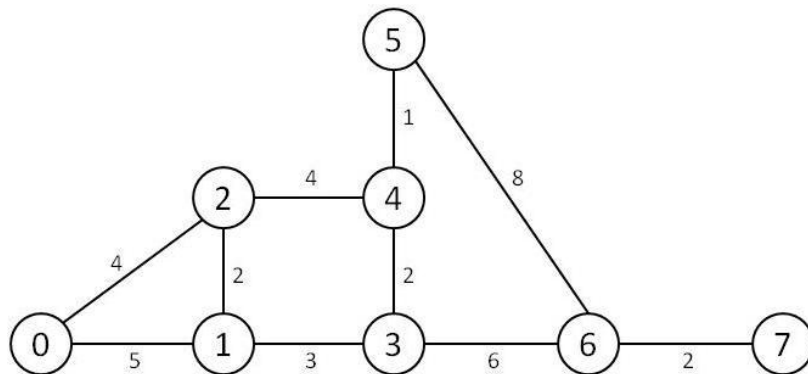


CS4404 Analysis of Algorithms

Unit 4 Assignment – Example Solution

Assignment

Develop an implementation of Prim's algorithms that determines the MST (Minimum Spanning Tree) of the graph from the Unit 2 assignment that we developed the data structure for.



(Note: I have changed the numbers on the vertices in this graph to match what is in the java code. The identifiers on the vertices is not important but rather the cost of spanning the graph)

For this assignment, develop an implementation using Java that first implements the graph in a data structure and then provides the algorithm that can determine the Minimum spanning tree within this graph in terms of cost. The cost will be the sum of the lengths of the edges that must be traversed. The cost of each edge is represented by the number on the edge. Your algorithm must output the total cost of spanning the tree as determined by your implementation of Prim's algorithm. The algorithm must produce output, which is the total cost of the path.

Grading Rubric

Rubric Items

Was a java implementation of a minimum spanning tree algorithm provided?

Is the code documented to give the reader an idea of what the author is trying to do within the code?

Does the java algorithm execute in the Java IDE environment

When executed does the algorithm produce output indicating the total cost of spanning the tree?

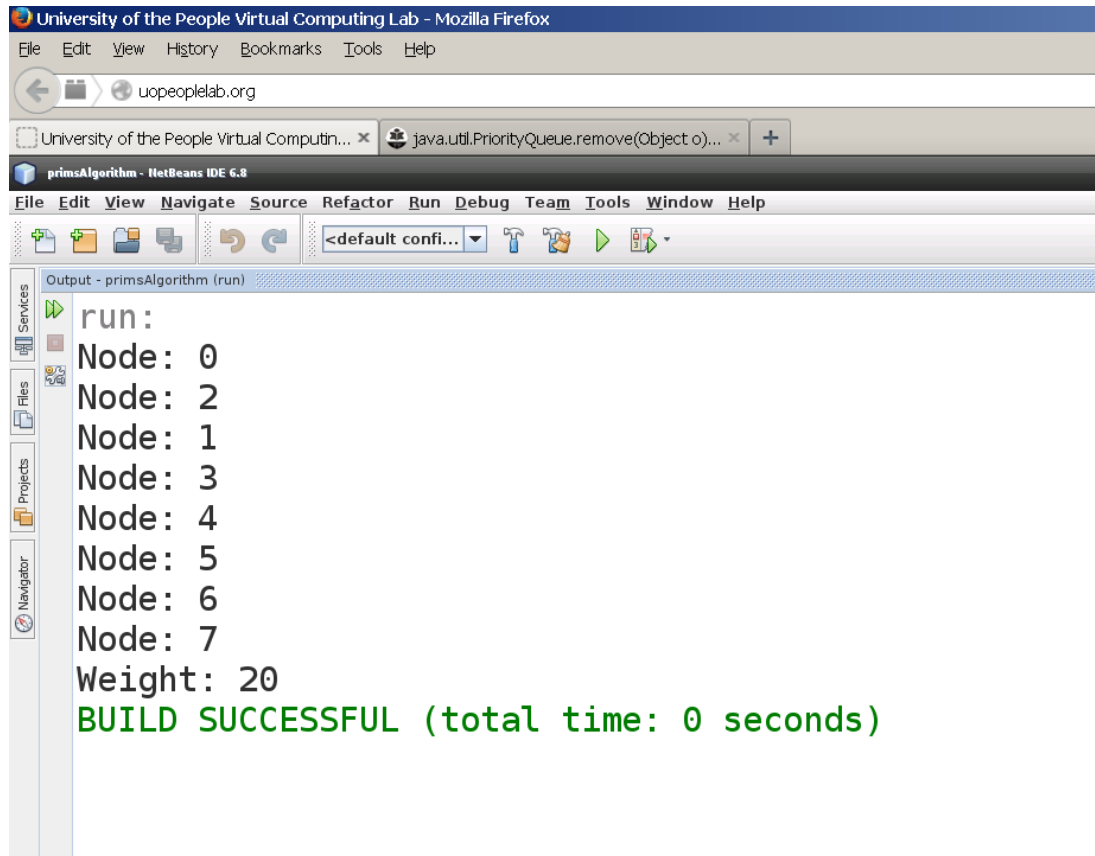
Does the cost reported by the algorithm match the cost provided by the instructor?

Does the assignment include an asymptotic analysis describing the complexity of the algorithm in terms of Big-O (Big- Θ , or Big- Ω as appropriate)?

Output

The following is the output of running the algorithm. What we see in the output is the cost of spanning the tree which is the total 'weight' of the edges selected as part of the MST.

In addition we see the order that the nodes were processed to produce the spanning tree.



The screenshot shows a web browser window titled "University of the People Virtual Computing Lab - Mozilla Firefox" with the address bar displaying "uopeoplelab.org". Below the browser, the NetBeans IDE 6.8 interface is visible, showing the "Output - primsAlgorithm (run)" window. The output text is as follows:

```
run:
Node: 0
Node: 2
Node: 1
Node: 3
Node: 4
Node: 5
Node: 6
Node: 7
Weight: 20
BUILD SUCCESSFUL (total time: 0 seconds)
```

Java Source Code

The following is the Java code that implements the solution. The code is broken into 4 pieces.

First is the Main class that has the implementation of Prim's algorithm. Second is the Node class which defines the structure and methods of a Node, the Vertex class which implements the structure and methods of a vertex and the Edge class which implements the structure and methods of an edge. The Node class is used as the structure for the Priority queue and includes both the node id and weight of the node.

```
/*
 * CS3304 Analysis of Algorithms
 * Unit 4: Assignment Solution
 * Implementation of Prim's algorithm for the tree provided in the assignment.
 * The tree is configured in two structures.
 * The first is a list of edges. For this we have defined an Edge class where
 * we instantiate an edge for each of the edges. An edge has a start and end node
 * and a weight. All of the edges are organized in an array.
```

```

*
* The second structure is all of the vertices. Each vertex is a object instantiated
* from the Vertex class and has a vertex id and a list of adjacent edges. The adjacent
* edges are simply indexes to the position of the edge in the edge array. So the edge
* that begins at node 0 and ends at node 1 is located in position 0 in the array so
* in the vertex the edges array would contain 0 to indicate that this edge connects an
* adjacent node this this node.
*
* One important point in the data is that I have renumbered the nodes to make it more convenient.
* So node 1 in the assignment is node 0 in the code. The following shows the mapping between
* the nodes in the assignment and the data in this program.

```

```

*
* Node in    Node in
* Assignment  Code
* 1          0
* 2          1
* 3          2
* 4          3
* 5          4
* 6          5
* 7          6
* 8          7
*

```

```

*/
package primsalgorithm;

import java.util.*;

public class Main {

    static int gid;
    static int gweight;
    public static void main(String[] args) {
        //natural ordering example of priority queue
        //PriorityQueue example with Comparator

        int i, t, n;
        int mstidx;
        int tweight;
        int[] mst = new int[9];
        Edge[] edges = new Edge[10];
        Vertex[] vertices = new Vertex[9];
        Edge tedge;
        Vertex v;
        Node tnode;
        Node tnode2;
        Iterator itr;

        //
        // Create object for each of the edge and put the object reference
        // into the array Edges
        //
        edges[0] = new Edge(0,1,5);
        edges[1] = new Edge(0,2,4);
        edges[2] = new Edge(1,2,2);
        edges[3] = new Edge(1,3,3);
        edges[4] = new Edge(2,4,4);
        edges[5] = new Edge(3,4,2);
        edges[6] = new Edge(3,6,6);
    }
}

```

```

edges[7] = new Edge(4,5,1);
edges[8] = new Edge(5,6,8);
edges[9] = new Edge(6,7,2);

//
// Create the nodes and identify the adjacent edges which we identify as the indices of the edge
// in the edges array. For example 0 refers to edge 1,2 with weight 5. 1 refers to edge 1,3 with
// weight 4 and so on.
//
vertices[0] = new Vertex(0,new int[] {0,1});
vertices[1] = new Vertex(1,new int[] {0,2,3});
vertices[2] = new Vertex(2,new int[] {1,2,4});
vertices[3] = new Vertex(3,new int[] {3,5,6});
vertices[4] = new Vertex(4,new int[] {4,5,7});
vertices[5] = new Vertex(5,new int[] {7,8});
vertices[6] = new Vertex(6,new int[] {6,8,9});
vertices[7] = new Vertex(7,new int[] {9});

//
// Build initial priority Queue. The first node has weight of 0 and the rest a weight of
// 9999 which simply is larger than any weight of any edge ensuring that it is at the end
// of the priority queue
//
PriorityQueue<Node> nodePriorityQueue = new PriorityQueue<Node>(7, idComparator);
gweight = 0;
for (i=0; i<8; i++) {
    gid = i;
    addDataToQueue(nodePriorityQueue);
    gweight = 9999;
}

//
// Get an array of the nodes in the priority queue
//
mstidx = 0;
tweight = 0;

//
// Process the graph.
// 1. Poll node from the priority queue
// 2. Get the adjacent edges for the node.
// 3. For each edge update the weight on the node if it is either 9999 or if the weight is less than current weight
// 4. Add the polled node to the minimum spanning tree array
// 5. Add the weight of the polled node to the global weight
//
do {
    tnode = nodePriorityQueue.poll();

    if (tnode != null) {
        t = tnode.getId();
        v = vertices[t];
        //
        // Process all of the edges that are adjacent to the polled node
        // the adjacent edges are in the vertex object
        //
        for (i=0; i<v.getEdges().length; i++) {

            tedge=edges[v.getEdges()[i]];
            //

```

```

// We need to look for the vertex that is opposite the one that
// is being processed. For example if we are processing node 1 in the graph
// and node one connects to node 0, 2, and 3 then we need to identify those nodes
// which can be in either the start or end of the edge.
//
if (tedge.getStart() == t) {
    n = tedge.getEnd();
} else {
    n = tedge.getStart();
}

//
// Setup an iterator to go through the priority queue. As we remove nodes from the priority
// queue by polling them, it will naturally implement the feature within Prim's where we only
// process those nodes which remain in the priority queue. As we iterate through the nodes
// we will identify any node that matches the node n that we need to update and then either update
// the weight if the weight in the edge is less than the weight for the node in the priority queue
// or will set the weight if it has not yet been set which we know because it will have the initial
// starting value of 9999. In this example, I am using 9999 as the 'infinity' that we saw in the
// lecture about prim's algorithm.
//
itr = nodePriorityQueue.iterator();

while (itr.hasNext()) {
    tnode2 = (Node) itr.next();
    if (tnode2.getId() == n) {
        if (tnode2.getWeight() > tedge.getWeight() || tnode2.getWeight() == 9999) {
            //
            // In this section we are removing the node whose weight needs to be updated
            // and adding a new node back into the queue. The reason that we are doing this
            // is because I found that if we just 'updated' the weight value it might not change
            // the order properly. By removing and then adding the node it forces the priority queue
            // to reorder the entries.
            //
            gid = tnode2.getId();
            gweight = tedge.getWeight();
            nodePriorityQueue.remove(tnode2);
            addDataToQueue(nodePriorityQueue);
            itr = nodePriorityQueue.iterator();
        }
    }
}

//
// Add the polled node to the minimum spanning tree array
// and add the weight to the total weight of the spanning tree
//
mst[mstidx++] = t;
tweight += tnode.getWeight();
}
} while (tnode != null);

//
// Print out the nodes (vertices) in the minimum spanning tree in the
// order that they were added and print the total weight of the
// spanning tree.
//
for (i=0; i<mstidx; i++) {
    System.out.println("Node: "+mst[i]);
}

```

```

    }
    System.out.println("Weight: "+tweight);
}

//Comparator anonymous class implementation
public static Comparator<Node>idComparator = new Comparator<Node>() {
    @Override
    public int compare(Node c1, Node c2) {
        return (int) (c1.getWeight() - c2.getWeight());
    }
};

//utility method to add random data to Queue
private static void addDataToQueue(Queue<Node> nodePriorityQueue) {
    nodePriorityQueue.add(new Node(gid, gweight));
}

//utility method to poll data from queue
private static Node pollDataFromQueue(Queue<Node> nodePriorityQueue) {
    Node node = nodePriorityQueue.poll();
    return(node);
}
}

```

```

package primsalgorithm;

```

```

public class Edge {
    private int _start;
    private int _end;
    private int _weight;

    public Edge(int s, int e, int w){
        this._start=s;
        this._end=e;
        this._weight=w;
    }
    public int getStart() {
        return _start;
    }
    public int getEnd() {
        return _end;
    }
    public int getWeight() {
        return _weight;
    }
}

```

```

package primsalgorithm;

```

```

public class Vertex {
    private int node;
    private int[] edges;

    public Vertex(int i, int[] arr){

```

```
    this.node=i;
    edges = arr;
}
public int getNode() {
    return node;
}
public int[] getEdges() {
    return edges;
}
}
```

```
package primsalgorithm;
```

```
public class Node {
    private int node;
    private int weight;

    public Node(int i, int w){
        this.node=i;
        this.weight=w;
    }
    public int getId() {
        return node;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int w) {
        this.weight = w;
    }
}
```