CS 3000: Algorithms & Data — Spring 2024

Homework 5
Due Tuesday, April 2 at 11:59pm via Gradescope

Name: Ryan Tietjen
Collaborators:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- This homework is due Tuesday, April 2 at 11:59pm via Gradescope. No late assignments will be accepted. Make sure to submit something before the deadline.
- Solutions must be typeset. If you need to draw any diagrams, you may draw them by hand as long as they are embedded in the PDF. We recommend that you use LaTeX, in which case it would be best to use the source file for this assignment to get started.
- We encourage you to work with your classmates on the homework problems, but also urge you to attempt all of the problems by yourself first. If you do collaborate, you must write all solutions by yourself, in your own words. Do not submit anything you cannot explain. Please list all your collaborators in your solution for each problem by filling in the `yourcollaborators` command.
- Finding solutions to homework problems on the web, or by asking students not enrolled in the class is strictly forbidden.

Problem 1. *(17 points) Wood Chuckin'*

Punxsutawney Phil and his family are getting ready to hold their annual 6 More Weeks of Winter party this February. In order to prepare, there are a set of j jobs that need to be completed before the party can occur. Some of the jobs depends on other jobs being completed beforehand (e.g. they can't construct the stage until they prepare the wood and shovel out the stage area). Formally, the jobs are labeled from 1 to j and the jobs that need to be completed before job i are stored in a list B_i. It is possible that B_i is empty for some values of i.

Because Phil has such a large family, there is no limit on how many jobs can be done simultaneously (up to j jobs can be worked on at the same time). Assuming each job takes 1 day to complete, how many days are necessary to complete all of the jobs in time for the party? Determine which jobs should be carried out on each day in order to complete all of the jobs by this time.

You may assume that there is no circular dependence among jobs.

(a) (**2 points**) Describe precisely what your algorithm is given as input and what it needs to output. [1]

Solution:

Input:

-The list (of lists) $B[1...j]$ where $B[i]$ is the list of jobs that must be completed before the beavers can start working on job i

Output:

-A list Day where $Day[i]$ is the list of jobs that must be carried out on day i and the length of Day is the number of days necessary to complete all of the jobs.

[1] **Check:** Make sure you have this right, before you move on to designing the algorithm.

(b) (**8 points**) Write your algorithm in pseudocode. Additionally, provide a few sentences describing your approach.

Solution:

Our goal is to figure out which jobs have 0 uncompleted prerequisites on a given day, and then figure out which jobs can then be completed the next day (and repeat this process). Hence, we will perform a topological sort on a DAG that represents the jobs and their prerequisties. We begin by determining how many jobs a given job is waiting on. If a job isn't waiting on any other jobs, then we push it (enqueue it). For each job completed on a given day, we remove that job from being a prerequisite of the jobs that depend on it. We will repeat this process until every job has been completed.

Algorithm 1: Wood Chuckin': Determines which jobs should be carried out on each day in order to complete all jobs in the minimum number of days

```
Function $MinimumDays(B[1...j])$:
    Initalize $numDependencies[1...j]$        ▷ Number of prerequisite jobs for job at index $i$
                                              ▷ Initalize $numDependencies[1...j]$ to 0
    For int $i$ from 1 to $j$:
        For each int $job$ in $B[i]$:
            $numDependencies$[job]++
    Initalize Queue $jobs$
    For int $i$ from 1 to $j$:
        If $numDependencies$[i] == 0 :
            $jobs$.push($i$)
    Initalize empty list $Day$                ▷ Array that will be returned
    int $d = 0$                               ▷ Day counter
    int $jobsToday = jobs$.size()             ▷ Number of jobs to pop (dequeue) on day $d$
    While $jobs$ is not empty
        Initalize empty list $tempList$
        For int $i$ from 1 to $jobsToday$:
            $currentJob = jobs$.pop()
            $tempList$.append($currentJob$)
            For each int $k$ in $B[currentJob]$:
                $numDependencies[k]--$
                If $numDependencies[k] == 0$ :
                    $jobs$.push($k$)
        $Day[d] = tempList$
        $d++$
        $jobsToday = jobs$.size()
    Return $Day$
```

(c) (**4 points**) Justify the correctness of your algorithm. Your justification does not need to be long or formal, just convincing.

Solution:

This algorithm essentially performs a topologicial sort on a DAG, thus ensuring that all jobs are completed the day after their last prerequisite is completed. By keeping track of the number of remaining prerequisites for each job, we ensure that each job is only added to the queue when it has 0 remaining prerequisites. Since jobs are popped from the queue and pushed day by day, the algorithm follows the dependency constraints, ensuring that a job cannot be completed before all of its prerequisites are. Since there are no cycles in a DAG, there cannot be any job dependencies that will cause an infinite loop. Hence, this algorithm successfully determines which jobs are to be completed on the given days.

Furthermore, this algorithm also correctly determines the total (minimum) number of days required for preparation. We begin by completing each job with no prerequisites on day 0. Then, each day, all jobs whose prerequisites have been completed the day before are pushed onto the queue. This ensures that there is no idle time where a given job could have been started. In other words, every job is being completed as soon as possible, thus guaranteeing that each job is completed in a minimum number of days.

(d) (**3 points**) Analyze the running time of your algorithm in terms of j and L, where L is the total length of the j lists. To obtain full credit for part (b), your algorithm needs to run in time $O(j + L)$.

Solution:

j represents the number of vertices in our DAG. Each vertex represents a job.

L represents the total number of edges in our DAG. Each edge represents a prerequisite.

-Initalizing $numDependencies$ takes $O(j + L)$ since we visit every edge job in every vertex $B[1...j]$

-Initalizing the queue $jobs$ takes $O(j)$ since the for loop runs from 1 to j.

-The main while loop takes $O(j + L)$. This is because every vertex (every job) is enqueued exactly once. Also, every edge from each vertex is considered exactly once when decrementing $numDependencies$.

Since the dominant term is $O(j + L)$, this algorithm runs in $O(j + E)$

Problem 2. *(16 points) My Team is Better Than Your Team*

An amusing website uses transitivity to demonstrate one sports team is better than another. Essentially the idea is that team A is better by transitivity than team Z if team A defeated a team which defeated a team which defeated a team, and so on, which defeated Z.

For example, Northeastern women's basketball is better by transitivity than last year's national champions South Carolina, because Northeastern beat William & Mary, who beat George Mason, who beat Florida, who beat Kentucky, who beat South Carolina. South Carolina is also better by transitivity than Northeastern, because South Carolina beat Elon, who beat Northeastern. As another example, Northeastern's men's basketball is better by transitivity than last year's national champions Kansas Unviersity, because Northeastern beat BU, who beat UMass Lowell, who beat Dayton, who beat Kansas.

You are given the n american universities which have a women's basketball program, and the results of all games last season (note that there are no draws in basketball, each game has a winner and a loser). We say that two teams are in the same class if they are better by transitivity than each other (so Northeastern and South Carolina are in the same class in the example above, and there are many other teams in this class as well). A class C is top-tier if every team in C is better by transitivity than each of the other $n-1$ teams. It is possible that a top-tier class only contains one team. If there is a top-tier class, we would like to return all teams in this class.

(a) (**2 points**) Give a set of results with 6 teams where there is a top-tier class of size 2. Identify the teams in the top-tier class.

Solution:

Suppose we have the 6 teams: A, B, C, D, E, F. Then A and B would be top-tier if:

-A beats B, C, D, E, F

-B beats A, C, D, E, F

-C beats D

-D beats E

-E beats F

-F beats C

(b) (**2 points**) Give a set of 6 results with 6 teams where there is no top-tier class of size 2.

Solution:

Suppose we have the 6 teams: A, B, C, D, E, F. Then there would be no top-tier class if:

-A beats B

-B beats C

-C beats A

-D beats E

-E beats F

-F beats D

(c) Design and analyze a polynomial-time algorithm to determine if there is a top-tier class. If there is, return the teams in that class. If there is no top-tier class, then your algorithm must indicate so.

(i) (**2 points**) Describe precisely what your algorithm is given as input and what it needs to output.[2]

Solution:

Inputs:

-V[1...n] - The list of n american universities with a women's basketball program.

-E - A list of pairs (u, v) that indicate the results of all games last season (i.e u beats v)

Output:

Returns the list of top-tier teams in the class, or outputs an error if there is no top-tier class.

(ii) (**5 points**) Describe your algorithm. You may invoke or modify any of the graph traversal algorithms studied in class. You do not need to use pseudocode.

Solution:

1. Create a directed graph G where each vertex reprsents a team and each edge $(u, v) \in E$ represents team u beating team v

2. Use the SCC algorithm (lecture 16/17) to partition G into strongly connected components.

3. Construct a component graph G' where each vertex represents a SCC in G, and there is an edge (A, B) from SCC A to SCC B if there is any edge from any vertex in A to any vertex in B. When constructing G', maintain a list $L[1...n']$ (or a hashmap) of how many edges there are pointing to each vertex.

4. Search L for any vertices that have 0 edges pointing to it. If there is exactly one vertex with 0 edges pointing to it, then return the list of all vertices in that SCC. Otherwise, return an empty list/an error stating that there is no top-tier class.

(iii) (**3 points**) Justify the correctness of your algorithm. Your justification does not need to be long or formal, just convincing.

Solution:

In essence, the different classes would be the strongly connected components (SCCs) since, by definition, every vertex in a SCC is reachable by every other vertex in that SCC. Hence, every team in a SCC is better (by transitivity) than every other team in that SCC.

By definition, the component graph G' is a DAG because all cycles were already accounted for when running the SCC algorithm.

Hence, if there is exactly one vertex in G' with no incoming edges, then this vertex must be able to reach every other vertex in G'. So, every team in that SCC must be better than all other teams (and thus be a part of the top-tier class). Furthermore,

[2]**Check:** Make sure you have this right, before you move on to designing the algorithm.

every vertex in G' with an incoming edge (let's denote these vertices as λ) must be out-classed (i.e. at least one team in the SCC pointing to λ has beat, but has not been beaten by, any team in λ).

If there is more than one vertex in G' with no incoming edges, then there are multiple teams that are not better than each other by transitivity, so there would be no top-tier class.

There cannot be no vertices with no incoming edges in a DAG. If there is, then there must not be any vertices in the DAG (not possible - that would be like saying there are no teams in the league).

(iv) (**2 points**) Analyze the running time of your algorithm in terms of the number of teams n and number of games (results) r. Your algorithm should run in $O(n+r)$ time to receive full credit.

Solution:

1. Creating G would take $O(n+r)$ since adding each edge and vertex would take $O(n+r)$

2. Performing the SCC algorithm takes $\theta(n+r)$

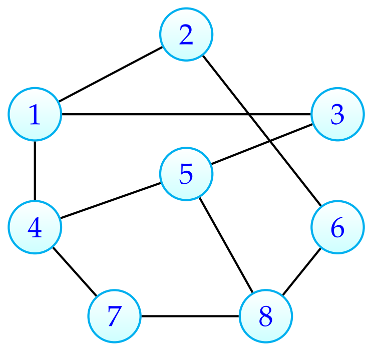
3. Constructing the component graph G' and maintaing L can be completed in $O(n+r)$ since we have to check all edges in E once.

4. Performing a linear search on L takes $O(n)$ time.

Hence, the dominant term is $O(n+r)$. So, the algorithm will run in $O(n+r)$ time as required.

Problem 3. *Graph Representations and Exploration (8 points)*

This problem is about the following graph. **Tip:** Use the LaTeXfor rendering the graph as a starting point if your solutions involve drawing a graph.



(a) (**2 points**) Draw the adjacency matrix of this graph. **Tip:** you can use bmatrix if using LaTeX.

Solution:

$$\begin{bmatrix} A & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 6 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 7 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 8 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(b) (**2 points**) Draw the adjacency list of this graph.

Solution:

$A[1] = \{2, 3, 4\}$

$A[2] = \{1, 6\}$

$A[3] = \{1, 5\}$

$A[4] = \{1, 5, 7\}$

$A[5] = \{3, 4, 8\}$

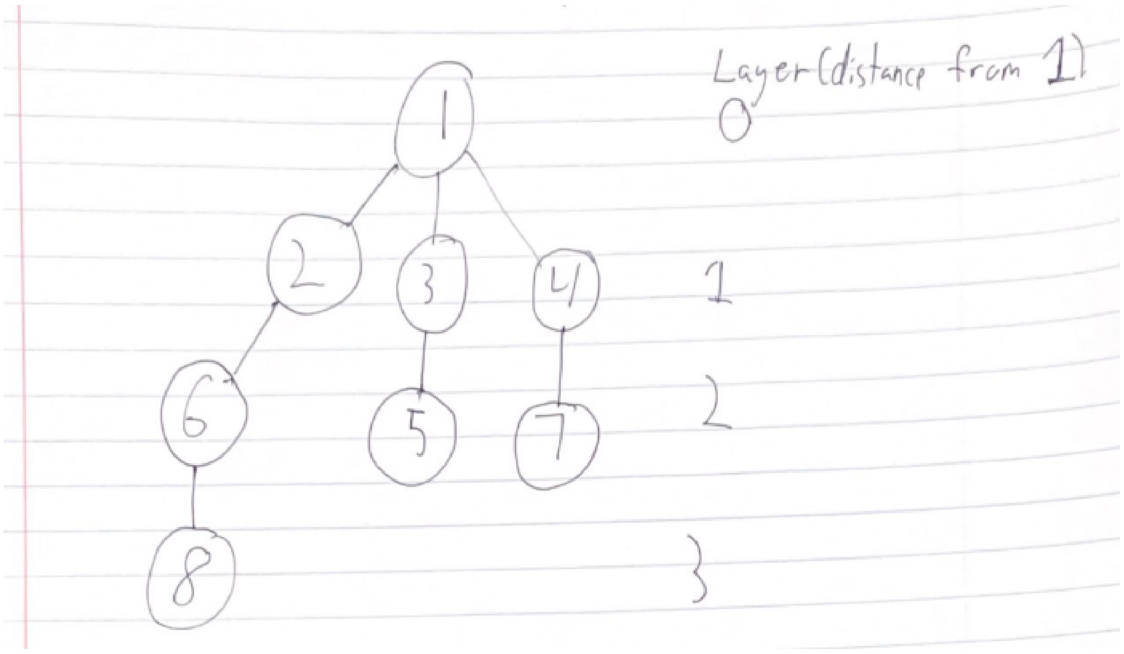
$A[6] = \{2, 8\}$

$A[7] = \{4, 8\}$

$A[8] = \{5, 6, 7\}$

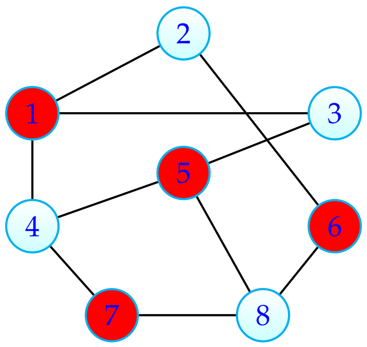
(c) (**2 points**) BFS this graph starting from the node 1. Always choose the lowest-numbered node next. Draw the BFS tree and label each level with its distance from 1.

Solution:



(d) (**2 points**) Is this graph 2-colorable? Either 2-color it or explain why you can't.

Solution: The graph is indeed 2-colorable:



Problem 4. *Graph Properties (9 points)*

Consider an undirected graph $G = (V, E)$. The degree of a vertex v is the number of edges adjacent to v—that is, the number of edges of the form $(v, u) \in E$. Recall the standard notational convention that $n = |V|$ and $m = |E|$.

(a) (**6 points**) Prove by induction that the sum of the degrees of the vertices is equal to $2m$.

Solution:

Inductive Hypothesis: Let $H(m)$ be the statement: The sum of the degrees of the vertices of a graph G with m edges is $2m$ for all $m \geq 0$.

Base Case: Let m = 0. Since there are no edges in G, then every vertex in G has a degree zero. Hence, the sum of the degrees of the vertices is $2m = 0$ as required. Thus, $H(0)$ holds.

Inductive Step: We will show that for every $m \geq 1$, $H(m) \implies H(m+1)$. Assume $H(m)$ holds.

Consider a graph G' that is obtained from adding one edge $e = (a, b)$ where a, b are any two vertices in V. Hence, G' has $m+1$ edges.

If $a = b$, the degree of a increases by 2 because the edge (a, a) contributes twice to the degree of a. So, the degree of G' is equal to the degree of G + 2, which is $(2m+2) = 2(m+1)$

Otherwise, $a \neq b$. So, the degree of a increases by 1 since it now has an additional edge. Similarly, the degree of b increases by 1 since it now has an additional edge. So, the degree of G' is equal to the degree of G + 1 + 1, which is $2(m+1)$.

In either case the sum of the degrees of the vertices in G' is $2m+1$ as required, thus establishing $H(m+1)$. Therefore, $H(m)$ is true for all $m \geq 0$ by induction.

(b) (**3 points**) Prove that there are an even number of vertices whose degree is odd.

Solution:

We will do a proof by contradiction:

Suppose a graph G has an odd number of vertices whose degree is odd. Let E denote the set of vertices of an even degree in G, and let O denote the set of vertices of an odd degree in G. Then:

-The sum of the degrees of vertices in E is even because adding any even numbers together results in an even number.

-The sum of the degrees of vertices in O is odd because adding any odd numbers together results in an odd number.

Hence, the sum of the degrees in G is an even number + an odd number, which must be an odd number.

However, in part a, we concluded that the sum of the degrees of the vertices of graph G with m edges is $2m$, but $2m$ must be an even number. This is a direct contradiction to the above statement (the sum of the degrees in G is odd.)

So, our assumption that G has an odd number of vertices whose degree is odd is false. Hence, G must have an even number of vertices whose degree is odd.