Name : Pratik Kanhaiya Rathod
Roll : 321056
PRN : 22010885
Batch : A2

Assignment No. 5

Aim : Implement Travelling Salesman problem using Genetic Algorithm

Theory:

- **What is Genetic Algorithm?**
  1) Genetic algorithm is a type of optimization algorithm that is inspired by the process of natural selection and evolution. It is a search-based optimization technique that is used to find the best solution to a problem by imitating the process of natural selection.
  2) The genetic algorithm starts with a population of candidate solutions, which are encoded as chromosomes. These chromosomes are typically represented as strings of binary digits, where each digit corresponds toa specific characteristic or feature of the solution.
  3) The genetic algorithm then uses a set of genetic operators, such as crossover and mutation, to create new solutions from the existing population. These new solutions are then evaluated based on a fitness function, which measures how well each solution performs.
  4) The best-performing solutions are selected to form the next generationof candidate solutions, and the process is repeated until a satisfactory solution is found or a stopping criterion is met.
  5) The genetic algorithm is widely used in various fields such as engineering, computer science, economics, and biology, to solve problems that are difficult to solve using traditional optimization techniques.

- **Travelling salesman problem:**
  1) The traveling salesman problem (TSP) is an algorithmic problem tasked with finding the shortest route between a set of points and locations that must be visited. In the problem statement, the points are the cities a salesperson might visit. The salesman‘s goal is to keep both the travel costs and the distance traveled as low as possible.

2) Focused on optimization, TSP is often used in computer science to findthe most efficient route for data to travel between various nodes. Applications include identifying network or hardware optimization methods. It was first described by Irish mathematician W.R. Hamilton and British mathematician Thomas Kirkman in the 1800s through the creation of a game that was solvable by finding a Hamilton cycle, whichis a non-overlapping path between all nodes.

Implementation of Travelling Salesman problem using Genetic algorithm inC++:

Main 5 phases of implementation are:

1) Creating initial population.
2) Calculating fitness.
3) Selecting the best genes.
4) Crossing over.
5) Mutating to introduce variations.

Implemetation :

```
#include<bits/stdc++.h>

#define INF INT_MAX

using namespace std;

class Chromosome{

    public:
    // Represent the path from city-to-city with start city same as end city.
    string pathString;

    // The cost of the given path is the sum of all the subpaths.
    int pathCost;


    Chromosome(){
        pathString = "";
        pathCost = 0;
    }

    Chromosome(string _pathString){ pathString = _pathString; }
```

```
    Chromosome(string _pathString, int _pathCost){
        pathString = _pathString;
        pathCost = _pathCost;
    }

n ostream& Returns the implicit ostream object with formated output to print
on console.

    friend ostream& operator <<(ostream& s, const Chromosome& _chromosome){
        s << "Path String: " << _chromosome.pathString << " |\t " << "Path
Cost: " << _chromosome.pathCost;
        return s;
    }

};


class TSP{

    public:

    //  Stores the total number of cities in a map of current instance of TSP.
    int numberOfCities;

    // Vector that stores the cost of an edge/path between pairs of cities of
a map.
    vector<vector<int>> map;


    TSP(int _numberOfCities, vector<vector<int>>& _map){
        srand(time(0));
        numberOfCities = _numberOfCities;
        map = _map;

        geneticAlgorithm();
    }

    string getOptimalPath(){ return optimalPath; }

    int getOptimalPathCost(){ return optimalPathCost; }

    private:

    // Constant variable defining the population size of the chromosomes in a
pool.
    const int POPULATION_SIZE = 10;
```

```
    // Constant variable defining the city to start tour from.
    const int START_END_CITY = 0;

    // Constant variable defining the maximum number of generation to perform
the mutation.
    const int GENERATION_THRESHOLD = 5;

    // Stores the optimal path or tour of cities.
    string optimalPath = "";

    // Stores the optimal tour's cost starting from start city to the end city
of the calculated path.
    int optimalPathCost = INF;

    // Indicates the current generation number.
    int currentGeneration = 1;

    // The cooling variable whose value decreases after every iteration of new
generation.
    int temperature = 10000;


    bool isCharRepeated(char ch, string str){
        int n = str.length();
        for(int i = 0; i < n; i+=3) if(str[i] == ch) return true;
        return false;
    }


    string createPathString(){
        string randomPathString = "";
        randomPathString += char(START_END_CITY+65);
        int currLen = 1;
        while(currLen < numberOfCities){
            char currCity = char(((1 + (rand() % (numberOfCities-1))) + 65));
            if(isCharRepeated(currCity, randomPathString)) continue;
            else{
                currLen++;
                randomPathString += "->";
                randomPathString += currCity;
            }
        }
        randomPathString += "->";
        randomPathString += char(START_END_CITY+65);
        return randomPathString;
    }

    int getPathCost(string path){
        int cost = 0;
```

```
        for(int i = 0, k = 0; k < numberOfCities; k++, i+=3){
            int r = path[i] - 65;
            int c = path[i+3] - 65;
            if(map[r][c] == INF) return INF;
            cost += map[r][c];
        }
        return cost;
    }


    string mutatePathString(string currentPathString){
        while(true){
            char cityOne = (1 + (rand() % (numberOfCities-1)));
            char cityTwo = (1 + (rand() % (numberOfCities-1)));
            if(cityOne != cityTwo){
                currentPathString[cityOne*3] ^= currentPathString[cityTwo*3];
                currentPathString[cityTwo*3] ^= currentPathString[cityOne*3];
                currentPathString[cityOne*3] ^= currentPathString[cityTwo*3];
                break;
            }
        }
        return currentPathString;
    }

    void geneticAlgorithm(){
        auto comp = [](Chromosome &ch1, Chromosome &ch2) { return ch1.pathCost
< ch2.pathCost; };
        vector<Chromosome> population;

        // Populating the pool with chromosomes.
        for(int i = 0; i < POPULATION_SIZE; i++){
            string currentPathString = createPathString();
            int currentPathCost = getPathCost(currentPathString);

            Chromosome currentChromosome =  Chromosome(currentPathString,
currentPathCost);
            population.push_back(currentChromosome);
        }

        // Printing the population.
        cout << "Initial Population: " << endl;
        for(auto ch: population) cout << ch << endl;
        cout << endl;

        // Performing population crossing and mutation.
        while(temperature > 1000 && currentGeneration <=
GENERATION_THRESHOLD){
            sort(population.begin(), population.end(), comp);
            vector<Chromosome> newPopulation;
```

```
            for(int i = 0; i < POPULATION_SIZE; i++){
                Chromosome currentChromosome = population[i];

                while(true){
                    // Mutating chromosomes
                    string mutatedPathString =
mutatePathString(currentChromosome.pathString);
                    int mutatedPathCost = getPathCost(mutatedPathString);
                    Chromosome mutatedChromosome =
Chromosome(mutatedPathString, mutatedPathCost);

                    if(mutatedChromosome.pathCost <=
currentChromosome.pathCost){
                        newPopulation.push_back(mutatedChromosome);
                        break;
                    }else{
                        float probability = pow(2.7, -1 *
((float)(mutatedChromosome.pathCost - currentChromosome.pathCost) /
temperature));

                        if(probability > 0.5){
                            newPopulation.push_back(mutatedChromosome);
                            break;
                        }
                    }

                }

            }
            // Temperature recaliberation
            temperature = (temperature*90)/100;
            population = newPopulation;

            cout << "Generation " << currentGeneration << " population: " <<
endl;
            for(auto ch: population) cout << ch << endl;
            cout << endl;
            currentGeneration++;
        }

        // Storing the optimal path  and its cost
        sort(population.begin(), population.end(), comp);
        optimalPath = population[0].pathString;
        optimalPathCost = population[0].pathCost;

    }
};
```

```
int main(){
    vector<vector<int>> map = { {0, 2, INF, 12, 5},
                                {2, 0, 4, 8, INF},
                                {INF, 4, 0, 3, 3},
                                {12, 8, 3, 0, 10},
                                {5, INF, 3, 10, 0} };
    int numberOfCities = 5;
    TSP testMap = TSP(numberOfCities, map);
    cout << "Optimal path is " << testMap.getOptimalPath() << ", and its cost
is " << testMap.getOptimalPathCost() << endl;
    return 0;
}
```

Output :

```
PS C:\Users\PRATIK RATHOD\OneDrive\Desktop\TY\DAA\gitassg\Assg5> cd "c:\Users\PRATIK RATHOD\OneDrive\Desktop\TY\DAA\gitassg\Assg5\" ; i
f ($?) { g++ TSP_GeneticProblem.cpp -o TSP_GeneticProblem } ; if ($?) { .\TSP_GeneticProblem }
Initial Population:
Path String: A->D->C->B->E->A |  Path Cost: 2147483647
Path String: A->C->B->E->D->A |  Path Cost: 2147483647
Path String: A->D->C->E->B->A |  Path Cost: 2147483647
Path String: A->D->E->C->B->A |  Path Cost: 31
Path String: A->D->E->C->B->A |  Path Cost: 31
Path String: A->B->E->C->D->A |  Path Cost: 2147483647
Path String: A->D->E->B->C->A |  Path Cost: 2147483647
Path String: A->E->D->B->C->A |  Path Cost: 2147483647
Path String: A->D->B->C->E->A |  Path Cost: 32
Path String: A->C->E->D->B->A |  Path Cost: 2147483647

Generation 1 population:
Path String: A->D->B->C->E->A |  Path Cost: 32
Path String: A->D->B->C->E->A |  Path Cost: 32
Path String: A->B->D->C->E->A |  Path Cost: 21
Path String: A->D->C->E->B->A |  Path Cost: 2147483647
Path String: A->C->B->D->E->A |  Path Cost: 2147483647
Path String: A->B->C->E->D->A |  Path Cost: 31
Path String: A->E->B->C->D->A |  Path Cost: 2147483647
Path String: A->D->B->E->C->A |  Path Cost: 2147483647
Path String: A->C->D->B->E->A |  Path Cost: 2147483647
Path String: A->B->E->D->C->A |  Path Cost: 2147483647
```

```
Generation 2 population:
Path String: A->B->C->D->E->A |  Path Cost: 24
Path String: A->B->C->D->E->A |  Path Cost: 24
Path String: A->B->D->C->E->A |  Path Cost: 21
Path String: A->C->B->E->D->A |  Path Cost: 2147483647
Path String: A->C->B->D->E->A |  Path Cost: 2147483647
Path String: A->B->E->C->D->A |  Path Cost: 2147483647

Generation 3 population:
Path String: A->B->C->D->E->A |  Path Cost: 24
Path String: A->B->D->C->E->A |  Path Cost: 21
Path String: A->B->C->E->D->A |  Path Cost: 31
Path String: A->E->C->D->B->A |  Path Cost: 21
Path String: A->D->B->C->E->A |  Path Cost: 32
Path String: A->D->E->C->B->A |  Path Cost: 31
Path String: A->D->B->C->E->A |  Path Cost: 32
Path String: A->C->E->B->D->A |  Path Cost: 2147483647
Path String: A->E->B->D->C->A |  Path Cost: 2147483647
Path String: A->B->C->E->D->A |  Path Cost: 31

Generation 4 population:
Path String: A->B->C->D->E->A |  Path Cost: 24
Path String: A->B->C->D->E->A |  Path Cost: 24
Path String: A->B->C->E->D->A |  Path Cost: 31
Path String: A->E->C->B->D->A |  Path Cost: 32
Path String: A->E->D->C->B->A |  Path Cost: 24
Path String: A->E->C->B->D->A |  Path Cost: 32
Path String: A->D->E->C->B->A |  Path Cost: 31
Path String: A->B->D->C->E->A |  Path Cost: 21
Path String: A->D->E->B->C->A |  Path Cost: 2147483647
Path String: A->D->B->E->C->A |  Path Cost: 2147483647

Generation 5 population:
Path String: A->E->D->C->B->A |  Path Cost: 24
Path String: A->B->C->E->D->A |  Path Cost: 31
Path String: A->B->C->E->D->A |  Path Cost: 31
Path String: A->B->D->C->E->A |  Path Cost: 21
Path String: A->E->C->B->D->A |  Path Cost: 32
Path String: A->E->D->C->B->A |  Path Cost: 24
Path String: A->B->C->E->D->A |  Path Cost: 31
Path String: A->E->C->D->B->A |  Path Cost: 21
Path String: A->D->C->B->E->A |  Path Cost: 2147483647
Path String: A->D->C->E->B->A |  Path Cost: 2147483647

Optimal path is A->B->D->C->E->A, and its cost is 21
```