Compsci 330 Design and Analysis of Algorithms
Assignment 3, Fall 2024 Duke University

TODO: Add your name(s) here

Due Date: Monday, September 23, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on Gradescope. You can use the internet, but looking up solutions or using LLMs is unlikely to help you prepare for exams. See the homework webpage for more details.

Grading. Theory problems will be graded by TAs on an S/I/U scale. Applied problems typically have a separate autograder where you can see your score. See the course policy webpage for details about homework grading.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Graph Modeling). Alex is shopping for a new house in the Triangle area. Its road network is represented by an undirected graph $G = (V, E)$ with positive edge weights given by $w : E \to \mathbb{R}^+$, where the vertices are points-of-interest, and edges are direct road segments between them, and $w(e)$ for an edge e is the amount of time in minutes it takes to travel down (following the speed limit). There are no one-way streets, so modeling the edges as undirected suffices.

Let $H \subset V$ be a subset of houses for sale, let $D \subset V$ be the subset of daycares, and let $p \in V$ be the parking lot for his new job's building. Every weekday, Alex has to drop his son off at a daycare on his way to work, then pick his son up (at the same daycare, of course) on his way home after work. Interested in minimizing his commute, Alex wants to find the house $h \in H$ with the shortest total driving time each day. That is, the shortest drive from a house $h \in H$ to a daycare $d \in D$, then to p, then back to d, and finally back to h.

Describe an $O((n + m) \log n)$-time algorithm to find the desired house for Alex, where $n = |V|$ and $m = |E|$. Note that the sizes of H and D **are not** constant; they may be as large as $\Omega(n)$. Justify its correctness and analyze its runtime complexity.

As always, you may use algorithms from lecture without restating them or arguing for its correctness. If you modify an algorithm from lecture, make sure to be precise about how and explain why any modifications are correct.

[Hints: Given that G is undirected, what does this imply about Alex's best route from home to work compared to Alex's best route from work back to home? Consider building a different graph than G, then solve a shortest-path problem on it in order to derive a solution to the given problem on the original graph, G.]

Solution 1: A solution with multiple calls to Dijkstra's. We solve this problem by reduction to shortest paths. The total travel time consists of four parts: the time from a house to a daycare, from that daycare to the parking lot p, and then *the same walk in reverse.* As we will see, we actually find the shortest walk, starting at p, back to a house $h \in H$; its reverse is the first part of the shortest desired route.

Using Dijkstra's algorithm in G from p, we obtain the shortest path distances $\text{dist}(p, d)$ from p to each node in G; in particular, to every daycare $d \in D$. What remains to find which daycare should be visited from p to then reach a house $h \in H$ with minimum total distance (i.e., travel time), *which includes the distance from p to that daycare.*

We construct an undirected graph $G' = (V', E')$ as follows:

$$
\begin{aligned}
V' =& V \cup \{S\} \\
E' =& E \cup \{S_d \mid d \in D\}
\end{aligned}
$$

That is, we add a single super-source vertex S that is connected to each daycare. The weights of existing edges are the same, and the new edges between S each daycare $d \in D$ has weight $\text{dist}(p, d)$.

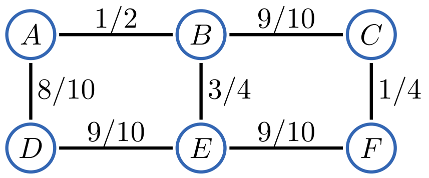
The key property is that, by construction of G', the shortest route among all houses $h \in H$ and $d \in D$ has length k if and only if the shortest path from S to any house (through a daycare) has length k.

Thus, using Dijkstra's algorithm starting at S in G', we obtain the shortest path distances from S to each house $h \in H$ in G'. Let $h^* \in H$ be the house with the minimum distance, and let $d^* \in D$ be the daycare incident to S on the corresponding shortest path to h^*. The desired route in G is then the shortest path in G from h^* to d^*, followed by the shortest path from d^* to p, followed by this overall *walk* from h^* to p in reverse.

The total runtime is the time to run Dijkstra's once in G, the time to construct G', and then the time to run Dijkstra's once in G'. Since G' has $n' = O(n)$ vertices and $m' = O(m)$ edges, the overall runtime is $((n' + m') \log n') = O((n + m) \log n)$ time.

Problem 2 (Probabilistic Routing). You are routing packets of information along a lossy network. You are given a connected, undirected, weighted graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, where the edge weights are real numbers in $(0, 1)$ corresponding to probabilities. The success probability along a path is the product of the probabilities of its edges.

For example, the success probability along the path A, B, C, F in the graph diagrammed below is $(1/2)(9/10)(1/4) \approx 11\%$ whereas the success probability along the path A, D, E, F is $(8/10)(9/10)(9/10) \approx 65\%$.



Describe an algorithm for computing the maximum probability with which a packet can be routed from a source s to a target t. Prove that the algorithm is correct and analyze analyze its running time. Make your algorithm as asymptotically efficient as possible.

As always, you may use algorithms from lecture without restating them or arguing for its correctness. If you modify an algorithm from lecture, make sure to be precise about how and explain why any modifications are correct.

[Hint: The standard shortest-path problem involves computing a path whose ***sum*** *of edge weights is smallest, but here the success probability is a* ***product*** *of edge weights. How can the edge weights be modified so that a shortest path corresponds to the path with maximum success probability?]*

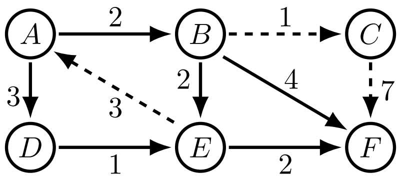
Solution 2. One solution is to transform all of the weights in the graph. In particular, let G' be the graph G where we replace each original edge weight p_e with $w_e = -\ln(p_e)$. Note that because $0 < p_e < 1$, we know that $-\ln(p_e)$ is nonnegative. Now, run Dijkstra's algorithm on G' with start vertex s and target vertex t. Suppose Dijskstra's algorithm finds a shortest path $P(s,t)$ in G'. We claim this is a maximum probability path in G.

Correctness. We assume the correctness of Dijsktra's algorithm to compute a shortest path $P(s,t)$ in G'. Then for any other path from s to t $P'(s,t)$

$$\begin{aligned} \sum_{e \in P(s,t)} w_e &\leq \sum_{e \in P'(s,t)} w_e \\ \implies \sum_{e \in P(s,t)} -\ln(p_e) &\leq \sum_{e \in P'(s,t)} -\ln(p_e) \\ \implies -\ln\left(\prod_{e \in P(s,t)} p_e\right) &\leq -\ln\left(\prod_{e \in P'(s,t)} p_e\right) \\ \implies \ln\left(\prod_{e \in P(s,t)} p_e\right) &\geq \ln\left(\prod_{e \in P'(s,t)} p_e\right) \\ \prod_{e \in P(s,t)} p_e &\geq \prod_{e \in P'(s,t)} p_e \end{aligned}$$

where the last inequality follows because log is a monotone increasing function for nonnegative numbers. So $P(s,t)$ must have maximum probability among all paths from s to t.

Problem 3 (Useless Edges). Let $G = (V, E)$ be a directed graph, not necessarily acyclic, with $|V| = n$, $|E| = m$ and with positive edge weights $w(u, v)$ for every $u \to v \in E$. For given vertices s and t, note that there may be multiple shortest paths from s to t. For example, there are three distinct shortest paths from A to F in the following figure (edges of these paths drawn as solid lines).



Describe an algorithm that, given vertices s and t, returns the number of edges in the graph that are *not* included in *any* shortest paths from s to t. In the example graph above with $s = A$ and $t = F$ there are three such edges (drawn in dashed lines). Prove that the algorithm is correct and analyze its running time. Make your algorithm as asymptotically efficient as possible.

As always, you may use algorithms from lecture without restating them or arguing for its correctness. If you modify an algorithm from lecture, make sure to be precise about how and explain why any modifications are correct.

Solution 3. We run single-source Dijkstra's algorithm twice: Once from s in the original graph to calculate the shortest path distances $d[s, u]$ from s to every other node u, and once from t in the reverse graph (with the same weights) to calculate the shortest path distances $d[u, t]$ from every other node u to t. We then count an edge $u \to v$ as useless if $d[v, t] = \infty$ (in which case it is impossible to traverse the edge and still reach t) or if $d[s, u] + w(u, v) + d[v, t] > d[s, t]$ (in which case traversing the edge necessarily results in a longer path than the shortest). We write as a procedure below.

```
1: procedure USELESS($G, s, t$)
2:     Run Dijkstra's algorithm on $G$ starting from $s$ to compute distances $d[s, u]$
3:     Compute the reverse graph $G^R = (V, E^R = \{v \to u \,|\, u \to v \in E\})$
4:     Run Dijkstra's algorithm on $G^R$ starting from $t$ to compute distances $d[u, t]$
5:     $c = 0$
6:     for $u \to v \in$ do
7:         if $d[v, t] = \infty$ or $d[s, u] + w(u, v) + d[v, t] > d[s, t]$ then
8:             $c = c + 1$
9:     return $c$
```

Correctness. We assume the correctness of Dijkstra's algorithm for computing the shortest path distance $d[s, u]$ from s to u in G for every $u \in V$ and for computing the shortest path distance from t to u in $G^R$for every $u \in V$. Note that the shortest path distance from t to u in G^R is equal to the shortest path distance $d[u, t]$ from u to t in G.

We want to argue that $u \to v$ is "useless" (not included in any shortest path from s to t) if and only if $d[v, t] = \infty$ or $d[s, u] + w(u, v) + d[v, t] > d[s, t]$.

- Suppose $d[v, t] = \infty$. Then there is no path from v to t. Then there is no path that traverses $u \to v$ and then reaches t. The edge $u \to v$ is therefore not on any path from s to t.

- Suppose $d[s,u]+w(u,v)+d[v,t] > d[s,t]$. Consider any path P from s to t that traverses the edge $u \to v$. Let $|P|$ be the sum of the weights of a path consisting of zero of more edges. The path can be decomposed into $P_{s \dashrightarrow u}$ (the path edges from s to u), $u \to v$, and $P_{v \dashrightarrow t}$ (the path edges from v to t). Since $d[s,u]$ is the shortest path distance from s to u, $|P_{s \dashrightarrow u}| \geq d[s,u]$. Similarly, $|P_{v \dashrightarrow t}| \geq d[v,t]$. So we conclude that

$$\begin{aligned} |P| &= |P_{s \dashrightarrow u}| + w(u,v) + |P_{v \dashrightarrow t}| \\ &\geq d[s,u] + w(u,v) + d[v,t] \\ &> d[s,t]. \end{aligned}$$

  To see the other direction, observe that if $u \to v$ is "useless" then it cannot have $d[s,u] + w(u,v) + d[v,t] \leq d[s,t]$, or else the path composed of the shortest path from s to u, then the edge $u \to v$, then the shortest path from v to t would be a shortest path.

Runtime. There are two runs of Dijkstra's algorithm, each of which takes $O((n+m)\log(n))$ time. Computing the reverse graph can be done in linear $O(n+m)$ time in a single pass over the adjacency list representation. The for loop on line 6 has m iterations and each iteration is constant time. The two runs of Dijkstra's algorithm dominate, so the overall runtime complexity is $O((n+m)\log(n)$.

Problem 4 (Applied). **Autograder available 9/17 @ 5pm.**
In this problem, we consider the problem of computing shortest paths in directed graphs with real edge weights (possibly negative). Let $G = (V, E)$ be a directed graph where $n = |V|$, $m = |E|$, and each edge $w \to v \in E$ has a real weight ℓ_{vw}, and let $s \in V$ be a specified vertex in G. In lecture, we presented[2] Bellman's equations for the shortest-path distances from s to all vertices $v \in V$:

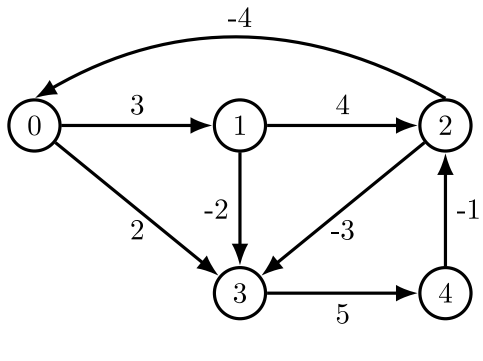
$$\begin{aligned} d[s] &= 0 \\ d[v] &= \min_{wv \in E} \{d[w] + \ell_{wv}\}, \text{for all } v \neq s. \end{aligned}$$

Under the assumption that the graph has no negative cycles (paths from one vertex to itself whose sum of weights is negative), we described how to solve these equations, yielding the shortest-path distances from s to all other vertices. First, we initialize values $\pi^{(0)}[s] = 0$ and $\pi^0[v] = \infty$ for all vertices $v \neq s$. Then we perform $n-1$ "Bellman iterations," where in each k-th iteration, for k from 1 to $n-1$, we compute a value $\pi^{(k)}[v]$ for each $v \in V$. In the end, we have $\pi^{(n-1)}[v] = d[v]$ for all vertices $v \in V$. The details and efficient implementations of the iterations were described in lecture.

However, it is possible to relax the assumption that G contains no negative cycles and modify the algorithm accordingly, as follows. The main change is to perform an additional n-th Bellman iteration (instead of only $n-1$), computing $\pi^{(n)}[v]$ for each vertex $v \in V$. If $\pi^{(n)}[v] = \pi^{(n-1)}[v]$ for all vertices $v \in V$, then it can be shown G **does not** contain any negative cycle reachable from s (since the values converged). Otherwise, it must be that $\pi^{(n)}[v] < \pi^{(n-1)}[v]$ for some vertex $v \in V$, and G **does** contain a negative cycle reachable from s (since the values did not converge). You may use these facts without proof when implementing your algorithm.

We represent graphs with vertices as integers from 0 to $n-1$, where n is the number of vertices. Your task is to **design and implement** an algorithm that, given $G = (V, E)$ with real edge weights and a specified vertex t, either reports that there is a negative cycle reachable from vertex 0 in G if one exists, and otherwise returns the shortest path from vertex 0 to vertex t in G as an array of vertices (ints). Your solution will need to have an empirical runtime that is within constant factors of an reference solution with $O(nm)$ worst-case runtime, where n and m are the numbers of vertices and edges in G, respectively.

The example graph below is a graph with negative weights but no negative cycles.



The shortest-path distances (the $d[v]$'s) for all vertices $0, 1, 2, 3, 4$ from $s = 0$ are $0, 3, 5, 1, 6$, respectively. The shortest path from vertex 0 to vertex 4 goes through vertices 1 then 3 before reaching

[2]We note that the bottom equations are slightly different from lecture, where the minimum is over all *incoming edges* to the vertex v as opposed to all vertices w. The equations here can be seen to be equivalent to the original definitions from lecture, by treating ℓ_{wv} as ∞ when there is no edge from w to v in the latter.

4, for a total weight of $3 + (-2) + 5 = 6$. Representing the path as an array of integers, this is $[0, 1, 3, 4]$.

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline.

- **Python.** You should submit a file called `bellman.py` to the Gradescope item "Homework 3 (Python)." The file should define (at least) a top level function `bellman` that takes in an int n, the number of vertices, an int t, the target vertex, and a list `edges` of tuples `(int, int, float)` representing the edges, where `edges[i]` represents an edge from `edges[i][0]` to `edges[i][1]` with weight `edges[i][2]`. The function returns:
  - If a negative cycle reachable from vertex 0 exists: `None`
  - Otherwise: A shortest path from vertex 0 to t, represented by a list of integers (vertices).

  The function header is:

  ```
  def bellman(n:int, t:int, edges:list[(int, int, float)]):  -> list[int].
  ```

  For example, your algorithm should return `[0,1,3,4]` for the example graph above with $t = 4$.

- **Java.** You should submit a file called `Bellman.java` to the Gradescope item "Homework 3 (Java)." The file should define (at least) a top level function `Bellman` that takes in an int n, the number of vertices, an int t, the target vertex, and two arrays that represent the edges and their real weights: a 2D $m \times 2$ array `int[][] edges` and a m-length 1D array `double[] weights`, where `edges[i]` is the directed edges from `edges[i][0]` to `edges[i][1]` and its weight is `weight[i]`. The method returns:
  - If a negative cycle reachable from vertex 0 exists: `null`
  - Otherwise: A shortest path from vertex 0 to t, represented by a list of integers (vertices).

  For example, your algorithm should return `[0,1,3,4]` for the example graph above. The method header is:

  ```
  public ArrayList<Integer> bellman(int n, int t, int[][] edges, double[] weights);
  ```

  For example, your algorithm should return an `ArrayList` with `[0,1,3,4]` for the example graph above with $t = 4$.

Solution 4. The overall algorithm to either detect a present negative cycle reachable from vertex 0 or to compute the shortest-path distance to t (and all other vertices) is described above. To obtain the shortest path from 0 to t itself, we additionally keep track of the "predecessor" vertices as we update the π-values. In particular, for each vertex $v \in V$, we maintain a value $pred(v)$, which is the vertex on the shortest path from 0 to v. In particular, this vertex $pred(v)$ is the one for which $\pi^{(n)}[v] = \pi^{(n-1)}[pred(v)] + \ell_{pred(v),v}$. Using these values, one can start at t, to $pred(t)$, and so on, back to vertex 0 to obtain the shortest path (in reverse order).