EC9600: APPLIED ALGORITHMS

GROUP ASSIGNMENT

TRAVELLING SALESMAN PROBLEM (TSP)

Submitted by:

Anuvathan S. (2018/E/006)

Maduranga W.P.N. (2018/E/073)

Rodrigo SM. (2018/E/102)

DEPARTMENT OF COMPUTER ENGINEERING

FACULTY OF ENGINEERING

UNIVERSITY OF JAFFNA

[DECEMBER] [2022]

PROBLEM

The travelling Salesman Problem (TSP) is a classical combinatorial optimization problem, which is simple to state but difficult to solve. The problem is to find the shortest possible tour through a set of N nodes. The constraints require that the salesman must enter and leave the city exactly once. A salesman is asked to cover all the nine cities given in the table. He has approached your team to find a suitable path for him to travel such that he covers all cities, while only entering and leaving the city exactly once.

City	1	2	3	4	5	6	7	8	9
1	-	225	304	236	213	339	187	197	226
2	225	-	140	153	15	175	84	160	110
3	304	140	-	152	132	41	121	190	108
4	236	153	152	-	143	188	70	73	63
5	213	15	132	143	-	166	74	145	102
6	339	175	41	188	166	-	157	226	144
7	187	84	121	70	74	157	-	81	43
8	197	160	190	73	145	226	81	-	90
9	226	110	108	63	102	144	43	90	-

However, this is extremely time consuming and as the number of cities grows, brute force quickly becomes an infeasible method. A TSP with just 10 cities has 9! or 362,880 possible routes, far too many for any computer to handle in a reasonable time. The TSP is an NP-hard problem and so there is no polynomial-time algorithm that is known to efficiently solve every travelling salesman problem.

SOLUTION

Greedy Implementation

Approach: This problem can be solved using Greedy Technique.

Below are the steps:

1. Create two primary data holders:
   - A list that holds the indices of the cities in terms of the input matrix of distances between cities.
   - Result array which will have all cities that can be displayed out to the console in any manner.
2. Perform traversal on the given adjacency matrix tsp[][] for all the city and if the cost of the reaching any city from current city is less than current cost the update the cost.
3. Generate the minimum path cycle using the above step and return their minimum cost.

Code:

```
import java.util.ArrayList;
import java.util.List;

public class TSPGreedy {
    // Function to find the minimum cost path for all the paths
    static void findMinRoute(int[][] tsp)
    {
        int sum = 0; // sum of the costs in min path
        int counter = 0;
        int j = 0, i = 0;
        int min = Integer.MAX_VALUE;
        List<Integer> visitedRouteList = new ArrayList<>();
        int startingNode=0;

        // Starting from the 0th indexed, city 1
        visitedRouteList.add(startingNode);
        int[] route = new int[tsp.length];

        // Traverse the adjacency matrix tsp[][]
        while (i < tsp.length && j < tsp[i].length) {

            // Corner of the Matrix
            if (counter >= tsp[i].length - 1) {
                break;
            }

            // If this path is unvisited then and if the cost is less then update the
cost
            if (j != i && !(visitedRouteList.contains(j))) {
                if (tsp[i][j] < min) {
                    min = tsp[i][j];
                    route[counter] = j + 1;
                }
            }
            j++;

            // Check all paths from the ith indexed city
            if (j == tsp[i].length) {
                sum += min;
                min = Integer.MAX_VALUE;
                visitedRouteList.add(route[counter] - 1);
                j = 0;
                i = route[counter] - 1;
                counter++;
            }
        }

        // Add the cost from the ending city to starting city
        i = route[counter - 1] - 1;
        sum += tsp[i][startingNode];


        // Print path from starting node to final node
        System.out.print("Shortest Path is : ");
        for (Integer visited : visitedRouteList) {
            System.out.print(visited+1+" -> ");
        }

        // Started from the node where we finished as well.
        System.out.print(startingNode+1);
```

```
        System.out.print("\nMinimum Cost is : ");
        System.out.println(sum);
    }

    // Driver Code
    public static void main(String[] args)
    {
        // Input Matrix
        int[][] tsp = {
                {-1, 225, 304, 236, 213, 339, 187, 197, 226},
                {225, -1, 140, 153, 15, 175, 84, 160, 110},
                {304, 140, -1, 152, 132, 41, 121, 190, 108},
                {236, 153, 152, -1, 143, 188, 70, 73, 63},
                {213, 15, 132, 143, -1, 166, 74, 145, 102},
                {339, 175, 41, 188, 166, -1, 157, 226, 144},
                {187, 84, 121, 70, 74, 157, -1, 81, 43},
                {97, 160, 190, 73, 145, 226, 81, -1, 90},
                {226, 110, 108, 63, 102, 144, 43, 90, -1}
        };

        // Function Call
        findMinRoute(tsp);
    }
}
```

Output:

```
Shortest Path is : 1 -> 7 -> 9 -> 4 -> 8 -> 5 -> 2 -> 3 -> 6 -> 1
Minimum Cost is : 1046

Process finished with exit code 0
```

Runtime:

```
Build completed successfully in 1 sec, 718 ms (moments ago)
```

Actually, as you can see greedy approach does not giving the optimal solution. The reason is greedy algorithm consists of a sequence of decisions and each choice made should be the best possible one **at the moment.** It does not consider about whole scenario. So even though the time complexity and space complexity are better, it is this method is not good for TSP problem.

Time complexity: O (N2 * LogN)

Dynamic Programming Implementation

Approach: This problem can be solved using Dynamic Programming approach.

Below are the steps:

- Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
- Now, we will generate all possible permutations of cities which are (n-1)!.
- Find the cost of each permutation and keep track of the minimum cost permutation.
- Return the permutation with minimum cost.

Here we assumed as the person is starting from city 1.

Code:

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class TSP_Dynamic {

    private final int N, start;
    private final double[][] distance;
    private List<Integer> tour = new ArrayList<>();
    private double minTourCost = Double.POSITIVE_INFINITY;
    private boolean ranSolver = false;

    public TSP_Dynamic(double[][] distance) {
        this(0, distance);
    }

    public TSP_Dynamic(int start, double[][] distance) {
        N = distance.length;

        this.start = start;
        this.distance = distance;
    }

    // optimal tour
    public List<Integer> getTour() {
        if (!ranSolver) solve();
        return tour;
    }

    // minimal tour cost
    public double getTourCost() {
        if (!ranSolver) solve();
        return minTourCost;
    }

    // solutions of  the traveling salesman problem
    public void solve() {

        if (ranSolver) return;

        final int END_STATE = (1 << N) - 1;
        Double[][] memory = new Double[N][1 << N];

        // Add all outgoing edges from the starting node to memory table.
        for (int end = 0; end < N; end++) {
            if (end == start) continue;
            memory[end][(1 << start) | (1 << end)] = distance[start][end];
        }

        for (int r = 3; r <= N; r++) {
            for (int subset : combinations(r, N)) {
                if (notIn(start, subset)) continue;
                for (int next = 0; next < N; next++) {
                    if (next == start || notIn(next, subset)) continue;
                    int subsetWithoutNext = subset ^ (1 << next);
                    double minDist = Double.POSITIVE_INFINITY;
                    for (int end = 0; end < N; end++) {
                        if (end == start || end == next || notIn(end, subset)) continue;
                        double newDistance = memory[end][subsetWithoutNext] +
```

```
distance[end][next];
                        if (newDistance < minDist) {
                            minDist = newDistance;
                        }
                    }
                    memory[next][subset] = minDist;
                }
            }
        }

        // path back to starting node and minimize cost
        for (int i = 0; i < N; i++) {
            if (i == start) continue;
            double tourCost = memory[i][END_STATE] + distance[i][start];
            if (tourCost < minTourCost) {
                minTourCost = tourCost;
            }
        }

        int lastIndex = start;
        int state = END_STATE;
        tour.add(start);

        // get the TSP path from memory table
        for (int i = 1; i < N; i++) {

            int index = -1;
            for (int j = 0; j < N; j++) {
                if (j == start || notIn(j, state)) continue;
                if (index == -1) index = j;
                double prevDist = memory[index][state] + distance[index][lastIndex];
                double newDist  = memory[j][state] + distance[j][lastIndex];
                if (newDist < prevDist) {
                    index = j;
                }
            }

            tour.add(index);
            state = state ^ (1 << index);
            lastIndex = index;
        }

        tour.add(start);
        Collections.reverse(tour);

        ranSolver = true;
    }

    private static boolean notIn(int elem, int subset) {
        return ((1 << elem) & subset) == 0;
    }

    public static List<Integer> combinations(int r, int n) {
        List<Integer> subsets = new ArrayList<>();
        combinations(0, 0, r, n, subsets);
        return subsets;
    }

    private static void combinations(int set, int at, int r, int n, List<Integer>
subsets) {

        // Return early if there are more elements left to select than what is available.
        int elementsLeftToPick = n - at;
```

```
        if (elementsLeftToPick < r) return;

        if (r == 0) {
            subsets.add(set);
        } else {
            for (int i = at; i < n; i++) {
                // Try to include this element
                set |= 1 << i;

                combinations(set, i + 1, r - 1, n, subsets);

                //  try with Backtrack
                set &= ~(1 << i);
            }
        }
    }

    public static void main(String[] args) {
        // Create distance weight matrix
        int n = 9;
        double[][] distanceMatrix = {
                { 0, 225, 304, 236, 213, 339, 187, 197, 226 },
                { 225, 0, 140, 153, 15, 175, 84, 160, 110 },
                { 304, 140, 0, 152, 132, 41, 121, 190, 108 },
                { 236, 153, 152, 0, 143, 188, 70, 73, 63 },
                { 213, 15, 132, 143, 0, 166, 74, 145, 102 },
                { 339, 175, 41, 188, 166, 0, 157, 226, 144 },
                { 187, 84, 121, 70, 74, 157, 0, 81, 43 },
                { 197, 160, 190, 73, 145, 226, 81, 0, 90 },
                { 226, 110, 108, 63, 102, 144, 43, 90, 0 },
        };

        int startNode = 0;
        TSP_Dynamic solution = new TSP_Dynamic(startNode, distanceMatrix);

        System.out.print("Path:\t" + solution.getTour());
        System.out.print("\nPath cost:\t" + solution.getTourCost());
    }
}
```

Output:

```
Path:   [0, 7, 3, 8, 2, 5, 1, 4, 6, 0]
Path cost:  933.0
Process finished with exit code 0
```

Runtime:

Dynamic approach gives the optimal solution for this problem. It means that provides the minimum time which required to visit all nodes (without visiting more than one time per node except the starting node). Dynamic approach considered complete scenario when giving the decried result. So, it has more time and space complexity. Due to that for a large data amount, Dynamic procedure will take more time to execute.

Time complexity: O (N2 * 2N)

COMPARISON

Greedy method is generally faster than Dynamic programming method. For above Travelling sales man problem, Greedy algorithm takes time complexity of O (N^2 * LogN) and Dynamic programming approach takes time complexity of O (N^2 * 2^N). If we consider the space complexity greedy is more efficient as it never looks back or revise previous choices. For DP it requires table for memorization ad it increases its memory complexity. If we compare the optimality, in greedy sometimes there is no such guarantee of getting optimal solution. For above TSP the greedy didn't given an optimal solution. But DP method given an optimal solution as it generally considers all possible cases and then choose the best. So, in the end of this comparison we can go for greedy approach if consider only time and space complexity or we can go for DP approach if we need an exact optimal solution for every time.

CONCLUSION

There is an important distinction between exact algorithms and heuristics. An exact algorithm is guaranteed to find the exact optimal solution. A heuristic is not, but it is designed to run quickly. DP is an exact algorithm, at least as it is usually used. There are DP algorithms for TSP. Thus, these algorithms will solve the problem exactly. The TSP cannot be solved exactly using greedy methods; hence any greedy method is a heuristic. By definition, therefore, DP will always find a better (or, no worse) feasible solution than a greedy heuristic will, for any instance of the TSP. However, DP is not the dominant approach for solving TSP. Many other algorithms exist that are much more efficient.

GitHub Link: https://github.com/Nadeesham332/TSP_PROBLEM

WORK DISTRIBUTION

2018/E/006	•Documentation of Greedy, DP, Comparison and Conclusion
2018/E/073	•TSP implementation in Greedy algorithm •Conclusion about greedy algorithm
2018/E/102	•TSP implementation in Dynamic Programming approach •Conclusion about DP approach