Oregon State University

CS 325 - Analysis of Algorithms

Prof. Julianne Schutford

Spring 2021

# Homework 6

Traveling Salesman Problem

by

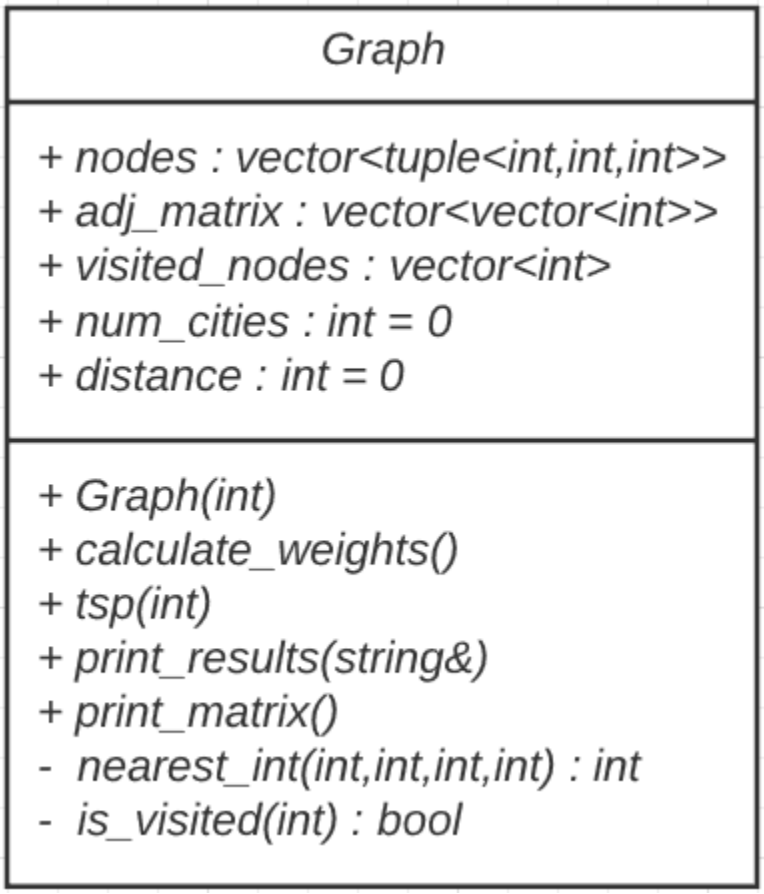
Abdullah Saydemir

saydemia@oregonstate.edu

June 8, 2021

*Data Structure*

I would like to briefly explain my data structure to make things clearer. It is very similar to the one I used in HW4 and the UML diagram of it is as follows.




- *nodes* is the vector that holds the given points
- *adj_matrix* is the adjacency matrix calculated using *nodes* and *calculate_weights* function
- *visited_nodes* holds the nodes the salesman visited
- *num_cities* is the number of nodes in the graph

- *distance* is the length of the path the salesman traveled
- *calculate_weights* fills the adjacency matrix by calculating the distances between each pair of nodes
- *tsp* determines the approximated traveling salesman path using nearest neighbor algorithm
- *print_results* outputs the calculated distance and the path to the file
- *print_matrix* prints the adjacency matrix
- *nearest_int* calculates the distance between two points
- *is_visited* checks if the given node is inside *visited_nodes*

### *Description*

Given a complete graph *G(C, R)*, *C* being the cities and *R* being the roads, let *P* be the path vector that initially has only one city $c_0$, a random city to start. Also, let *d* be the distance the salesman travels.

Until each city is included in *P*

- Take the latest city *c* included to path
- Check all the edges emanating from *c* and find the nearest adjacent city $c_{NEAR}$
- Include $c_{NEAR}$ to *P*
- Update distance *d* by adding the weight of road *R* between *c* and $c_{NEAR}$

Add the distance between the last city salesman visited and the first city the salesman started travelling.

*I didn't include the pseudocode of the functions that I used in HW4. They are used as they are without a modification. Pseudocodes of them can be checked from HW4.*

*Pseudocode*

```
FUNCTION main()
    FILE input := argv[ 1 ]
    IF file is NOT correctly opened THEN
        print "Error reading file"
        exit
    END IF

    LET output_file = argv[ 1 ] + ".tour"
    LET num_cities <- cin
    LET graph <- new Graph( num_cities )

    // take points representing the vertices and put them into the graph
    FOR i <- 0 to num_cities DO
        LET id <- cin
        LET x <- cin
        LET y <- cin
        graph.nodes.insert( <id,x,y> )
    END FOR

    // calculate weights of all edges
    graph.calculate_weights()

    // choose a "random" integer and run the tsp algorithm
    LET start be a random integer
    graph.tsp( start )

    // print the distance and the path to the file
    graph.print_results( output_file )

END FUNCTION
```

*This function belongs to the Graph class. So, all attributes are available to the function and only the starting index is given to the function. If you cannot remember what an attribute is responsible for please have a look at the above definitions.*

```
FUNCTION tsp( int start )        // start is also used as the index of the latest included node

    visited_nodes.insert( start )

    LET min_dist <- 0              // minimum distance to a node from the current node
    LET index    <- 1              // index of the next node salesman visit

    // Graph has num_cities node. Starting node is
    // included. Therefore, we will visit num_cities -1
    // nodes
    FOR i <- 0 to num_cities - 1 DO
        LET min_dist <- inf

        // for all adjacent nodes
        FOR j <- 0 to num_cities DO
            IF node j is NOT visited yet       AND
               cost to j is less than min_dist THEN

                min_dist <- cost to go to j
                index <- j
            END IF
        END FOR

        distance += min_dist                // include the last path traveled
        visited_nodes.insert( index )       // include the visited node to the path
        start <- index                      // update the current index
    END FOR
    distance += distance between last visited node and starting node

END FUNCTION
```

*Theoretical Running Time and Approximation Bound*

Let $V$ be the number of cities on the graph. Since the graph is complete, there will be $V^2$ roads.

The TSP function has to traverse every city in the graph. Outer loop in the function code does this job *(Figure 1, line #80)*. For the city the salesman is currently visiting, the inner loop *(Figure 1, line #85)* looks up to distances to every other city and checks if that city is visited or not *(Figure 1, line #87)*. In my implementation, checking a city takes $O(V)$ time using an array, but it could be reduced to $O(1)$ time, on average, if a hashtable or a node struct was used.

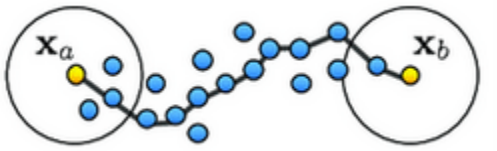
Therefore, the running time of my implementation takes $O(V^3)$ time; however, with some more work, nearest neighbor algorithm can be implemented in $O(V^2)$ time as well.

```
68      auto tsp(int start)                         -> void
69      {
70          // start is also used as the index of the latest included node
71          visited_nodes.push_back(start);
72
73          auto min_dist = 0;                  // minimum distance to a node from the current node
74          auto index = 1;                     // index of the next node salesman visit
75
76
77          // Graph has num_cities node. Starting node is
78          // included. Therefore, we will visit num_cities -1
79          // nodes
80          for (auto i = 0; i < num_cities-1; ++i)
81          {
82              min_dist = INT_MAX;
83
84              // for all adjacent nodes
85              for (auto j = 0; j < num_cities; ++j)
86              {
87                  if ( !is_visited(j) && adj_matrix[start][j] < min_dist)
88                  {
89                      min_dist = adj_matrix[start][j];
90                      index = j;
91                  }
92              }
93
94              distance += min_dist;               // include the last path traveled
95              visited_nodes.push_back(index);     // include the visited node to the path
96              start = index;                      // update the current index
97          }
98
99          // add the distance between the last
100         // visited node and the starting node
101         distance += adj_matrix[visited_nodes.back()][visited_nodes[0]];
102     }
```

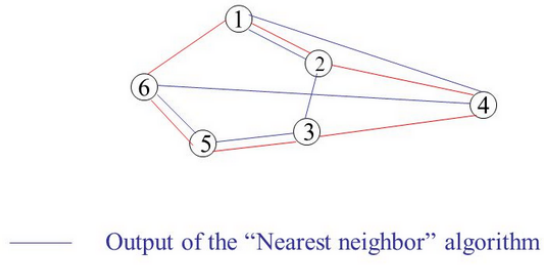
*Figure 1*

Though nearest neighbor algorithm works very fast compared to others, it does not guarantee an approximation bound. It is a greedy algorithm and there are cases where the algorithm can be tricked. Here are two examples of malfunction I found on internet:

1.




2.




Starting point is also important in the nearest neighbor algorithm. So, if things somehow start to go wrong, all goes wrong.

*Summary*

Before going into analyzing each graph I would like to talk about my starting point in the function. I used the *rand()* function to randomly select the starting city. I know that the rand function does not provide random results if you don't seed it. I tried the function without seeding it and it turned out benefiting me in the approximation. So, the starting city in each case is "randomly" selected.

*Example 0*
*Starting node : 3*
*Rho Ratio : 1.00*

This graph is really small. That's why even the NN algorithm finds the optimal solution.

```
Example 0

Starting node: 3
Each item appears to exist in both the input file and the output file.
solution found of length
14
Rho Ratio 0: 1.0000
```

*Example 1*
*Starting node : 59*
*Rho Ratio : 1.39*

This graph has 76 cities in the graph. NN algorithm goes off from the optimal solution but still the ratio is inside the reasonable bounds.

```
Example 1

Starting node: 59
Each item appears to exist in both the input file and the output file.
solution found of length
150842
Rho Ratio 1: 1.3946
```

*Example 2*
*Starting node : 183*
*Rho Ratio : 1.20*

This graph has 280 nodes. However, notice that the approximation ratio is better than the previous case, which has a smaller number of cities.

```
Example 2

Starting node: 183
Each item appears to exist in both the input file and the output file.
solution found of length
3118
Rho Ratio 2: 1.2089
```

*Example 3*
*Starting node : 33*
*Rho Ratio : 1.16*

There are 50 nodes in this case and the approximation ratio is really good.

```
Example 3

Starting node: 33
Each item appears to exist in both the input file and the output file.
solution found of length
6211
Rho Ratio 3: 1.1646
```

*Example 4*
*Starting node : 83*
*Rho Ratio : 1.20*

There are 100 nodes in this case. Again, it seems like the rand function did a good job picking a non-problematic node to start.

```
Example 4

Starting node: 83
Each item appears to exist in both the input file and the output file.
solution found of length
8902
Rho Ratio 4: 1.2011
```

*Example 5*
*Starting node : 383*
*Rho Ratio : 1.25*

A thousand nodes! I thought for a large number of cities there would be a problem, but it is better than case 1 in terms of approximation ratio. That means the cities in the node are distributed well.

```
Example 5

Starting node: 383
Each item appears to exist in both the input file and the output file.
solution found of length
28814
Rho Ratio 5: 1.2527
```