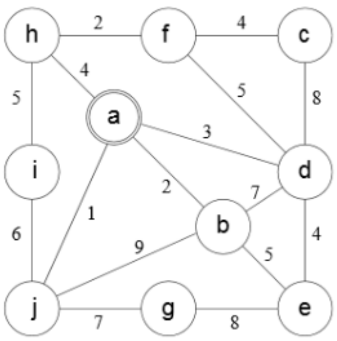
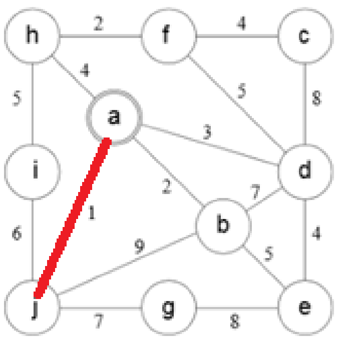
CS 325 Homework Assignment 5

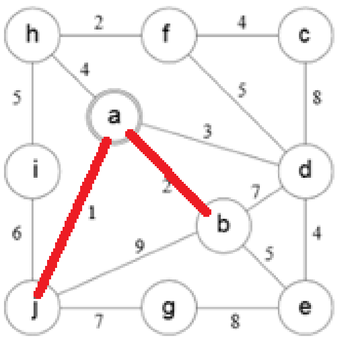
Peter Moldenhauer

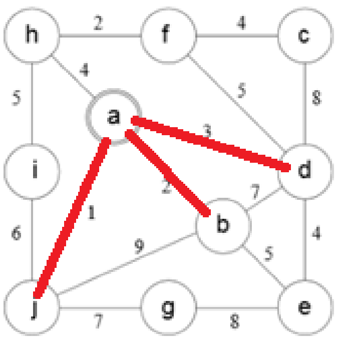
10/30/17

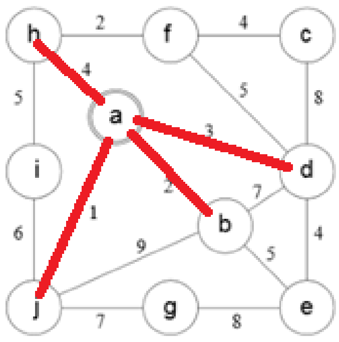
1 Demonstrate Prim's algorithm on the graph below by showing the steps in subsequent graphs as shown in Figures 23.5 on page 635 of the text. What is the weight of the minimum spanning tree? Start at vertex a.

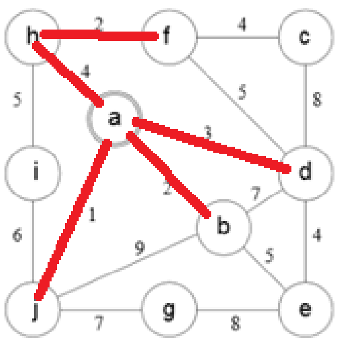


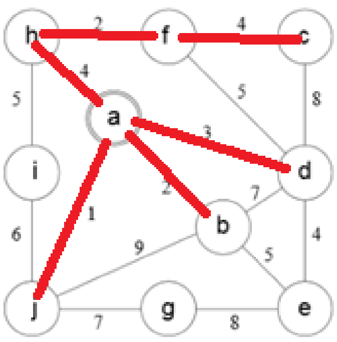


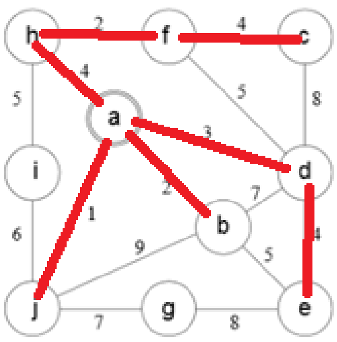


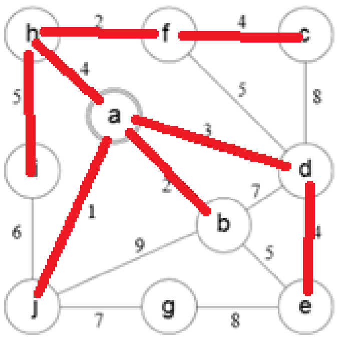


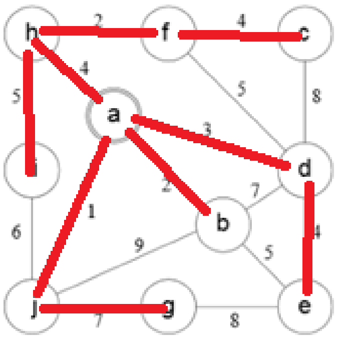












The total weight of the minimum spanning tree is 32

2 Consider an undirected graph G = (V, E) with nonnegative edge weights w(u,v) >= 0. Suppose that you have computed a minimum spanning tree G, and that you have also computed shortest paths to all vertices from vertex s in V. Now suppose each edge weight is increased by 1: the new weights w'(u,v) = w(u,v) + 1.

a) Does the minimum spanning tree change? Give an example it changes or prove it cannot change.

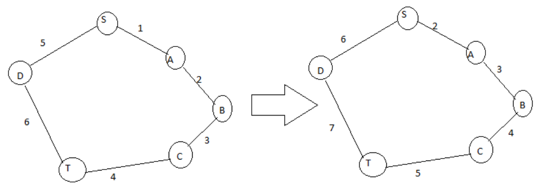
No, the minimum spanning tree does NOT change. This can be seen by using Kruskal's Algorithm which is an edge based algorithm. With this algorithm, the edges are added one at a time in increasing weight order while a forest of trees is maintained. An edge is accepted if it connects vertices of distinct trees. In this problem, even though all of the edge weights are increased by 1, Kruskal's Algorithm will still consider all of the edges in the same order (as if it were running on the original edge weights to find the minimum spanning tree). This is because all of the edge weights were increased by the same amount (this would not be the case if different edges were increased by different amounts). Therefore, we have the same graph structure as the original graph and Kruskal's Algorithm will add the same edges to the minimum spanning tree.

We can prove the correctness of Kruskal's Algorithm by poof by contradiction. We can assume that the algorithm is wrong and does not produce a minimum spanning tree. If we assume this, then the algorithm adds a wrong edge at some point in the process. If the algorithm adds a wrong edge, then there must be a lower weight edge. However, since the algorithm chooses the lowest weight edge at each step this is a contradiction.

Therefore, after proving the correctness of Kruskal's Algorithm and by also showing that Kruskal's Algorithm will yield the same result if all of the edge weights are increased by 1, we can say that the minimum spanning tree does not change.

b) Do the shortest paths change? Give an example where they change or prove they cannot change.

Yes, it is possible for the shortest paths to change. This can be seen in the following example:



In this example above, there are two paths from s to t. The path on the left side uses only two edges while the path on the right side uses four edges. Since each edge weight is increased by 1, the total length of a path increases based on the number of edges of that given path. Therefore, a path with fewer edges will grow less compared to a path with more edges. Since the path on the right has twice as many edges than the path on the left, the total path weight will grow when each edge weight is increased by 1.

3 In the bottleneck-path problem, you are given a graph G with edge weights, two vertices s and t and a particular weight W; your goal is to find a path from s to t in which every edge has at least weight W.

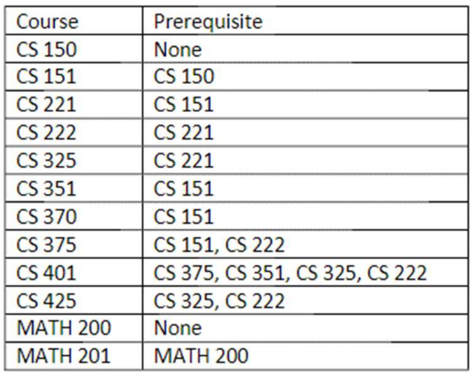
a) Describe an efficient algorithm to solve this problem.

An efficient algorithm to solve this problem would be to us the Breadth-First Search algorithm. However, instead of using the “traditional” rules of Breadth-First Search, we would modify it so that it would ignore any edges that have a weight less than W. This will ensure that every edge in the final path from s to t has at least weight W.

b) What is the running time of your algorithm.

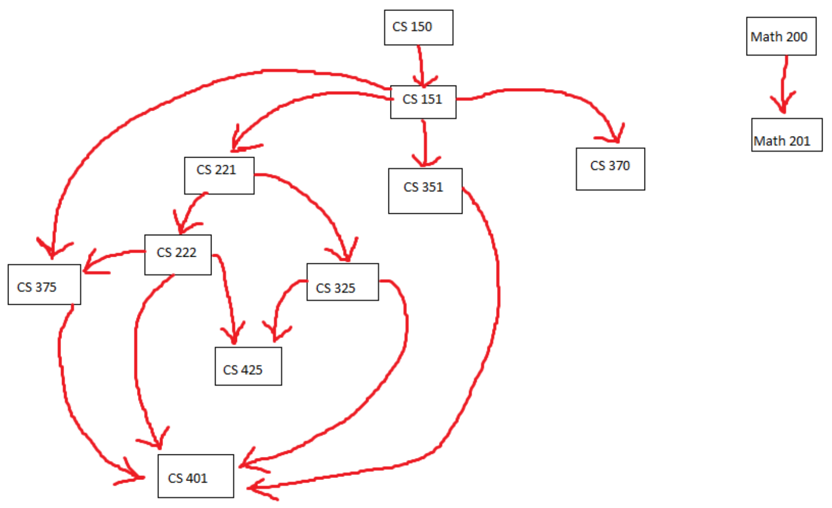
The running time of this algorithm will be the same as the running time of Breadth-First Search which is O(V + E). This is where V is the number of vertices and E is the number of edges.

4 Below is a list of courses and prerequisites for a factious CS degree.



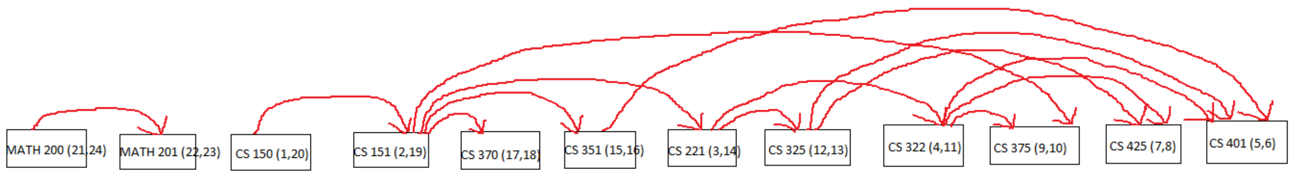
Course	Prerequisite
CS 150	None
CS 151	CS 150
CS 221	CS 151
CS 222	CS 221
CS 325	CS 221
CS 351	CS 151
CS 370	CS 151
CS 375	CS 151, CS 222
CS 401	CS 375, CS 351, CS 325, CS 222
CS 425	CS 325, CS 222
MATH 200	None
MATH 201	MATH 200

a) Draw a directed acyclic graph (DAG) that represents the precedence among the courses.



b) Give a topological sort of the graph

Math 200, Math 201, CS 150, CS 151, CS 370, CS 351, CS 221, CS 325, CS 222, CS 375, CS 425, CS 401



c) If you are allowed to take multiple courses at a time as long as there is no prerequisite conflict, find an order in which all the classes can be taken in the fewest number of terms.

Term 1: Math 200, CS 150
Term 2: Math 201, CS 151
Term 3: CS 221, CS 351, CS 370
Term 4: CS 222, CS 325
Term 5: CS 375, CS 425
Term 6: CS 401

d) Determine the length of the longest path in the DAG. How did you find it? What does this represent?

The length of the longest path in the above DAG is 5. You can get this path by going from CS 150 to CS 151 to CS 221 to CS 222 to CS 375 and then to CS 401. I found this longest path by visually looking at the DAG above and by tracing it through. Another way to find this longest path is to start at vertex CS 150 and to run Breadth-First Search until you get to vertex CS 401. The longest path of 5 represents the number of terms minus 1 that would be needed to finish all of the courses. Or in other words, it represents the longest chain of prerequisites in the CS degree. This is because each edge is composed of 2 vertices and also because it is a DAG, there are no cycles in the graph.

5 Suppose there are two types of professional wrestlers: “Babyfaces” (“good guys”) and “Heels” (“bad guys”). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n wrestlers and we have a list of r pairs of rivalries.

a) Give pseudocode for an efficient algorithm that determines whether it is possible to designate some of the wrestlers as Babyfaces and the remainder as Heels such that each rivalry is between a Babyface and a Heel. If it is possible to perform such a designation, your algorithm should produce it.

An algorithm to solve this problem could be described by the following: We need to create a graph where each vertex represents a wrestler and each edge represents a rivalry. The graph will then contain n vertices and r edges. We then run “Breadth-First Search” on the graph to visit all of the vertices. Throughout this process we will assign “Babyfaces” to all wrestlers whose distance to the starting vertex is even. This includes assigning the starting vertex as a “Babyface” as well. Similarly, we assign “Heels” to all wrestlers whose distance to the starting vertex is odd. We then also check each edge to verify that it goes between a Babyface and a Heel. If we find that any child shares the same label (Babface/Heel) as it’s parent, then we have a non-bipartite graph. In this case, it means that it is not possible to designate the wrestlers in a manner such that each rivalry is between a Babyface and a Heel. The algorithm returns whether or not this is possible and if so, it then prints the different “teams” to the terminal.

The pseudocode for this can be seen below:

```
Graph = createGraph(inputData)
BreadthFirstSearch(Graph, startingVertex)
    Initialize vertices
    Q = { startingVertex }
    while (Q not empty)
        u = DEQUEUE(Q)
        for each v in Graph.Adj[u] {
            if (v.color == WHITE)
                v.color = GREY
                v.d = u.d + 1
                v.p = u
            if (v.d is even)
```

```
                v.team = Babyface
            else
                v.team = Heel
            if(v.team == u.team)
                isPossible = False
if(isPossible)
    print(True + teams)
else
    print(Not possible)
```

b) What is the running time of your algorithm?

   This algorithm takes O(n + r) time for the Breadth-First Search, O(n) time to designate each wrestler as a Babyface or Heel and O(r) time to check the edges. This Results in O(n + r) time overall.

c) Implement: Babyfaces vs Heels.

   [Implementation submitted to TEACH separately]