DSC 40B - Quiz 08

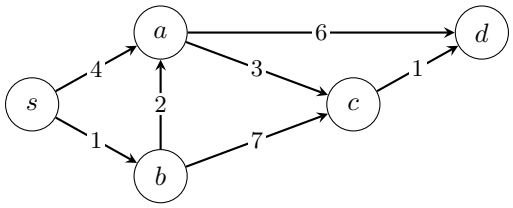
Name: Student ID:

About the quizzes:

- Quizzes in DSC 40B are *optional* and graded pass/fail.
- A score of 70% or higher earns a "pass" and 1.5 credits toward your final grade.
- If you don't pass, no credits are earned, but it doesn't hurt your grade.
- You have 30 minutes to complete the quiz.
- Unfortunately, we can't answer clarifying questions during the quiz. If you think a question has a bug or is unclear, please let us know in a private post on Campuswire after the quiz, and we'll take it into account when grading.

Problem 1.

Suppose the Bellman-Ford algorithm is run on the weighted, directed graph below using node s as the source.



Suppose `graph.edges` produces the edges in this order:

$$(a, c), (c, d), (s, a), (b, a), (s, b), (a, d), (b, c).$$

After **one** iteration of the outer loop, what is the estimated distance to node d?

◯ 7 ◯ 8 ● 10 ◯ ∞

Solution: 10.

At the start, s's estimate is 0 and every other estimate is ∞. In the first iteration, the edges are updated in order:

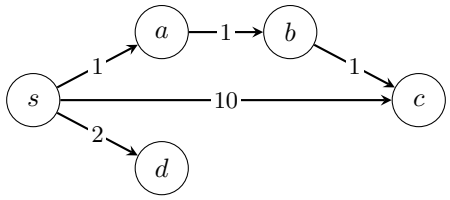
- (a, c) and (c, d) change nothing, since a and c are still at ∞;
- (s, a) sets a to $0 + 4 = 4$;
- (b, a) changes nothing, since b is still at ∞;
- (s, b) sets b to $0 + 1 = 1$;
- (a, d) sets d to $4 + 6 = 10$;
- (b, c) sets c to $1 + 7 = 8$.

So after one iteration, d's estimate is 10. (Its actual distance is 7, along s, b, a, c, d, which later iterations find.)

This problem was similar to Problem #306 in the Quiz 08 practice problems.

Problem 2.

Suppose the Bellman-Ford algorithm is run on the weighted, directed graph below using node s as the source.



After 2 iterations of the outer loop, which one of these nodes is **not** guaranteed to have a correct estimated distance (no matter the order in which `graph.edges` produces the edges)?

◯ a ◯ b ● c ◯ d

Solution: c.

After α iterations, every node whose *actual* shortest path from s has at most α edges is guaranteed to be estimated correctly. The shortest paths to a and d have one edge, and the shortest path to b (s, a, b)

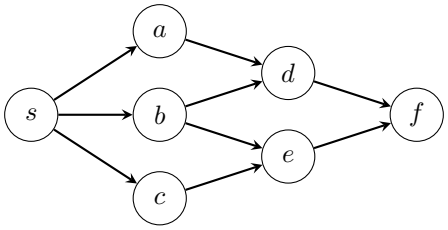
has two, so these are correct after 2 iterations. The shortest path to c is s, a, b, c, of length 3, which has three edges, so c isn't guaranteed to be correct, even though there is an edge from s to c.

For example, suppose the edges are produced in the order (b, c), (a, b), (s, a), (s, c), (s, d). In the first iteration, (b, c) and (a, b) change nothing, then a becomes 1, c becomes 10, and d becomes 2. In the second, (b, c) changes nothing, since b is still at ∞, and then b becomes 2. So after 2 iterations, c's estimate is still 10, not 3.

This problem was similar to Problem #307 in the Quiz 08 practice problems.

Problem 3.

Suppose Bellman-Ford **with early stopping** is run on the directed graph below using node s as the source. The edge weights are not shown: they are unknown to us (but known to the algorithm), and the edges may be produced by `graph.edges` in any order.



a) What is the **fewest** number of iterations of the outer `for`-loop that can possibly be performed?

○ 1 ● 2 ○ 3 ○ 4 ○ 6

Solution: 2.

The number of iterations depends on how lucky or unlucky we are with the order in which `graph.edges` produces the edges. The luckiest order is one in which the edges of each shortest path come in the order they appear along the path: then a single iteration finds every shortest path.

For example, suppose the edges are produced in the order (s, a), (s, b), (s, c), (a, d), (b, d), (b, e), (c, e), (d, f), (e, f). Every edge out of a node comes after all of the edges into it, so each node's estimate is already correct by the time the edges out of it are updated. So whatever the weights are, every estimate is correct after the first iteration.

But the algorithm only stops after an iteration in which no estimate changes, and the first iteration always changes something (a's estimate drops from ∞, for example). So a second iteration runs. It changes nothing, and the algorithm stops. No order can do better than 2.

b) What is the **greatest** number of iterations of the outer `for`-loop that can possibly be performed?

○ 2 ○ 3 ● 4 ○ 5 ○ 6

Solution: 4.

After α iterations, every node whose shortest path from s has at most α edges is guaranteed to be estimated correctly. This graph has no cycles, and its longest path from s has 3 edges (e.g., s, a, d, f), so every node's shortest path has at most 3 edges, and every estimate is correct after 3 iterations. The fourth iteration therefore changes nothing, and the algorithm stops after at most 4 iterations.

4 can happen: if every weight is 1 and the edges are produced in the reverse of the order in part

(a), each iteration fixes only the nodes one edge further from s (a, b, c, then d, e, then f). The third iteration still changes f's estimate, so a fourth one runs.

The outer loop is programmed to run $|V| - 1 = 6$ iterations, but early stopping ends it sooner here.

This problem was similar to Problem #302 in the Quiz 08 practice problems.

Problem 4.

A **directed** graph has nodes $1, 2, \ldots, n$, and there is an edge from node u to node v whenever $u < v$. Suppose the Bellman-Ford algorithm *without* early stopping is run on this graph, using node 1 as the source. What is its time complexity, as a function of n?

◯ $\Theta(n)$ ◯ $\Theta(n^2)$ ● $\Theta(n^3)$ ◯ $\Theta(n^4)$

Solution: $\Theta(n^3)$.

Bellman-Ford without early stopping takes $\Theta(|V| \cdot |E|)$ time: its outer loop runs $|V| - 1$ times, and each iteration updates all $|E|$ edges, each in $\Theta(1)$ time. Here $|V| = n$. Node u has an edge to each of the $n - u$ nodes after it, so

$$|E| = (n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2} = \Theta(n^2).$$

The time is $\Theta(n \cdot n^2) = \Theta(n^3)$.

This problem was similar to Problem #305 in the Quiz 08 practice problems.

Problem 5.

The version of Bellman-Ford from lecture that detects negative cycles runs its outer loop up to $|V|$ times, with early stopping. Suppose it is run on a graph with 8 nodes, and in the **8th** iteration of the outer loop, one of the estimated distances changes. What can you conclude?

● The graph has a negative cycle that is reachable from the source.

◯ The graph has no negative cycles.

◯ The graph has negative edge weights, but no negative cycles.

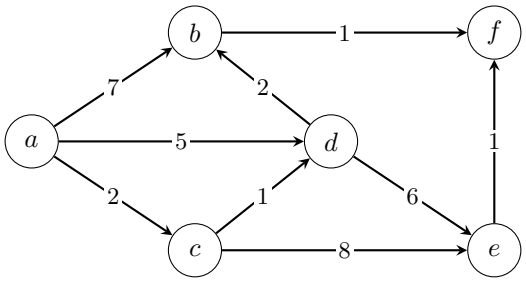
◯ The edges were produced in an unlucky order, and a 9th iteration would make every estimate correct.

Solution: The graph has a negative cycle that is reachable from the source.

If a graph has no negative cycles, every node has a shortest path with at most $|V| - 1 = 7$ edges, so every estimate is correct after 7 iterations, whatever the order of the edges. Estimates never go below the actual distances, so nothing can change in the 8th iteration. Since something did change, the graph must have a negative cycle reachable from the source, and some estimates would keep decreasing forever.

Problem 6.

Suppose Dijkstra's algorithm is run on the weighted, directed graph below using node a as the source.



What will be the **fifth** node popped from the priority queue? (The first node popped is the source, a.)

◯ b ◯ c ◯ d ◯ e ● f

Solution: f.

Dijkstra's algorithm always pops the node with the smallest estimated distance, then updates the edges out of it.

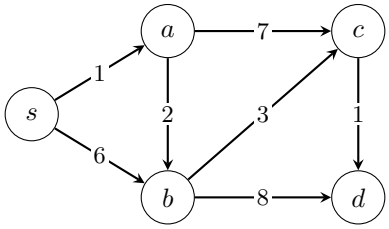
1. Pop a (0): b becomes 7, c becomes 2, and d becomes 5.
2. Pop c (2): d becomes $2 + 1 = 3$, and e becomes 10.
3. Pop d (3): b becomes $3 + 2 = 5$, and e becomes 9.
4. Pop b (5): f becomes 6.
5. Pop f (6).
6. Pop e (9).

So the fifth node popped is f.

This problem was similar to Problem #315 in the Quiz 08 practice problems.

Problem 7.

Suppose Dijkstra's algorithm, as written in lecture, is run on the weighted, directed graph below using node s as the source.



When the algorithm finishes, what is d's predecessor (that is, `pred['d']`)?

◯ s ◯ a ◯ b ● c

Solution: c.

Dijkstra's algorithm pops the node with the smallest estimated distance, then updates the edges out of it. An update that finds a shorter path to v also sets v's predecessor.

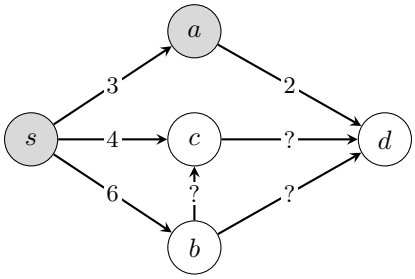
1. Pop s (0): a becomes 1 and b becomes 6, both with predecessor s.
2. Pop a (1): b becomes 3 and c becomes 8, both with predecessor a.
3. Pop b (3): c becomes 6 (predecessor b), and d becomes 11 (predecessor b).
4. Pop c (6): d becomes 7, and its predecessor changes to c.
5. Pop d (7).

So d's predecessor is first set to b, but ends up as c: the shortest path to d is s, a, b, c, d.

This problem was similar to Problem #315 in the Quiz 08 practice problems.

Problem 8.

Dijkstra's algorithm is run on the directed graph shown below using node s as the source. The weights marked "?" are unknown to us (but known to the algorithm), and are assumed to be **positive**. The algorithm has been run partway: the shaded nodes, s and a, have been popped from the priority queue, and the edges out of them have been updated.



a) At this point, the estimated distances of the other nodes are $b : 6$, $c : 4$, and $d : 5$. Which one of these statements must be true, whatever the unknown weights are?

- ● c's estimated distance is its actual distance from s.
- ○ d's estimated distance is its actual distance from s.
- ○ b's estimated distance is greater than its actual distance from s.
- ○ The next node popped from the priority queue is d.

Solution: c's estimated distance is its actual distance from s.

Of the nodes that haven't been popped, c has the smallest estimate, 4. Any other path from s to c would have to go through b (its only other way in), and just reaching b costs at least 6. Since the weights are positive, the rest of the path can't make it shorter. So no path to c is shorter than 4. This is Dijkstra's loop invariant: the node outside of C with the smallest estimate has a correct estimate, which is why it is popped next.

The others don't have to be true. d's estimate of 5 drops when c is popped if the weight of (c, d) is less than 1. b's only edge in is from s, so its estimate of 6 is already correct, not too large. And the next node popped is c, which has the smallest estimate, not d.

b) True or False: it is possible that the shortest path from s to d goes through b, depending on what the unknown weights are.

○ True ● False

Solution: False.

The path s, a, d ☹ has length $3 + 2 = 5$, so the shortest path to d has length at most 5. Any path through b costs 6 just to reach b, and since the weights are positive, the rest of the path can only add to that. So it is longer than 5, and the shortest path can't go through b, whatever the unknown weights are. This is how Dijkstra's algorithm can rule out paths without looking at them.

This problem was similar to Problem #316 in the Quiz 08 practice problems.

Problem 9.

Below is the priority queue implementation of Dijkstra's algorithm from lecture.

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}
    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])
    return est, pred
```

In lecture, the priority queue was a binary min-heap. Suppose your friend claims to have invented a new data structure for priority queues, where `PriorityQueue(est)` takes $\Theta(V)$ time, and both `.extract_min()` and `.change_priority()` take $\Theta(1)$ time.[1] If the claim were true, what would be the time complexity of the code above when it uses your friend's priority queue?

◯ $\Theta(V)$ ◯ $\Theta(E)$ ● $\Theta(V + E)$ ◯ $\Theta(V \cdot E)$

Solution: $\Theta(V + E)$.

Setting up `est`, `pred`, and the priority queue takes $\Theta(V)$ time. The `while`-loop runs once for each node, $|V|$ times in all, and each `.extract_min()` takes $\Theta(1)$ time, for $\Theta(V)$ in total. Each node's neighbors are looped over once, when the node is popped, so there are $\Theta(E)$ updates in total, each taking $\Theta(1)$ time, along with at most one $\Theta(1)$ `.change_priority()` each. Altogether, the time is $\Theta(V + E)$.

(With a binary min-heap, the last two operations take $O(\log V)$ time each, which is where lecture's $\Theta((V + E) \log V)$ comes from.)

This problem was similar to Problem #305 in the Quiz 08 practice problems.

[1] Your friend, unfortunately, must have made a mistake: no such priority queue can possibly exist. With it, we could sort n numbers in $\Theta(n)$ time: put them in the priority queue, then call `.extract_min()` n times. But we know that no comparison-based algorithm can sort faster than $\Theta(n \log n)$.