CS325 Project

Travelling Salesman Problem

by - Group 60
Fransisco Hernandez
Sheng Tse Tsai
Swetha Jayapathy

1. **Algorithms Analysed :**

- Prims Algorithm/Nearest Neighbor
- Brute Force
- Dynamic Programming

Description

We tried the following 4 Algorithms for solving the TSP

Prims Algorithm for TSP :
With this implementation our tour cannot be lower than the MST generated by prims + the traversal weight from the last node to the beginning.
Input: Graph with vertices that have an X and Y coordinate
Generate an MST using Prim's algorithm until every vertice is included.
Approximate the TSP by doing a preorder of the MST starting at the 0th vertice until the end.
Once at the end of the MST you connect to the 0th vertice to show the best tour.

Nearest Neighbor for TSP :
We first generate a random vertex and start from there. We calculate the shortest edge from this start vertex u to v. We mark the covered vertex and track it in an array called parent. Again from v, we calculate the shortest edge to the remaining uncovered vertex. We repeat this step untill we cover all the vertices and from the final vertex we calculate the edge back to the start vertex to complete the tour.
We randomly generate the start vertex with permutations and take the tour which has the minimum tour distance.

Brute Force Algorithm for TSP :
For brute force we find every single permutation of possible tours the salesman is able to take. Given cities from 0 to n-1 there are (n-1)! possible tours and the shortest tour will be the optimal tour. For every vertex we visit, we keep track of the vertices that we can visit next and recursively visit all of them. Brute force will give us the best possible tour but when the number of cities increases it becomes almost impossible to compute.

```
Given a weighted graph G
```

```
While there are possible tours of G
      Generate hamiltonian circuit
      find cost of each circuit
      Pick lowest cost tour
```

Dynamic Programming Algorithm for TSP :

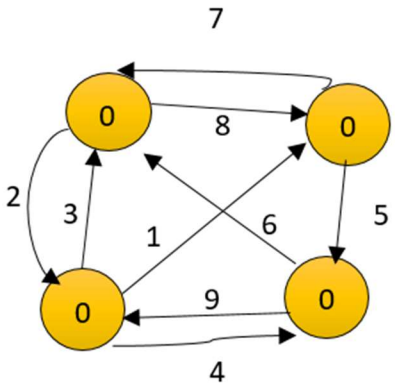
First, we need to calculate the edges of all the vertices with the given input x and y coordinates. The edges are stored in a 2D array. We create subsets of size 0 to n-1 and evaluate the minimum cost for each subset.

Calculating the minimum distance from the Start vertex :

1. Subset size 0 - We first calculate the minimum cost for every vertex from the start vertex using the given 2D array and store it.
2. Subset with size 1 - We then calculate the minimum distance for subset size 1 (1 vertex already covered) and mark its parent as the vertex in the subset.
3. Then we calculate the minimum cost for the subset of size 2 by calculating all the possible values in the set as the parent vertex and taking the minimum distance of it.

We repeat the above process up to n-1 subsets and then we select the one which has the minimum cost to the Start vertex.

Example :



2D Array for edge cost :

i/j	0	1	2	3
0	0	1	15	6
1	2	0	7	3
2	9	6	0	12
3	10	4	8	0

DP - creating sets and calculating its cost from parent

Here we take the start vertex as 0 :

Sets	Cost	Parent Vertex
[1,φ]	0	0
[2,φ]	15	0
[3,φ]	6	0
[2,{1}]	8	1
[3,{1}]	4	1
[1,{2}]	21	2
[3,{2}]	27	2
[1,{3}]	10	3
[2,{3}]	14	3
[3,{1,2}]	20	2
[1,{2,3}]	20	2
[2,{1,3}]	12	3

[0,{1,2,3}] 21 2

If we track the parent vertices, we get the route as below :

0 -> 1 -> 3 -> 2 -> 0
Min Distance = 21

Pseudocode

a) Prims Approximation

```
Function generateMstTour( Vertices with (X,Y) coordinates){
        //Holds the size of how many vertices there are
        Int VSize = Vertices.size()

        Int primMST[VSize + 1]

        //Holds key values
        Int key[VSize]

        //Represents the set of vertices that are inside the MST
        Bool mstSet

        //Setting all of the key values to max and the vertices in mstSet to NOT in the set
        For (int i = 0; i < VSize; i++){
                Key[i] = INT_MAX
                mstSet[i] = false
        }

        //setting first city to root of the MST
        Key[0] = 0
        primMST[0] = -1;

        For (int count = 0; count < VSize; count++){
                //Finding the minimum key value in Vertices
                Int minK = minKey(key, mstSet, VSize)
                Int tempDistance = 0;
                //Adding the minK element into the MST as true
                mstSet[minK] = true

                For (int currentVert = 0; currentVert < VSize; currentVert++){
                        //This loop is finding the smallest distance vertice from our Vertice[minK]
                        //To generate the MST
```

```
                        tempDistance = euclidDist(Vertices[minK], key[currentVert])
                        If (mstSet[currentVert] == false && distance < key[currentVert]){
                                primMST[currentCity] = minK
                                key[currentVert] = distance;
                        }
                }
        }

        //Representing that the tour is ending at the 0th vertice
        primMST[VSize + 1] = 0;

        Int distanceSum = 0;

        //This loop using a preorder traversal from the 0th vertice to find the sum
        For (int i = 0; i < size; i++){
                distanceSum += euclidDist(Vertices[primMST[i]], Vertices[primMST[i+1]])
        }
        //Taking the last vertice in the MST and getting the distance from it to the 0th
        distanceSum += euclidDist(Vertices[primMST[VSize]], Vertices[primMST[0]])

        Return distanceSum and primMST
}
```

This will result in the following
primMST = the MST of Vertices which is the size of Vertices + 1 where the first and last int are the starting and ending point.
distanceSum = the sum of distance resulting from a preorder traversal of the MST.

b) Dynamic Programming

```
Function minTour(int graph[][])
   {

        //Creating Hashmap to store values of the minimum distance and the

        Hashmap minCost;
        // Creating Hashmap to store values of parent from  which we got the minimum distance
        Hashmap Parent;

        //generating all combinations of sets in the graph
        allSets = generateSets()

      for(each set in allSets) {
        for(currentVertex 1 to graph.length)
```

```
            if(set.contains(currentVertex)) {
          Ignore; do nothing
           }

       //index - stores the vertex number from the input
       int minCost = INFINITY;
       int parent = 0;

       for(int prevVertex : set) {
          int cost = graph[prevVertex][currentVertex] + getCost(set, prevVertex, minCostDP);
          if(cost < minCost) {
             minCost = cost;
             parent = prevVertex;
          }
       }

       if(set is empty) {
          minCost = graph[0][currentVertex];
       }

       minCostDP.put(index, minCost);
       parent.put(index, parent);
    }
}

Set<Integer> set = new HashSet<>();
for(int i=1; i < graph.length; i++) {
    set.add(i);
}
int min = Integer.MAX_VALUE;
int prevVertex = -1;

for(int k : set) {
    int cost = graph[k][0] + getCost(set, k, minCostDP);
    if(cost < min) {
       min = cost;
       prevVertex = k;
    }
}

parent.put(Index.createIndex(0, set), prevVertex);
printTour(parent, graph.length);
return min;
```

```
}
```

c) Nearest Neighbour Algorithm :

```
Main Method (){
  //Read file and store the total number of lines as V
  for (int i = 0; i < V; i++) {
        Read line by line and storing the 1st row as id, and the x & y coordinates
        g.addVertex(new Vertex(id=column 1, x=column 2, y=column3));
   }

  //Adding an edge for each pair of vertices
      for(int i = 0; i < vertexNames.size(); i++)
        for(int j = i + 1; j < vertexNames.size(); j++) // ...every other
          g.addUndirectedEdge(vertexNames.get(i).id, // Add edge
              vertexNames.get(j).id, 0);

     g.calAllEDistances(); // compute distances

              path = g.NearestNeighbourTsp();
}

public List<Edge> NearestNeighbourTsp() {

    double smallestDistance = Double.MAX_VALUE;
    List<Edge> shortestPath = null;

    // For each vertex, perform nearest neighbor, check its length
    // against current smallest path. If smaller, replace.
    for(Vertex v : getVertices())
    {
      List<Edge> path = NNUtil(v); // NN path
      double length = PathLength(path); // Path's length

      // If smaller than current smallest, replace
      if(length < smallestDistance)
      {
        smallestDistance = length;
        shortestPath = path;
      }
```

```
    }

    // Return the shortest nearest neighbor path
    return shortestPath;

       }

       private List<Edge> NNUtil(Vertex v)
{

    Vertex currentVertex = v;
    v.known = true;
    List<Edge> path = new LinkedList<>();


    while(path.size() < vertexNames.size() - 1)
    {
       double sLength = Double.MAX_VALUE; // Shortest edge length
       Edge sEdge = null;                 // Shortest edge

       // Cycle through all adjacent edges whose targets are unknown.
       // If distance is smaller than current smallest, replace length
       // and the edge
       for(Edge e : currentVertex.adjacentEdges)
       {
          if (!e.target.known && e.cost < sLength)
          {
             sLength = e.cost;
             sEdge = e;
          }
       }

       sEdge.target.known = true;
       path.add(sEdge);
       currentVertex = sEdge.target;
    }

    // Add last edge
    // the cycle
    for(Edge e : currentVertex.adjacentEdges)
    {
       if(e.target == v)
       {
          path.add(e);
```

```
            break;
        }
    }

    for(Vertex x : getVertices())
        x.known = false;
    return path;
}


private double PathLength(List<Edge> path)
{
    double sum = 0.0;

    // Add distances of each edge to the sum and return the sum
    for(Edge e : path)
        sum += e.cost;
    return sum;
}


public void getNNeighbors() {

for (Vertex vertex : getVertices()) {
vertex.known = false; }

Vertex start = vertexNames.get(0);
start.known = true;
Vertex current = start;

while (true) {
List<Edge> unknownEdges = current.adjacentEdges.stream().filter(x -> !x.target.known)
                                        .collect(Collectors.toList());
if (unknownEdges.isEmpty()) {
start.prev = current;
return;  }

Edge shortestEdge = Collections.min(unknownEdges, Comparator.comparing(x -> x.cost));
shortestEdge.target.prev = shortestEdge.source;
current = shortestEdge.target;
current.known = true;
                    }
}
```

3. Why the above algorithm was selected?

We selected the nearest Neighbour algorithm as the other two algorithms- Dynamic Programming and Prim's didn't work out for such a huge input. The Dynamic Programming algorithm works for small inputs, but for huge input, we are facing issues with Java heap size. This is due to the generation of n! Sets. Similarly, the modified Prims algorithm worked fine for small inputs but took a lot of time for huge inputs.

Hence, we implemented the Nearest Neighbour algorithm.

4. Best tours for the three example instances and the time it took to obtain these tours. No time limit.

Running time in Microseconds :
Example 1 - 76 ms
Example 2 - 1766 ms
The best tour for the examples has been attached in the zip file as filename_op.

4. Best solution for each test instance (which algorithm achieved the best results). Time limit 5 minutes for at least one algorithm.

All the test cases were run by Nearest Neighbour Algorithm :

Test Case Name	Running Time Micro Seconds	Distance
tsp_example_1	76	130921
tsp_example_2	1766	3094
tsp_example_3		
test-input-1	67	5913
test-input-2	140	7964
test-input-3	571	14853
test-input-4	7325	20016
test-input-5	52356	27552
test-input-6	276851	39454
test-input-7		