John Hunsaker

CPSC 335

Algorithm Engineering

TRAVELING SALESPERSON PROJECT

I. Introduction

The goal of this paper is to design, implement, and analyze two algorithms for the Rectilinear Traveling Sales Person Problem (RTSP). The following steps will be covered throughout this paper for each algorithm:

A. Describe the design of method for each algorithm and represent it with pseudocode.

B. Analyze and mathematically prove the efficiency of the pseudocode using Big-Oh notation.

C. Source Code Implementation of this pseudocode using Java.

D. Test cases for various input sizes and output for each input.

II. Rectilinear Traveling Sales Person Problem (RTSP)

A rectilinear graph is a complete, weighted, undirected graph G = (V, E) where V is a set of points $V \subset R^2$, E is the set of all possible edges between distinct points

$$E = \bigcup_{p,q \in V,\ p \neq q} \{ \{p, q\} \}$$

And the weight w(p, q) of an edge between p and q is the rectilinear distance (also called Manhattan distance or city-block metric), defined as follows:

$$w\left[\left(x_p, x_q\right), \left(y_p, y_q\right)\right] = \left|x_p - x_q\right| + \left|y_p - y_q\right|$$

A rectilinear graph is complete, so it may be defined entirely by the points comprising its vertices. There is no need to explicitly store a set of edges, since every pair of distinct vertices is connected, and the weight of such an edge is computed using the rectilinear distance function.

The *RTSP Problem* is:

Input: a positive integer *n* and a list *P* of *n* distinct points representing vertices of a rectilinear graph

Output: a list of *n* points from *P* representing a Hamiltonian cycle of minimum total weight for the graph

III. Exhaustive Optimization Algorithm (EOA)

The exhaustive optimization algorithm will list all possible Hamiltonian cycles (leaving out the exact reversals), find the weight of each cycle, and choose the one with the smallest weight. To generate all possible Hamiltonian paths is equivalent to generating all permutations of vertices. Steps of the EOA:

1. Obtain an infinite length (or very large number) by calculating the distance between the farthest pair of vertices (DIST) and multiplying it by the total number of vertices (N).

2. Generate all permutations, calculate the length of the Hamiltonian cycle generated by each permutation, and compare it with the current best solution.

3. Output the best solution.

A. Pseudocode

```
ALGORITHM Exhaustive Optimization Algorithm (EOA)
// Input: a positive integer n and a list P of n distinct points representing vertices of a
rectilinear graph
// Output: a list of n points from P representing a Hamiltonian cycle of minimum total weight for
the graph

// Calculate the farthest pair of vertices
n                   ← >2                                //# of points on graph entered by user
Strt                ← nanotime()                        //Record beginning time
biggestDist         ← 0
currDist            ← 0
x                   ← 0
y                   ← 0

for i ← 0 to n
      for c ← 0 to n
            x ← | point[i][0] – point[c][0] |
            y ← | point[i][1] – point[c][1] |
            currDist ← x + y
```

```
        if currDist < biggestDist
                biggestDist ← currDist

// Set largest possible distance path to use as max weight
farthestDist ← n * biggestDist


// Create & populate array A with values in the range
A[] ← new int[n]
For i ← 0 to n
        A[i] = i

// Calculate number of permutations
numPerms ← 1
for i ← 1 to n
        numPerms ← numPerms * i

// Create temporary output file
permutationsFile                                                        //temporary file

// Generate all permutations of indices recursively
permutations (int n, int[], int sizeA, int farthestDist, permutationsFile)
        i ← 0
        if n = 1
                for i ← 0 to sizeA
                        print A[i]
        else
                for i ← 0 to n – 1
                permutations ((n-1), A, sizeA, farthestDist, out)
                        if (n % 2 = 0)                                  // swap A[i] & A[n-1]
                                temp ← A[i]
                                A[i] ← A[n-1]
                                A[n-1] ← temp
                        else                                            // swap A[0] & A[n-1]
                                temp ← A[0]
                                A[0] ← A[n-1]
                                A[n-1] ← temp

                permutations ((n-1), A, sizeA, farthestDist, out)

// Close Permutations file
permutationsFile.close                                                  // Close file

// Create array to store the permutation indexes
B[][] ← new int[numPerms][n]


// Read permutation indices from file to new array
currRow ← 0
currCol ← 0
s ← permFile.nextLine()
        while s.length > 0
                insert character ← B[currRow][currCol]
permFile.close()


//Find permutation with minimum weight path
C[] ← new int[n]
C = bestPermutation(n, point, B, numPerms, farthestDist)
```

```
//Calculates weight of every permutation and returns best path
bestPermutation(int n, double[][] point, int[][] B, int numPerms, double farthestDist)
x,y
weight ← 0
currRow ← 0
temp[][] = new double[n]
s[] ← new int [n]


while currRow < newPerms
      for i ← 1 to n
            x = | point[(B[currRow][i])][0] - point[(B[currRow][i - 1])][0] |
            y = | point[(B[currRow][i])][0] - point[(B[currRow][i - 1])][0] |
            weight ← x + y

      weight ← weight + | point[B[currRow][n - 1]][0] - point[B[currRow][0]][0] |
                      + | point[B[currRow][n - 1]][1] - point[B[currRow][0]][1] |


      if weight < farthestDist
            farthestDist = weight
            for i ← 0 to n
            s[i] ← B[currRow][i]

weight = 0
      curRow++;

return s

// Sum total weight of vertexes from beginning to end
farthestDist ← farthestDist + | point[C[n-1]][0] – point[C[0][0] |
  + | point[C[n-1]][1] – point[C[0][1] |

// Record end time
end ← nanoTime()
totalTime ← (end – start)/ 1000000000
```

B. Efficiency

We will now calculate the efficiency of the entire algorithm based of pseudocode's operations defined above:
****Note: For clarity, we will leave the count of operations in every loop as 1 because constants will be factored out in the final calculation of efficiency analysis anyway****

$$C(n) = \sum_{i=0}^{n}\sum_{c=0}^{n} 1 + \sum_{i=0}^{n} 1 + \sum_{i=0}^{n} 1 + n! + (n-1)! + \sum_{i=0}^{n} 1 \qquad [Eq.1]$$

We solve each of summations separately and will sum them at the end:

$$\sum_{i=0}^{n}\left(\sum_{c=0}^{n} 1\right) = \sum_{i=0}^{n}(n+1) = n\sum_{i=0}^{n} 1 + \sum_{i=0}^{n} 1 = [n(n+1)] + (n+1) = n^2 + 2n + 1 \qquad [Eq.1.1]$$

$$\sum_{i=0}^{n} 1 = n + 1 \qquad [Eq.1.2]$$

$$\sum_{i=0}^{n} 1 = n + 1 \qquad [Eq. 1.3]$$

$$\sum_{i=0}^{n} 1 = n + 1 \qquad [Eq. 1.5]$$

Sum of all summations:

$$C(n) = (n^2 + 2n + 1) + (n + 1) + (n + 1) + (n + 1) + n! + (n - 1)! \qquad [Eq. 1.6]$$

$$C(n) = (n - 1)! + n! + n^2 + 5n + 4 \rightarrow \boldsymbol{(n + 1)(n - 1)! + n^2 + 5n + 4}$$

B.1. Proof by Mathematical definition

$$(n + 1)(n - 1)! + n^2 + 5n + 4 \ \epsilon \ O(n \ \cdot n!) \ if \ (n + 1)(n - 1)! + n^2 + 5n + 4 \leq c(n \cdot n!) \ for \ all \ n \geq n_0$$

$$(n + 1)(n - 1)! + n^2 + 5n + 4 \ \epsilon \ O\big((n + 1)(n - 1)! + n^2 + 5n + 4\big) \qquad \rightarrow [Trivial]$$

$$\epsilon \ O(\max((n + 1)(n - 1)!, n^2, 5n, 4)) \qquad \rightarrow [Drop \ dominated \ terms]$$

$$\epsilon \ O\big((n + 1)(n - 1)!\big) \qquad \rightarrow [Drop \ multiplicative \ constants]$$

$$\epsilon \ O(n \cdot n!)$$

$$n \ \cdot n! \ \epsilon \ O(n \cdot n!) \qquad \rightarrow [Trivial]$$

$$n \ \cdot n! \ \leq O(n \ \cdot n! \ )$$

$$\therefore By \ definition \ (n + 1)(n - 1)! + n^2 + 5n + 4 \ \epsilon \ O(n \ \cdot n! \ )$$

B.2. Proof by Limits Theorem

$$\lim_{n\to\infty} \frac{(n + 1)(n - 1)! + n^2 + 5n + 4}{n \ \cdot n!} = \lim_{n\to\infty} \frac{n^2 + 5n + 4 + n! + (n - 1)!}{n \ \cdot n!}$$

$$= \lim_{n\to\infty} \frac{2}{(n - 1)!} + \frac{5}{n!} + \frac{4}{n \ \cdot n!} + \frac{1}{n} + \frac{1}{n^2} = 0 \qquad \rightarrow [constant]$$

$$\therefore By \ limits \ theorem, (n + 1)(n - 1)! + n^2 + 5n + 4 \ \epsilon \ O(n \ \cdot n!)$$

IV. Improved Neatest Neighbor Algorithm (INNA)

In this algorithm, we are willing to give up our requirement for an optimal solution in the interest of time and settle for a "good" solution which may not be optimal. For approximate algorithms, we will be interested in a quantity called the relative error.

$$Relative\ Error = \frac{Difference\ between\ the\ solution\ \&\ the\ optimal\ solution}{Optimal\ Solution}$$

The relative error tells us how close the approximate solution is to the optimal solution. The Relative Error is important but one will be able to calculate the Optimal Solution only for simple cases. Steps for INNA:

1. Calculate the farthest pair of vertices. Consider A and B to be such vertices.

2. Start at vertex A.

3. Let us consider a generic node V which is the current node. Starting from node V, travel to the vertex that you haven’t been to yet but is the closest to V. If there is a tie, pick randomly the next vertex.

4. Continue until you travel to all vertices

5. Travel back to your starting vertex A.

6. Output the solution

Pseudocode

```
ALGORITHM Improved Nearest Neighbor Algorithm (INNA)
// Input: a positive integer n and a list P of n distinct points representing vertices of a
rectilinear graph
// Output: a list of n points from P representing a Hamiltonian cycle of minimum total weight for
the graph

// Calculate the farthest pair of vertices & their distance
n                   ← >2                                            //# of points on graph entered by user
Strt                ← nanotime()                                    //Record beginning time
farthestDist        ← 0
currDist            ← 0
x                   ← 0
y                   ← 0
startVertex[]       ← new double[2]
otherVertex[]       ← new double[2]
nextVertex[]        ← new double[2]

for i ← 0 to n
       for c ← 0 to n
              x ← | point[i][0] – point[c][0] |
              y ← | point[i][1] – point[c][1] |
              currDist ← x + y
       if currDist < farthestDist
              farthestDist ← currDist
```

```
            startVertex[0] ← point[i][0]
            startVertex[1] ← point[i][1]
            otherVertex[0] ← point[c][0]
otherVertex[1] ← point[c][1]

// Create variable to flag visited vertices, set all to false
bool Visited[] ← new Boolean[n]
for i ← 0 to n
      Visited[i] ← false

// Starting from "startVertex" travel to the nearest non-visited vertex & record path at each step
currDist                 ← 0
weight                   ← 0
lastShortestDist         ← farthestDist
otherVertex[0]           ← startVertex[0]
otherVertex[1]           ← startVertex[1]
indexChosen              ← -1
indexOrderSelected[]     ← new int[n]

for c ← 0 to n
      for i ← 0 to n
            x ← | otherVertex[0] – point[i][0] |
            y ← | otherVertex[1] – point[i][1] |
            currDist ← x + y
if currDist < lastShortestDist
            if Visited[i] != true
            lastShortestDist ← currDist
            nextVertex[0] ← point[i][0]
            nextVertex[1] ← point[i][1]
            indexChosen   ← i

Visted[indexChosen]      ← true
indexOrderSelected[c]    ← indexChosen
      otherVertex[0]           ← newVertex[0]
      otherVertex[1]           ← newVertex[1]
      lastShortestDist         ← farthestDist

//Calculate weight of minimal order
farthestDist ← 0
for i ← 0 to n
      x ← | point[indexOrderSelected[i]][0] – point[indexOrderSelected[i-1]][0] |
y ← | point[indexOrderSelected[i]][1] – point[indexOrderSelected[i-1]][1] |
      farthestDist ← farthestDist + x + y

// Sum total weight of vertexes from beginning to end
farthestDist ← farthestDist + |point[indexOrderSelected[n-1]][0] – point[indexOrderSelected[0][0]|
+ |point[indexOrderSelected[n-1]][1] – point[indexOrderSelected[0][1]|

// Record end time
ed           ← nanoTime()
totalTime    ← (ed – strt)/1000000000
```

B. Efficiency

We will now calculate the efficiency of the entire algorithm based of pseudocode's operations defined above:
****Note: For clarity, we will leave the count of operations in every loop as 1 because constants will be factored out in the final calculation of efficiency analysis anyway****

$$C(n) = \sum_{i=0}^{n}\sum_{c=0}^{n} 1 + \sum_{i=0}^{n} 1 \ + \sum_{c=0}^{n}\sum_{i=0}^{n} 1 + \sum_{i=0}^{n} 1 \qquad [Eq.2]$$

We solve each of summations separately and will sum them at the end:

$$\sum_{i=0}^{n}\left(\sum_{c=0}^{n} 1\right) = \sum_{i=0}^{n}(n+1) = n\sum_{i=0}^{n} 1 + \sum_{i=0}^{n} 1 = [n(n+1)] + (n+1) = n^2 + 2n + 1 \qquad [Eq.2.1]$$

$$\sum_{i=0}^{n} 1 = n + 1 \qquad [Eq.2.2]$$

$$\sum_{c=0}^{n}\left(\sum_{i=0}^{n} 1\right) = \sum_{c=0}^{n}(n+1) = n\sum_{c=0}^{n} 1 + \sum_{c=0}^{n} 1 = [n(n+1)] + (n+1) = n^2 + 2n + 1 \qquad [Eq\ 2.3]$$

$$\sum_{i=0}^{n} 1 = n + 1 \qquad [Eq.2.4]$$

Sum of all summations:

$$C(n) = (n^2 + 2n + 1) + (n + 1) + (n^2 + 2n + 1) + (n = 1) \to \boldsymbol{2n^2 + 6n + 4} \qquad [Eq.2.4]$$

$$C(n) = \boldsymbol{2n^2 + 6n + 4}$$

B.1. Proof by Mathematical definition

$$2n^2 + 6n + 4 \ \epsilon \ O(n^2) \ if \ 2n^2 + 6n + 4 \le cn^2 \ for \ all \ n \ge n_0$$

$$2n^2 + 6n + 4 \ \epsilon \ O(2n^2 + 6n + 4) \qquad \to [Trivial]$$

$$\epsilon \ O(\max(2n^2, 6n, 4)) \qquad \to [Drop \ dominated \ terms]$$

$$\epsilon \ O(2n^2) \qquad \to [Drop \ multiplicative \ constants]$$

$$\epsilon \ O(n^2)$$

$$n^2 \epsilon \ O(2n^2) \qquad \to [Trivial]$$

$$n^2 \le O(n^2)$$

$$\therefore \boldsymbol{By \ definition}, \boldsymbol{2n^2 + 6n + 2 \ \epsilon \ O(n^2)}$$

B.2. Proof by Limits Theorem

$$\lim_{n\to\infty} \frac{2n^2 + 6n + 4}{n^2} \ = \lim_{n\to\infty} \frac{4n + 6}{2n} = 2 + \frac{3}{n} = 2 \ge 0 \qquad \to [constant]$$

$$\therefore By \ limits \ theorem, 2n^2 + 6n + 2 \ \epsilon \ O(n^2)$$

D. Test Case Input/output

D.1. EOA Test Case 1 → N=4

```
EOA - Notepad
File  Edit  Format  View  Help
CPSC335-01 - Programming Assignment #3.

Rectilinear Traveling Salesperson Problem.
Using Exhaustive Optimization Algorithm.

The purpose of this program is to collect a set of points on a rectilinear graph,
then use two different algorithms to output a list of points
representing a Hamiltonian cycle of minimum total weight.

Enter the number of points (larger than 2) to be on the graph, then press enter:
4

Enter each point and press enter; make sure that they are distinct:
x = 2.0
y = 0.0
x = 1.0
y = 1.0
x = 3.0
y = 1.0
x = 0.1
y = 0.0

Input: n
n = 4

The Hamiltonian cycle of the minimum length:
(1.0, 1.0) (0.1, 0.0) (2.0, 0.0) (3.0, 1.0) (1.0, 1.0)
Relative minimum length: 7.8
Elapsed Time: 0.013451701 seconds
```

D.2. EOA Test Case 2 → N = 8

EOA - Notepad

File Edit Format View Help

```
CPSC335-01 - Programming Assignment #3.

Rectilinear Traveling Salesperson Problem.
Using Exhaustive Optimization Algorithm.

The purpose of this program is to collect a set of points on a rectilinear graph,
then use two different algorithms to output a list of points
representing a Hamiltonian cycle of minimum total weight.

Enter the number of points (larger than 2) to be on the graph, then press enter:
8

Enter each point and press enter; make sure that they are distinct:
x = 0.0
y = 4.0
x = 2.0
y = 1.0
x = 1.0
y = 6.0
x = 2.0
y = 7.0
x = 3.0
y = 5.0
x = 3.0
y = 2.0
x = 5.0
y = 2.0
x = 6.0
y = 5.0

Input: n
n = 8

The Hamiltonian cycle of the minimum length:
(3.0, 5.0) (2.0, 7.0) (1.0, 6.0) (0.0, 4.0) (2.0, 1.0) (3.0, 2.0) (5.0, 2.0) (6.0, 5.0) (3.0, 5.0)
Relative minimum length: 24.0
```

D.3 INNA Test Case 1 → N = 4

INNA - Notepad

File Edit Format View Help

```
CPSC335-01 - Programming Assignment #3.

Rectilinear Traveling Salesperson Problem.
Using Improved Nearest Neighbor Algorithm.

The purpose of this program is to collect a set of points on a rectilinear graph,
then use two different algorithms to output a list of points
representing a Hamiltonian cycle of minimum total weight.

Enter the number of points (larger than 2) to be on the graph, then press enter:
4

Enter each point and press enter; make sure that they are distinct:
x = 2.0
y = 0.0
x = 1.0
y = 1.0
x = 3.0
y = 1.0
x = 0.1
y = 0.0

Input: n
n = 4

The Hamiltonian cycle of the minimum length:
(3.0, 1.0) (2.0, 0.0) (0.1, 0.0) (1.0, 1.0) (3.0, 1.0)
Relative minimum length: 7.8
Elapsed Time: 1.661E-5 seconds
```

D.4 INNA Test Case 1 → N = 8

INNA - Notepad

File Edit Format View Help

```
CPSC335-01 - Programming Assignment #3.

Rectilinear Traveling Salesperson Problem.
Using Improved Nearest Neighbor Algorithm.

The purpose of this program is to collect a set of points on a rectilinear graph,
then use two different algorithms to output a list of points
representing a Hamiltonian cycle of minimum total weight.

Enter the number of points (larger than 2) to be on the graph, then press enter:
8

Enter each point and press enter; make sure that they are distinct:
x = 0.0
y = 4.0
x = 2.0
y = 1.0
x = 1.0
y = 6.0
x = 2.0
y = 7.0
x = 3.0
y = 5.0
x = 3.0
y = 2.0
x = 5.0
y = 2.0
x = 6.0
y = 5.0

Input: n
n = 8

The Hamiltonian cycle of the minimum length:
(2.0, 1.0) (3.0, 2.0) (5.0, 2.0) (6.0, 5.0) (3.0, 5.0) (1.0, 6.0) (2.0, 7.0) (0.0, 4.0) (2.0, 1.0)
Relative minimum length: 26.0
```