Christopher Merrill

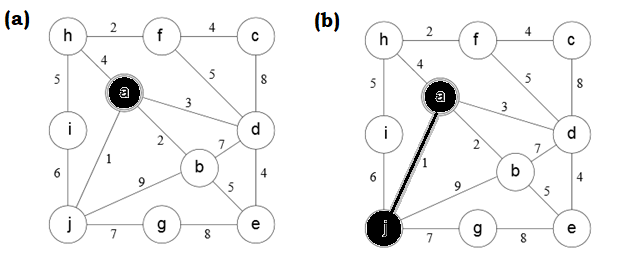
CS325

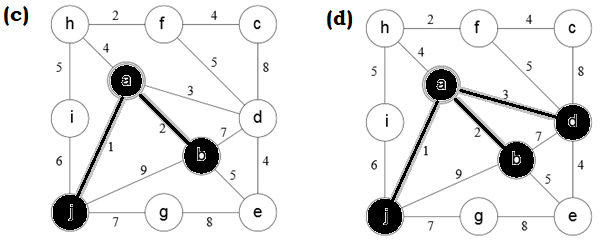
11/5/17

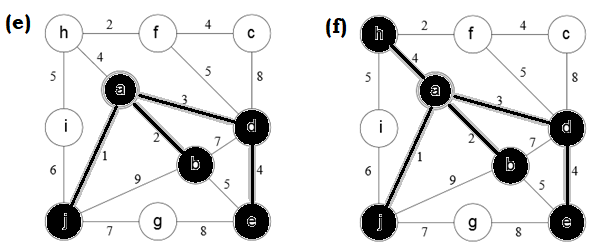
**Homework 5**

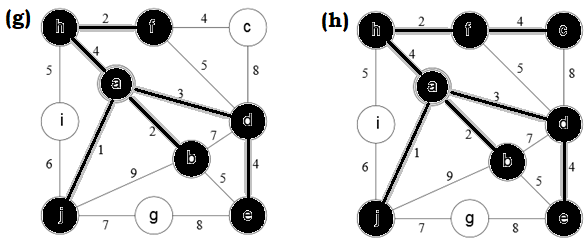
**1. *(3 points)* Demonstrate Prim’s algorithm on the graph below by showing the steps in subsequent graphs as shown in Figures 23.5 on page 635 of the text. What is the weight of the minimum spanning tree? Start at vertex a.**

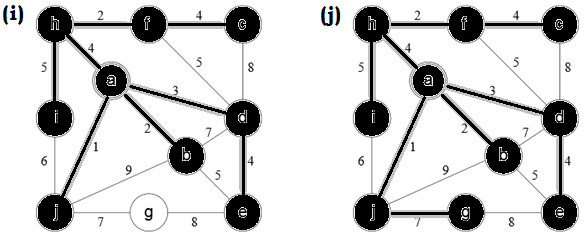












Weight of Minimum Spanning Tree**:** 32

**2. *(6 points)* 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∈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. When growing a minimum spanning tree from u to v, you will always choose the lightest available edges that connect to each vertex. If the following edges weights are available for w(u, v) to connect to the next vertex:

e1 < e2 < … < en

then e1 would be the edge that is safely added to the MST. Therefore, when considering edge weights w’(u,v) where 1 has been added to each edge, we would still be left with the same ordering of edges from lightest to heaviest at any given point when growing the tree:

e1’ < e2’ < … < en’ => e1 + 1 < e2 + 1 < … < en + 1

In this case we would still choose e1 over any other available edge. Thus, there would be no difference in the MST if each edge weight is increased by 1.

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

No, the shortest paths do not change. Dijkstra’s algorithm applies in this case, and is therefore guaranteed to provide the shortest path between any two vertices on graph G. Dijkstra’s algorithm utilizes a greedy strategy where it always chooses the “lightest” or “closest” vertex in V-S to add to a set S (pg 659 of the textbook). Considering the case off adding a new vertex x that connects u to v, when considering the following edges to traverse:

e1 < e2 < … < en

Dijkstra’s algorithm will always traverse e1 to reach vertex x. Therefore, when considering edge weights w’(u,v) where 1 has been added to each edge, we would still be left with the same ordering of edges from lightest to heaviest at any given point when creating our shortest path:

e1’ < e2’ < … < en’ => e1 + 1 < e2 + 1 < … < en + 1

In this case we would still choose e1 over any other available edge. Thus, there would be no difference in any shortest path between two vertices if each edge weight is increased by 1.

**3. *(4 points)* 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.**

In order to solve this problem we can do the following:

1. Loop through all edges in the graph. If the weight of an edge is less than W, remove it from the graph.
2. Perform a breadth-first search on the remaining graph to find a path from s to t.

By removing all edges less than W, we are guarunteed that all remaining paths will meet the necessary criteria. The reason a breadth-first search is favored over a depth-first search, even though both have a worst case runtime of O(V + E), is that we may not need to search the entire graph to find a path from s to t. In the case where not all paths need be examined, a breadth-first search should be more efficient.

**(b) What is the running time of your algorithm.**

In step 1 we only need to examine each edge once, so the runtime of this step should be ϴ(E). For step 2, it is already known that a BFS has a worst case run time of O(V + E). Therefore the total runtime requires examining all vertexes once and all edges twice (in the worst case), which would give us O(V + 2E).

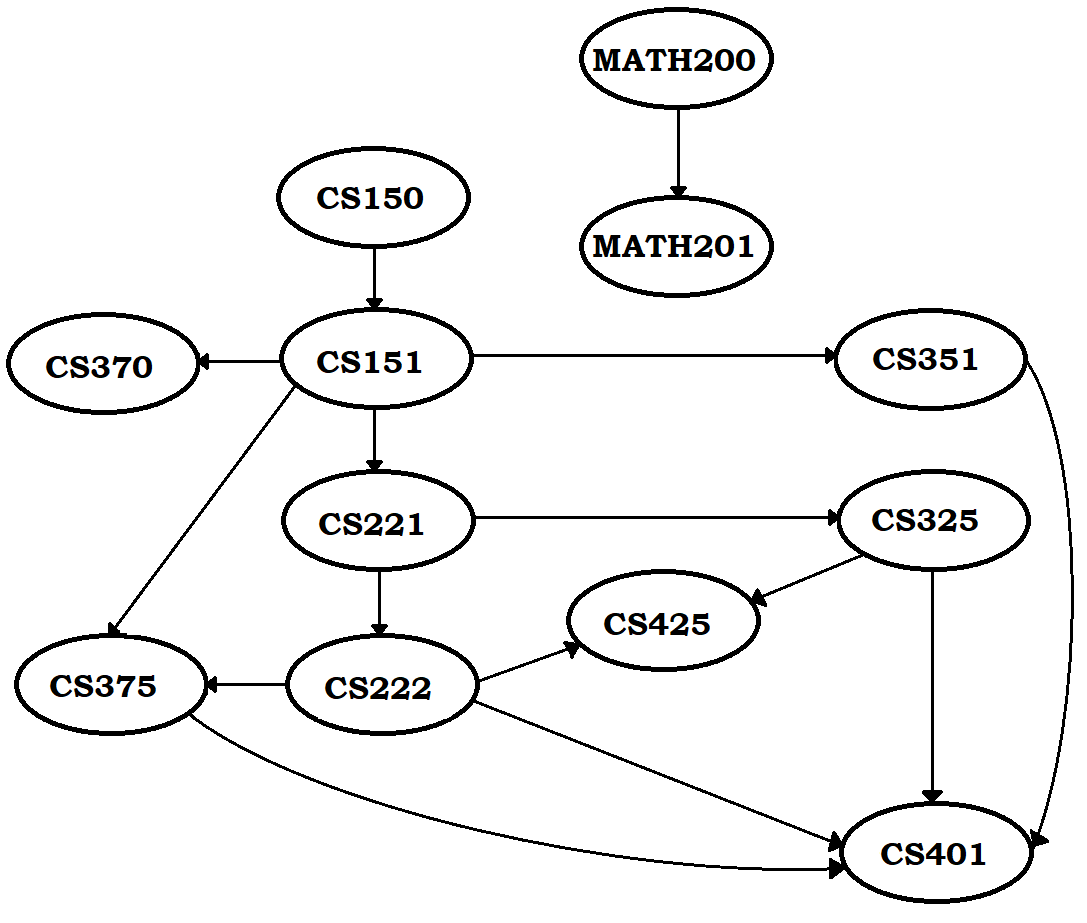
For large E, the leading “2” would become insignificant and we would be left with:

O(V + E)

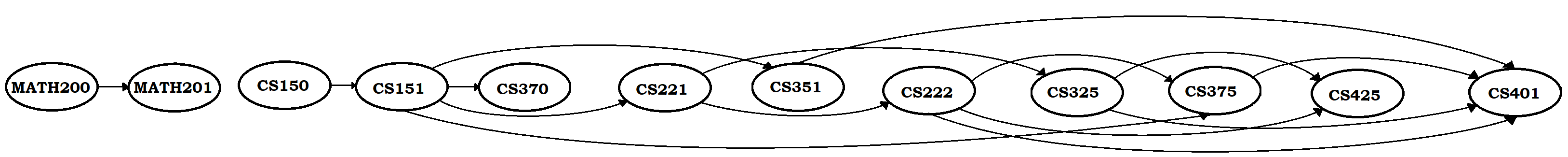
**4. *(5 points)* Below is a list of courses and prerequisites for a factious CS degree.**

|  |  |
| --- | --- |
| 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.**

****

**(c) If you are allowed to take multiple courses at one 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** | **2** | **3** | **4** | **5** | **6** |
| **Courses** | CS150 | CS151 | CS221 | CS222 | CS375 | CS401 |
| MATH200 | MATH201 | CS351 | CS325 | CS425 |  |
|  |  | CS370 |  |  |  |

Fewest Number of Terms: 6

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

The longest path in the DAG is as followed:

CS150 -> CS151 -> CS221 -> CS222 -> CS375 -> CS401

I found it by hand tracing through the first round of a depth first search, in which CS401 was the deepest vertex reached. The length of this path represents the shortest number of terms required to complete all of the courses.

**5. *(12 points)* 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.**

The following steps should produce an efficient algorithm for this problem:

1. Create a graph where every vertex is a wrestler and every edge is a rivalry.
2. Run as many BFS’s as necessary to find all vertices.
3. Assign all even-distance wrestlers as BabyFaces.
4. Assign all odd-distance wrestlers as Heels.
5. Check each edge to make sure it is between a BabyFace and a Heel

PSEUDOCODE:

generateTeams(Graph G, Size V) {

FOR all unvisited vertices i

BFS(i, G)

FOR each vertex v

IF distance is even

v = BabyFace

ELSE

v = Heel

FOR each edge e

IF teamA == teamB

RETURN Not Possible

RETURN Team Names

}

**(b) What is the running time of your algorithm?**

It takes O(n + r) time for the BFS, O(n) time to designate each wrestler’s team, and O(r) time to check edges. Thus the total running time is O(n + r).

**(c) Implement: Babyfaces vs Heels.**

Submitted to TEACH