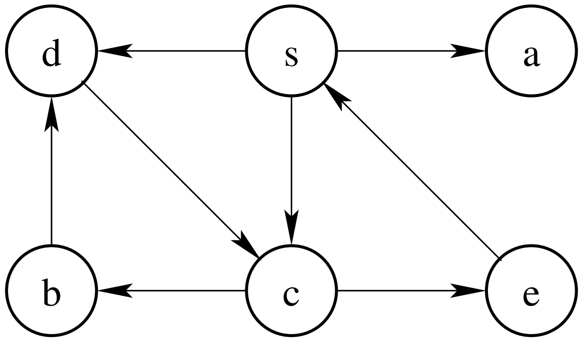
Quiz 2 Solutions

Problem 1. BFS/DFS [10 points] (2 parts)

Give the visited node order for each type of graph search, starting with s, given the following adjacency lists and accompanying figure:

$adj(s) = [a, c, d]$,
$adj(a) = [\,]$,
$adj(c) = [e, b]$,
$adj(b) = [d]$,
$adj(d) = [c]$,
$adj(e) = [s]$.



(a) Breadth First Search

Solution: s a c d e b

(b) Depth First Search

Solution: s a c e b d

Problem 2. Miscellaneous True/False [30 points] (10 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (No credit if no explanation given.)

(a) T F While running DFS on a directed graph, if from vertex u we visit a finished vertex v, then the edge (u, v) is a cross-edge.
Explain:

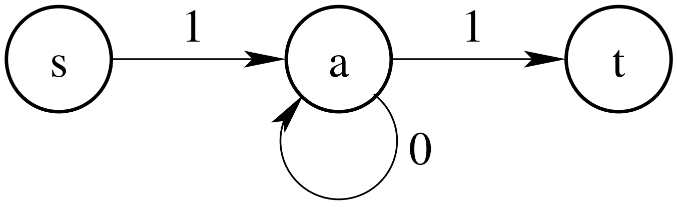
Solution: False. The edge could be either a cross-edge or a forward edge.

Note: The edge *cannot* be a back edge—a back edge goes to a vertex that has started, but not finished (a "gray vertex", in CLRS terms).

(b) T F Changing the RELAX function to update if $d[v] \geq d[u] + w(u, v)$ (instead of strictly greater than) may produce different shortest paths, but will not affect the correctness of the Bellman-Ford outputs d and π.
Explain:

Solution: False. The parent pointers may not lead back to the source node if a zero-length cycle exists.

In the example below, relaxing the (s, a) edge will set $d[a] = 1$ and $\pi[a] = 1$. Then, relaxing the (a, a) edge will set $d[a] = 1$ and $\pi[a] = a$. Following the π pointers from t will no longer give a path to s, so the algorithm is incorrect.

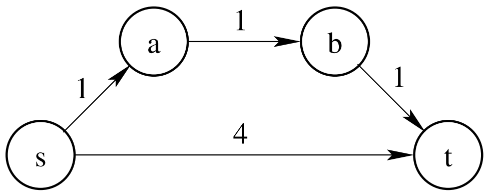


(c) T F The running time of Radix sort is effectively independent of whether the input is already sorted.
Explain:

Solution: True. All input orderings give the worst-case running time; the running time doesn't depend on the order of the inputs in any significant way.

(d) T F Let P be a shortest path from some vertex s to some other vertex t in a directed graph. If the weight of each edge in the graph is increased by one, P will still be a shortest path from s to t.
Explain:

Solution: False. See the counterexample below.



(e) T F If a weighted directed graph G is known to have no shortest paths longer than k edges, then it suffices to run Bellman-Ford for only k passes in order to solve the single-source shortest paths problem on G.

Solution: True. The ith iteration finds shortest paths in G of i or fewer edges, by the path relaxation property (see p. 587 in CLRS).

(f) T F If a topological sort exists for the vertices in a directed graph, then a DFS on the graph will produce no back edges.
Explain:

Solution: True. Both parts of the statement hold if and only if the graph is acyclic.

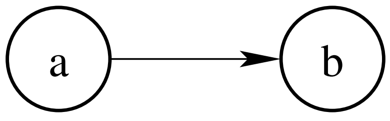
(g) T F Dynamic programming is more closely related to BFS than it is to DFS.
Explain:

Solution: False. DFS is more closely related. The top-down approach to dynamic programming is effectively performing DFS on the subproblem dependence graph. The bottom-up approach means solving subproblems in the order of a reverse topological sort, which is also related to DFS.

(h) T F A depth-first search of a directed graph always produces the same number of tree edges (i.e. independent of the order in which the vertices are provided and independent of the order of the adjacency lists).
Explain:

Solution: False. The DFS forest may contain different numbers of trees (and tree edges) depending on the starting vertex and upon the order in which vertices are searched.

Consider the example below. If the DFS starts at a, then it will visit b next, and (a, b) will become a tree edge. But if the DFS visits b first, then a and b become separate trees in the DFS forest, and (a, b) becomes a cross edge.



(i) T F Suppose we do a DFS on a directed graph G. If we remove all of the back edges found, the resulting graph is now acyclic.
Explain:

Solution: **True.** If DFS finds no back edges, then the graph is acyclic. Removing any back edges found doesn't change the traversal order of DFS, so running DFS again on the modified graph would produce no back edges.

(j) T F Both DFS and BFS require $\Omega(V)$ storage for their operation. (That is, for working storage, above and beyond the storage needed to represent the input.)
Explain:

Solution: **True.** Each needs to keep track of the vertices that have already been visited.

Problem 3. Miscellaneous Short Answer [20 points] (3 parts)

(a) Suppose you want to get from s to t on weighted graph G with nonnegative edge weights, but you would like to stop by u if it isn't too inconvenient. (Here too inconvenient means that it increases the length of your travel by more than 10%.)

Describe an efficient algorithm that would determine an optimal s to t path given your preference for stopping at u along the way if not too inconvenient. It should either return the shortest path from s to t, or the shortest path from s to t containing u.

Solution: Run Dijkstra's algorithm 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. Compare the length of this path with the length of the path from s to t, and choose the one to return based on their lengths.

(b) Explain how the "rod-cutting" problem described in class can still be solved by dynamic programming, even if cuts now cost \$1 each. (In class, we assumed cuts were free.) Here is the pseudocode for the original solution, where the cuts were free:

```
r = [0] * (n + 1)
for k in range(1, n + 1):
    ans = p[k]
    for i range(1, k):
        ans = max(ans, p[i] + r[k - i])
    r[k] = ans
```

(It suffices to explain how to express r_n, the maximum revenue achievable for a rod of size n, in terms of $r_1, r_2, ..., r_{n-1}$ and the prices p_i that the market will pay for a piece of length i, for $i = 1, 2, ...n$.)

Solution: Modify the fifth line to account for the loss in revenue due to the cost of the cut:

```
ans = max(ans, p[i] + r[k - i] - 1)
```

(c) Suppose that you implement Dijkstra's algorithm using a priority queue algorithm that requires $O(V)$ time to initialize, worst-case $f(V, E)$ time for each EXTRACT-MIN operation and worst-case $g(V, E)$ time for each DECREASE-KEY operation. How much (worst-case) time does it take to run Dijkstra's algorithm on an input graph $G = (V, E)$?.

Solution: $O(V \cdot f(V, E) + E \cdot g(V, E))$

Problem 4. A Series of Tubes [20 points] (2 parts)

Consider a network of computers represented by a directed graph G. Each vertex $v \in V$ represents a computer, and each edge $(u, v) \in E$ represents a network link from u to v.

(a) Let each edge (u, v) in G have a weight $w(u, v) \in (0, 1]$, representing the probability that a packet going from u to v is successfully delivered. Give an efficient algorithm to find the path along which a packet has the *highest* probability of reaching its destination (i.e., the path for which the product of the edge weights is *maximized*).

Hint: Transform the weights and use a shortest path algorithm.

Solution: Construct a new graph G', letting $w'(u, v) = -\log w(u, v)$. Shortest paths in G' correspond to highest probability paths in G, because $\sum_i w'(v_i, v_{i+1}) = -\sum_i \log w(v_i, v_{i+1}) = -\log\left(\prod_i w(v_i, v_{i+1})\right)$, which is minimized when the product term is maximized.

The new weights are non-negative, so run Dijkstra's algorithm on G' to find the highest probability paths.

(b) Now, instead of weighted edges, consider weighted vertices. In other words, network links never drop packets, but nodes might. Edges are not weighted, but each vertex v has a weight $w(v) \in (0, 1]$, representing the probability that an incoming packet is *not* dropped by that node. Give an efficient algorithm to find the path along which a packet has the highest probability of reaching its destination (i.e., the path for which the product of the vertex weights is maximized).

Hint: Transform the graph and use the algorithm from part (a).

Solution: Construct a new edge-weighted graph G' by assigning each edge the weight $w'(u, v) = w(v)$, then run the algorithm from part (a).

Note 1: It's reasonable to assume here that the weight of the source vertex, s, is 1, because the originating computer won't drop the packet. Even without this assumption, the problem isn't much harder—adjust the weight of each edge originating at s by multiplying it by $w(s)$.

Note 2: Using the weight function $w'(u, v) = w(u) \cdot w(v)$ is problematic because the weight of each vertex, except for the source and the target, is included in two edges of the path, so those vertices are double-counted.

Problem 5. Fast Flyer [20 points] (3 parts)

You are given a list of all scheduled daily flights in the US, giving departure airports, departure times, destination airports, and arrival times. We want an algorithm to compute travel times between airports, including waiting times between connections. Assume that if one flight arrives at time t and another departs at time $t' \geq t$, travelers can make the connection. Further assume that at a given airport, no two events (arrivals or departures) occur at the same time, and that there are at most 100 events at any airport during a given day. All times are given in GMT; don't worry about time zones.

Construct a weighted graph so that given a departure airport, a departure time T, and a destination airport, we can efficiently determine the earliest time T' that a traveler can be guaranteed (according to the schedules!) of arriving at her destination *on that day* (ignore overnight flights and overnight airport stays).

(a) What do vertices represent? What do edges in your graph represent, and what is the weight of an edge?

Solution: There are many ways to represent the problem with a weighted graph. Here is one good solution. We create a vertex for every flight departure and every flight arrival. That is, each vertex corresponds to a particular time (arrival or departure of a flight) at a particular airport. We also create one vertex for the departure time at the departure airport.

We create an edge for every flight. The weight of these edges is the flight time. We also create an edge from every event (departure or arrival) at an airport to the next event at the same airport. The weight of these is the time between the events.

(b) Give an upper bound on the number of edges and vertices in your graph if there are n airports in the US and m daily flights. Justify your bound.

Solution: For the construction given above, the number of vertices is clearly $2m+1$. The number of edges is a bit harder to bound. One bound is $1 + 99n + m$, since there are at most 100 events at an airport (so 99 intervals between them, plus one more from the start time) and m flight edges. There are other bounds that we can derive for the number of edges here. This part was graded based on the construction in the first part.

(c) What algorithm would you use to compute the shortest travel times, and what is its running time in terms of the number of vertices, V, and the number of edges, E?

Solution: For the construction above, we need to sort the events at each airport to create the edges. This takes $O(V \lg V)$, but this is likely to be a loose bound since we don't solve one big sorting problem but many small ones. If we use Radix sort, the cost drops to $O(V)$.

Once the graph is constructed, we can find the shortest travel times using BFS or DFS; since our vertices are associated with a time and an airport, all the reachable vertices from the starting vertex represent times in which a traveler can be at an airport. By finding the earliest reachable vertex per airport we find the solution.

We can also use Dijkstra (and we can use Fibonacci heaps), or a shortest-paths algorithm for DAGs (this construction is a graph, and is acyclic because travelers always move forward in time).

Problem 6. Articulation Points [20 points] (3 parts)

We define an articulation point as a vertex that when removed causes a connected graph to become disconnected. For this problem, we will try to find the articulation points in an *undirected* graph G.

(a) How can we efficiently check whether or not a graph is disconnected? (Hint: think of a recent problem set question)

Solution: Run BFS or DFS on a node in G (since G is undirected, any node will do). If the size of the set of visited nodes does not equal $|V|$, then the graph is disconnected.

(b) Describe an algorithm that uses a brute force approach to find all the articulation points in G in $O(V(V + E))$ time.

Solution: For each vertex v in G, remove v and all the edges connected to v. Run the algorithm from *part a*. If the graph G is disconnected, then v is an articulation point. Add v and all its edges back into G and repeat.

An observer comments that the polynomial time algorithm is rather slow and suggests to use a linear time DFS approach instead. Let's explore this idea some more.

(c) Suppose we run DFS on graph G. Consider the types of edges that can exist in a DFS tree produced from an *undirected* graph, recalling that cross edges can't happen in the DFS of an undirected graph. Argue that a non-root, non-leaf vertex u is an articulation point *if and only if* there exists a subtree rooted at a child of u that has no back edges to a proper ancestor of u.

Solution: The first thing to note is that the claim is an *if and only if* statement so we need to prove it in *both* directions.

Claim 1: If u is an articulation point, then there exist a subtree rooted at a child of u that has no back edges to a proper ancestor of u.

We can prove this by contradiction. Assume all the subtrees have back edges to a proper ancestor of u. If we remove u, every subtree of u can still reach a proper ancestor of u through the back edge. Since the graph is still connected, u cannot be an articulation point, so our initial assumption is violated. Therefore, there must be a subtree with no back edge to a proper ancestor of u.

Claim 2: If there exist a subtree rooted at a child of u that has no back edges to a proper ancestor of u, then u is an articulation point.

The important fact to realize is that a DFS on an undirected graph only produces tree edges and back edges. Because there are no cross edges, then there is no path from one subtree rooted at a child of u to another subtree rooted at a child of u, nor a path to a vertex that is *neither* an ancestor or descendent of u. Since the assumption says there are no back edges either, then there must only be tree edges. If we remove u, then the subtree that has no back edges will be disconnected since we are effectively removing the connecting tree edge. Hence, u is an articulation point.

Since Claim 1 and Claim 2 are true, then the original claim is true.

(FYI: The above idea can be extended to yield an $O(V + E)$ algorithm to find all articulation points, but we don't have time for all of that on this quiz!)