**Lab 13**

**Solam Jung Rana**

In this lab you will develop your graph package of software. Based on direction from the

slides, finish the implementation of the operations on a Graph:

boolean areAdjacent(Vertex u, Vertex v)

List getListOfAdjacentVerts (Vertex u)

Graph getSpanningTree()

List getConnectedComponents()

boolean isConnected()

boolean hasPathBetween(Vertex u, Vertex v)

boolean containsCycle()

boolean isTree()

boolean isBipartite()

DFS and the spanning tree algorithm have already been implemented. You will need to

use observations given in the slides to provide the connected components of the graph,

determine whether the graph has a cycle, and to determine if there is a path joining two

given vertices.

You will also implement BFS, and, in a subclass, implement the additional work needed

to determine if the graph has an odd cycle (so you can determine whether it is a bipartite

graph).

Finally, I have provided a second constructor in Graph that accepts an array of Edges (in

the form of Objects). One use for this constructor is that it allows you to return a

spanning tree as a Graph object after performing your spanning tree algorithm.

The toString method that is provided may not be suitable – you should modify it as

necessary so that you can display test results in a useful way.

**Lab 14**

1. Must every dense graph be connected? Prove your answer.

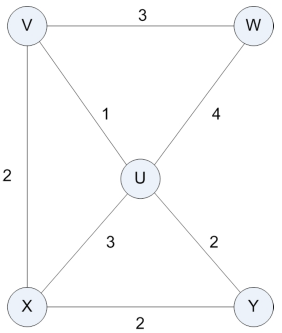
**NO.**

**A graph G is dense if e = θ(n2)**

**Let G = K1 U Kn-1. This shows a graph may not be connected but have θ(n2) edges.**

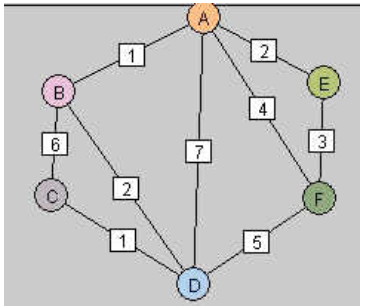
1. Carry out the steps of Dijkstra's algorithm to compute the length of the shortest path between vertex V and vertex Y in the graph I gave in class (reproduced below).

Display the evolution of the values for D[] in a table.



|  |  |
| --- | --- |
| **Vertex** | **0** |
| **V** | **∞** |
| **U** | **∞, 1** |
| **W** | **∞, 3** |
| **X** | **∞, 2** |
| **Y** | **∞, 3** |

1. Carry out the steps of Kruskal's algorithm to compute a minimum spanning tree for the graph shown below. Express the tree as a set of edges, and display the evolution of clusters in a table.



**Priority Queue:**

**AB**

**CD**

**AE**

**BD**

**EF**

**AF**

**FD**

**BC**

**AD**

|  |  |
| --- | --- |
| **C(A)** | **{A} {AB} {ABE} {ABCDE} {ABCDEF}** |
| **C(B)** | **{B} {AB} {ABE} {ABCDE} {ABCDEF}** |
| **C(C)** | **{C} {CD} {ABCDE} {ABCDEF}** |
| **C(D)** | **{D} {CD} {ABCDE} {ABCDEF}** |
| **C(E)** | **{E} {ABE} {ABCDE} {ABCDEF}** |
| **C(F)** | **{F} {ABCDEF}** |