**CSC 464 Program #2**

**Minimum Cost Spanning Trees (MCST)**

**Prim and Kruskal’s Algorithms**

I am asking you to implement both Prim’s and Kruskal’s algorithms to produce a minimum cost spanning tree for a given graph. The program will open a file with the following format and will then carry out Prim’s algorithm followed by Kruskal’s algorithm and will output the MCST for each algorithm along with an estimate of the time each algorithm used in carrying out its computation.

**Input File Format**

An input file will begin with a line containing a two integers X and Y. The first integer represents the number of vertices in the graph. We will assume that vertices will consist of integers in the range of 0 – (X-1). The second integer will represent the number of edges in the graph.

The next Y lines will present information for one edge of the graph. Edges will consist of three integers A B C. These three integers will represent an edge from vertex A to vertex B with a cost/weight of C. The weight/cost will be represented by an integer (negative edge weights ***ARE*** possible).

An example data file is shown below for the weighted, undirected graph presented :

**![Diagram

Description automatically generated]()**

Graph with 9 vertices (0-8) and 14 edges

**9 14**

**8 2 2**

**0 1 4**

**8 6 6**

**7 8 7**

**1 2 8**

**5 4 10**

**3 5 14**

**7 6 1**

**6 5 2**

**2 5 4**

**2 3 7**

**0 7 8**

**3 4 9**

**1 7 11**

**Kruskal’s algorithm**

Kruskal’s algorithm assumes that the edges of the graph are available in sorted order from lowest cost to highest cost. You should use a Heap-like data structure to assist in the implementation of Kruskal’s algorithm. Using Java’s ***PriorityQueue*** class would seem to make sense. Simply ***add( )*** each edge into the priority queue and, as needed, delete the minimum remaining weighted edge via the ***poll( )*** method.

**Implementing a UnionFind Class for Kruskal’s algorithm**

Kruskal’s algorithm also requires that you be able to identify whether an edge under consideration would create a cycle in the evolving MCST. We will need to implement a data structure known as a **Union-Find (AKA Disjoint Set)**.

As you might infer from its name, a Union-Find data structure relates to sets. It supports two operations. ***Find( )*** allows you to search for an element in a collection of subsets and allows you to identify the set which contains the element. The second operation is ***Union( S1, S2)*** which combines the two subsets S1 and S2 into a new set which represents the Union of sets S1 and S2.

Please see/read the Union-Find slides on canvas for more detailed coverage of this topic.

**public** **class** **UnionFind** {

**private** **int**[] parent; // array that holds/represents the subsets

**public** ***UnionFind***(**int** size) {

        parent = **new** **int**[size];

**for** (var i = 0; i < size; i++) {

            parent[i] = i; // create subsets so each element 1-(size-1) are in their own sets…

        }

    }

**public** **int** ***Find***(**int** x) {

**if** (x == parent[x]) {

**return** x;

        }

*// compress the paths as we seek out the root*

**return** parent[x] = Find(parent[x]); // now all members point to root

    }

**public** **void** ***Union***(**int** x, **int** y)  {

        var px = Find(x); // what subset is x in?

        var py = Find(y); // what subset is y in?

**if** (px != py) { // if the sets are NOT the same

            parent[px] = py; // join set x into set y

        }

    }

**public** **int** ***Size***() { *// number of subsets*

**int** ans = 0;

**for** (**int** i = 0; i < parent.length; ++ i) {

**if** (i == parent[i]) ans ++; // just count the roots

        }

**return** ans;

    }

}

**Kruskal’s Algorithm Outline**

Declare all necessary variables. (see Prim’s algorithm description for declaring priority queue.)

Read first line of input (number of vertices and number of edges)

Instantiate and construct **UnionFind** object **UF** passing it number of vertices.

Read edges and **.add( )** each one as an **EdgeNode** (see below for EdgeNode description) into **PriorityQueue** object.

while (PriorityQueue is not empty AND KruskalMCST has less than (# of vertices) - 1 )

{

**.poll( )** minimum edge E from PriorityQueue object.

**.getVertex1( )** into int **x**

**.getVertex2( )** into int **y**

If ( Find(x) != Find(y) ) // edge will not cause a cycle in KruskalMCST

{

Add edge E to ***KruskalMCST*** **Set** and increment a ***KruskalCost*** with edge E’s weight.

Union( x , y )

}

}

**Implementing EdgeNode Class**

**public class EdgeNode implements Comparable<EdgeNode>**

**{**

**public int vertex1, vertex2, weight;**

**public EdgeNode next;**

**public EdgeNode( int vertex1, int vertex2, int weight )**

**{**

**this.vertex1 = vertex1;**

**this.vertex2 = vertex2;**

**this.weight = weight;**

**this.next = null;**

**}** // end constructor

The class **EdgeNode** should probably also at least have a “setter” and getter method ***for all of the various fields*** such as

**public void setNext( EdgeNode n )**

**{**

**this.next = n ;**

**}**

**public EdgeNode getNext( )**

**{**

**return this.next;**

**}**

**// add setters and getters for vertex1, vertex2 and weight**

**…**

**// next add the following compareTo( ) method required for implementing the Comparable**

**// interface.**

**public int compareTo(EdgeNode n) {**

**if(this.weight < n.weight)**

**return -1;**

**else if(this.weight > n.weight)**

**return 1;**

**else**

**return 0;**

**}** // end compareTo

**}** *// end class EdgeNode*

**Prim’s Algorithm**

Prim’s algorithm is a Greedy algorithm. At each iteration, the algorithm looks at a local decision for the best edge and selects that edge. It can be started on any vertex in the graph but ***we will always assume that we start the algorithm on vertex 0***.

The basic algorithm will need to use a Priority Queue (you should use Java’s built in ***PriorityQueue*** class) or heap as well as three ***HashSet*** objects (see Java’s Set class if you need more information).

This algorithm will assume that the graph is ***implemented as an Adjacency List*** (see slides if you need to review this concept). You should create the class ***EdgeNode*** which was described earlier.

**Implementing Prim’s Algorithm**

//The **Adjacency List** itself will be an ***array of EdgeNode objects*** shown below.

// read first line of input file for number of vertices and number of edges

**EdgeNode AdjList[ ];** // *instantiate this array to the size of the number of vertices in the graph*

***// initialize each entry of AdjList[ ] to null***

Initialize a ***HashSet*** object **S** to an empty set ***then .add( ) vertex 0 to*** **S**.

Initialize a ***HashSet*** object **VminusS** to all *other* vertices 1 through ( # of vertices - 1 ).

Initialize a ***HashSet*** ***<EdgeNode>*** object **PrimMCST** to an empty set.

//Now read the input file’s edges in a loop and build the adjacency list.

// assuming an edge input line has just been read with vertices **u** and **v** with weight **w**

// create an ***EdgeNode*** with vertex1 = **v**, vertex2 = **u** and weight = **w** as arguments and insert it at the front of **AdjList[ u ]**

// create a second ***EdgeNode*** with vertex1 = **u**, vertex2 = **v** and weight = **w** as arguments and insert it at front of **AdjList[ v]**

*// these two edges were both added because we have an* ***undirected*** *graph*

*// note the order of the vertices u and v in each of these insertion operations!!*

We will also require a **PriorityQueue** (i.e. heap). I would recommend the built-in Java ***PriorityQueue*** class though you may, if you like, use a variation of your Heap class that you wrote/used in our previous program. The PriorityQueue will be storing objects of the ***EdgeNode*** class that was described earlier.

// at this point we can declare the **PriorityQueue** object named ***PQ***.

**PriorityQueue<EdgeNode> PQ = new PriorityQueue<>();**

***int Curvertex*** = 0; //we will always start at vertex 0

// below we put 0’s edges into the priority queue

EdgeNode temp = AdjList[ Curvertex ];

while ( temp != null ) // walk through the list of edges

{

PQ.add( temp ); // add the edge to the priority queue

temp = temp.getNext( );

}

*// declare a variable to store minimum cost edges*

EdgeNode min;

*// while there are edges in the priority queue and the MCST is not complete*

**while ( (PQ.size( ) > 0) && (PrimMCST.size() < numv-1 ))**

**{**

**min = PQ.poll();**  *// grab the minimum cost edge from the priority queue*

**x = min.getVertex1();**  *// place vertex #1 into x*

**if ( ! S.contains(x))** *// check if x is NOT in S*

**{**

// now add x to hashset S

// add the min edge to the MCST

// increment the PrimCost by the weight of the min edge

// now use .**remove( x )** to remove x from the **VminusS** hashset.

// Next we need to add edges involving new vertex x to

// the priority queue but only if edge connects x to vertex NOT in S….

**temp = AdjList[x];**

**while ( temp != null )** *// walk through x’s adjacency list*

**{**

**if ( ! S.contains(temp.getVertex1()))** *// test to see if adjacent vertex is NOT in S*

**{**

**PQ.add( temp );**

**}**

**temp = temp.getNext( );**

**} // end while temp != null**

**} // end if (!S.contains(x))**

**} // end while loop**

**Prog2/Main Program**

**Getting execution time information in Java**

I want you to time the execution of both Prim’s and Kruskal’s algorithms. A general and reasonable approximation of execution time for a method can be accomplished in the following way.

long **startTime** = System.nanoTime();

call\_to\_method\_you\_are\_timing(); // you will do this for both algorithm method calls

long **endTime** = System.nanoTime();

long **duration** = (double) (**endTime** - **startTime**) / 1000000.0 ;

//divided by 1000000 to get milliseconds.

**Setting up Prog2/Main/Driver**

There are a handful of variables that should probably be defined as global. BY that, I mean make them private static variables.

**int KruskalCost = 0;**

**int PrimCost = 0;**

**Set PrimMCST = new HashSet( );**

**Set KruskalMCST = new HashSet( );**

public static void main ( String args[ ] ) {

Prompt user for File path and read their response.

Create and open the file containing the graph data.

Record the start time with System.nanoTime( ) for Prim’s algorithm

Call Prim’s algorithm method and pass it any necessary objects for file reading

Record the end time with System.nanoTime( ) for Prim’s algorithm

Print out Data for Prim’s including PrimCost, PrimMCST and elapsed time in milliseconds.

Reset input or close input file and reopen for Kruskal’s algorithm.

Record start time with System.nanoTime( ) for Kruskal’s algorithm

Call Kruskal’s algorithm method and pass it any necessary objects for reading the input file

Record end System.nanoTime( ) for Kruskal’s algorithm

Print out data for Kruskal’s including KruskalCost, KruskalMCST and elapsed time in milliseconds

}

**Submitting Program**

Your program should have 3 classes (4 if you decide to create your own Heap class). You MUST name your classes UnionFind, EdgeNode, and Prog2. These ARE case-sensitive! You may also include a README file to explain any issues or problems that you know your program may have. You should place the following files into a folder named **LastName\_FirstName\_Prog2**

***UnionFind.java***

***EdgeNode.java***

***Prog2.java***

***README (optional)***

***DO NOT INCLUDE ANY OTHER FOLDERS OR FILES!***  Compress/zip the folder and submit/upload the zipped document.

**Sample #1 Input**

**9 14**

**8 2 2**

**0 1 4**

**8 6 6**

**7 8 7**

**1 2 8**

**5 4 10**

**3 5 14**

**7 6 1**

**6 5 2**

**2 5 4**

**2 3 7**

**0 7 8**

**3 4 9**

**1 7 11**

**Sample #1 Output**

**Prim's Algorithm Cost : 37**

**3 2 Weight : 7**

**6 7 Weight : 1**

**4 3 Weight : 9**

**1 0 Weight : 4**

**5 6 Weight : 2**

**8 2 Weight : 2**

**7 0 Weight : 8**

**2 5 Weight : 4**

**Total Time Elapsed : 34.8073 milliseconds.**

**Kruskal's Algorithm Cost : 37**

**8 2 Weight : 2**

**1 2 Weight : 8**

**3 4 Weight : 9**

**6 5 Weight : 2**

**2 5 Weight : 4**

**2 3 Weight : 7**

**7 6 Weight : 1**

**0 1 Weight : 4**

**Total Time Elapsed : 2.7258 milliseconds.**

**Note that your output should match mine if you use the classes I have discussed/provided but the order of the edges shown and, in some cases, the individual edges, may differ (e.g. if there are multiple edges with the same weight, as in the graph for example #1) as may the time it took an algorithm to execute. Note that the actual MCST Cost should NEVER differ however.**

**Sample input #2**

**4 5**

**0 1 10**

**1 3 15**

**0 3 5**

**0 2 6**

**2 3 4**

**Sample output #2**

**Prim's Algorithm Cost : 19**

**1 0 Weight : 10**

**3 0 Weight : 5**

**2 3 Weight : 4**

**Total Time Elapsed : 32.296341 milliseconds.**

**Kruskal's Algorithm Cost : 19**

**0 3 Weight : 5**

**0 1 Weight : 10**

**2 3 Weight : 4**

**Total Time Elapsed : 1.605831 milliseconds.**