Part 3

Algorithm and Data Structure

Aneesh Garg - agarg6@uncc.edu - 800815619

4/29/2014

1.0 Assumptions 3
2.0 Graph Data Structures 3
3.0 Single Source Shortest Path 4
3.1 Input 4
3.2 Algorithm 4
3.3 Code 5
3.4 Output 6
3.5 Runtime Complexity 6
4.0 Minimum Spanning Tree - Kruskal Algorithm 7
4.1 Input 7
4.2 Algorithm 8
4.3 Runtime Complexity 8
4.4 Code 9
4.5 Output 10

1.0 Assumptions

1. Input file uses single space as separator. If something else is used please change SEPARATOR string in ImplementPart3.java
2. If graph is undirected graph then this means graph is for question2 and only Kruskal's algorithm will be applied.
3. If graph is directed graph then this means graph is for question and only shortest path will be calculated.
4. Input will be taken only from input.txt file. However I have included sample graph text files DAG Input.txt and MST Input.txt for question 1 and 2 respectively. Copy the contents of these files into input.txt file and then run ImplementPart3.java file.
5. You can also use your own graph but copy and paste the graph representation in input.txt file.

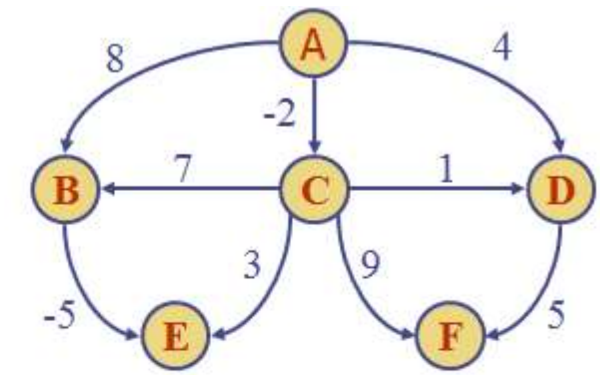
2.0 Graph Data Structures

I am using adjacency list data structure for graphs.

```java
public class Graph {
    public enum Type {
        DIRECTED, UNDIRECTED
    }

    private Type graphType;
    private ArrayList<VertexListObject> vertexList;
    private ArrayList<EdgeListObject> edgeList;
    private Vertex source;
    private int noOfVertices;
    private int noOfEdges;

    public void insertVertex(String key) {...

    public void insertEdge(String startKey, String endKey, int weight) {...

    public List<Edge> incidentEdges(Vertex vertex) {...

    public List<Edge> edges() {...

    public List<Vertex> vertices() {...

    public void printGraph() {...

    public Vertex opposite(Vertex u, Edge edge) {...

    /**
     * This method calculates the cost of the graph i.e. sum of weights of all
     * edges of the graph.
     *
     * @return
     */
    public int calculateCost() {...
```

3.0 Single Source Shortest Path

3.1 Input




```
6 9 D
A B 8
A D 4
A C -2
C B 7
C D 1
B E -5
C E 3
C F 9
D F 5
[Source A]
```

3.2 Algorithm

```
Algorithm DagDistances(G, s)
  for all  v ∈ G.vertices()
    if  v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  Perform a topological sort of the vertices
  for u ← 1 to n do      {in topological order}
    for each  e ∈ G.outEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if  r < getDistance(z)
        setDistance(z,r)
```

3.3 Code

```java
public class DAGDistance {

    /**
     * This method calculates and stores the shortest distance of every vertex
     * from start vertex. It also stores edges leading to minimum distance so
     * that shortes path can be printed
     * @param g
     *            : Directed Acyclic Graph
     * @param start
     *            : Source vertex
     */
    public void dagDistances(Graph g, Vertex start) {
        List<Vertex> vertexList = g.vertices();
        for (Vertex v : vertexList)
            if (v.equals(start))
                v.setDistance(0);
            else
                v.setDistance(Integer.MAX_VALUE);

        Vertex[] sortedVertices = topologicalDFS(g);

        for (Vertex u : sortedVertices) {
            List<EdgeListObject> outEdges = u.getOutEdges();
            if (outEdges != null) {
                for (EdgeListObject elo : outEdges) {
                    Vertex z = g.opposite(u, elo.getEdge());
                    int r = u.getDistance() + elo.getEdge().getWeight();
                    if (r < z.getDistance()) {
                        z.setDistance(r);
                        z.setParent(elo.getEdge());//To Track Shortest Path
                    }
                }
            }
        }
    }
    /**
     * This method sorts the vertices of a DAG in topological order and returns
     * an array of vertices in order
     * @param g
     *           Directed Acyclic Graph
     * @return Array of vertices in topologically sorted order
     */
    private Vertex[] topologicalDFS(Graph g) {

    private int topologicalDFS(Graph g, Vertex v, int n) {
}
```

3.4 Output

```
----- Actual Graph -----

Vertices: [A, B, D, C, E, F]
Edges: [A B 8, A D 4, A C -2, C B 7, C D 1, B E -5, C E 3, C F 9, D F 5]
Source: [A]
Type: [Directed]

----- Shortest Path from A -----
Vertex: A
Path: []
Cost: 0

Vertex: B
Path: [A C -2, C B 7]
Cost: 5

Vertex: D
Path: [A C -2, C D 1]
Cost: -1

Vertex: C
Path: [A C -2]
Cost: -2

Vertex: E
Path: [A C -2, C B 7, B E -5]
Cost: 0

Vertex: F
Path: [A C -2, C D 1, D F 5]
Cost: 4

---- Performance ----
Time taken for graph creation(ms): 15
Time taken to find shortest paths(ms): 5
Total time taken(ms): 21
```

3.5 Runtime Complexity

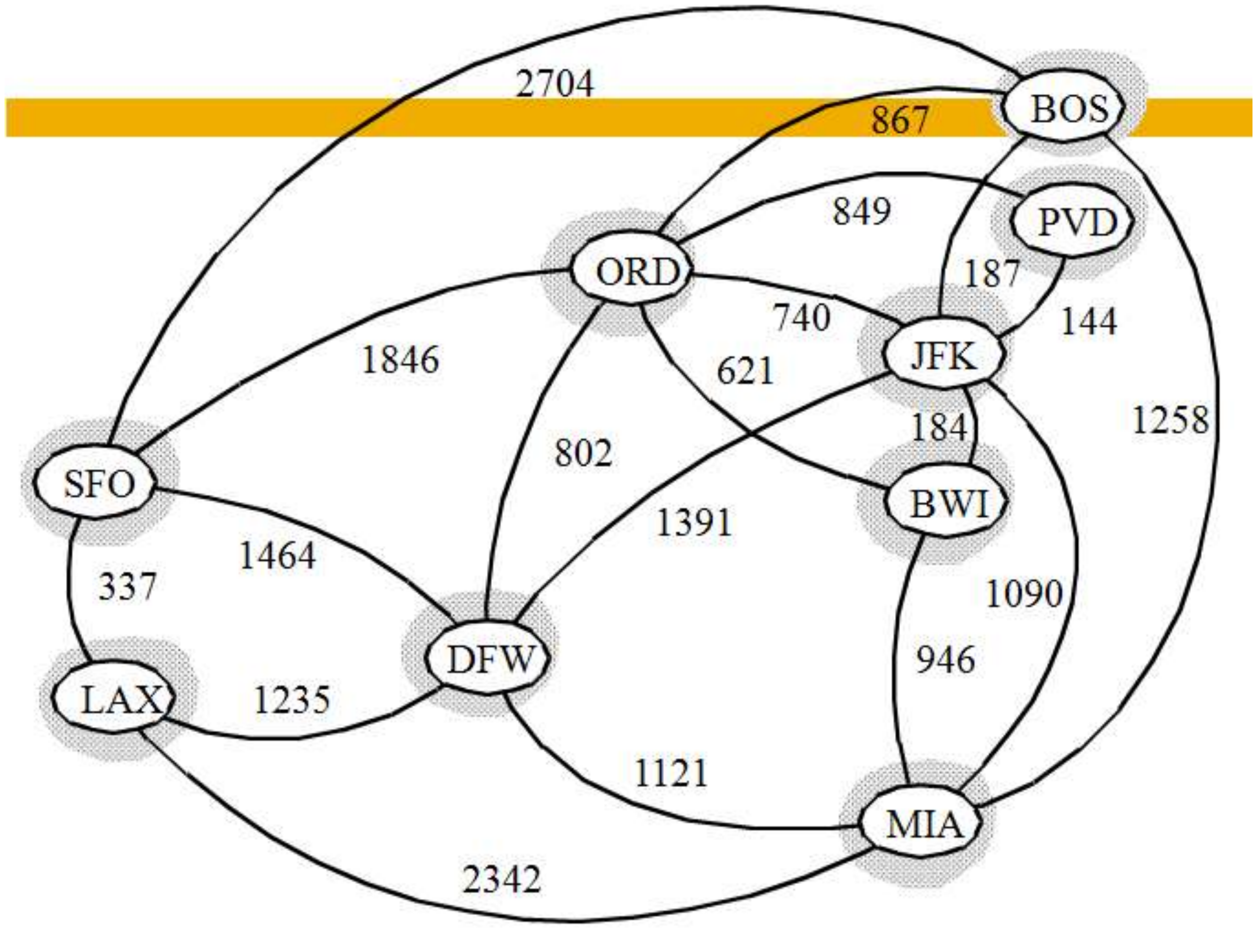
n : No of vertices
m: No of edges

Graph Creation: O (n + m)
Topological Sorting: O (n + m)
Shortest Path Running Time: O (n + m)

Net Runtime Complexity: O (n + m)

4.0 Minimum Spanning Tree - Kruskal Algorithm

4.1 Input




```
9 19 U
SFO LAX 337
SFO BOS 2704
SFO ORD 1846
SFO DFW 1464
LAX DFW 1235
LAX MIA 2342
DFW ORD 802
DFW JFK 1391
DFW MIA 1121
MIA BWI 946
MIA JFK 1090
MIA BOS 1258
ORD BOS 867
ORD PVD 849
ORD JFK 740
ORD BWI 621
BWI JFK 184
JFK BOS 187
JFK PVD 144
```

4.2 Algorithm

```
Algorithm Kruskal(G):
  Input: A weighted graph G.
  Output: An MST T for G.
Let P be a partition of the vertices of G, where each vertex forms a separate set.
Let Q be a priority queue storing the edges of G, sorted by their weights
Let T be an initially-empty tree
while Q is not empty do                              Running time: O(m log m)
   (u,v) ← Q.removeMinElement()
   if P.find(u) != P.find(v) then                    Running time: O(m log n)
         Add (u,v) to T
         P.union(u,v)
return T                                 Minimum Spanning Trees
```

4.3 Runtime Complexity

n : No of vertices
m: No of edges

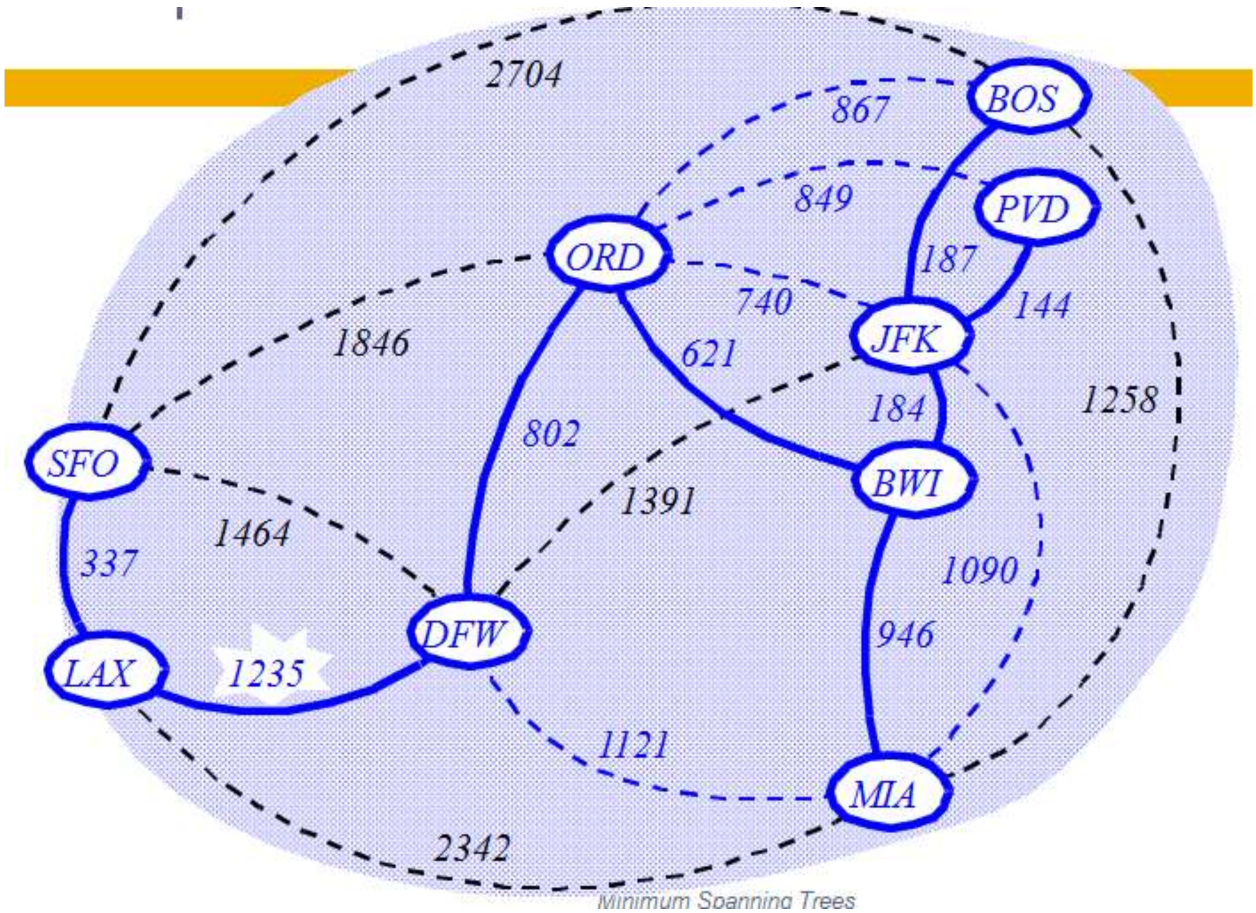
Graph Creation: O (n + m)
Removing minimum weight edges: O (m log m)
Creating Tree Time: O (m log n)

Net Runtime Complexity: O (n + m + m log m + m log n)

4.4 Code

```java
public class KruskalMST {
    public Graph findMST(Graph g) {

        List<Edge> edges = g.edges();
        List<Vertex> vertices = g.vertices();
        /**
         * partition of the vertices of G, where each vertex forms a separate
         * set
         */
        Partition p = new Partition(vertices);
        /**
         * priority queue storing the edges of G, sorted by their weights
         */
        PriorityQueue<Edge> q = new PriorityQueue<Edge>(edges);
        /**
         * an initially-empty tree
         */
        Graph mst = new Graph();

        while (!q.isEmpty()) {
            Edge minEdge = q.remove();
            System.out.println("Removed Edge: " + minEdge);
            Vertex u = minEdge.getStartVertex();
            Vertex v = minEdge.getEndVertex();
            int weight = minEdge.getWeight();

            if (p.find(u) != p.find(v)) {
                mst.insertEdge(u.getKey(), v.getKey(), weight);
                p.union(u, v, weight);
            }
        }

        return mst;
    }

    private static class Partition {
        Set<Graph> partitionSet;

        public Partition(List<Vertex> vertices) {...

        public void union(Vertex u, Vertex v, int weight) {...

        public Graph find(Vertex u) {...
    }
}
```

4.5 Output




```
----- Actual Graph -----

Vertices: [SFO, LAX, BOS, ORD, DFW, MIA, JFK, BWI, PVD]
Edges: [SFO LAX 337, SFO BOS 2704, SFO ORD 1846, SFO DFW 1464, LAX DFW 1235, LAX MIA 2342, DFW ORD 802, DFW JFK 1
Source: [null]
Type: [Undirected]

----- Minimum Spanning Tree Kruskal -----

Vertices: [JFK, PVD, BWI, BOS, SFO, LAX, ORD, DFW, MIA]
Edges: [JFK PVD 144, BWI JFK 184, JFK BOS 187, SFO LAX 337, ORD BWI 621, DFW ORD 802, MIA BWI 946, LAX DFW 1235]
Source: [null]
Type: [Undirected]
Cost: 4456

---- Performance ----
Time taken for graph creation(ms): 18
Time taken to implement kruskal(ms): 19
Total time taken(ms): 37
```