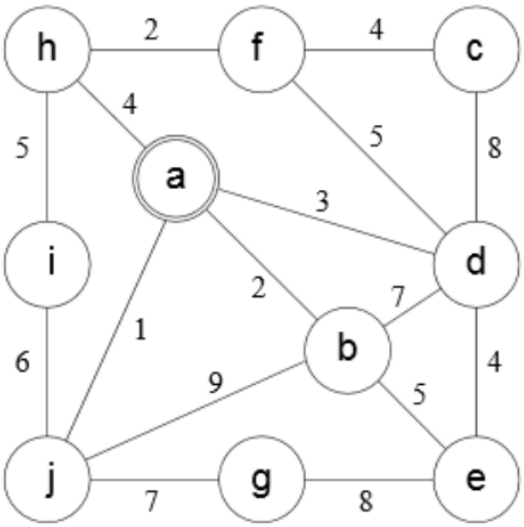
James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Homework 5

Due Date: Sunday, May 13, 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.



ANSWER:

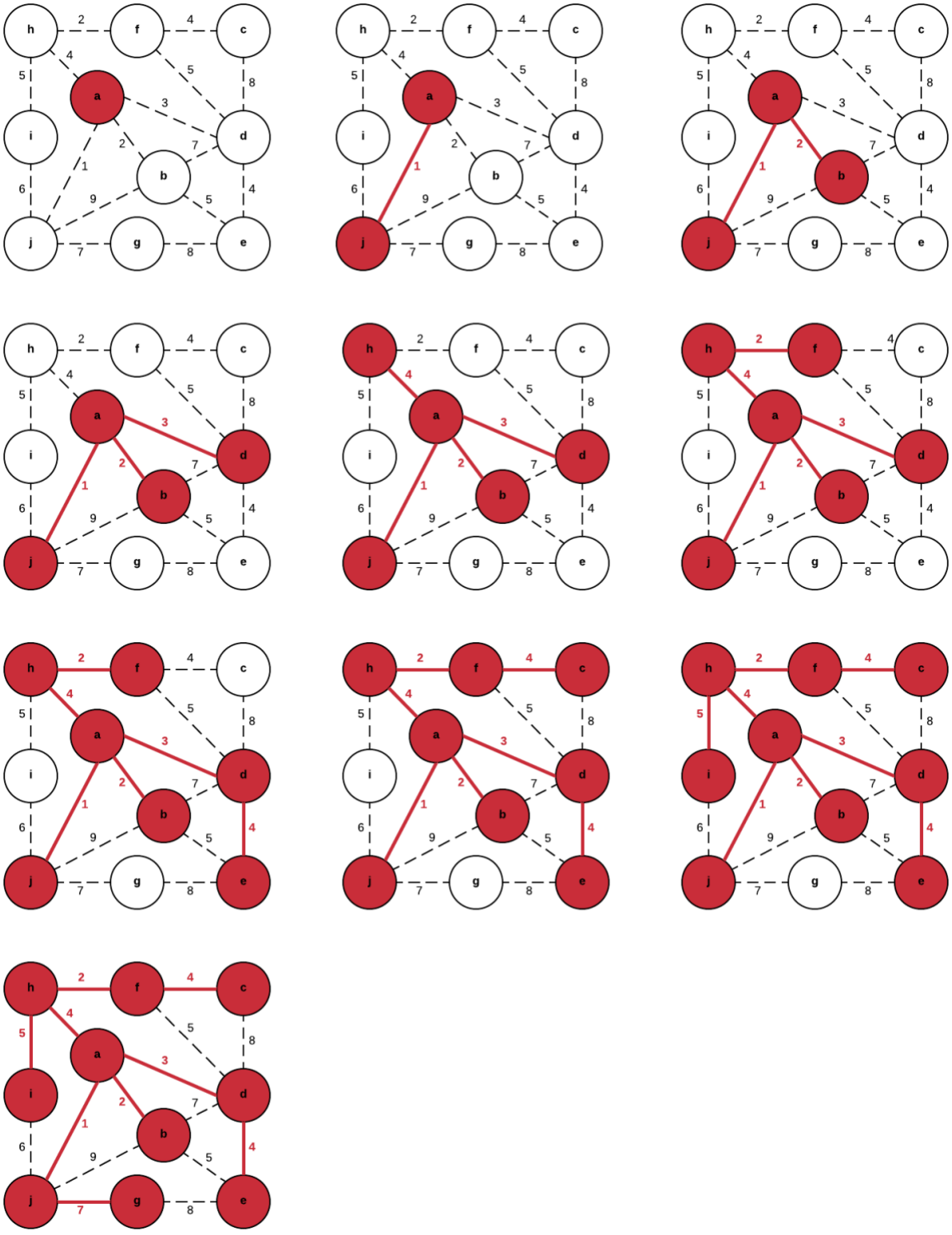
The weight of the minimum spanning tree (MST), starting at vertex '*a*', is 32 with the selection of the following edge weight values.

$$\mathbf{1 + 2 + 3 + 4 + 2 + 4 + 4 + 5 + 7 = 32}$$

A step-by-step demonstration of Prim's algorithm is presented below on the next page. All graphs in this assignment were created in Lucidchart (https://www.lucidchart.com).

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Prim's Algorithm:



2. *(6 points)* Consider an undirected graph $G = (V, E)$ with nonnegative edge weights $w(u, v) \geq 0$. 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.

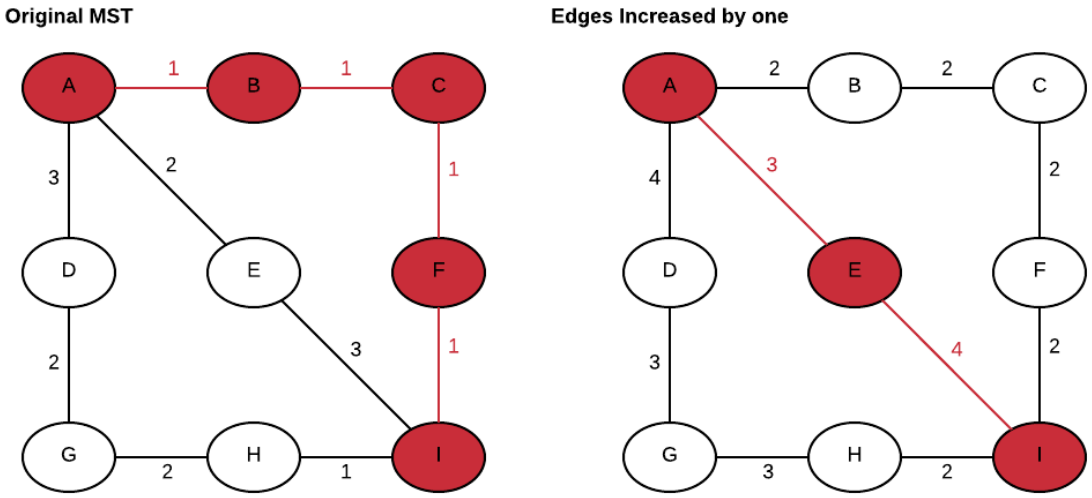
ANSWER:

Incrementing each edge weight by 1 will not change the minimum spanning tree. The relationship between vertices would remain equivalent as all edge weights are uniformly incremented by a value of one. This can be proven through Kruskal's algorithm. The algorithm selects the most minimal edge weight that does not invoke cycling. Kruskal's algorithm would still build the same Minimum Spanning Tree before and after the edge weights are increased because the smallest edge will not deviate and constantly remain as the smallest edge.

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

ANSWER:

Incrementing each edge weight by 1 could, however, potentially change the shortest path. Consider an original shortest path comprised of many edges with marginal weights and another longer path with minimal edges but with higher associated weights. Incrementing the weights by one for all edges would cause the original shortest path to grow faster than the previous heavier path and could potentially change the shortest path. Below is an example of when the explained scenario might occur.



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.

ANSWER:
The Breadth-First Search (BFS) Algorithm could be implemented in this circumstance to solve the bottleneck-path problem but would need to be modified to arbitrarily ignore edges with weight values that are less than that of weight W. The algorithm would operate exactly like the standard implementation of the breadth-first search but whenever it encounters and edge value that is less than the specified weight of W, the edge will not be considered and will not be appended to the queue.

b) What is the running time of your algorithm?

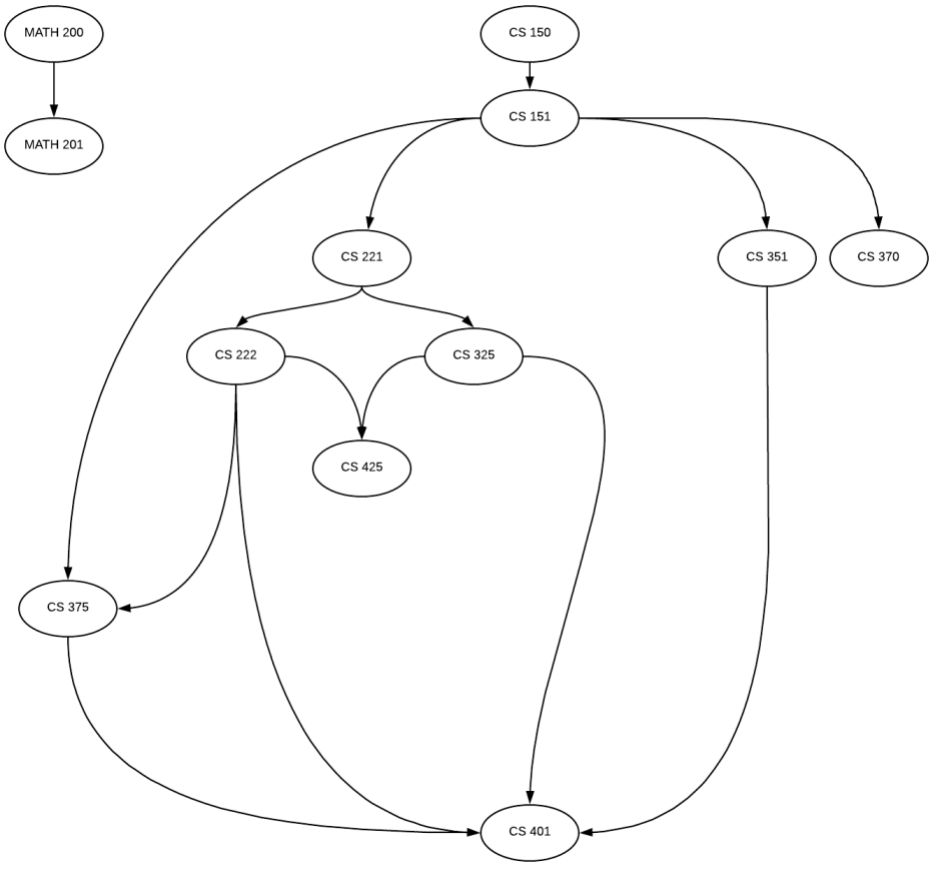
ANSWER:
The running time for the modified Breadth-First Search algorithm mentioned above would be the same as the standard Breadth-First search algorithm, $O(V + E)$. Checking the weight value in comparison to W is the only difference from the traditional Breadth-First search but since this is done in constant time, its impact is negligible and can be ignored from our running time complexity equation.

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.

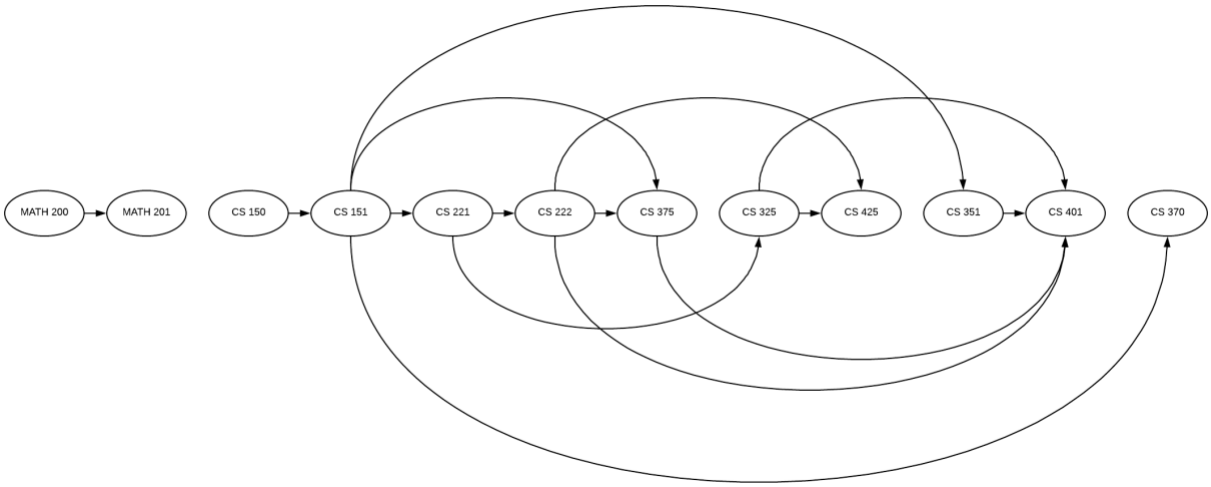
ANSWER:



b) Give a topological sort of the graph.

ANSWER:
I gave a topologically sorted version of the graph below.

MATH 200 → MATH 201 → CS 150 → CS 151 → CS 221 → CS 222 → CS 375 → CS 325 → CS 425 → CS 351 → CS 401 → CS 370



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.

ANSWER:
If we are allowed to take multiple courses at once as long as there is no prerequisite conflict the following ordering would allow all of the classes to be taken in as few as 6 terms.

TERM 01: MATH 200, CS 150
TERM 02: MATH 201, CS 151
TERM 03: CS 221, CS 351, CS 370
TERM 04: CS 222, CS 325
TERM 05: CS 375, CS 425
TERM 06: CS 401

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

ANSWER:
The longest path in the Directed Acyclic Graph (DAG) has a value of 5 edges and follows the path listed below. If we look at the DAG graph presented in part A we can see that it takes the most required prerequisite classes before a student is eligible to take the CS 401 course (5 total).

Longest Path:
CS 150 → CS 151 → CS 221 → CS 222 → CS 375 → CS 401

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.

ANSWER:

An efficient algorithm that determines whether it is possible to designate some of the wrestlers as Babyfaces and heels would be the Breadth-First Search algorithm. The objective would be to create a graph where all the wrestlers are represented by individual vertices and the edges between then represent a potential rivalry. The BFS algorithm will process through the graph until all the vertices have been reached (visited). While traversing the vertices, any wrestler with an even distance in edges will be assigned as a Babyface and any wrestler with an odd distance in edges will be assigned as a heel. Finally, each edge will be checked to confirm that they are a relationship between a babyface a heel wrestling pair.

PSEUDOCODE:

```
BFS(start):
babyfaces = []
distance = 0
explored = []
queue = [start]
babyfaces += start
visited += start
while queue is not empty:
        distance = distance + 1
        node = queue[0]
        explored += node
        neighbors = graph[node]
        for each neighbor in neighbors:
                if not visited before:
                        add to queue += neighbor
                        add to visited += neighbor
                        ff the edge distance is even:
                                add to babyfaces += neighbor
                        else:
                                add to heels += heels
```

b) What is the running time of your algorithm?

ANSWER:

The running time complexity of this algorithm is the time it takes to assign each wrestler as a babyface or a heel O(n) plus the time it takes to check each rivalry or edge O(r). This gives a total running time complexity of $O(n + r)$ for the wrestling algorithm.

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

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

Submit a copy of your files including a README file that explains how to compile and run your code in a ZIP file to TEACH.

ANSWER:

My code for the wrestling program has been submitted to TEACH. I've attached a screen shot of my BFS function below.

Just a heads up, it's pretty flakey and I'm not 100% confident in its implementation. I can get the graph built and traverse the vertices and separate wrestlers into the category of babyfaces and heels but I'm running into difficulties when determining if rivalries are possible. I'm going to submit what I have now before the deadline and hopefully at least get some partial credit. I'll try to work on it again tomorrow (Monday, May 14, 2018) and see if I can get it working before the end of the day. If not, we'll just have to take this one on the chin and hope for the best for the remainder of the term.

```
"""
/*****************************************************************************
* Description: Rivalries class
* Initializes a default dictionary and creates a graph of connections
* between each wrestler and their matches.  Uses a Breadth-First
* Search (BFS) to traverse through each node in the graph.
*****************************************************************************/
"""

class Rivalries:
    def __init__(self):
        self.graph = defaultdict(list)
        self.babyfaces = []
        self.heels = []
        self.visited = []

    def add_connection(self, wrestler1, wrestler2):
        self.graph[wrestler1].append(wrestler2)

    def BFS(self, start):
        distance = 0
        self.babyfaces.append(start)
        explored = []
        queue = [start]
        self.visited.append(start)
        while queue:
            distance += 1
            node = queue.pop(0)
            explored.append(node)
            neighbours = self.graph[node]
            for neighbour in neighbours:
                if neighbour not in self.visited:
                    queue.append(neighbour)
                    self.visited.append(neighbour)
                    if distance % 2 == 0:
                        self.babyfaces.append(neighbour)
                    else:
                        self.heels.append(neighbour)
```