Quiz 2 Solutions

Problem 1. True or false [24 points] (8 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (Your explanation is worth more than your choice of true or false.)

(a) T F Instead of using counting sort to sort digits in the radix sort algorithm, we can use any valid sorting algorithm and radix sort will still sort correctly.

Solution: False. Need stable sort.

(b) T F The depth of a breadth-first search tree on an undirected graph $G = (V, E)$ from an arbitrary vertex $v \in V$ is the diameter of the graph G. (The ***diameter*** d of a graph is the smallest d such that every pair of vertices s and t have $\delta(s, t) \leq d$.)

Solution: False. An arbitrary vertex could lay closer to the 'center' of the graph, hence the BFS depth will be underestimating the diameter. For example, in graph $G = (V, E) = (\{a, v, b\}, \{(a, v), (v, b)\})$, a BFS from v will have depth 1 but the graph has diameter 2.

(c) T F Every directed acyclic graph has exactly one topological ordering.

Solution: False. Some priority constraints may be unspecified, and multiple orderings may be possible for a given DAG. For example a graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (a, c)\})$ has valid topological orderings $[a, b, c]$ or $[a, c, b]$. As another example, $G = (V, E) = (\{a, b\}, \{\})$ has valid topological orderings $[a, b]$ or $[b, c]$.

(d) T F Given a graph $G = (V, E)$ with positive edge weights, the Bellman-Ford algorithm and Dijkstra's algorithm can produce different shortest-path trees despite always producing the same shortest-path weights.

Solution: True. Both algorithms are guaranteed to produce the same shortest-path weight, but if there are multiple shortest paths, Dijkstra's will choose the shortest path according to the greedy strategy, and Bellman-Ford will choose the shortest path depending on the order of relaxations, and the two shortest path trees may be different.

(e) T F Dijkstra's algorithm may not terminate if the graph contains negative-weight edges.

Solution: False. It always terminates after $|E|$ relaxations and $|V|+|E|$ priority queue operations, but may produce incorrect results.

(f) T F Consider a weighted directed graph $G = (V, E, w)$ and let X be a shortest s-t path for $s, t \in V$. If we double the weight of every edge in the graph, setting $w'(e) = 2w(e)$ for each $e \in E$, then X will still be a shortest s-t path in (V, E, w').

Solution: True. Any linear transformation of all weights maintains all relative path lengths, and thus shortest paths will continue to be shortest paths, and more generally all paths will have the same relative ordering. One simple way of thinking about this is unit conversions between kilometers and miles.

(g) T F If a depth-first search on a directed graph $G = (V, E)$ produces exactly one back edge, then it is possible to choose an edge $e \in E$ such that the graph $G' = (V, E - \{e\})$ is acyclic.

Solution: True. Removing the back edge will result in a graph with no back edges, and thus a graph with no cycles (as every graph with at least one cycle has at least one back edge). Notice that a graph can have two cycles but a single back edge, thus removing *some* edge that disrupts that cycle is insufficient, you have to remove specifically the back edge. For example, in graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (b, c), (a, c), (c, a)\})$, there are two cycles $[a, b, c, a]$ and $[a, c, a]$, but only one back edge (c, a). Removing edge (b, c) disrupts one of the cycles that gave rise to the back edge ($[a, b, c, a]$), but another cycle remains, $[a, c, a]$.

(h) T F If a directed graph G is cyclic but can be made acyclic by removing one edge, then a depth-first search in G will encounter exactly one back edge.

Solution: False. You can have multiple back edges, yet it can be possible to remove one edge that destroys all cycles. For example, in graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (b, c), (b, a), (c, a)\})$, there are two cycles ($[a, b, a]$ and $[a, b, c, a]$) and a DFS from a in G returns two back edges ((b, a) and (c, a)), but a single removal of edge (a, b) can disrupt both cycles, making the resulting graph acyclic.

Problem 2. Short answer [24 points] (6 parts)

(a) What is the running time of RADIX-SORT on an array of n integers in the range $0, 1, \ldots, n^5 - 1$ when using base-10 representation? What is the running time when using a base-n representation?

Solution: Using base 10, each integer has $d = \log n^5 = 5 \log n$ digits. Each COUNTING-SORT call takes $\Theta(n + 10) = \Theta(n)$ time, so the running time of RADIX-SORT is $\Theta(nd) = \Theta(n \log n)$.

Using base n, each integer has $d = \log_n n^5 = 5$ digits, so the running time of RADIX-SORT is $\Theta(5n) = \Theta(n)$.

2 points were awarded for correct answers on each part. A point was deducted if no attempt to simplify running times were made (e.g. if running time for base-10 representation was left as $\Theta(\log_{10} n^5(n + 10))$

Common mistakes included substituting n^5 as the base instead of 10 or n. This led to $\Theta(n^5)$ and $\Theta(n^6)$ runtimes

(b) What is the running time of depth-first search, as a function of $|V|$ and $|E|$, if the input graph is represented by an adjacency matrix instead of an adjacency list?

Solution: DFS visits each vertex once and as it visits each vertex, we need to find all of its neighbors to figure out where to search next. Finding all its neighbors in an adjacency matrix requires $O(V)$ time, so overall the running time will be $O(V^2)$.

2 points were docked for answers that didn't give the tightest runtime bound, for example $O(V^2 + E)$. While technically correct, it was a key point to realize that DFS using an adjacency matrix doesn't depend on the number of edges in the graph.

(c) Consider the directed graph where vertices are reachable tic-tac-toe board positions and edges represent valid moves. What are the in-degree and out-degree of the following vertex? (It is O's turn.)

X	O	X
	O	
	X	

Solution: There were three possible vertices that could have pointed into this board position:

	O	X
	O	
	X	

X	O	
	O	
	X	

X	O	X
	O	

And there are four possible vertices that could have pointed out from this board position as O has four spaces to move to. In-degree is 3, out-degree is 4.

(d) If we modify the RELAX portion of the Bellman-Ford algorithm so that it updates $d[v]$ and $\pi[v]$ if $d[v] \geq d[u] + w(u, v)$ (instead of doing so only if $d[v]$ is strictly greater than $d[u] + w(u, v)$), does the resulting algorithm still produce correct shortest-path weights and a correct shortest-path tree? Justify your answer.

Solution: No. There exists a zero-weight cycle, then it is possible that relaxing an edge will mess up parent pointers so that it is impossible to recreate a path back to the source node. The easiest example is if we had a vertex v that had a zero-weight edge pointing back to itself. If we relax that edge, v's parent pointer will point back to itself. When we try to recreate a path from some vertex back to the source, if we go through v, we will be stuck there. The shortest-path tree is broken. 1 point was awarded for mentioning that shortest-path weights do get preserved, but also thinking the tree was correct..

(e) If you take 6.851, you'll learn about a priority queue data structure that supports EXTRACT-MIN and DECREASE-KEY on integers in $\{0, 1, \ldots, u-1\}$ in $O(\lg\lg u)$ time per operation. What is the resulting running time of Dijkstra's algorithm on a weighted direct graph $G = (V, E, w)$ with edge weights in $\{0, 1, \ldots, W-1\}$?

Solution: The range of integers that this priority queue data structure (van Emde Boas priority queue) will be from 0 to $|V|(W-1)$. This is because the longest possible path will go through $|V|$ edges of weight $W-1$. Almost the entire class substituted the wrong value for u. Dijkstra's will call EXTRACT-MIN $O(V)$ times and DECREASE-KEY $O(E)$ times. In total, the runtime of Dijkstra's using this new priority queue is $O((|V| + |E|) \lg\lg(|V|w))$

2 points were deducted for substituted the wrong u, but understanding how to use the priority queue's runtimes to get Dijkstra's runtime

(f) Consider a weighted, directed acyclic graph $G = (V, E, w)$ in which edges that leave the source vertex s may have negative weights and all other edge weights are non-negative. Does Dijkstra's algorithm correctly compute the shortest-path weight $\delta(s, t)$ from s to every vertex t in this graph? Justify your answer.

Solution: Yes, For the correctness of Dijkstra, it is sufficient to show that $d[v] = \delta(s, v)$ for every $v \in V$ when v is added to s. Given the shortest $s \leadsto v$ path and given that vertex u precedes v on that path, we need to verify that u is in S. If $u = s$, then certainly u is in S. For all other vertices, we have defined v to be the vertex not in S that is closest to s. Since $d[v] = d[u] + w(u, v)$ and $w(u, v) > 0$ for all edges except possibly those leaving the source, u must be in S since it is closer to s than v.

It was not sufficient to state that this works because there are no negative weight cycles. Negative weight edges in DAGs can break Dijkstra's in general, so more justification was needed on why in this case Dijkstra's works.

Problem 3. You are the computer [12 points] (4 parts)

(a) What is the result of relaxing the following edges?

(i)
$$\begin{tikzcd} 4 \arrow[r, "3"] & 7 \end{tikzcd}$$

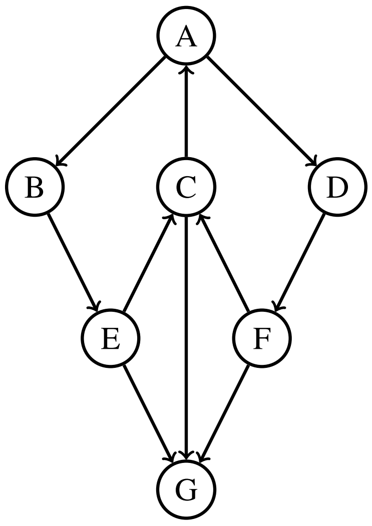
(ii)
$$\begin{tikzcd} 12 \arrow[r, "4"] & 17 \end{tikzcd}$$

(iii)
$$\begin{tikzcd} 9 \arrow[r, "5"] & 11 \end{tikzcd}$$

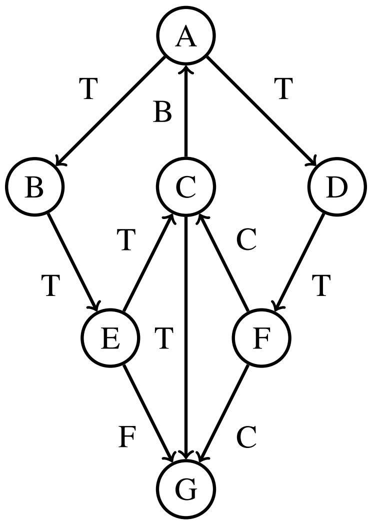
Solution: 7, 16, 11 for the new value of the right vertex

one point for each edge

(b) Perform a depth-first search on the following graph starting at A. Label every edge in the graph with T if it's a tree edge, B if it's a back edge, F if it's a forward edge, and C if it's a cross edge. To ensure that your solution will be exactly the same as the staff solution, assume that whenever faced with a decision of which node to pick from a set of nodes, pick the node whose label occurs earliest in the alphabet.



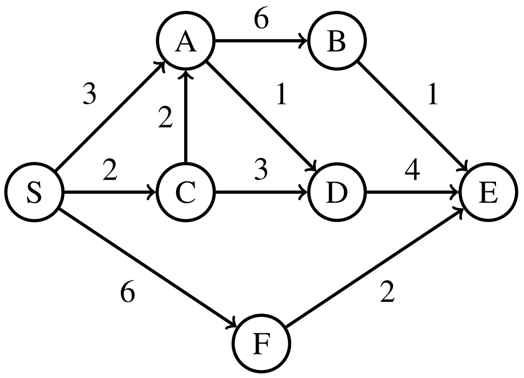
Solution:



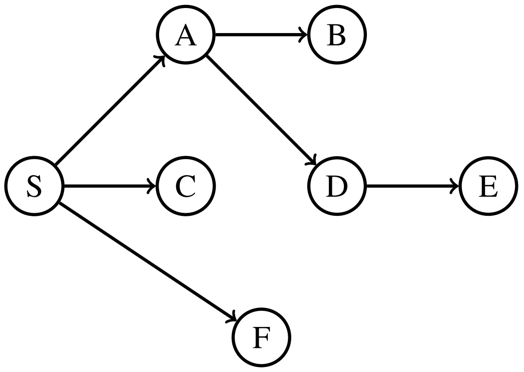
-1 for minor errors in labeling, sometimes resulting from incorrect choice of which node to visit

-2 for major errors in labeling

(c) Run Dijkstra's algorithm on the following directed graph, starting at vertex S. What is the order in which vertices get removed from the priority queue? What is the resulting shortest-path tree?



Solution: Dijkstra will visit the vertices in the following order: S, C, A, D, F, E, B. Dijkstra will relax the edge from D to E before the edge from F to E, since D is closer to S than F is. As a result, the parent of each node is:



-1 for minor errors, such as a missing vertex in the ordering of vertices removed from the priority queue or an incorrect edge in the shortest-path tree

-2 for major errors, such as not providing the shortest-path tree (some people mistakenly provided the shortest-path length in the tree)

(d) Radix sort the following list of integers in base 10 (smallest at top, largest at bottom). Show the resulting order after each run of counting sort.

Original list	First sort	Second sort	Third sort
583			
625			
682			
243			
745			
522			

Solution:

Original list	First sort	Second sort	Third sort
583	682	522	243
625	522	625	522
682	583	243	583
243	243	745	625
745	625	682	682
522	745	583	745

-1 for minor errors

-2 for major errors, such as not using a stable sort for the individual sorts.

Problem 4. Burgers would be great right about now [10 points]

Suppose that you want to get from vertex s to vertex t in an unweighted graph $G = (V, E)$, but you would like to stop by vertex u if it is possible to do so without increasing the length of your path by more than a factor of α.

Describe an efficient algorithm that would determine an optimal s-t path given your preference for stopping at u along the way if doing so is not prohibitively costly. (It should either return the shortest path from s to t or the shortest path from s to t containing u, depending on the situation.)

If it helps, imagine that there are burgers at u.

Solution: Since the graph is unweighted, one can use BFS for the shortest paths computation. We run BFS twice, once from s and once from u. The shortest path from s to t containing u is composed of the shortest path from s to u and the shortest path from u to t. We can now compare the length of this path to the length of the shortest path from s to t, and choose the one to return based on their lengths. The total running time is $O(V + E)$.

An alternative is to use Dijkstra algorithm. This works, but the algorithm becomes slower. Same for Bellman-Ford.

Problem 5. How I met your midterm [10 points]

Ted and Marshall are taking a roadtrip from Somerville to Vancouver (that's in Canada). Because it's a 52-hour drive, Ted and Marshall decide to switch off driving at each rest stop they visit; however, because Ted has a better sense of direction than Marshall, he should be driving both when they depart and when they arrive (to navigate the city streets).

Given a route map represented as a weighted undirected graph $G = (V, E, w)$ with positive edge weights, where vertices represent rest stops and edges represent routes between rest stops, devise an efficient algorithm to find a route (if possible) of minimum distance between Somerville and Vancouver such that Ted and Marshall alternate edges and Ted drives the first and last edge.

Solution: There are two correct and efficient ways to solve this problem. The first solution makes a new graph G'. For every vertex u in G, there are two vertices u_M and u_T in G': these represent reaching the rest stop u when Marshall (for u_M) or Ted (for u_T) will drive next. For every edge (u, v) in G, there are two edges in G': (u_M, v_T) and (u_T, v_M). Both of these edges have the same weight as the original.

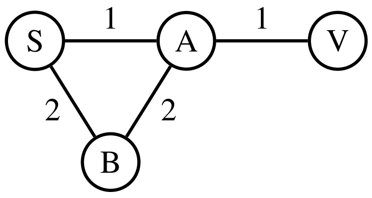
We run Dijkstra's algorithm on this new graph to find the shortest path from Somerville$_T$ to Vancouver$_M$ (since Ted drives *to* Vancouver, Marshall would drive next if they continued). This guarantees that we find a path where Ted and Marshall alternate, and Ted drives the first and last segment. Constructing this graph takes linear time, and running Dijkstra's algorithm on it takes $O(V \log V + E)$ time with a Fibonacci heap (it's just a constant factor worse than running Dijkstra on the original graph).

The second correct solution is equivalent to the first, but instead of modifying the graph, we modify Dijkstra's algorithm. Dijkstra's algorithm will store two minimum distances and two parent pointers for each vertex u: the minimum distance d_{odd} using an odd number of edges, and the minimum distance d_{even} using an even number of edges, along with their parent pointers π_{odd} and π_{even}. (These correspond to the minimum distance and parent pointers for u_T and u_M in the previous solution). In addition, we put each vertex in the priority queue twice: once with d_{odd} as its key, and once with d_{even} as its key (this corresponds to putting both u_T and u_M in the priority queue in the previous solution).

When we relax edges in the modified version of Dijkstra, we check whether $v.d_{\text{odd}} > u.d_{\text{even}} + w(u, v)$, and vice versa. One important detail is that we need to initialize Somerville.d_{odd} to ∞, not 0. This algorithm has the same running time as the previous one.

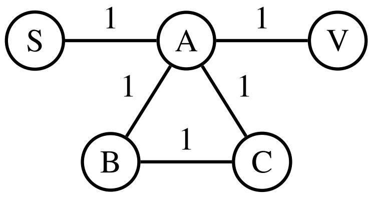
A correct but less efficient algorithm used Dijkstra, but modified it to traverse two edges at a time on every step except the first, to guarantee a path with an odd number of edges was found. Many students incorrectly claimed this had the same running time as Dijkstra's algorithm; however, computing all the paths of length 2 (this is the ***square*** of the graph G) actually takes a total of $O(VE)$ time, whether you compute it beforehand or compute it for each vertex when you remove it from Dijkstra's priority queue. This solution got 5 points.

The most common mistake on the problem was to augment Dijkstra (or Bellman-Ford) by keeping track of either the shortest path's edge count for each vertex, or the parity of the number edges in the shortest path. This is insufficient to guarantee that the shortest odd-edge-length path is found, and this solution got 2 points. Here is an example of a graph where the algorithm fails: once the odd-edge-count path of weight 1 to A is found, Dijkstra will ignore the even-edge-count path of weight 4 to A since it has greater weight. As a result, the odd-edge-count path to V will be missed entirely.



Another common mistake was to use Dijkstra, and if the path Dijkstra found had an even number of edges, to attempt to add or remove edges until a path with an odd number of edges was obtained. In general, there is no guarantee the shortest path with an odd number of edges is at all related to the shortest path with an even number of edges.

Some algorithms ran Dijkstra, and if Dijkstra found a path with an even number of edges, removed some edge or edges from the graph and re-ran Dijkstra. This algorithm fails on the following graph, where the shortest path with an odd number of edges uses *all* the edges and vertices (note that we visit A twice; the first time, Ted drives to A, and the second time, Marshall drives to A):



One last common mistake was to attempt to use Breadth-First Search to label each vertex as an odd or even number of edges from Somerville (or sometimes to label them as odd, even, or both). This does not help: the smallest-weight path with an odd number of edges could go through any particular vertex after having traversed an odd or even number of edges, and BFS will not correctly predict which. These solutions got 0 points.

Algorithms which returned the correct answer but with exponential running time got *at most* 2 points.

Problem 6. Just reverse the polarity already [10 points]

Professor Kirk has managed to get himself lost in his brand new starship. Furthermore, while boldly going places and meeting strange new, oddly humanoid aliens, his starship's engines have developed a strange problem: he can only make "transwarp jump" to solar systems at distance exactly 5 from his location.

Given a starmap represented as an unweighted undirected graph $G = (V, E)$, where vertices represent glorious new solar systems to explore and edges represent transwarp routes, devise an efficient algorithm to find a route (if possible) of minimum distance from Kirk's current location s to the location t representing Earth, that Kirk's ship will be able to follow. Please hurry—Professor Kirk doesn't want to miss his hot stardate!

Solution: In general, the idea is to convert $G = (V, E)$ into a graph $G' = (V, E')$ representing all the feasible transwarp jumps that Kirk can make, i.e., with an edge (u, v) if there is a simple path in G from u to v of length exactly 5. (Note that this definition is the notion of "distance" in the problem, as clarified during the quiz.) Once we have such a graph G', we simply run breadth-first search on G' from s, and follow parent pointers from t to recover the shortest route (if there one) for Kirk to follow. The running time of this breadth-first search is $O(V + E') = O(V^2)$.

The central question is how to compute G'. The best solutions we know run in $O(V^3)$ time. There are two ways to achieve this bound.

The first $O(V^3)$ algorithm is a modification of breadth-first search from every vertex. For each vertex v, we construct the set $N_1(v)$ of all neighbors of v. Next we construct the set $N_2(v)$ of all vertices reachable by a path of length 2 from v, by taking the union of $N_1(u)$ for each $u \in N_1(v)$. Then we construct $N_3(v)$, $N_4(v)$, and $N_5(v)$ similarly. Constructing $N_1(v)$ costs $O(V)$ time, while constructing $N_k(v)$ for $k \in \{2, 3, 4, 5\}$ costs $O(v^2)$ time. The key here is that we remove duplicate vertices in each set $N_k(v)$, so each such set has size $O(V)$. Because we do this for every vertex v, we spend $O(v^3)$ time total. Finally we set $E' = \{(v, w) : w \in N_5(v)\}$.

The second $O(V^3)$ algorithm is to compute the adjacency matrix A, and compute $A^5 = A \cdot A \cdot A \cdot A \cdot A$. Each matrix multiplication costs $O(V^3)$ time, for a total of $O(V^3)$ time. The nonzero entries in this matrix correspond to the edges in G'.

A simpler $O(V^4E) = O(V^6)$ algorithm is much simpler: for each vertex v_0, for each neighbor v_1 of v_0, for each neighbor v_2 of v_1, for each neighbor v_3 of v_2, for each neighbor v_4 of v_3, for each neighbor v_5 of v_4, add the edge (v_0, v_5) to E'. The first two loops cost a factor of $O(E)$, and the next four loops cost a factor of $O(V^4)$.

The grading scheme was as follows. A $O(V^3)$ solution was worth a nominal value of 10/10. A $O(V^4)$ solution was worth a nominal value of 9/10. Very few students achieved such solutions. A $O(V^4E)$ or $O(V^6)$ solution was worth a nominal value of 7/10. These nominal values were adjusted according to clarity, quality, and/or errors. The idea of computing a graph like G' was worth a nominal value of 4/10. Executing this idea by performing a depth-5 BFS or DFS was worth a nominal value of 5/10. Simply running BFS and focusing on the layers divisible by 5 was worth a nominal value of 1/10.

Problem 7. The price is close enough [10 points]

As part of a new game show, contestants take turns making several integer guesses between 0 and $1{,}000{,}000$ (inclusive). In scoring each round, the show's host, Professor Piotrik Kellmaine, needs to know which two guesses were closest to each other. Provide an asymptotically time-optimal algorithm that answers this question, argue that it is correct, and give and explain its time complexity.

Solution: **algorithm:** We first radix sort the input n guesses using base 10. Then we go through the list of n sorted integers and compare adjacent ones to see which pair of adjacent integers are closest to each other, and output that pair of gusses.

correctness: We see that the closest pair of guesses have to be adjacent to each other in the sorted list because or else there will be some integers in between them making them not the closest pair. In other word, say a and b are the closest pair, then if $a < c < b$, we see $b - c$ and $c - a$ are less than $b - a$, therefore contradicting the fact that a and b are the closest pair of guesses.

runtime: radix sort takes $O(7 \cdot (n + 10))$ time if we take base 10 which is $O(n)$ time. Going through the list once and compare all adjacent pairs only take $O(n)$ time because there are only $n - 1$ pairs we have to compare and find the minimum absolute difference between them. So the total running time is $O(n)$.

grading: one point is taken off for not mentioning the base of radix sort or using counting sort instead because 1000000 is a relatively big constant factor in the case of this problem. three points are taken off if students did not present an explanation on how to iterate through the sorted list to find the min difference.
three points are given if the student gave the naive algorithm which takes all $\frac{(n)(n-1)}{2}$ pairs and find the minimum. four points are given for students who choose a sorting algorithm that takes $O(n \lg n)$ time.

Problem 8. Call it the scenic route [10 points]

In the ***longest path problem***, we're given a weighted directed graph $G = (V, E, w)$, a source $s \in V$, and we're asked to find the longest simple path from s to every vertex in G. For a general graph, it's not known whether there exists a polynomial-time algorithm to solve this problem. If we restrict G to be acyclic, however, this problem can be solved in polynomial time. Give an efficient algorithm for finding the longest paths from s in a weighted directed acyclic graph G, give its runtime, and explain why your solution doesn't work when G is not acyclic.

Solution: Algorithm: We map this to a single-source shortest paths problem by creating a new graph, G', with the same vertices and edges as G but whose weight function is the negative of the original.

Now we can run the single-source shortest paths algorithm for DAG's shown in class to find the shortest paths in $\Theta(V + E)$. This algorithm relaxes the edges of G' in topologically sorted order only once. See class notes to see why this works for finding the shortest paths in a DAG.

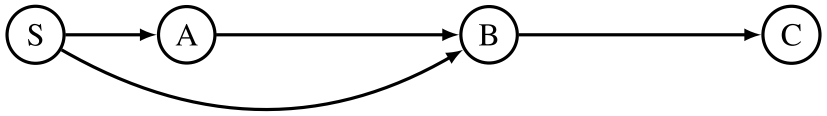
We could alternatively use Bellman Ford here, although that will give us a suboptimal runtime.

Runtime: Creating G' is a simple process and only requires $\Theta(V + E)$ time to iterate over all the edges and vertices to create our new graph and weight function. Topologically sorting the edges takes $\Theta(V + E)$ since topological sort is done using a modification of the DFS algorithm. Finally, relaxing all the edges once only takes $\Theta(E)$ time. Thus, the runtime is $\Theta(V + E)$ overall.

Why G needs to be acyclic: We can't use the single-source shortest paths algorithm for DAG's if G is not acyclic since we would no longer have a DAG. But, even assuming we used Bellman Ford, which can handle negative weight cycles, we would still be in trouble. The main reason we need G to be acyclic is that we're looking for the longest **simple** path (i.e. no vertex is repeated). Negative weight cycles in G' wouldn't be much of an issue if we didn't restrict our paths to be simple. Simply detecting them and marking those paths as infinite is easy to do in asymptotically the same time as Bellman Ford.

Grading: Overall, 6 points were given to the algorithm and 4 points were given to the explanation of why we need G to be acyclic.

Many students tried BFS or DFS, both of only work on unweighted graphs. Another large portion of students attempted to use Dijkstra or a modified Dijkstra algorithm. The problem with a Dijkstra approach is that Dijkstra for shortest paths relies on the fact that once we visit a vertex, we wont ever find a shorter path to that vertex. This requires non neg. edge weights, however. So, in the longest path problem, we would need all neg. edge weights in order to be able to have a similar invariant. But, there's nothing in the problem statement that allows us to make this assumption. If you're still skeptical, here's a counterexample to the Dijkstra approaches seen:



Dijkstra will not find the longest path from S to C.

Otherwise, the majority of students did not give an adequate explanation to why the graph needs to be acyclic. We were mainly looking for some comment about the problem specifying **simple** paths, since that's at the heart of the matter. Any solution that didn't mention this, or touch on something close to this, lost credit.

Finally, while Bellman Ford is a correct approach, it is not optimal. Only a point was docked for this.

Problem 9. Rated M for "Masochistic" [10 points]

You're playing the hit new platform video game, *Mega Meat Man*, and are having trouble getting through Level 6006. You've decided to model the level as a directed graph, where each vertex represents a platform you can reach, and each edge represents a jump you can try to make. After extensive experimentation, you've labeled each edge with the probability (a number in $[0, 1]$) that you can successfully make the jump. Unfortunately, if you fail to make any jump, you instantly die, and have to start over. Describe an efficient algorithm to find a path from the start platform s to the goal platform t that maximizes the probability of a successful traversal.

Solution: Intuitively, we'd like to maximize $\prod_i p_i$ over the vertices in the path we take from s to t. Since the $\log$ function is monotonic, this is the same as maximizing $\sum_i \log p_i$, which is the same as minimizing $-\sum_i \log p_i = \sum_i (-\log p_i)$. Therefore, if we create an auxiliary graph in which the weight w of each edge is replaced with $-\log w$, the shortest s-t path is the maximum probability path. Additionally, $p_i \in [0, 1] \implies \log p_i \leq 0 \implies -\log p_i \geq 0$, so all edge weights are nonnegative. The negative logarithm goes to ∞ as p_i goes to 0, which suits us just fine; if we never make the jump, we should never try that path. Because all of our edge weights are nonnegative, we can use Dijkstra to find the shortest s-t path; the creation of the auxiliary graph takes $O(|E|)$ time, so the total time complexity of the algorithm is $O(|E| + |V| \lg |V|)$ (if we use a Fibonnaci heap).

Solutions that did not provide the time complexity of the algorithm or that used a less efficient algorithm, solutions that did not convincingly argue that edge weights were nonnegative (while using Dijkstra), and solutions that did not convincingly argue that the shortest s-t path in the auxiliary graph corresponded to the solution to the original problem lost points. Some solutions tried to modify Dijkstra instead of reducing the problem given to a standard shortest-path problem. This itself did not cause any loss of credit, but often led to mistakes (subtle or otherwise) or a lack of clarity.