TRAVELLING SALESMAN PROBLEM USING WHALE OPTIMIZATION ALGORITHM

A PROJECT REPORT

submitted by

ADITYA BISHT (20114013)

DEEPANSHU MEENA (20114031)

SONALI GUPTA (20114094)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

ROORKEE-247667

APRIL 2024

PROBLEM STATEMENT

Travelling Salesman Problem: Given a set of cities and the distances between each pair of cities, the problem asks for the determination of the shortest possible route that starts at one city, visits each city exactly once, and returns to the origin city.
Solve it using the Whale Optimization Algorithm (WOA). Apply parallelization to enhance the efficiency of the algorithm by distributing computation tasks across multiple processing units or cores.

DATASETS

For our problem statement, we are utilizing two datasets:

- Dataset 1: The first dataset is available at the following link: five_d.txt.
- Dataset 2: The second dataset can be accessed from the following link: p01_d.txt.

These datasets contain information about the distances between each pair of cities in the respective problem instances. They will be used to formulate and solve the Travelling Salesman Problem (TSP), aiming to find the shortest possible route that visits each city exactly once and returns to the origin city.

BRUTE-FORCE SOLUTION OF TSP

As we know, the Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. But here in TSP, we have to find a minimum weight Hamiltonian Cycle. This problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem.

We can implement a simple brute-force solution as given below:

- Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
- Generate all (n-1)! permutations of cities.
- Calculate the cost of every permutation and keep track of the minimum cost permutation.
- Return the permutation with minimum cost.

WHALE OPTIMIZATION ALGORITHM

The step-by-step procedure to obtain an optimum value (maximum or minimum) of an objective function is called an Optimization Algorithm.

The Whale Optimization Algorithm (WOA) is a meta-heuristic optimization technique inspired by the hunting behavior of humpback whales. Foraging behavior of Humpback whales is called bubble-net feeding method. It mimics the whale's strategies for hunting prey, such as creating bubble nets to encircle and trap schools of fish.

In WOA, candidate solutions are iteratively improved by simulating three main behaviors: searching for prey, encircling the prey, and executing spiral bubble-net movements. These behaviors guide the exploration and exploitation of the search space to find optimal solutions.

WOA is widely used in solving optimization problems across various domains due to its simplicity, efficiency, and ability to handle complex search spaces.

- **Encircling Prey:** Current best candidate solution is assumed to be closed to target prey and other solutions update their position towards the best agent.

t = current iteration
A, C = coefficient vectors
X = position vector
r1, r2 = random vectors

$$\vec{D} = |\vec{C} . \vec{X}_{best}(t) - \vec{X}(t)|$$
$$\vec{X}(t+1) = \vec{X}_{best}(t) - \vec{D} . \vec{A}$$

- **Bubble-net attacking method:** It is an exploitation phase. In order to mathematically model the bubble-net behaviour of humpback whales, two approaches are designed as follows:
  - **Shrinking encircling mechanism:** This behaviour is achieved by decreasing the value of $\vec{a}$. a is decreased from 2 to 0 over the course of iterations.
  - **Spiral updating position:** This approach first calculates the distance between the whale located at (X,Y) and prey located at (X*,Y*). A spiral equation is then created between the position of whale and prey to mimic the helix-shaped movement of humpback whales as follows:

$$\vec{D} = |\vec{X}_{best}(t) - \vec{X}(t)|$$
$$\vec{X}(t+1) = \vec{X}_{best}(t) - \vec{D} . e^{bl} . \cos(2\pi l) + \vec{X}_{best}(t)$$

- **Searching for prey:** Humpback whales search randomly according to the position of each other.

$$\vec{D} = |\vec{C} . \vec{X}_{random}(t) - \vec{X}(t)|$$
$$\vec{X}(t+1) = \vec{X}_{random}(t) - \vec{D} . \vec{A}$$

TRAVELLING SALESMAN PROBLEM USING WOA

Here are the steps to find the best path for TSP using WOA:

1. **Initialization:**

a. Generate a population of random solutions for the TSP.

b. Set variables to track the best solution found (`bestPath`) and its distance (`bestDistance`).

2. Main Loop:

a. Evaluate each solution in the population and update the best solution found if necessary.

b. Apply Whale Optimization Algorithm logic to each solution to potentially improve it.

3. Whale Optimization Algorithm (WOA):

a. Evaluate each solution's fitness.

b. Update solutions based on a combination of strategies: making them closer to the best solution found so far and performing local optimizations.

4. **Termination:** Repeat the process for a specified number of iterations, then return the best solution found (bestPath) and its distance (bestDistance).

PSEUDO CODE:

```
function tspWhaleOptimization(distanceMatrix, bestPath, bestDistance, populationSize, maxIterations):
    numCities = size of distanceMatrix
        population = empty array of arrays of integers
    initializePopulation(population, populationSize, numCities)
bestDistance = positive infinity

for each iteration in range maxIterations:
        for each individual in population:
            dist = calculatePathDistance(individual, distanceMatrix)
            if dist < bestDistance:
            bestDistance = dist
            bestPath = copy of individual

        for each individual in population:
            updateSolutionUsingWOA(individual, bestPath, distanceMatrix)
```

Fitness Function

The fitness function for the Traveling Salesman Problem (TSP) is defined as the total distance travelled in a given solution. It calculates the sum of distances between consecutive cities

visited in the order specified by the solution. The goal is to minimize this total distance, as it represents the overall tour length required for the salesman to visit all cities exactly once and return to the starting point.

Mathematically, the fitness function f(x) for a solution x can be represented as:

$$f(x) = \sum_{i=1}^{n-1} d(x_i, x_{i+1}) + d(x_n, x_1)$$

Where:

- n is the number of cities.
- x_i represents the i^{th} city in the solution order.
- $d(x_i, x_{i+1})$ denotes the distance between city x_i and city x_{i+1}.
- $d(x_n, x_1)$ represents the distance from the last city back to the starting city to complete the tour.

The fitness of a solution is lower when the total distance is shorter, indicating a better tour. Therefore, the objective is to find the solution with the lowest fitness value, corresponding to the optimal tour for the TSP.

Here are the steps to update the population for TSP using WOA:

1. **Encircling Prey Strategy:**

a. With a probability of 0.5, the algorithm chooses to employ the "encircling prey" strategy.

b. For each city in the solution:

i. There's a 10% chance (probability 0.1) to swap it with the corresponding city in the `bestSolution`.

ii. The corresponding city in `bestSolution` is the city at the same index as in the current solution.

2. **Bubble-net Attacking Strategy:**

a. With a probability of 0.5, the algorithm chooses the "bubble-net attacking" strategy.

b. It randomly selects two cities in the solution and swaps them.

c. If the resulting solution is worse than the best solution found so far (as measured by the distance calculated using the `distanceMatrix`), it reverts the swap, restoring the original solution.

These strategies aim to explore and exploit the solution space by either making solutions closer to the best solution found so far or performing local optimizations.

PSEUDO CODE:

```
function updateSolutionUsingWOA(solution, bestSolution, distanceMatrix):
    if random number between 0 and 1 < 0.5:
        for each city in solution:
            with probability 0.1:
                idx = index of bestSolution's corresponding city in solution
                swap city and city at idx
    else:
        randomly select two cities in solution and swap them
        if the new solution is worse:
            swap the selected cities back to their original positions
```

APPLYING PARALLELIZATION

1. Parallel For Directive:

a. The `#pragma omp parallel for` directive parallelizes the loop that iterates over the population of solutions.

b. It divides the iterations of the loop among multiple threads, allowing each thread to independently evaluate the fitness of a subset of solutions.

2. Critical Section:

a. The `#pragma omp critical` directive ensures that only one thread at a time can execute the critical section.

b. In this code, the critical section updates the `bestDistance` and `bestPath` variables if a solution's distance is better than the current best solution found so far.

c. Without the critical section, multiple threads might attempt to update these variables simultaneously, leading to race conditions and incorrect results.

By parallelizing the evaluation of solution fitness, the code can leverage multiple processor cores to speed up the computation, particularly for large population sizes and numbers of iterations. Each thread works on a subset of solutions concurrently, improving the overall efficiency of the algorithm.

RESULTS

Dataset 1:-

0.0 3.0 4.0 2.0 7.0

3.0 0.0 4.0 6.0 3.0

4.0 4.0 0.0 5.0 8.0

2.0 6.0 5.0 0.0 6.0

7.0 3.0 8.0 6.0 0.0

Dataset 2:-

0	29	82	46	68	52	72	42	51	55	29	74	23	72	46
29	0	55	46	42	43	43	23	23	31	41	51	11	52	21
82	55	0	68	46	55	23	43	41	29	79	21	64	31	51
46	46	68	0	82	15	72	31	62	42	21	51	51	43	64
68	42	46	82	0	74	23	52	21	46	82	58	46	65	23
52	43	55	15	74	0	61	23	55	31	33	37	51	29	59
72	43	23	72	23	61	0	42	23	31	77	37	51	46	33
42	23	43	31	52	23	42	0	33	15	37	33	33	31	37
51	23	41	62	21	55	23	33	0	29	62	46	29	51	11
55	31	29	42	46	31	31	15	29	0	51	21	41	23	37
29	41	79	21	82	33	77	37	62	51	0	65	42	59	61
74	51	21	51	58	37	37	33	46	21	65	0	61	11	55
23	11	64	51	46	51	51	33	29	41	42	61	0	62	23
72	52	31	43	65	29	46	31	51	23	59	11	62	0	59
46	21	51	64	23	59	33	37	11	37	61	55	23	59	0

BRUTE FORCE APPROACH:-

FOR DATASET 1:-

Optimal Path: 0 2 1 4 3 0

Total Distance: 19

TIME TAKEN: 0m0.008s

FOR DATASET 2:-

Time taken is very long and solutions can not be obtained.

SEQUENTIAL WHALE OPTIMIZATION APPROACH:-

FOR DATASET 2:-

Optimal Path: 5 7 9 13 11 2 6 4 8 14 1 12 0 10 3 5

Total Distance: 291

TIME TAKEN: 0m10.226s

PARALLELIZED WHALE OPTIMIZATION APPROACH:-

FOR DATASET 2:-

FOR NUM OF THREADS = 2

Optimal Path: 5 7 9 13 11 2 6 4 8 14 1 12 0 10 3 5

Total Distance: 291

TIME TAKEN: 0m6.291s

FOR NUM OF THREADS = 4

Optimal Path: 5 7 9 13 11 2 6 4 8 14 1 12 0 10 3 5

Total Distance: 291

TIME TAKEN: 0m6.254s

FOR NUM OF THREADS = 64

Optimal Path: 5 7 9 13 11 2 6 4 8 14 1 12 0 10 3 5

Total Distance: 291

TIME TAKEN: 0m5.437s

Here we can see that increasing the number of threads is increasing the efficiency of algorithm as parallel tasks can run on different threads.

VARYING THE POPULATION SIZE ON DATASET 2:-

POPULATION = 1000

Optimal Path: 5 7 9 13 11 2 6 4 8 14 1 12 0 10 3 5

Total Distance: 291

TIME TAKEN: 0m7.437s

POPULATION = 900

Optimal Path: 5 3 10 0 12 1 7 9 8 14 4 6 2 11 13 5

Total Distance: 307

TIME TAKEN: 0m6.001s

POPULATION = 500

Optimal Path: 1 12 0 10 3 5 9 13 11 2 6 4 14 8 7 1

Total Distance: 321

TIME TAKEN: 0m3.217s

We can see that as we are decreasing the population the result is wrong and we are getting the sub-optimal solution. So we had to choose the optimal number of population for the correct results.