1. (a) Describe an algorithm that given a dag $G = (V, E)$ with one source $s \in V$ and one sink $b \in V$ determines whether all paths from s to b have the same length.

Solution (one-shot dynamic programming): Given a tuple $X = (x_1, x_2, \ldots)$, let $X.i$ denote the component x_i.

Let $\mathit{MinMaxLength}(u)$ denote the pair (ℓ_1, ℓ_2) where ℓ_1 and ℓ_2 are the lengths of the shortest and longest paths from u to b, respectively. We need to compute whether $\mathit{MinMaxLength}(s).1 = \mathit{MinMaxLength}(s).2$. The function satisfies the following recurrence.

$$\mathit{MinMaxLength}(u) = \begin{cases} (0,0) & \text{if } u = b \\ \begin{pmatrix} \min\limits_{u \to v} (1 + \mathit{MinMaxLength}(v).1), \\ \max\limits_{u \to v} (1 + \mathit{MinMaxLength}(v).2) \end{pmatrix} & \text{otherwise} \end{cases}$$

We can memoize this function by storing each value $\mathit{MinMaxLength}(u).1$ in a field $u.\mathit{Min}$ and $\mathit{MinMaxLength}(u).2$ in a field $u.\mathit{Max}$. Evaluating the entires in postorder takes $O(|V| + |E|)$ ***time total***. ■

Rubric: 4 points: standard dynamic programming rubric.

Solution (reduction to optimal paths): All paths from s to b have the same length if and only if the shortest path has the same length as the longest path. We compute $b.\mathit{dist}$, the distance from s to b using, say, DagSSSP(s). We also compute $s.\mathit{longest}$ which denotes the length of the *longest* path from s to b by running DagLongestPaths(b) from lecture or by giving all edges a weight of -1 and running DagSSSP again. We then compare the optimal path lengths and return if they're equal.

Each optimal paths computation and therefore the whole algorithm takes $O(|V| + |E|)$ ***time total***. ■

Rubric: 4 points: standard dynamic programming rubric.

(b) Describe an algorithm to compute the maximum number of relics the PC can acquire without dying while following some path from s to b or $-\infty$ if there is no path from s to b that the PC can survive. Your running time should be a function of $|V|$, $|E|$, and k. For simplicity, you may assume $Damage(s) = 0$ and $Relic(s) = \text{FALSE}$.

Solution (dynamic programming): Let $MaxRelics(u, \ell)$ denote the maximum number of relics the PC can acquire along a path from u to b starting with $\ell \geqslant 1$ health (without counting damage or relics at u itself) or $-\infty$ if no such path exists. We need to compute $MaxRelics(s, k)$. The function satisfies the recurrence

$$MaxRelics(u,\ell) = \begin{cases} 0 & \text{if } u = b \\ \max\limits_{\{u\to v \mid Damage(v)<\ell\}} \left(\begin{array}{l} [Relic(v)] + \\ \\ MaxRelics(v, \ell - Damage(v)) \end{array} \right) & \text{otherwise} \end{cases}$$

where $[P] = 1$ if proposition P is TRUE and 0 otherwise, and a max over the empty set evaluates to $-\infty$.

We can memoize this function by storing all $MaxRelics(u, \cdot)$ in an array $u.MaxRelics[0..k]$. We can evaluate subproblems in a postorder as the outer loop and any ordering of the index ℓ as the inner loop. ***The algorithm takes $O(k(|V| + |E|))$ time total.*** ■

Rubric: 6 points: standard dynamic programming rubric.

Solution (graph reduction): We create a new directed acyclic graph $G' = (V', E')$ with edge weights $w' : E' \to \mathbb{R}$ defined as follows:

- $V' := (V \times \{0, \ldots, k\}) \sqcup t$ where $t \notin V$ is a new vertex
- $E' := \{(u,\ell)\to(v, \ell - Damage(v) \mid Damage(v) < \ell\} \cup \{(b,\ell)\to t \mid \ell \in \{1, \ldots, k\}\}$
- $w'((u,\cdot)\to(v,\cdot)) := 1$ for all $(u,\cdot)\to(v,\cdot) \in E'$ and $Relic(v) = \text{TRUE}$
- $w'((u,\cdot)\to(v,\cdot)) := 0$ for all $(u,\cdot)\to(v,\cdot) \in E'$ and $Relic(v) = \text{FALSE}$
- $w'((b,\cdot)\to t) := 0$ for all $(b,\cdot) \in V'$

The maximum number of relics obtainable on a path from s to b in G is the maximum length of a path from (s, k) to t in G'. We run DAGLONGESTPATHS(t) as given in lecture and return $(s, k).varlength$. The algorithm runs in $O(|V|' + |E|') = O(k(|V|' + |E|'))$ ***time***. ■

Rubric: 6 points: standard graph reduction rubric.

2. (a) Describe an algorithm to determine whether G contains an energetic cycle.

Solution: Let $w' : E \to \mathbb{R}$ be a second set of real-valued edge weights where $w'(e) = 2$ if e is red and $w'(e) = -1$ if e is blue. A path or cycle W with r red edges and b blue edges is energetic if and only if $2r - b < 0$ if and only if $\sum_{e \in W} w'(e) < 0$.

We'll assume there is a vertex s that can reach everything in G (if not, add a new vertex s with 0 weight (colorless) edges going to each vertex of G). We run Bellman-Ford from s using the edge weights w' and return whether if G has a negative weight cycle. ***The running time is $O(|E||V|)$.*** ■

Rubric: 3 points: standard graph reduction rubric. No penalty for assuming G is strongly connected.

(b) Prove the following claim: If G contains an energetic closed walk (possibly with repeated vertices), then the *shortest* energetic closed walk in G is actually a simple cycle (with no repeated vertices).

Solution: Let W be the shortest energetic closed walk in G (assuming an energetic closed walk exists). Suppose W is not a simple cycle. Let v be a vertex repeated on W, let W_1 be a closed walk from an instance of v on W to its next instance on W, and let W_2 be the remainder of W starting from the end of W_1. Edge weights are positive, so both W_1 and W_2 weigh strictly less than W.

Let r_i and b_i be the number of red and blue edges, respectively, in W_i. We have $2r_1 + 2r_2 < b_1 + b_2$, implying either $2r_1 < b_1$ or $2r_2 < b_2$. Either W_1 or W_2 is an energetic closed walk that is strictly lighter than W, contradicting W not being simple.

■

Rubric: 2 points. This is more detail than necessary for full credit.

(c) Describe and analyze an algorithm to compute the length of the *shortest* energetic cycle in G.

Solution: According to part (b) (and us asking for a cycle), the shortest energetic cycle contains at most $|V| - 1$ edges. Therefore, we can afford to build a layered graph that tracks twice the number of red edges minus the number of blue edges on any walk that might be part of a shortest energetic cycle.

We reduce to shortest paths in a new directed graph $G' = (V', E')$ with positive edge weights $w' : E' \to \mathbb{R}^+$ as follows.

- $V' \coloneqq V \times \{-|V| + 1, \ldots, 2|V| - 2\}$. Vertex (v, i) means we've taken a walk from some (u) where i is twice the number of red edges minus the number of blue edges in the walk.
- $E' \coloneqq \{(u,i)\to(v,i+2) \mid u\to v \text{ is a red edge and } i \leqslant 2|V| - 4\}$
  $\cup\, \{(u,i)\to(v,i-1) \mid u\to v \text{ is a blue edge and } i \geqslant -|V| + 2\}$.
- $w'((u,\cdot)\to(v,\cdot)) \coloneqq w(u\to v)$ for all $u\to v \in E$.

The shortest energetic cycle containing a particular vertex v in G is a shortest path from $(v, 0)$ to some (v, i) with $i < 0$ in G'. Edge weights are positive, so for each vertex $v \in V$, we run Dijkstra's algorithm from $(v, 0)$ and record the least of those distances in a value $v.shortest$. We return $\min_{v \in V} v.shortest$. Our algorithm runs in $O(|V|(|E|' \log |V|')) = \boldsymbol{O(|E||V|^2 \log |V|)}$ ***time***. ■

Rubric: 5 points: standard graph reduction rubric. An $O(|E||V|^4)$ time algorithm is worth 3 points.

(c) ***Practice only. Do not submit solutions.***

Suppose instead that the lengths of edges can be negative after all. Describe an algorithm that *either* computes the length of the shortest *walk* W in G for which $c(W)$ is accepted by the DFA M or correctly reports that no shortest walk *of that type* exists.

Solution: For this problem, we need to open the Bellman-Ford black box. Consider the graph $G' = (V', E')$ constructed in the previous solution. We need to find the length of the shortest walk from s' to t' or report that no shortest walk exists.

For any $v' \in V'$ and non-negative integer i, let $dist(v', i)$ denote the minimum length of any walk from s' to v' that uses at most i edges. We claim no shortest walk from s' to t' exists if and only if there exists a vertex v' such that $dist(v', kn+2) < dist(v', kn+1)$ and v' can reach t'.

Suppose there is such a vertex v'. In that case, the shortest walk W' from s' to v' that uses at most $kn+2$ edges uses exactly $kn+2$ edges and therefore repeats at least one vertex u'. Let C' be the portion of W' from the first occurrence of u' to the second occurrence of u'. Closed walk C' has negative length; otherwise, there would be a walk of no greater length that uses fewer edges, contradicting $dist(v', kn+2) < dist(v', kn+1)$. We can construct an arbitrarily short walk from s' to t' by walking from s' to u' along W', going around C' an arbitrarily large number of times, walking from u' to v' along W', and finally walking from v' to t' along an arbitrary path.

Now, suppose no such v' exists. Inductively, every vertex w' that can reach t' will have $dist(w', i) = dist(w', i-1)$ for all $i \geqslant kn+2$, implying $dist(t', kn+1)$ is the length of the shortest walk from s' to t'.

We compute $dist(v', kn+2)$ and $dist(v', kn+1)$ for all vertices $v' \in V'$ using the standard evaluations for Bellman-Ford. We also compute which vertices can reach t' by doing a whatever-first search like BFA from t' in the *reversal* of G'. If $dist(v', kn+2) < dist(v', kn+1)$ for any vertex v' that can reach t', we report no shortest walk exists. Otherwise, we return $dist(t', kn+1)$.

Our algorithm runs in $O(|E'||V'|) = \boldsymbol{O(k^2mn)}$ ***time***. ■

Rubric: Practice only: no points.

3. ***Practice only. Do not submit solutions.***

After a grueling midterm at the See-Bull Center for Commuter Silence, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Sham-Poobanana. Unfortunately, no single bus visits both the See-Bull Center and your home; you must change buses at least once.

For the following questions, your input consists of the exact time when the midterm ends at See-Bull and two arrays $Time[1..b, 1..n]$ and $Stop[1..b, 1..n]$, where $Time[i, j]$ is the scheduled time of the ith bus's jth stop, and $Stop[i, j]$ is the location of that stop. Report the running times of your algorithms as functions of the parameters n and b.

(a) Describe and analyze an algorithm to determine the sequence of bus rides that gets you home as early as possible. Your goal is to minimize your *arrival time*, not the time you spend traveling.

Solution: First we compute, for each bus stop location i, a list of which buses stop at that location at what times, in chronological order. We can compute these arrays in $O(nb \log nb)$ time essentially by brute force. The extra logarithmic factor comes from either sorting these lists chronologically, or using a priority queue to walk through the input arrays in chronological order.

We build a directed graph G with nb vertices, each labeled by a bus stop and a time when some bus visits that stop, plus two additional nodes: a start node s representing the See-Bull Center, and a target node t representing home. G has four types of edges:

- For every bus, G contains a sequence of $n-1$ directed *ride* edges joining its visits in chronological order. The weight of a ride edge is the time between the two visits represented by its endpoints.
- For every stop, G contains a sequence of directed *wait* edges joining its visits by buses in chronological order. The weight of a wait edge is the time between the two visits represented by its endpoints.
- G has a *start* edge from s to the node representing the first visit to the bus stop closest to See-Bull after your midterm ends. The weight of this edge is the difference between that first visit time and the end time of your midterm.
- Finally, G has *target* edges from *every* node representing the bus stop closest to home to t; these edges all have weight 0.

There are exactly $b(n-1)$ ride edges. In addition, every node has at most one outgoing wait edge, at most one incoming start edge, and at most one outgoing home edge. Thus, the total number of edges is at most $3nb = O(nb)$.

Every ride home is represented by a path from s to t in G. The length of this path is equal to the arrival time at home. Thus, we need to compute the shortest path in G from s to t.

We can compute this path using Dijkstra's algorithm. The distance value $dist(t)$ tells us the time spent outside during the best trip home; the predecessor pointers allow us to reconstruct the actual ride path. Because there are no

negative-weight edges, Dijkstra's algorithm runs in $O(E \log V) = \boldsymbol{O(nb \log nb)}$ ***time***.

We can make this algorithm slightly faster by observing that every edge leads from an earlier time to a later time, which implies that G is a directed *acyclic* graph. Thus, we can compute the shortest path in G from s to t using dynamic programming, as described in the textbook, in $O(V + E) = O(nb)$ time. However, the overall algorithm still runs in $\boldsymbol{O(nb \log nb)}$ ***time***, because we need that much time to construct G. ■

Rubric: 5 points: graph-reduction rubric, including 1 point for nontrivial preprocessing, and 1 point for proving G is a dag for solutions that use dynamic programming. These are not the only correct solutions.

(b) Describe how to modify your algorithm from part (a) to minimize *the total time you spend waiting at bus stops*.

Solution: We build almost exactly the same directed graph G as in part (a), with the following differences:

- All ride edges now have weight 0.
- G now has a separate *start* edge from s to *every* node representing the bus stop closest to See-Bull after your midterm ends. Each of these edges has weight 0.

The total number of edges is still $O(nb)$.

Every ride home is represented by a path from s to t in G. The length of this path is equal to total time spent waiting at bus stops, except at the stop closest to See-Bull. So we still need to compute the shortest path in G from s to t.

We can compute this path using either Dijkstra's algorithm or dynamic programming. (Each edge in G still connects an earlier even to a later event.) As in part (a), the overall algorithm runs in $\boldsymbol{O(nb \log nb)}$ ***time***. ■

Rubric: 5 points: graph-reduction rubric, including 1 point for nontrivial preprocessing, and 1 point for proving G is a dag for solutions that use dynamic programming. Yes, this is enough detail for full credit. This is not the only correct solution.