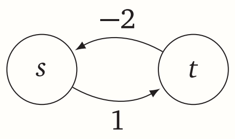
1. Let $G = (V, E)$ a directed graph with edge lengths $\ell(e), e \in E$. The edge lengths can be negative. Let $R \subset E$ be red edges (an edge can be red or uncolored). Given s, t and an integer h_r, the goal is is to find the length of a shortest s-t *walk* that contains at most h_r red edges. Note that if the same red edge is repeated ℓ times in a walk then it is counted ℓ in the bound h_r.

   - Describe an instance or example in which the shortest walk length is $-\infty$.

     Solution: If a graph has a negative cycle that doesn't use any red edges, a shortest walk can repeatedly take this cycle and continually get shorter and shorter, meaning the shortest walk length would be $-\infty$ in this case.

     In the picture below there is a negative cycle with no red edges. The shortest s-t walk length is $-\infty$: for every s-t walk, we can shorten it by walking through the cycle more times.

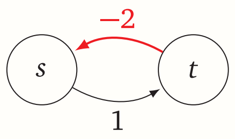
     

     ■

   - Describe an instance or example in which the shortest walk length is finite but there is no path achieving it (in other words one needs a cycle in the walk).

     Solution: If the graph has negative cycles, but each negative cycle contains a red edge, then one can go through a negative cycle at most h_r times. Then the shortest walk length is finite but going through negative cycles may be necessary to obtain the shortest walk.

     In the picture below there is a negative cycle using one red edge, and the shortest s-t walk goes through it h_r times.

     ■

- Describe an efficient algorithm that finds the shortest walk length from s to t under the given constraints, or reports that it is $-\infty$. The running time of your algorithm should be polynomial in m, n, h_r where $m = |E|, n = |V|$. *Hint:* Under what conditions will the answer be $-\infty$? If walk length is finite what is the maximum number of edges it can contain?

Solution (Graph Modeling): We will first check if the shortest walk length from s to t is $-\infty$.

This is possible if and only if there is a negative cycle C containing no red edges, such that one can reach C from s, and then t from C. So let $G_{\text{no}R}$ be G with all red edges removed. If there exists such a negative cycle C, then the entirety of C is contained in a strongly-connected component (scc) of $G_{\text{no}R}$.

For each scc of $G_{\text{no}R}$, we can check if it contains a negative cycle via standard Bellman-Ford. By running Basic (aka Whatever-First) Search in the scc metagraph $G_{\text{no}R}^{\text{SCC}}$ of $G_{\text{no}R}$ from the metavertex for s, and then in the reverse of $G_{\text{no}R}^{\text{SCC}}$ from the metavertex for t, we know which metavertices lie on a path from the metavertex for s to the metavertex for t. We then check if there is such a metavertex that corresponding to an scc with a negative cycle. If so then we report that the shortest walk length is $-\infty$.

This algorithm is dominated by the time for running Bellman-Ford in each scc of $G_{\text{no}R}$. Number the scc's arbitrarily from 1 to k, and let n_i and m_i be the number of vertices and edges, respectively, in the i-th scc. Observe that $\sum_{i=1}^{k} n_i \leq n$ and $\sum_{i=1}^{k} m_i \leq m$. Then running Bellman-Ford over all the scc's takes order $\sum_{i=1}^{k} m_i n_i \leq \sum_{i=1}^{k} \sum_{j=1}^{k} m_i n_j = (\sum_{i=1}^{k} m_i)(\sum_{j=1}^{k} n_j) \leq mn$ time. So the running time can be summarized as just $O(mn)$.

In the case where we have determined that the shortest walk length will be finite, we now need to *find* the shortest walk length. We use a graph layering construction similar to lab. We will have layers numbered 0 through h_r, where the i-th layer corresponds to having used i red edges. Naturally,

$$V' := V \times \{0, \ldots, h_r\}.$$

As for the copies of each edge e, if $e \in R$, each copy of e will point towards the next layer, otherwise the copy points within the same layer.

$$\begin{aligned} E' := &\left\{(u,i) \to (v,i) \;\middle|\; u \to v \in E \setminus R, 0 \leq i \leq h_r\right\} \\ &\cup \left\{(u,i) \to (v,i+1) \;\middle|\; u \to v \in R, 0 \leq i < h_r\right\}. \end{aligned}$$

We then run a standard Bellman-Ford on $G' = (V', E')$ to find the shortest path from $(s, 0)$ to every (t, i) for $0 \leq i \leq h_r$ and then report the minimum distance to be $\min_{0 \leq i \leq h_r} d_{G'}((s,0),(t,i))$.

Overall the running time is dominated by the second part of the algorithm, which takes $\boldsymbol{O(mnh_r^2)}$ ***time***, as G' has $n \cdot (h_r + 1)$ vertices and $m \cdot h_r$ edges. ■

Solution (Modifying Bellman-Ford): We modify the Bellman-Ford recurrence by adding an extra parameter to account for the number of allowed red edges. Thus let $d(v, j, k)$ denote the shortest walk length from s to v using at most j hops and at most k red edges.

We need to figure out what the parameters for the return values should be. Obviously the third parameter should be h_r, but what about the second parameter? Recall that for the Standard Bellman-Ford recurrence for shortest paths, the answer was $dist(s, v) = d(v, n-1)$ since a path cannot use more than $n-1$ edges. In our case, if the walk length is finite, then all negative cycles must contain a red edge. As seen in the solution to the previous part, it may be the case that this negative cycle contains n edges. So in the worst case, if the walk length is finite then it can contain at most $n-1$ edges for the part that does not go through any negative cycles, plus nh_r edges for the part that does go through a negative cycle, for a total of $n(h_r+1)-1$ edges. Assuming that the shortest-walk length is finite, we should set the minimum distance from s to $v \in V$ as $dist(s, v) := d(v, n(h_r+1)-1, h_r)$.

The function $d(v, j, k)$ satisfies the recurrence

$$d(v,j,k) = \begin{cases} 0 & \text{if } v = s,\ j = 0 \\ \infty & \text{if } v \neq s,\ j = 0 \\ \infty & \text{if } k < 0 \\ \min \left\{ \begin{matrix} \min_{u\to v \in E\setminus R} \left\{ d(u, j-1, k) + \ell(u\to v) \right\} \\ \min_{u\to v\in R} \left\{ d(u, j-1, k-1) + \ell(u\to v) \right\} \\ d(v, j-1, k) \\ d(v, j, k-1) \end{matrix} \right\} & \text{otherwise} \end{cases}$$

This recurrence can be memoized into a three-dimensional array, in increasing order of j and increasing order of k in the outer two loops, and over the vertices in the inner loop.

To complete the algorithm we need to check if the shortest-walk length from s to t is $-\infty$. Recall that Bellman-Ford does negative-cycle detection by taking advantage of the fact that if there is a negative cycle somewhere, then running one more iteration will find some vertex whose distance from s will be shortened. However, here we need to see if the distance to t can be shortened. In the worst case, we need to walk around an additional negative cycle of length n to shorten the distance. We will take an additional n iterations of Bellman-Ford where we *ignore red edges*. If at the end of this, the distance to t has been reduced, then we should report that the shortest-walk length is $-\infty$; otherwise we return $dist(s, t) := d(t, n(h_r+1)-1, h_r)$.

The running time of the algorithm is $\boldsymbol{O(mnh_r^2)}$.

Modifying the iterative Bellman-Ford algorithm to reflect the new recurrence and the check for if the shortest-walk length is $-\infty$, gives the pseudocode on the next page. With some care one can reduce the space usage similar to what was done in lecture; we will omit the details of this optimization.

```
ShortestWalkWithRed($G, s, t, R, h_r$):
    for $u \in V$
        $d[u,0,0] \leftarrow \infty$
    $d[s,0,0] \leftarrow 0$
    $N \leftarrow n(h_r+1)-1$

    for $j$ from 1 to $N$
        for $v \in V$
            $d[v,j,-1] \leftarrow \infty$
        for $k$ from 0 to $h_r$
            for $v \in V$
                $d[v,j,k] \leftarrow \min\{d[v,j-1,k], d[v,j,k-1]\}$
                for $u{\to}v \in E$
                    if $u{\to}v \in R$
                        $d[v,j,k] \leftarrow \min\{d[v,j,k], d[v,j-1,k-1] + \ell(u{\to}v)\}$
                    else
                        $d[v,j,k] \leftarrow \min\{d[v,j,k], d[v,j-1,k] + \ell(u{\to}v)\}$

    for $j$ from 1 to $n$
        for $v \in V$
            for $u{\to}v \in E \setminus R$
                $d[v,N+j,h_r] \leftarrow \min\{d[v,N+j,k], d[v,Nj-1,h_r] + \ell(u{\to}v)\}$

    if $d[t,N+n,h_r] < d[t,N,h_r]$
        return $-\infty$
    else
    return $d[t,N,h_r]$
```

■

Solution (Perspectives on Bellman-Ford and Layering): Recall that one way of interpreting the Bellman-Ford algorithm is as follows. We have copies of the vertices labeled 0 through $n-1$, and then for each edge $u{\to}v$ we have a copy $(u,i){\to}(v,i+1)$ for $0 \le i < n-1$. Then the base part of Bellman-Ford can be interpreted as running the linear-time Dynamic Programming algorithm for Shortest Paths in DAGs on this layered DAG. For the distance between s and $v \in V$, take the minimum over $0 \le i \le n-1$ of the distance from $(s,0)$ to (v,i).

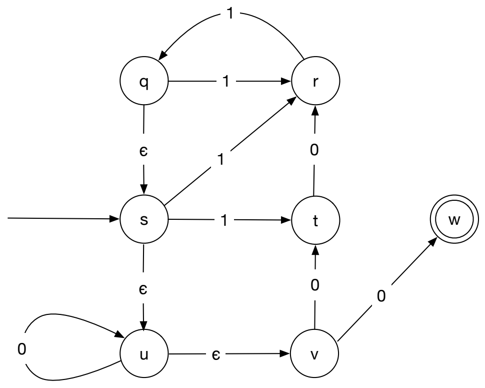
We can apply this perspective to understand that, for the parts where the shortest-walk length is assumed to be finite, the previous two solutions are really doing the same thing: each layer in the graph modeling solution corresponds to one possible value of the last parameter in the modified Bellman-Ford solution, and the reason we set $dist(s,v) := d(v, n(h_r+1)-1, h_r)$ in the modified Bellman-Ford solution is exactly because there are $n(h_r+1)$ vertices in the layered graph, i.e., any path in the layered graph can have at most $n(h_r+1)-1$ edges.

We can *further* apply this perspective to see that both of these are equivalent to creating a ***DAG*** consisting of $O(nh_r) \times O(h_r)$ copies, where being in the (j,k)-th copy means having taken j edges, of which k are red, and then running the linear-time algorithm for Shortest Paths in DAGs on this doubly-layered graph. The distance between s and $v \in V$ is the minimum over j,k of the distance from $(s,0,0)$ to (v,j,k). You can think about how to figure out if the shortest-walk length from s to t is $-\infty$. The running time is (once again) $\boldsymbol{O(mnh_r^2)}$. ■

Rubric: 10 points.

- 1 point for the first part.
- 1 points for the second part.
- 8 points for the third part. Scaled Graph Modeling Rubric or Dynamic Programming rubric, as appropriate. No penalty for slower algorithms.
  - Additonally, -1 if the solution only reports the correct answer if the shortest-walk length is finite (e.g., if there is a negative cycle containing no red edges, it just reports "negative cycle" instead of determining if the distance to t is affected).

2. Recall that an NFA N is specified as $(Q, \delta, \Sigma, s, F)$ where Q is a finite set of states, Σ is a finite alphabet, $s \in Q$ is the start state, $F \subseteq Q$ is the set of final (or accepting) states and $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the transition function. Recall that δ^* extends δ to strings: $\delta^*(q, w)$ is the set of states reachable from state q on input string w.



- In the NFA shown in the figure what is $\delta^*(q, 0)$?

Solution: $\delta^*(q, 0) = \{u, v, t, w\}$. ■

- Describe an efficient algorithm that takes three inputs, a description of an NFA $N = (Q, \Sigma, \delta, s, F)$, a state $q \in Q$, and a symbol $a \in \Sigma$, and computes $\delta^*(q, a)$ (in other words the set of all states that can be reached from q on input a which is now interpreted as a string). You may want to first think about how to compute $\delta^*(q, \epsilon)$ (the ϵ-reach of state q). Express the running time of your algorithm in terms of n the number of states of N and $m = \sum_{p \in Q} \sum_{b \in \Sigma \cup \{\epsilon\}} |\delta(p, b)|$ which is the natural representation size of the NFA's transition function.

Solution: We will create a graph consisting of two copies of the states, representing the number of a-transitions taken. Edges within each layer will correspond to ε-transitions; edges between the two layers will correspond to a-transitions. Accordingly,

$$\begin{aligned} V &:= Q \times \{0, 1\} \\ E &:= \big\{(p, i) \to (q, i) \;\big|\; i \in \{0, 1\}, q \in \delta(p, \varepsilon)\big\} \\ &\quad\; \cup \big\{(p, 0) \to (q, 1) \;\big|\; q \in \delta(p, a)\big\} \end{aligned}$$

which can be built in $O(m + n)$ time by brute force.

Then $\delta^*(q, a)$ corresponds to vertices in $\text{rch}((q, 0)) \cap (Q \times \{1\})$, which can be found in $\boldsymbol{O(m + n)}$ time via Basic (aka Whatever-First) Search followed by some filtering. ■

- Describe an efficient algorithm that takes two inputs, a description of a NFA $N = (Q, \Sigma, \delta, s, F)$, and a string $w \in \Sigma^*$, and outputs whether N accepts w. Express your running time as a function of $\ell = |w|$ and n and m as in the preceding part.

Solution: We will first generalize the algorithm from the previous part to be able to compute $\delta^*(X, a) = \bigcup_{q \in X} \delta^*(q, a)$ for some set of states $X \subseteq Q$. Naïvely, we can do this in $O(n(m+n))$ time by running Basic Search once in the graph from each $q \in X$, but we can do better. We add a new vertex s^* with edges $s^* \to (q, 0)$ for each $q \in X$. Then $\delta^*(X, a)$ corresponds to vertices in $\text{rch}(s^*) \cap (Q \times \{1\})$, which (as before) can be computed in $O(m+n)$ time.

We can also adapt the algorithm to compute $\varepsilon\text{reach}(X) = \delta^*(X, \varepsilon)$ for $X \subseteq Q$. This is simple enough: we simply omit the second layer, i.e., take the graph consisting of the states and only the ε-transitions, then add a new vertex s^* with edges $s^* \to q$ for each $q \in X$. Then $\varepsilon\text{reach}(X) = \text{rch}(s^*)$. Once again the running time is $O(m+n)$.

To determine if N accepts w, we compute $\delta^*(s, w)$ by stepping through w character by character via the identity $\delta^*(q, ax) = \delta^*(\delta^*(q, a), x)$, and then checking if the result contains an accepting state.

```
AcceptNFA($N, w[1..\ell]$):
  $X \leftarrow \varepsilon\text{reach}(\{s\})$ ⟨⟨ $= \delta^*(\{s\}, \varepsilon)$ ⟩⟩
  for $i$ from 1 to $\ell$
      $X \leftarrow \delta^*(X, w[i])$
  if $F \cap X \neq \varnothing$
      return "ACCEPT"
  else
      return "REJECT"
```

Why do start with $X \leftarrow \varepsilon\text{reach}(\{s\})$ instead of simply $X \leftarrow \{s\}$? Well, consider the case where $w = \varepsilon$, i.e., $\ell = 0$. Then starting with $X \leftarrow \{s\}$ instead of $X \leftarrow \varepsilon\text{reach}(\{s\})$ may result in incorrectly reporting that N rejects $w = \varepsilon$!

If we insist that Q is a set of indices like we do for graphs, the F is a set/list of indices. Thus we can check if $F \cap X \neq \varnothing$ in $O(n)$ time by enumerating over a constant number of arrays if necessary. The overall running time is dominated by the computations of $\delta^*(X, w[i])$, for a total of ***$O(\ell(m+n))$ time***.

Some commentary. In some sense this is the idea behind the incremental subset construction seen in the first part of the course. One can imagine that instead of a for-loop over some string w, we keep a queue of state sets X to process. Then while the queue is non-empty, we pop a state set X off the queue, and for each $a \in \Sigma$, compute $\delta^*(X, a)$ and add it to the queue if we have not seen it before. ■

Solution: We will generalize the solution from the previous part to be able to compute $\delta^*(q, w)$ for $w \in \Sigma^*$ of length ℓ, and use it to see if $\delta^*(s, w)$ contains any accepting states.

We will create a graph consisting of $\ell + 1$ copies of the states, numbered 0 through ℓ, representing the number of symbols of w read so far. Edges within each layer will correspond to ε-transitions; edges between the $(i-1)$-th layer and the i-th layer will correspond to $w[i]$-transitions. Accordingly,

$$\begin{aligned}V &:= Q \times \{0, 1, \dots, \ell\}\\ E &:= \big\{(p,i)\to(q,i) \bigm| 0 \le i \le \ell, q \in \delta(p,\varepsilon)\big\}\\ &\quad \cup \big\{(p,i-1)\to(q,i) \bigm| 1 \le i \le \ell, q \in \delta(p,w[i])\big\}\end{aligned}$$

which can be built in $O(\ell(m+n))$ time by brute force.

We need to check if $\delta^*(s,w) \cap F = \varnothing$, which is equivalent to checking $\mathrm{rch}((s,0)) \cap (F \times \{\ell\}) = \varnothing$. This set can be found in $O(V+E) = \boldsymbol{O(\ell(m+n))}$ time via Basic (aka Whatever-First) Search followed by some filtering. ■

Rubric: 10 points.

- 2 point for the first part.
- 4 points each for the other parts:
  - 3 points for the algorithm:
    - 2 points max for a slower polynomial-time algorithm, 1 point max for an exponential-time algorithm.
    - −1 for minor errors, −2 for major errors.
  - 1 point for running time analysis.

3. In number theory Oppermann's conjecture states the following: For every $n > 1$ there is at least one prime number between $n(n-1)$ and n^2 and at least one prime number between n^2 and $n(n+1)$. This conjecture is still open although it was postulated in 1877. The goal of this problem is to show why being able to solve the Halting problem can easily resolve such open problems.

   - Write pseudocode for a program CheckOppermann($int\ n$) that checks whether Oppermann's conjecture is true or false for a given integer n. You can use a subroutine IsPrime($int\ n$) that checks whether an integer n is a prime number (you can write one yourself if you want). In writing this routine you are not concerned about efficiency but your program should terminate and give the correct answer.

Solution: Assuming a black-box subroutine IsPrime(n), the following is an implementation of CheckOppermann:

```
CheckOpperman($n$):
  $firstPrimeExists \leftarrow$ False
  for $i$ from $n(n-1)$ to $n^2$
      $firstPrimeExists \leftarrow firstPrimeExists$ or IsPrime($i$)
  $secondPrimeExists \leftarrow$ False
  for $j$ from $n^2$ to $n(n+1)$
      $secondPrimeExists \leftarrow secondPrimeExists$ or IsPrime($j$)
  return $firstPrimeExists$ and $secondPrimeExists$
```

Let $P(n)$ denote the running time of IsPrime(n), and $M(k)$ denote the running time of multiplying two k-digit numbers. It is known that IsPrime(n) can be implemented to run in $O((\log n)^c)$ for some constant c, e.g., via the AKS primality test. Furthermore, multiplying two k-digit numbers can be implemented to run in $O(k^{\log_2 3})$ time via Karatsuba's method, or $O(k \log k)$ time via the recent result of Harvey and van der Hoeven. Once we have computed $n(n-1)$, n^2, and $n(n+1)$ in $O(M(\log n))$ time, each number i between $n(n-1)$ and $n(n+1)$ takes $O(\log i) = O(\log n)$ time to compute and write down, and $P(i)$ time to check primality. Overall the running time is $\boldsymbol{O\left(M(\log n) + n\log n + \sum_{i=n(n-1)}^{n(n+1)} P(i)\right)}$. Via the implementations mentioned above, this simplifies to $O(n(\log n)^c)$ for some constant c. Note that the running time is exponential in the input size. ■

   - Using the program in the preceding part as a sub-routine, write pseudocode for a program OppermannConj() that does not take any input and tries all n in a sequence looking for a counter example. Your program should have the feature that it will halt (if run on a computer) if and only if Oppermann's conjecture is false.

Solution: The algorithm halts if and only if Oppermann's conjecture is false. The running time is ***unknown***, as Oppermann's conjecture is open.

```
OppermannConj():
  $i \leftarrow 2$
  while True
      if not CheckOppermann($i$)
          return "$i$ is a counterexample"
      $i \leftarrow i + 1$
```

■

- Suppose there was a program P that given another program Q can always answer correctly whether Q will halt or not. How can you check whether Oppermann's conjecture is true or not by using P and the preceding part?

  Solution: Suppose P exists. Then Oppermann's conjecture holds if and only if P reports "does not halt" when given the program written in the previous part. So we would be able solve the conjecture by simply running P on said program. ■

Rubric: 10 points.

- 4 points each for the first two parts:
  - 3 points for a correct algorithm. −1 for minor errors, −2 for major errors.
  - 1 point for any semblance of a reasonable time analysis (this includes any variant of "unknown" for the second part).
- 2 points for the last part.