pragma omp parallel for
   for(int i = 0; i < cities_cnt; ++i){
       for(int j = i + 1; j < cities_cnt; ++j){
           float dist = sqrt(sqr(cities[i].x - cities[j].x) + sqr(cities[i].y -
cities[j].y));
           graph[i][j] = dist;
           graph[j][i] = dist;
       }
   }
```

Input:

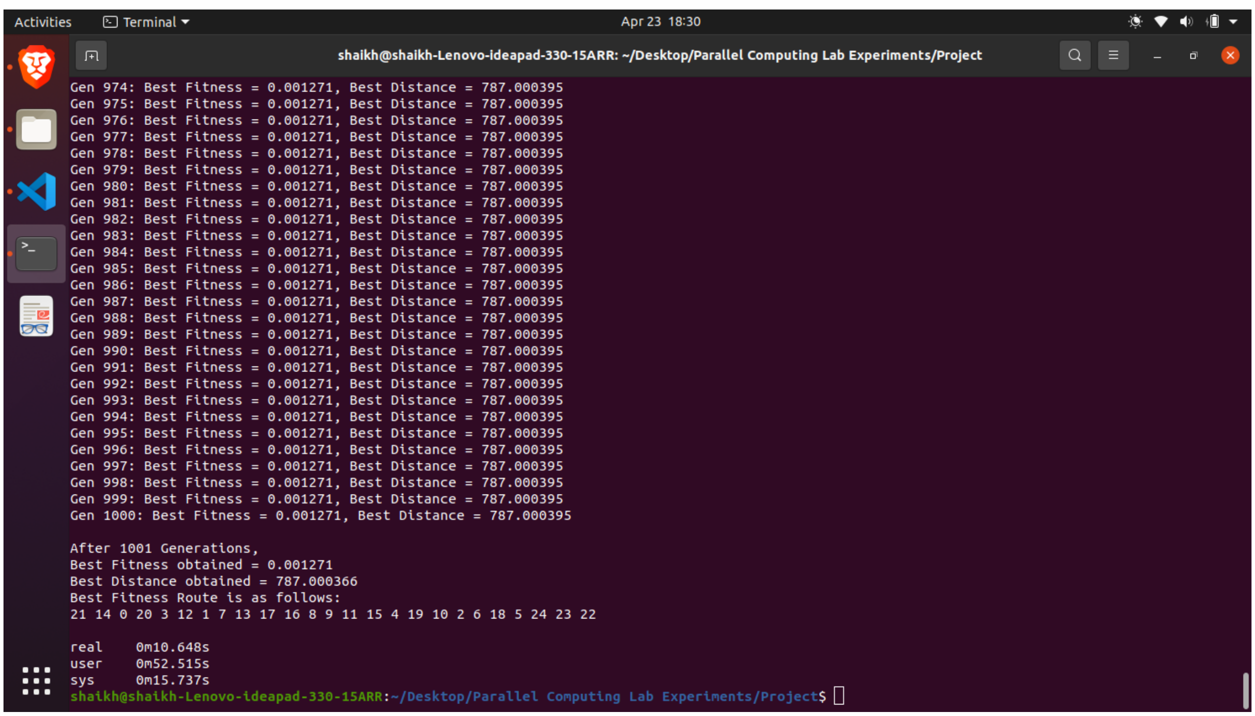
```
25
171 190
135 90
25 15
137 128
147 4
25 134
8 97
98 94
114 77
138 69
90 44
158 76
139 99
93 93
97 160
151 16
106 68
84 87
16 132
136 22
156 163
60 162
57 164
41 159
29 171
```

Output:

Population size = 1000, Max no. of iterations (generations) = 1000, No. of cities = 25
For Serial,

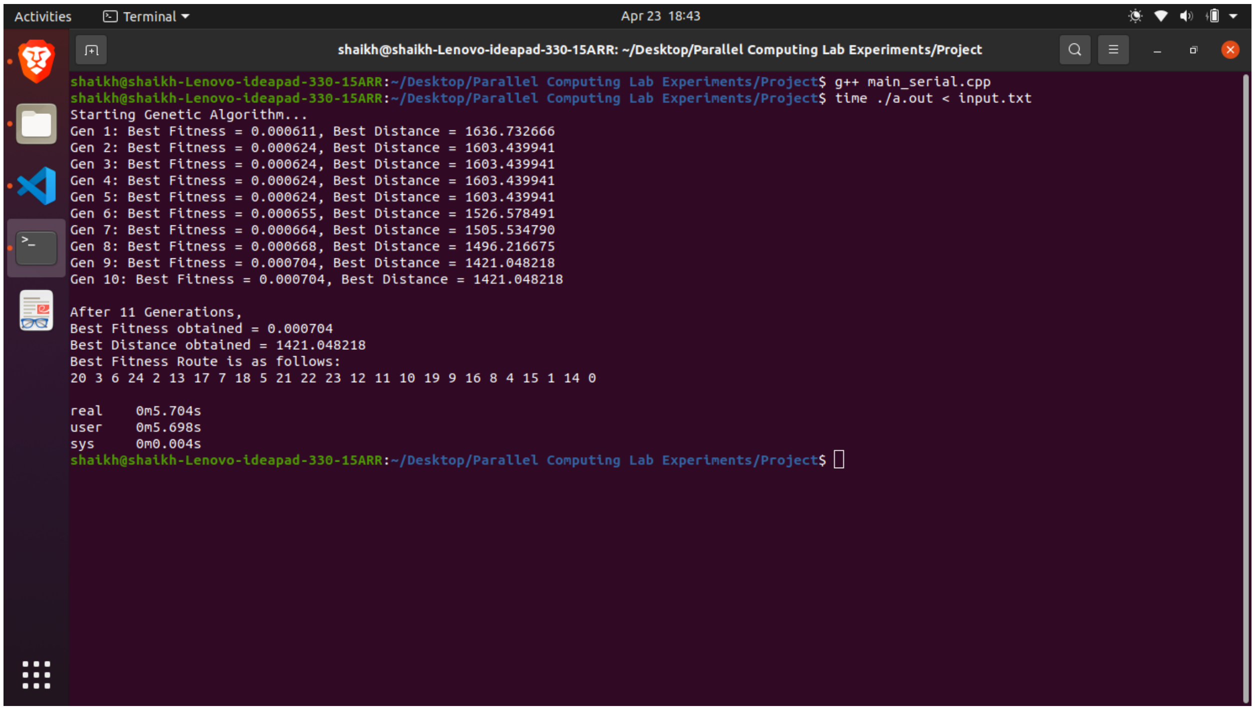


For Parallel,

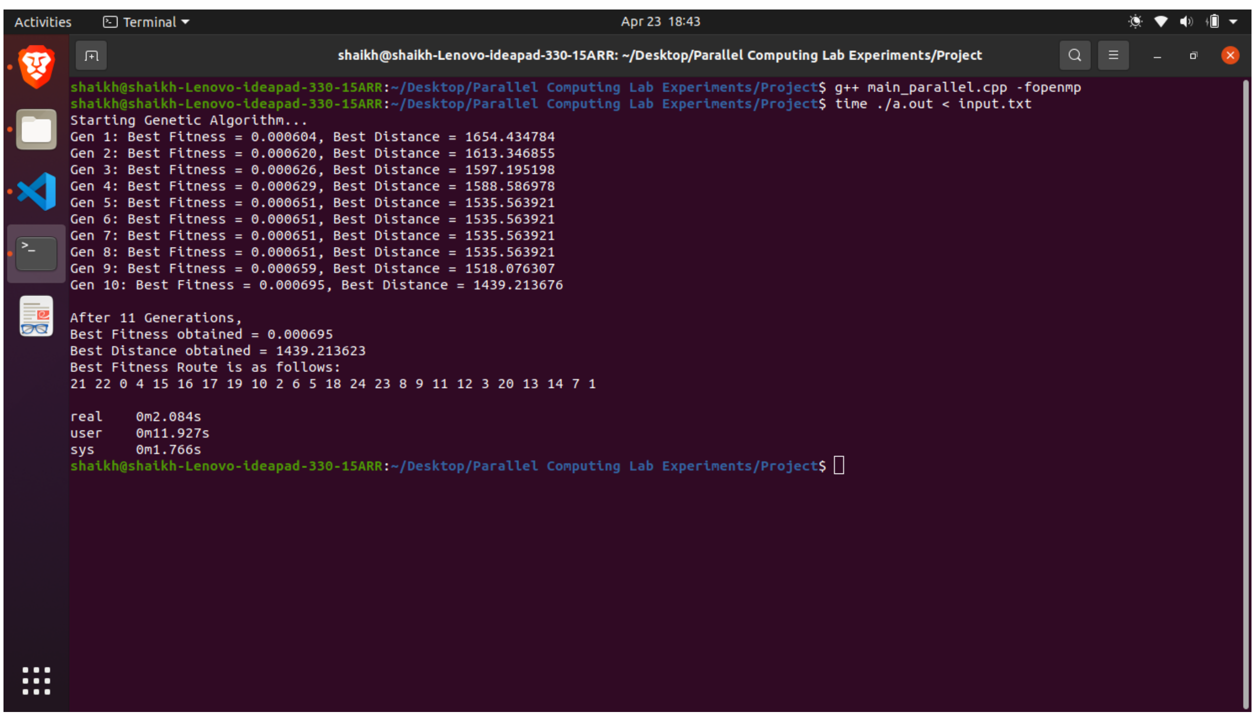


Speedup = Serial-time/Parallel-time = 13.435/10.648 = 1.2617
Minimum value of distance = 787.00366

As I have parallelized the iterations which depend on population size, better speed up is obtained when there is a bigger value of population size. For example:
When, Population size = 10000, Max no. of iterations (generations) = 10, No. of cities = 25
For Serial,



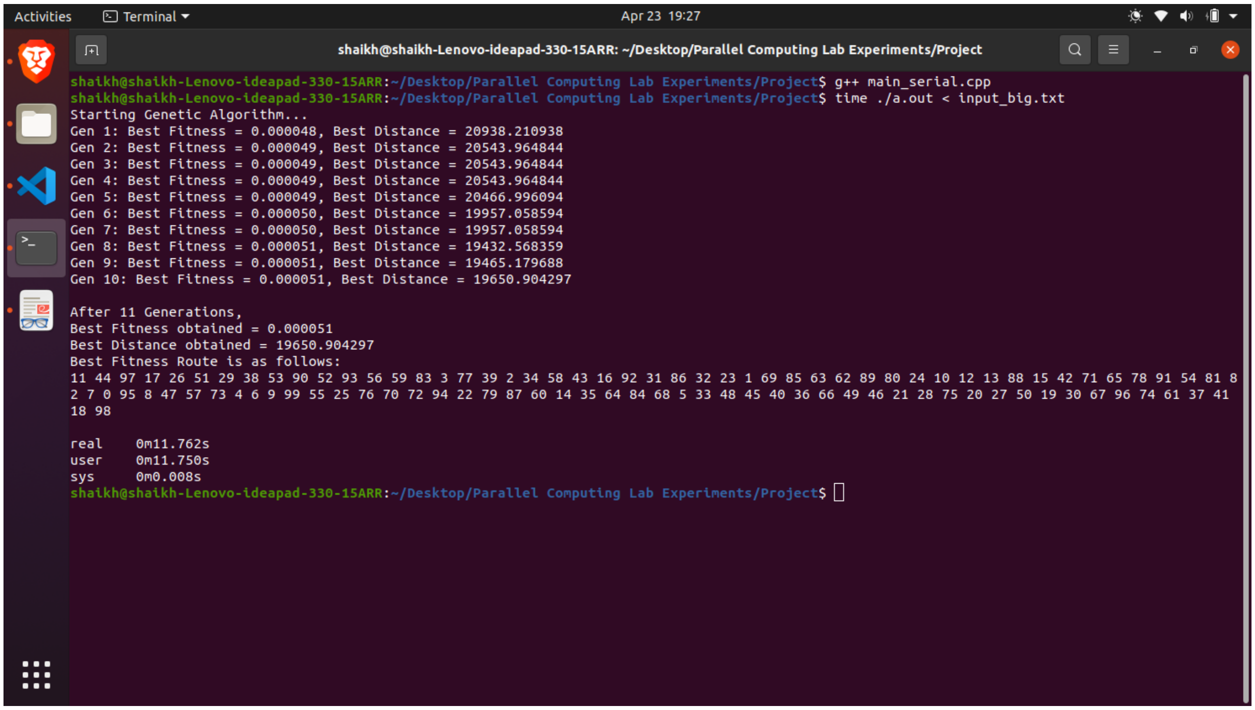
For Parallel,



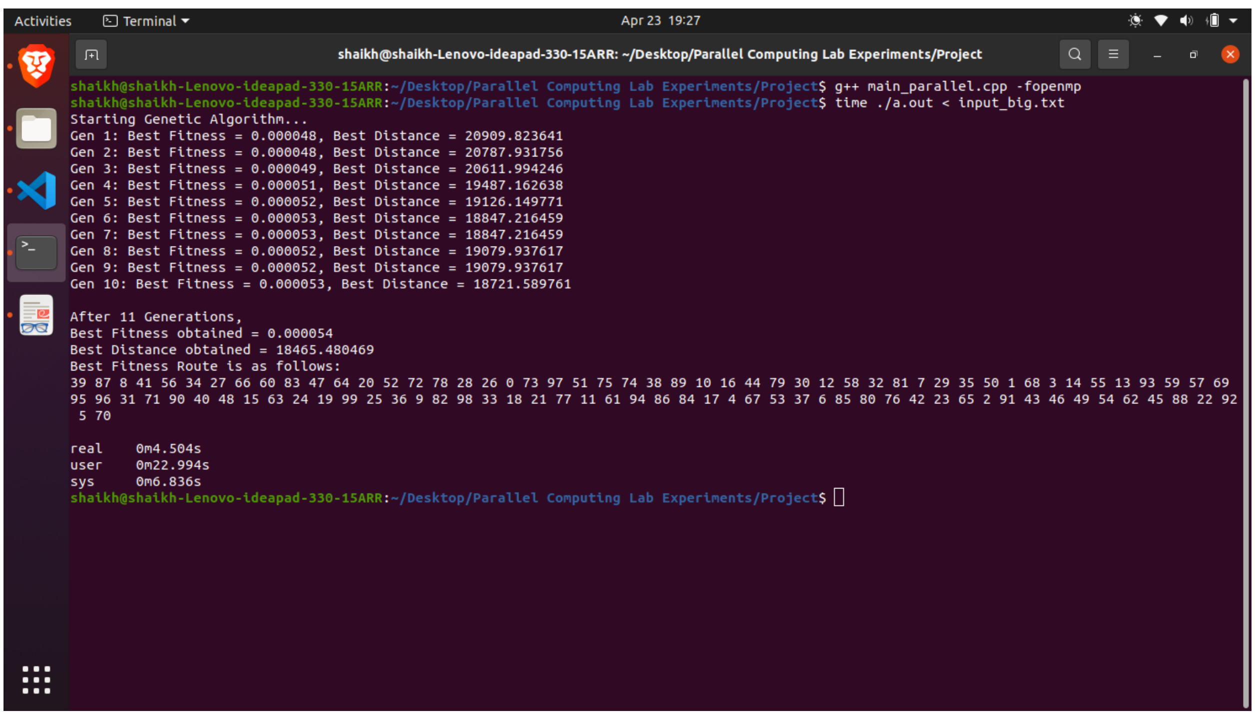
Speed up = Serial-time/Parallel-time = 5.704 / 2.084 = 2.7370 which is very near to 3

When, Population size = 10000, Max no. of iterations (generations) = 10, No. of cities = 100

For Serial,



For Parallel,



Speed up = Serial-time/Parallel-time = 11.762 / 4.504 = 2.6115

Code:

```
#include <cstdio> /* for using printf and scanf */
#include <cmath> /* for using sqrt */
#include <vector> /* for using vector */
#include <numeric> /* for using iota function */
#include <ctime> /* for using time(0) for seed srand() function */
#include <algorithm> /* for using sort function */

#define INF 999999 /* defining infinity to be 999999*/
#define sqr(x) ((x) * (x)) /* it computes square of the given number */
using namespace std;

/* my genetic algorithm parameters */
int elite_size = 20; /* it indicates the no. of elite people that will be
passed on to the next generation */
float mutation_rate = 0.01; /* for introducing variation in our population by
randomly swapping two cities in a route */
int max_generations = 1000; /* max no. of iterations of genetic algorithm */
int pop_size = 1000; /* population size of each generation */


int cities_cnt; /* no. of cities that the salesman has to visit */
int unique_id; /* used for alloting ids to routes in a population */
vector<vector<float>> graph; /* for adjacency matrix representation of the
distances between the cities */

/* A structure for representing a city with id and (x, y) coordinates */
struct City{
   int x, y, id;
};

/* A structure for representing a route with id, fitness, its distance of
travelling through the route and the permutaion of cities */
struct Route{
   int id;
   float fitness = 0;
   float dist = 0;
   vector<int> cities; /* storing just the ids of the cities */
};

/* A structure for representing a population. Every population has a generation
number, several routes and a best route out of several routes*/
struct Population{
```

```
    int gen_no; /* generation number */
    Route best_route;
    vector<Route> routes;
};

/* It computes fitness of the given route. */
void computeFitness(Route &route){
    float dist = 0;
    for(int i = 1; i <= cities_cnt; ++i){
        dist += graph[route.cities[i % cities_cnt]][route.cities[i - 1]];
    }
    route.dist = dist;
    route.fitness = 1 / dist; /* fitness of a routes is defined to be the
inverse of the distance of covering the route */
}

/* It evaluates the fitness of each route in the given population. It also
finds the route with the best fitness */
void evaluateFitness(Population &population){
    float best_fitness = 0;
    for(auto &r:population.routes){
        computeFitness(r);
        if(r.fitness > best_fitness){ /* if fitness of current route is better
that current best_fitness, then */
            best_fitness = r.fitness; /* update best_fitness */
            population.best_route = r; /* and best route */
        }
    }
}

/* It generates a random route. It is used during the creation of 1st
generation */
Route generateRandomRoute(int id){
    Route route;
    route.id = id;
    route.cities = vector<int>(cities_cnt);
    vector<int> temp(cities_cnt);
    iota(temp.begin(), temp.end(), 0); /* fills the temp vector with values 0,
1, 2, ..., cities_cnt-1 */
    for(int i = 0; i < cities_cnt; ++i){
        int random_index = rand() % temp.size();
        route.cities[i] = temp[random_index];
        swap(temp[random_index], temp[temp.size() - 1]);
        temp.pop_back();
    }
```

```
    return route;
}

/* It generates a random population. It is used for creating the 1st generation
of population */
Population generateRandomPopulation(int gen_no, int pop_size){
    Population population;
    population.gen_no = gen_no;
    population.routes = vector<Route>(pop_size);
    for(int i = 0; i < pop_size; ++i){
        population.routes[i] = generateRandomRoute(i);
    }
    return population;
}

/* It is used to mutate the offspring passed as argument. For each route, each
city
will be swapped with another city with a probability of mutation_rate */
void mutate(vector<Route> &offsprings){
    for(auto &route:offsprings){
        for(int i = 0; i < cities_cnt; ++i){
            int random = rand() % 10000;
            float rand_val = random / 10000.0;
            if(rand_val  < mutation_rate){
                int j = rand()%cities_cnt;
                swap(route.cities[i], route.cities[j]);
            }
        }
    }
}

/* It is used to shuffle the given parents. It is used to bring randomness in
the order of parents.
Randomness is needed so that the creation of new generation is not affected by
the order of parents */
void shuffle(vector<Route> &parents){
    for(int i = 0; i < parents.size(); ++i){
        int index1 = rand() % parents.size();
        int index2 = rand() % parents.size();
        swap(parents[index1], parents[index2]);
    }
}

/* It is used for perfoming crossover during breeding. It uses two point
crossover.
```

```
First, a subarray of genes of first parent are taken and the rest of the genes
are taken from the
second parent, such that cities do not get repeated */
void performCrossOver(Route p1, Route p2, Route &child){

   int geneA = rand()%cities_cnt;
   int geneB = rand()%cities_cnt;

   int startGene = min(geneA, geneB); /* startGene and endGene correspond to
the starting index and */
   int endGene = max(geneA, geneB); /* ending index of the subarray to be taken
from parent 1 */

   for(int i = startGene; i < endGene; ++i){
       child.cities.push_back(p1.cities[i]);
   }

   for(auto &city:p2.cities){ /* fill the empty places with cities from 2nd
parent, such that no city gets repeated */
       if(find(child.cities.begin(), child.cities.end(), city) ==
child.cities.end()){
           child.cities.push_back(city);
       }
   }
}

/* It performs Breeding. First, elite_size number of top parents are passed
on/promoted to the next generation.
Then parents order is shuffled to bring some randomness. The pop_size -
elite_size no. of childs are created using crossover operation*/
vector<Route> performBreeding(vector<Route> &parents){
   unique_id = 0;
   vector<Route> new_generation;
   for(int i = 0; i < elite_size; ++i){ /* pass on the elite people to the next
generation */
       new_generation.push_back(parents[i]);
       new_generation.back().id = unique_id++;
   }
   shuffle(parents);

   for(int i = 0; i < parents.size() - elite_size; ++i){
       Route child;
       child.id = unique_id++;
       performCrossOver(parents[i], parents[parents.size() - 1 - i], child);
       new_generation.push_back(child);
```

```
    }
    return new_generation;
}

/* It selects the fittest people to become parents. The basis rule for
selecting people is that
people having higher fitness value have higher chances of becoming parents */
vector<Route> selectFittest(vector<Route> people){
    sort(people.begin(), people.end(), [](Route &a, Route &b){ /* first, we
arrange people in descending order of fitness */
        return a.fitness > b.fitness;
    });

    /* now, we compute cumulative fitness for each people */
    float fitness_sum = 0.0, prev_probability = 0.0;
    for(auto &r:people)
        fitness_sum += r.fitness;

    for(auto &r:people){
        r.fitness = prev_probability + (r.fitness / fitness_sum);
        prev_probability = r.fitness;
    }

    vector<Route> fit_people(people.size()); /* declaring array for storing fit
peopel */

    for(int i = 0; i < elite_size; ++i){ /* elite_size of top people will be
selected to become parents */
        fit_people[i] = people[i];
    }

    /* remaining number of people will be selected using routlette wheel
selection (fitness proportionate selection) */
    for(int i = elite_size; i < people.size(); ++i){
        int num = rand()%10000;
        float rand_num = num/10000.0;
        for(auto &person:people){
            if(rand_num < person.fitness){
                fit_people[i] = person;
                break;
            }
        }
    }
    return fit_people;
}
```

```
/* It prints the data of population passed as argument */
void printPopulation(Population population){
   printf("The generation %d is as follows:\n", population.gen_no);
   for(auto &r:population.routes){
       printf("%d: ", r.id);
       for(auto &city:r.cities){
           printf("%d ", city);
       }
       printf("| dist = %f\n", r.dist);
   }
}

int main(){
   scanf("%d", &cities_cnt); /* take the number of cities as input */
   vector<City> cities(cities_cnt); /* declaring a vector of cities to store
the (x, y) coordinates of cities given as input */
   for(int i = 0; i < cities_cnt; ++i){
       cities[i].id = i;
       scanf("%d %d", &cities[i].x, &cities[i].y);
   }

   /* allocating cities_cnt x cities_cnt space for adjacency matrix
representation of the distances among the given cities*/
   graph = vector<vector<float>>(cities_cnt, vector<float> (cities_cnt, INF));

   /* computing distances among the cities and filling our adjacency matrix */
   for(int i = 0; i < cities_cnt; ++i){
       for(int j = i + 1; j < cities_cnt; ++j){
           float dist = sqrt(sqr(cities[i].x - cities[j].x) + sqr(cities[i].y -
cities[j].y));
           graph[i][j] = dist;
           graph[j][i] = dist;
       }
   }

   srand(time(0)); /* seeding our rand() function */
   Population population = generateRandomPopulation(1, pop_size); /* generate
the first population */
   printf("Starting Genetic Algorithm...\n");
   while(population.gen_no <= max_generations){
       evaluateFitness(population); /* evaluate the fitness, f(x), for all x
belonging to current generation; */
       printf("Gen %d: Best Fitness = %f, Best Distance = %f\n",
population.gen_no, population.best_route.fitness, population.best_route.dist);
```

```
        vector<Route> fittest_people = selectFittest(population.routes); /*
select pop_size routes for becoming parent */
        vector<Route> offsprings = performBreeding(fittest_people); /* perform
cross-over among the selected parents to produce offspring; */
        mutate(offsprings); /* mutate the offsprings with probability as
mutation_rate */
        population.routes = offsprings; /* update population; */
        population.gen_no += 1; /* advance to the next generation */
    }

    evaluateFitness(population); /* evaluate the fitness, f(x), for all x
belonging to current generation; */

    /* print the final results */
    printf("\nAfter %d Generations,\n", population.gen_no);
    printf("Best Fitness obtained = %f\n", population.best_route.fitness);
    printf("Best Distance obtained = %f\n", population.best_route.dist);
    printf("Best Fitness Route is as follows:\n");
    for(auto &city:population.best_route.cities){
        printf("%d ", city);
    }
    printf("\n");
    return 0;
}
```