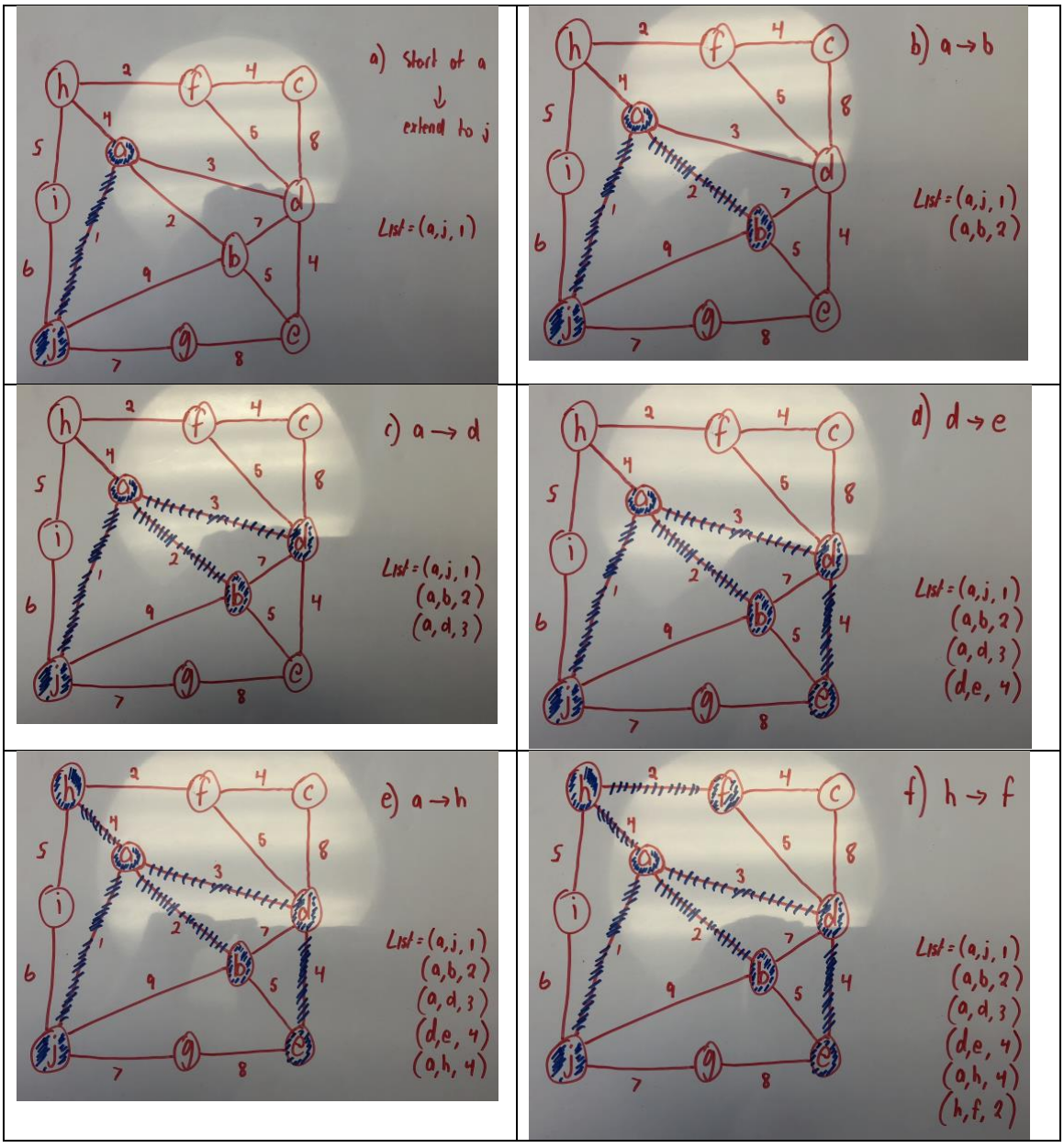
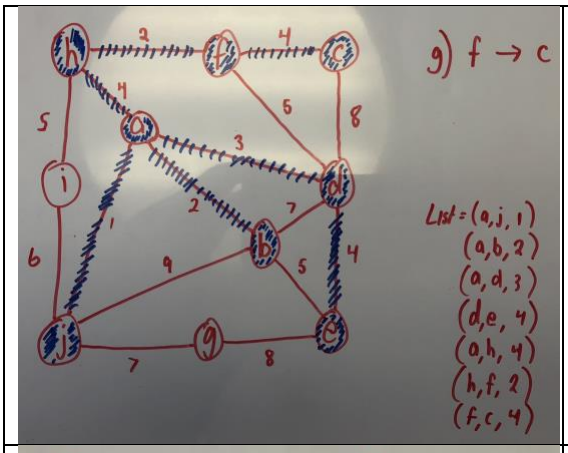
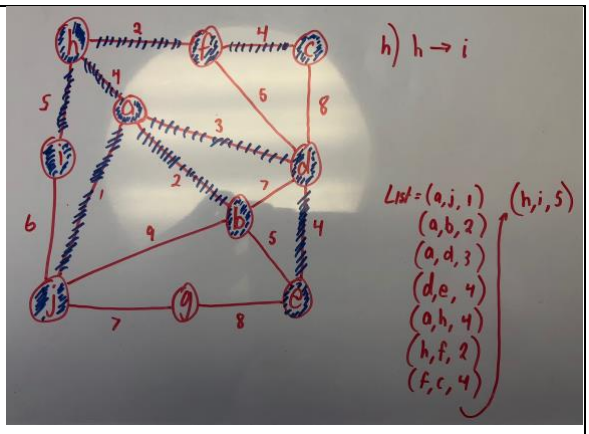
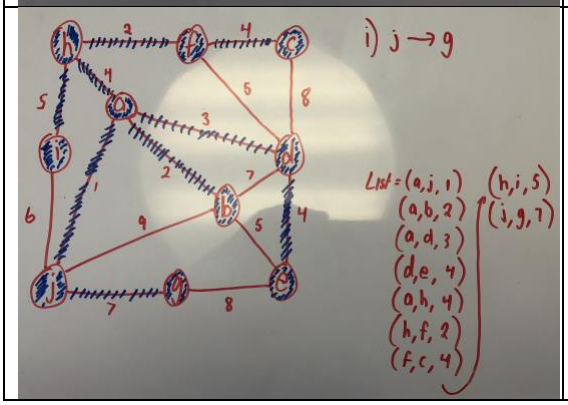
Christopher Ragasa
CS 325, HW 5
May 8, 2018

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.

The weight of the minimum spanning tree is **32**. For details, see pictures below.



<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td> 
</td>
<td>After adding all of the selected edges, we can calculate that the total weight of the MST is 32.</td>
</tr>
</table>

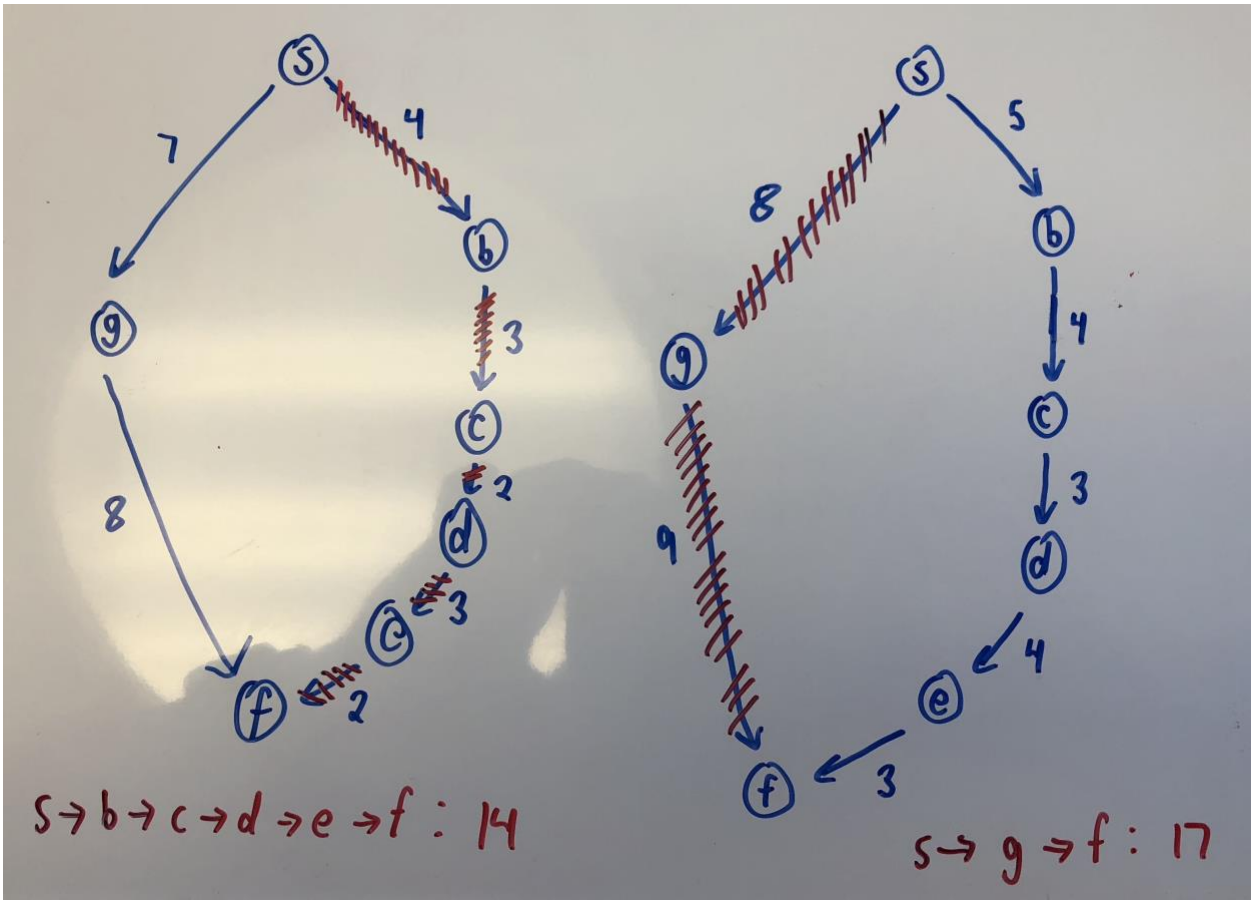
2) *(6 points)* Consider an undirected graph G=(V, E) with nonnegative edge weights w(u,v)>=0. Suppose that you have computed a minimum spanning tree G, and that you have also computed shortest paths to all vertices from vertex s ∈ V. Now suppose each edge weight is increased by 1: the new weights w'(u,v) = w(u, v) + 1.

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

a. No, the minimum spanning tree does not change. Consider running Kruskal's algorithm to find the MST. Since there are distinct edge weights and all edge weights are increased by 1, there is exactly one MST. As all edge weights are increased by the same amount, the graph structure does not change. Therefore, Kruskal's algorithm will add the same edges to the MST. We know that Kruskal's algorithm is proven to be correct, so, therefore, the MST cannot change.

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

a. Yes, it is possible that the shortest path may change. Since all edge weights are increased by 1, the length of a path increases by the number of edges that are on the path. Let's take a look at an example:



In the figure above, observe the shortest path from s → f of the graph on the left side. The shortest path is s → b → c → d → e → f, with a total weight of 14. After incrementing each of the edge weights of the graph by 1 (graph on right side), the shortest path has changed to s → g → f, with a total weight of 17. If we decided to use the same path as the one highlighted on the left graph of s → b → c → d → e → f, the weight would be 19, which is clearly heavier than the path of s → g → f.

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.

Dijkstra's Algorithm is a perfect algorithm to solve this problem. This algorithm is precisely designed to take a graph G and finds the shortest path between vertices s and t. The algorithm starts with vertex s and initializes it's distance value, d, to 0. The distance values for all other nodes in the graph are initialized to infinity.

First, we look at all nodes accessible by the current node (in this case of the first iteration, s). For each of these accessible nodes, if the distance value is greater than the distance value of the current node plus the weight of the connecting edge, update the

distance value of that node. After the distance values of all accessible nodes are calculated, visit the node with the smallest distance value. Repeat this process until you have visited t or have cycled through all of the nodes. The distance value at t will be the shortest path. Let's take a look at the pseudocode:

```
dijsktra(graph, initial):
    visited = {initial: 0}
    path = {}

    nodes = set of all nodes in the graph

    while nodes:
        min_node = None
        for node in nodes:
            if min_node is None:
                min_node = node
            elif visited[node] < visited[min_node]:
                min_node = node

    if min_node is None:
        break

    nodes.remove(min_node)
    current_weight = visited[min_node]

    for edge in graph.edges[min_node]:
        weight = current_weight + graph.distance[(min_node, edge)]
        if edge not in visited or weight < visited[edge]:
            visited[edge] = weight
            path[edge] = min_node

return visited, path
```

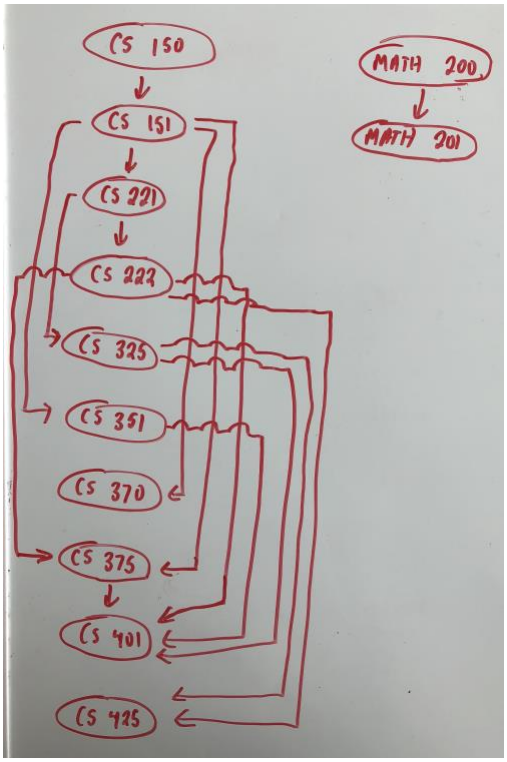
(b) What is the running time of your algorithm.

The running time of this algorithm is $O(V^2)$.

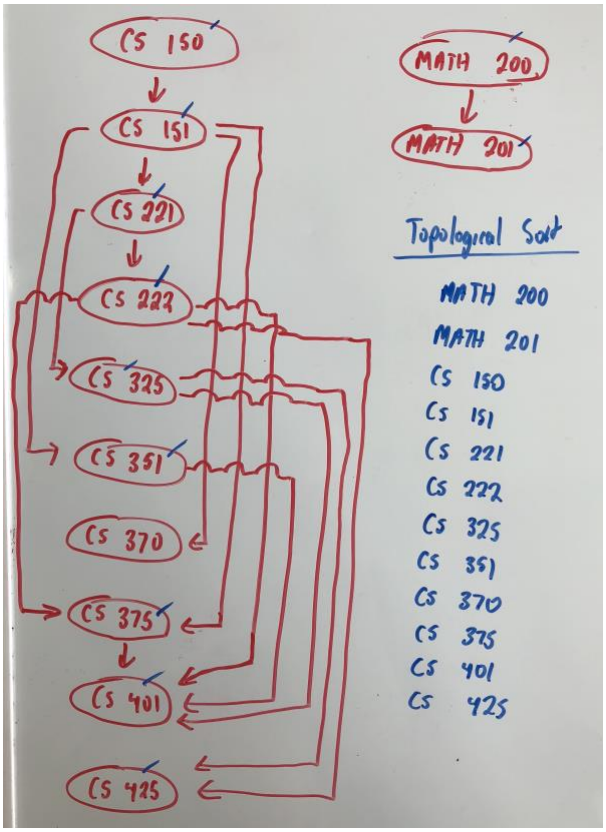
4. *(5 points)* Below is a list of courses and prerequisites for a factious CS degree.

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: MATH 200, CS 150
Term 2: MATH 201, CS 151
Term 3: CS 221, CS 370, CS 351
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 do you find it? What does this represent?

The longest path in this DAG is CS 150 → CS 151 → CS 221 → CS 222 → CS 375 → CS 401, with a length of 5. I identified this path by visually looking for a path that had the most connected edges. This longest path represents the longest continuous list of prerequisites.

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.

```
Initialize a queue
Initialize standard information for start vertex
can_designate = True
Process the neighbors of the starting vertex
While the length of the queue is greater than 0
        Visit the node that is at the front of the queue
        For each of the neighbors of this node
                add the neighbor to the queue
                update the distance
                if the distance is even
                        designate node as Babyface
                if the distance is odd
                        designate node as Heel
                if the child node and parent node are of the same type
                        can_designate = False
if can_designate = True
        the teams can be designated
```

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

Due to BFS, the running time of the algorithm is O(V+E), where V is the number of vertices in the graph, and E is the number of edges in the graph.

(c) **Implemen**t: Babyfaces vs Heels.

See CANVAS and source code attached.

Input: Input is read in from a file specified in the command line at run time. The file contains the number of wrestlers, n, followed by their names, the number of rivalries r and rivalries listed in pairs. *Note: The file only contains one list of rivalries*
Output: Results are outputted to the terminal.

- Yes, if possible followed by a list of the Babyface wrestlers and a list of the Heels .
- No, if impossible.

Sample Input file:

```
5
Ace
Duke
Jax
Biggs
Stone
6
Ace Duke
Ace Biggs
Jax Duke
Stone Biggs
Stone Duke
Biggs Jax
```

Sample Output:

```
Yes
Babyfaces: Ace Jax Stone
Heels: Biggs Duke
```