Oregon State University

CS 325 - Analysis of Algorithms

Prof. Julianne Schutford

Spring 2021

Homework 4

Graph Algorithms

by

Abdullah Saydemir

saydemia@oregonstate.edu

May 16, 2021

Q1)

a)

To find the fastest route from the distribution center to each town we need to run a shortest path algorithm. I used Dijkstra and the table is as follows.

D(A) : shortest distance to node A from the set of selected nodes
p(A) : previous node to reach A using the shortest path

Step		Nodes	D(A),p(A)	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)	D(H),p(H)
	0	G	inf	inf	9,G	7,G	2,G	8,G	3,G
	1	G,E	inf	11,E	9,G	5,E		8,G	3,G
	2	G,E,H	inf	6,H	9,G	5,E		8,G	
	3	G,E,H,D	inf	6,H	8,D			8,G	
	4	G,E,H,D,B	inf		8,D			8,G	
	5	G,E,H,D,B,F	15,F		8,D				
	6	G,E,H,D,B,F,C	12,C						
	Result	G,E,H,D,B,F,C	12,C	6,H	8,D	5,E	2,G	8,G	3,G

Starting with node G, in each step we determine the costs to accessible nodes and choose the closest one to include to the shortest path. After having all the nodes in the set, the shortest path to a node for all nodes is determined. Resulting route table is as follows.

	From G	Route	Cost
To:	A	G => E => D => C	12
	B	G => H	6
	C	G => E => D	8
	D	G => E	5
	E	Direct	2
	F	Direct	8
	H	Direct	3

b)

Given a map *G(T,R)*, *T* being the towns *(vertices)* and *R* being the roads *(weighted edges)* :

1. Let $time_{min}$, the minimum time to travel to the furthest node, be infinite.
2. Let $T_{optimal}$ be the optimal location, initially set to null.
3. For a town *T*, run a shortest path algorithm to find the furthest town.
4. Get the time needed to travel to that town and let it be *time*.
5. If $time < time_{min}$, set $T_{optimel} = T$
6. Do steps 3-5 until all the towns are visited.

The algorithm depends on the shortest path algorithm and the data structure used in order to store the graph information. For the time complexity analysis, I will assume Dijkstra's is used and we have an adjacency matrix. That means, running time of the algorithm is $O(T^2)$.

We are running the shortest path algorithm for each town and do a constant job with the values we collected. Therefore, the total running time is $O(T^3)$.

c)

In the above graphs, the furthest distance is to node A which is at the left side of the graph. To replace the station one of the nodes that is in the path to A should be chosen. The shortest path calculations for C, D and E yields the following tables.

	From C	Route	Cost
To:	A	Direct	4
	B	C => D => E => G => H	14
	D	Direct	3
	E	C => D	6
	F	Direct	2
	G	C => D => E	8
	H	C => D => E => G	11

	From D	Route	Cost
To:	A	D => C	7
	B	D => E => G => H	11
	C	Direct	3
	E	Direct	3
	F	D => C	5
	G	D => E	5
	H	D => E => G	8

	From E	Route	Cost
To:	A	E => D => C	10
	B	E => G => H	8
	C	E => D	6
	D	Direct	3
	F	E => D => C	8
	G	Direct	2
	H	E => G	5

Based on above tables, the best location to put the distribution center is E.

d)

The K-Centers problem is known to be an NP-hard problem, which we cannot have an exact solution in polynomial time. I looked it up on the internet and found a solution[1] that has an approximation value of 2. That is, if the optimal solution is *OPT* then the algorithm gives a solution up to *2*OPT*.

At start, let's say we have all the vertices and their shortest distances to each other. We will initialize an array that consists of each vertices' distance to the closest distribution center, which is infinity at start. After that, we will choose an arbitrary vertex V_0 to build the first distribution center and update all vertices' distances to the first distribution center. Next, we will select the second vertex such that it is the farthest vertex from V_0.

[1] Solution is adapted from a lecture PDF from University of Maryland, Analysis of Algorithms class that Professor David Mount gives. The PDF can be found using the following link : https://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect08-greedy-k-center.pdf

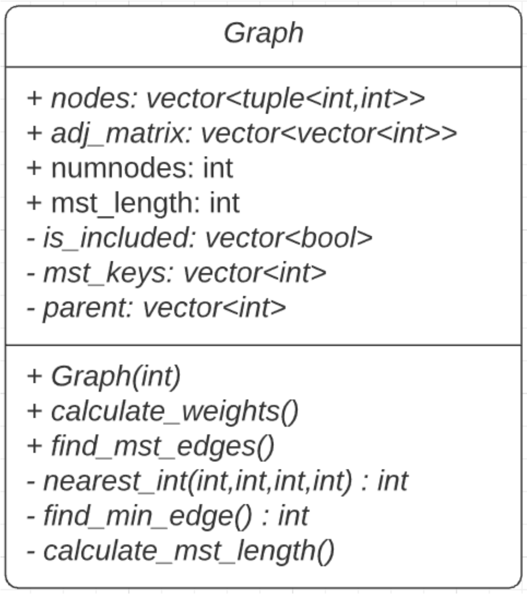
The running time of this algorithm is $O(kV)$, V being the number of vertices and k is the number of distribution centers which is 2 for our case. Because, we will select k cities and after choosing a city we have to update V distances.

e)

I solved this question by just looking at the graph. Distribution centers should be located at towns C and H. By doing so, the distance to the furthest town is 5 which is the cost of going to E from H.

Q2)

To make things clearer I would like to explain my data structure first. The UML diagram of it is as follows.



- *numnodes* is the number of nodes in the graph.
- *nodes* is the vector that holds the given points.
- *adj_matrix* is the adjacency matrix calculated using *nodes* and *calculate_weights* function.
- *nearest_int* calculates the distance between two points.

- *is_included* holds boolean values that implies if that node is included in the MST
- *mst_keys* holds integer values that the distance of each node from the MST
- *parent* holds indices of parent nodes for MST. e.g. index of node 2's parent is parent[2]
- *find_mst* finds the edges that are in MST using Prim's algorithm
- *find_min_edge* finds the edge that requires the smallest cost to include to MST
- *calculate_mst_length* uses adjacency matrix and MST to calculate the length of MST.
- *mst_length* is the length of the MST.

Algorithm Description

For each test case, we read the points and save them to *nodes.* After all points are read from the file, the distances between points are calculated and the *adj_matrix* is filled using *calculate_weights* function. Then, Prim's algorithm is used to find the *Minimum Spanning Tree* of the graph such that:

- Let *is_included* be the set that holds which vertices are included to the minimum spanning tree, initially all false.
- Let *mst_keys* be another set that contains key values for each vertex. For the first vertex key is initialized to 0 and infinite for all the others.
- While there is a vertex that is not included in the *MST*
    - Find the vertex *u* that has the minimum key value in *mst_keys* and include it to the *MST*.
    - For each vertex *v* that is adjacent to *u*, update key values. If the weight of the edge between *v* and *u* is less than the key value, update it to the weight of the edge between *u* and *v* and make *u parent* of *v*.
- Add the weights of all edges that are in the *MST* using the info in *parent* and *adj_matrix*.

As the last step, print the test case number and the length of the *MST*.

Pseudocode

Functions (except for main) belong to Graph class. So, all attributes are available to functions and no input parameters are given to functions. If you cannot remember what an attribute is responsible for please have a look at the above definitions.

```
FUNCTION main()
    FILE input := read(graph.txt)
    IF file is NOT correctly opened THEN
        print "Error reading file"
        exit
    END IF

    LET num_tests <- cin                                  // number of tests

    FOR i <- 0 to num_tests DO
        LET numV <- cin                                   // number of vertices
        LET graph <- new Graph( numV )                    // create a graph with  numV vertices

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

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

        // use Prim's algorithm to find which edges are in the MST
        graph.find_mst()

        // print the info
        LET length <- graph.mst_length
        print "Test Case " + (i+1) + ": MST weight " + length

    END FOR
END FUNCTION
```

```
FUNCTION calculate_weights()
    FOR i <- 0 to numnodes - 1 DO
        LET ( $x_1$, $y_1$ ) <- nodes[ i ]

        FOR j <- i+1 to numnodes DO                              // for each node pair i and j
            LET ( $x_2$, $y_2$ ) <- nodes[ j ]
            LET d <- round( sqrt( $(x_1 - x_2)^2 + (y_1 - y_2)^2$ ) )      // calculate the distance
            adj_matrix[ i ][ j ] <- d                             // update the adjacency matrix
            adj_matrix[ j ][ i ] <- d
        END FOR

    END FOR
END FUNCTION

FUNCTION find_mst()
    mst_keys[ 0 ] <-   0                                         // choose and index to start
    parent[ 0 ]     <-  -1                                       // make it the root

    FOR i <- 0 to numnodes-1 DO
        LET u <- find_min_edge()                                 // find the min edge in mst_keys
        is_included[ u ] <- true                                 // include it to MST

        // update the keys for each neighbor of u
        FOR v <- 0 to numnodes DO

            IF there is an edge between u and v          AND
               v is NOT included in MST                  AND
               the edge weight is less than mst_keys[ v ]  THEN

                mst_keys[ v ] <- adj_matrix[ u ][ v ]      // update the MST key
                parent[ v ] <- u                           // make u parent of v
            END IF

        END FOR
    END FOR
    calculate_mst_length()

END FUNCTION
```

```
FUNCTION find_min_edge()
    LET min    <-   INF
    LET index <-   -1

    // from the edges that are not included in MST
    // return the edge that has the smallest cost
    FOR i <- 0 to numnodes DO
        IF i is NOT included in MST  AND
            mst_keys[ i ] < min          THEN

                min = mst_keys[ i ]
                index = i
        END IF
    END FOR

    RETURN index
END FUNCTION

FUNCTION calculate_mst_length()

    // for each node find the distance to its parent in MST
    // and add them up to mst_length
    FOR  i <- 1 to numnodes DO
        mst_length += adj_matrix[ i ][ parent[ i ] ];
    END FOR

END FUNCTION
```

Theoretical Running Time

Let's evaluate the running time of each function one by one in order they appear in the code. During calculation, we will have V vertices and E edges.

calculate_weights

This function iterates through the vertices and fills the adjacency matrix by calculating the distance between two nodes. Since the graph is a complete graph, we have to do calculations for each pair of vertices. Therefore, running time of this algorithm is $\Theta(V^2)$

find_mst

This algorithm starts from a vertex and iterates through all vertices in both of the for loops. Finding the minimum cost edge takes $O(V)$ time since we use arrays and we have to look at all the elements to get the minimum value. This operation is done for each vertex which means we have at least $O(V^2)$ running time.

Additionally, decreasing the key takes $O(1)$ time and this operation is done for each edge. Since the graph is complete we have V^2 edges. That means we have another $O(V^2)$ that comes from decreasing the key.

Overall, the running time of the algorithm is $O(2V^2)$ which is actually $O(V^2)$. We are looking at complete graphs; therefore, we can use $\Theta(V^2)$ as well.