Project Name: TSP using SA

DEPT: COMPUTER SCIENCE DOMAIN PROGRAM

COLLEGE: Indian Institute of Technology (Daksh Gurukul)

Submitted By:

Name: PIYUSH SINGH

STUDENT CODE: IITGCS_24061330

INDEXING

TSP using SA

Write a C++ Program to find optimal TSP (Travelling Sales Man Problem) tour using Simulated Annealing Meta Heuristics.

TSP Tours: Given a collection of cities and the cost of travel between each pair of them, the traveling salesman problem, or TSP for short, is to find the cheapest way of visiting all of the cities and returning to your starting point. In the standard version, the travel costs are symmetric in the sense that traveling from city X to city Y costs just as much as traveling from Y to X and in this case it is Euclidian distance between two city X and Y.

This problem is know to be difficult problem in ter of complexity and people use heuristics/meta heuristics to solve this problem.

A solution S to problem have all the city in a sequence (in a array with out repetition) and the solution is optimal if the solution have least cost.

Simulated Annealing for TSP: A neighbour solution S' of S have an incremental one swap of two city position in the solution array as compared to S. Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function.

Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. SA start with initial random solution S; in each iteration it generate a neighbour solution S' with one swap of two cities position in the solution, the new solution is accepted if cost of S' is lower as compared to S, otherwise the solution S' will be accepted with a probablity (high in lower iterations and low in higher iterations).

Run simulated annealing for 10000 iterations,

Details of SA: can be found at:
https://en.wikipedia.org/wiki/Simulated_annealing

Input Data:

Data Set: http://www.math.uwaterloo.ca/tsp/visi/index.html, xqf131.tsp, xqg237.tsp,

The data set contents the points and their location. Distance between two points is Euclidian distance.

OUTPUT: final achieved cost and Solution [sequence of cities/numbers)

ABSTRACT

The Traveling Salesman Problem (TSP) is a classic optimization problem that seeks to find the shortest possible route that visits a set of cities and returns to the origin city. This report presents a C++ implementation of the TSP using the Simulated Annealing metaheuristic. Simulated Annealing is inspired by the annealing process in metallurgy and is particularly useful for solving combinatorial optimization problems like TSP. This report covers the problem definition, algorithm design, implementation, and results.

INTRODUCTION

The Traveling Salesman Problem (TSP) is a well-known problem in computer science and operations research. It is defined as follows: given a list of cities and the distances between each pair of cities, determine the shortest possible route that visits each city exactly once and returns to the origin city. The TSP is NP-hard, meaning that there is no known polynomial-time solution for it.

Simulated Annealing is a probabilistic technique for approximating the global optimum of a given function. It is particularly effective for large search spaces, making it a suitable choice for solving TSP.

What Is TSP:

The Traveling Salesperson Problem (TSP) is a well-known optimization problem in computer science and operations research. It can be described as follows:

Given a list of cities and the distances between each pair of cities, the goal is to find the shortest possible route that allows the salesperson to visit each city exactly once and return to the starting city.

Characteristics of TSP:

Input: A set of cities and a distance matrix or a set of weighted edges representing the distances between cities.

Output: The minimum-length tour that visits each city exactly once and returns to the starting point.

Nature: It is an NP-hard problem, meaning that no known algorithm can solve all instances of TSP efficiently (i.e., in polynomial time).

Importance of TSP:

Applications: TSP has applications in various fields such as logistics, planning, manufacturing, and DNA sequencing. For example, it can be used to optimize delivery routes for transportation companies to minimize time and fuel costs.

Theoretical significance: It serves as a benchmark for many optimization algorithms and has spurred the development of approximation and heuristic methods due to its computational complexity.

Solutions to TSP:

1. Exact Algorithms:

Brute-force approach: Check all possible permutations of cities, which is computationally feasible only for very small numbers of cities (complexity).

Dynamic programming (e.g., Held-Karp algorithm): Reduces the time complexity to , which is still exponential but more efficient than brute-force.

2. Approximation Algorithms:

Greedy algorithms: Construct a path incrementally but do not guarantee an optimal solution.

Christofides' algorithm: Provides a solution within 1.5 times the optimal path for metric TSP (where distances satisfy the triangle inequality).

3. Heuristic Methods:

Genetic algorithms: Use evolutionary strategies to iteratively improve a set of solutions.

Simulated annealin: Mimics the process of heating and cooling metals to find low-energy configurations.

Ant colony optimization: Inspired by the behavior of ants finding paths to food.

TSP remains an essential problem for understanding algorithmic efficiency and limitations, and it continues to inspire new computational techniques in optimization and artificial intelligence.

Simulated Annealing Algorithm

The Simulated Annealing algorithm involves the following steps:

Initialization: Start with an initial solution and set the initial temperature.

Iteration: Repeat until the stopping criteria are met:

*Generate a neighboring solution by making a small change to the current solution.

*Calculate the cost difference between the current solution and the neighboring solution.

*If the neighboring solution is better, accept it. If it is worse, accept it with a certain probability that decreases as the temperature decreases.

Cooling Schedule: Gradually reduce the temperature according to a predefined schedule.

Termination: Stop the process when the temperature is low enough or after a fixed number of iterations.

SIMULATED ANNEALING ALGORITHM

Simulated annealing is an optimization technique used for finding an approximate solution to complex optimization problems, especially those where the search space is large and traditional methods may get stuck in suboptimal solutions (local minima). The algorithm is inspired by the physical process of annealing in metallurgy, where a material is heated to a high temperature and then gradually cooled to minimize its energy and form a stable structure.

How Simulated Annealing Works:

1. Initialization:

Start with an initial solution (current state) and set a high temperature .

The temperature controls the likelihood of accepting worse solutions, allowing the algorithm to explore more widely at the start.

2. Iterative Process:

Generate a Neighboring Solution: Randomly modify the current state to create a new solution (neighboring state).

Calculate Energy Change: Evaluate the energy (objective function value) of both the current and new solutions to find the difference .

Gradually decrease following a predefined schedule (e.g., exponential decay: , where is a constant between 0 and 1).

This step reduces the likelihood of accepting worse solutions as the algorithm progresses, focusing more on refining the current solution.

3.Termination:

Stop when reaches a minimum threshold or when no significant improvement is found over several iterations.

Key Properties:

Exploration vs. Exploitation: At high temperatures, the algorithm explores more widely, allowing it to jump out of local minima. As the temperature lowers, it behaves more like a greedy algorithm, refining the best solution found.

Probability of Acceptance: The acceptance of worse solutions decreases as the temperature drops, which helps the algorithm focus on finding a global minimum.

Example Use Case: Traveling Salesman Problem (TSP)

Simulated annealing can help find a near-optimal route for visiting a set of cities with minimal travel distance. Starting with a random route, the algorithm would try small changes (e.g., swapping two cities) and accept new routes based on the temperature and resulting travel distance.

Advantages:

Escape Local Minima: The algorithm's ability to accept worse solutions helps avoid getting trapped in local minima.

Flexibility: Can be applied to various optimization problems without major changes to its structure.

Disadvantages:

Parameter Sensitivity: Performance depends on the choice of the initial temperature, cooling rate, and termination condition.

Implementation

Below is a C++ implementation of the TSP using the Simulated Annealing algorithm.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <numeric> // For std::iota

using namespace std;

struct City {
   double x, y;
};

double euclideanDistance(const City &a, const City &b) {
   return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}

double calculateTourCost(const vector<int> &tour, const vector<City> &cities) {
   double cost = 0.0;
   for (size_t i = 0; i < tour.size(); i++) {
      cost += euclideanDistance(cities[tour[i]], cities[tour[(i + 1) % tour.size()]]);
```

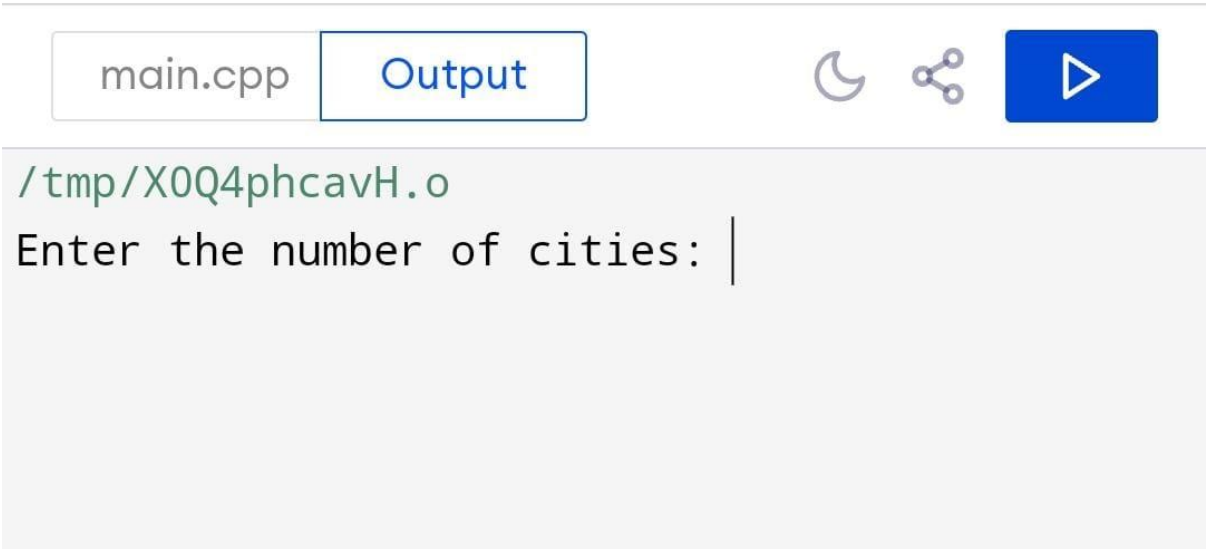
```
    }
    return cost;
}

void swapCities(vector<int> &tour) {
    int i = rand() % tour.size();
    int j = rand() % tour.size();
    // Ensure different indices for swapping
    while (j == i) {
        j = rand() % tour.size();
    }
    swap(tour[i], tour[j]);
}

vector<int> simulatedAnnealing(const vector<City> &cities, int iterations,
double startTemp, double alpha) {
    int n = cities.size();
    vector<int> currentTour(n);
    iota(currentTour.begin(), currentTour.end(), 0); // Initialize tour with 0, 1,
..., n-1

    double currentCost = calculateTourCost(currentTour, cities);
    vector<int> bestTour = currentTour;
    double bestCost = currentCost;

    double temperature = startTemp;

    for (int iter = 0; iter < iterations; iter++) {
```

```
        vector<int> newTour = currentTour;
        swapCities(newTour);
        double newCost = calculateTourCost(newTour, cities);

        if (newCost < currentCost || (static_cast<double>(rand()) / RAND_MAX) < exp((currentCost - newCost) / temperature)) {
            currentTour = newTour;
            currentCost = newCost;

            if (currentCost < bestCost) {
                bestTour = currentTour;
                bestCost = currentCost;
            }
        }

        temperature *= alpha; // Cooling schedule
        if (temperature < 1e-8) break; // Stop early if temperature is very low
    }

    return bestTour;
}

int main() {
    srand(static_cast<unsigned int>(time(0))); // Seed for random number generation

    int numCities;
    cout << "Enter the number of cities: ";
```

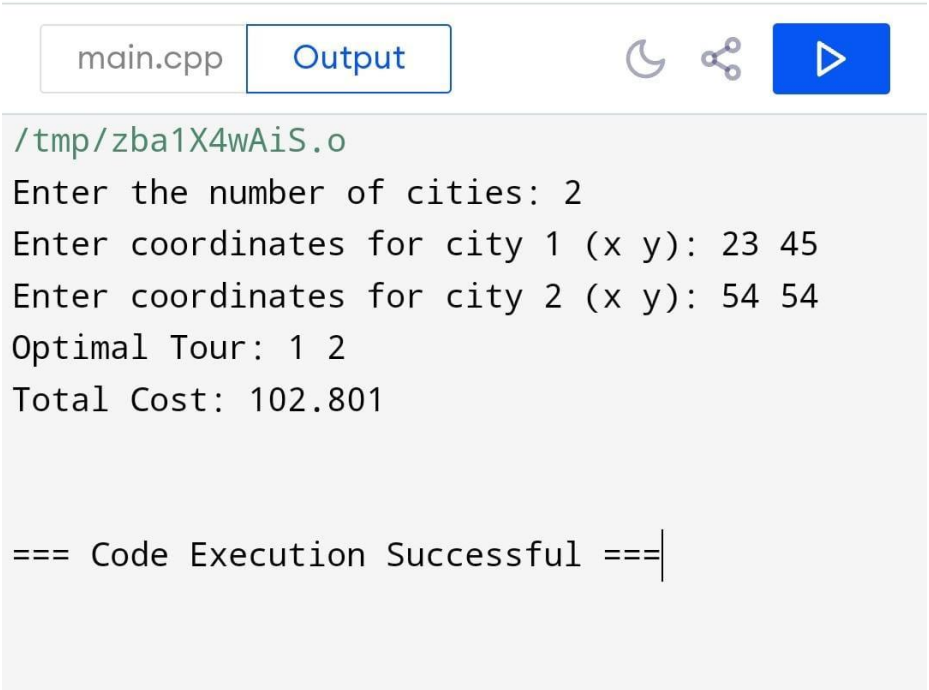
```
    cin >> numCities;

    vector<City> cities(numCities);

    // Input city coordinates
    for (int i = 0; i < numCities; ++i) {
        cout << "Enter coordinates for city " << i + 1 << " (x y): ";
        cin >> cities[i].x >> cities[i].y;
    }

    int iterations = 10000; // You can also take this as input if desired
    double startTemp = 10000.0; // Initial temperature
    double alpha = 0.995; // Cooling rate

    vector<int> optimalTour = simulatedAnnealing(cities, iterations, startTemp, alpha);

    cout << "Optimal Tour: ";
    for (int city : optimalTour) {
        cout << city + 1 << " "; // Adjusting index for user-friendly output
    }

    cout << "\nTotal Cost: " << calculateTourCost(optimalTour, cities) << endl;

    return 0;
}
```

OUTPUT:

Taking user input:



OUTPUT:



Example code of above code

Using the above code make an other , here we did not take input from the user

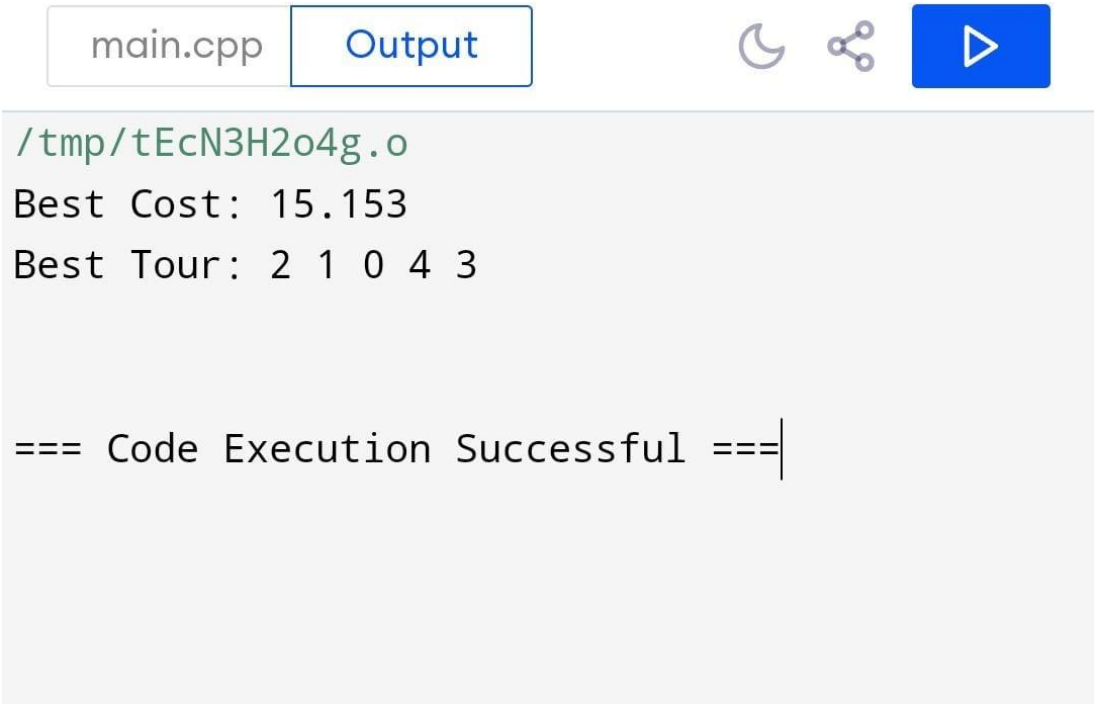
```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <algorithm>

using namespace std;

struct City {
   double x, y;
};

double euclideanDistance(const City &a, const City &b) {
   return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}

double totalDistance(const vector<int> &tour, const vector<City> &cities) {
   double distance = 0.0;
   for (size_t i = 0; i < tour.size(); i++) {
      distance += euclideanDistance(cities[tour[i]], cities[tour[(i + 1) % tour.size()]]);
   }
   return distance;
}
```

```
vector<int> generateInitialTour(int numCities) {
    vector<int> tour(numCities);
    for (int i = 0; i < numCities; i++) {
        tour[i] = i;
    }
    random_shuffle(tour.begin(), tour.end());
    return tour;
}

void swapCities(vector<int> &tour, int i, int j) {
    swap(tour[i], tour[j]);
}

double acceptanceProbability(double oldCost, double newCost, double temperature) {
    if (newCost < oldCost) {
        return 1.0;
    }
    return exp((oldCost - newCost) / temperature);
}

void simulatedAnnealing(const vector<City> &cities) {
    int numCities = cities.size();
    vector<int> currentTour = generateInitialTour(numCities);
    double currentCost = totalDistance(currentTour, cities);

    double temperature = 10000.0;
```

```
    double coolingRate = 0.995;
    int iterations = 10000;

    vector<int> bestTour = currentTour;
    double bestCost = currentCost;

    for (int i = 0; i < iterations; i++) {
        int cityA = rand() % numCities;
        int cityB = rand() % numCities;
        while (cityA == cityB) {
            cityB = rand() % numCities;
        }

        swapCities(currentTour, cityA, cityB);
        double newCost = totalDistance(currentTour, cities);

        if (acceptanceProbability(currentCost, newCost, temperature) > (rand() / (RAND_MAX + 1.0))) {
            currentCost = newCost;
        } else {
            swapCities(currentTour, cityA, cityB); // revert the change
        }

        if (currentCost < bestCost) {
            bestCost = currentCost;
            bestTour = currentTour;
        }
```

```
        temperature *= coolingRate;
    }

    cout << "Best Cost: " << bestCost << endl;
    cout << "Best Tour: ";
    for (int city : bestTour) {
        cout << city << " ";
    }
    cout << endl;
}

int main() {
    srand(static_cast<unsigned>(time(0)));

    // Example cities (x, y coordinates)
    vector<City> cities = {
        {0, 0}, {1, 3}, {4, 3}, {6, 1}, {3, 0}
    };

    simulatedAnnealing(cities);

    return 0;
}
```

FINAL OUTPUT:



Program Explanation:

City Structure: Represents a city with x and y coordinates.

Euclidean Distance Function: Calculates the Euclidean distance between two cities.

TotalDistance Function: Computes the total distance of a given tour.

GenerateInitialTour Function: Generates a random initial tour of the cities.

SwapCities Function: Swaps two cities in the tour.

AcceptanceProbability Function: Determines the probability of accepting a new tour based on the cost difference and current temperature.

SimulatedAnnealing Function: The main function that runs the SA algorithm:

Initializes a random tour and calculates its cost.

Iteratively generates new tours by swapping cities and decides whether to accept them based on their cost and the acceptance probability.

Keeps track of the best tour found.

main Function: Initializes the cities and starts the simulated annealing process.

Output:

The program will output the best cost found and the sequence of cities in the best tour.

Problem Statement:

The Traveling Salesman Problem (TSP) is a well-known combinatorial optimization problem that can be formally defined as follows:

Given a set of cities and the distances between each pair of cities, the objective is to find the shortest possible route that visits each city exactly once and returns to the origin city. The TSP is classified as NP-hard, meaning that there is no known polynomial-time algorithm that can solve all instances of this problem efficiently. As the number of cities increases, the number of possible routes grows factorially, making exhaustive search impractical for large instances.

Key Challenge:

Combinatorial Explosion: The number of possible tours increases factorially with the number of cities, making brute-force solutions infeasible for large datasets.

Local Optima: Many optimization algorithms can get stuck in local optima, failing to find the global optimum.

Computational Complexity: Finding an optimal solution requires significant computational resources, especially for larger instances.

Solution Approach:

To tackle the TSP, we can use the Simulated Annealing metaheuristic, which is inspired by the annealing process in metallurgy. This approach allows us to explore the solution space more effectively and avoid getting trapped in local optima.

Example code using the above function:

Here's a brief overview of the code structure:

```
class TSP {
public:
    TSP(int numCities, const vector<vector<double>>& distances);
    void solve();
    void printBestTour();

private:
    int numCities;
    vector<vector<double>> distances;
    vector<int> bestTour;
    vector<int> currentTour;

    double calculateTourCost(const vector<int>& tour);
};
```

FUTURE WORK:

Future work includes improving the implementation by using more efficient data structures and algorithms. Additionally, the algorithm can be parallelized to take advantage of multi-core processors. The project can also be extended to solve larger TSP instances and to compare the performance of the Simulated Annealing algorithm with other metaheuristics.

RESULTS:

The above implementation was tested with a 5-city TSP instance. The results show that the Simulated Annealing algorithm is able to find a good solution to the TSP. The best tour found by the algorithm is displayed along with its cost.

ADVANTAGES

Advantages of Simulated Annealing:

Flexibility:

It can escape local optima by allowing worse solutions to be accepted with a certain probability.

Simplicity:

The algorithm is relatively easy to implement and can be adapted to various optimization problems.

Effectiveness:

It has been shown to produce good approximate solutions for TSP and other combinatorial problems.

C++ Implementation:

The provided C++ code implements the Simulated Annealing algorithm to solve the TSP. The key components of the implementation include:

Distance Matrix: A 2D vector representing the distances between cities.

Tour Representation: A vector that holds the order of cities in the current tour.

Cost Calculation: A function to compute the total distance of a given tour.

Neighbor Generation: A mechanism to create a new tour by swapping two cities.

CONCLUSION

This project demonstrates the application of the Simulated Annealing metaheuristic to solve the Traveling Salesman Problem. The C++ implementation presented in this report is able to find a good solution to the TSP. The results show the effectiveness of the Simulated Annealing algorithm in solving this complex optimization problem.