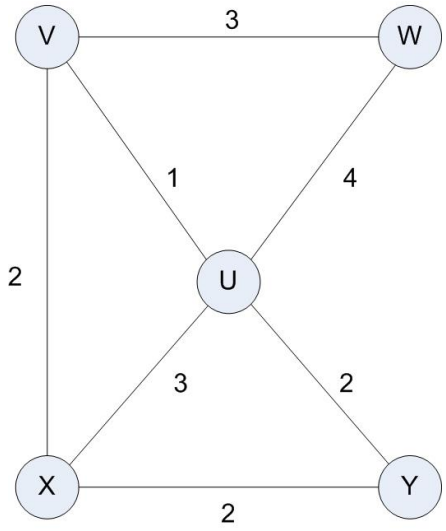
Lab 13

1. Must every dense graph be connected? Explain.

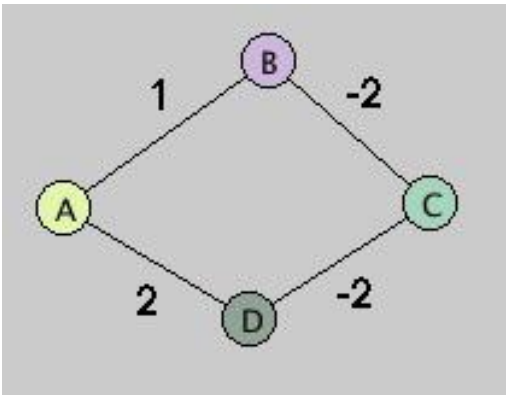
2. Carry out the steps of Dijkstra's algorithm to compute the length of the shortest path between vertex V and vertex Y in the graph below. Your final answer should consist of three elements:

    a) The length of the shortest path from V to Y
    b) The list A[] which shows shortest distances between V and every other vertex
    c) The list B[] which shows shortest paths between V and every other vertex



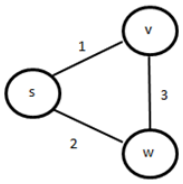
3. *Points about Dijkstra's Algorithm*

    a. What is the shortest path from A to C in the graph below?



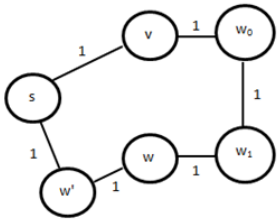
b.

Why is Dijkstra's approach to the shortest path problem better than simply using BFS, as described in the previous lesson?



[BFS approach: Making all edge weights = 1 is same as removing all weights. Perform BFS with start vertex s and compute distance to each vertex by returning its *level* in the BFS spanning tree. These computed values should be same as values found using Dijkstra]

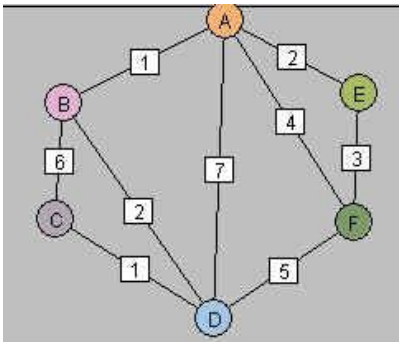
↓ BFS Style



4. Describe an algorithm for deleting a key from a heap-based priority queue that runs in O(log n) time, where n is the number of nodes. (Hint: You may use auxiliary storage as the priority queue is built and maintained. Assume there are no two nodes have the same key.) This technique is needed for the optimized Dijkstra algorithm discussed in the slides.

Kruskal's Algorithm and Disjoint Sets

5. Carry out the steps of Kruskal's algorithm for the following weighted graph, using the tree-based DisjointSets data structure to represent clusters. Keep track of edges as they are added to T and show the state of representing trees through each iteration of the main while loop.



6. The goal of this exercise is to devise a feasible algorithm that decides whether an input integer is prime. The key fact that you will make use of is the following:

   Fact: There is a function *f*, which runs in O(log *n*) (that is, O(length(*n*))), such that for any odd positive integer *n* and any *a* chosen randomly in [1, *n* - 1], if *f*(*a*, *n*) = 1, then *n* is composite, but if *f*(*a*,*n*) = 0, *n* is "probably" prime, but is in fact composite with probability < ½.
   A first try at such an algorithm would be:

   Algorithm **FirstTry**:
   Input: A positive integer n
   Ouptut: TRUE if n is prime, FALSE if n is composite

```
if n % 2 = 0 return FALSE
a ← random number in [1, n-1]
if f(a,n) = 1
   return FALSE
return TRUE
```

   Notice that **FirstTry** runs in O(log *n*). It also produces a correct result more than half the time.

   What could be done to improve the degree of correctness of **FirstTry** but still preserve a reasonably good running time? Explain.