Homework 8

CS/ECE 374 B

Due 8 p.m. Tuesday, November 19

Question 1: ... or die trying . . . .

(This question is the same as the textbook question 6(a)(b).)

(5) (a) Describe and analyze a modification of Bellman–Ford that actually returns a negative cycle if any such cycle is reachable from s, or a shortest-path tree if there is no such cycle. The modified algorithm should still run in $O(VE)$ time.

Solution:

```
def BellmanFordTreeOrNegativeCycle(s):
    pred(s) = NULL
    for all vertices v:
        dist(v) = inf
    dist(s) = 0
    for i = 1 to 2*V:
        for all edges u—>v:
            if dist(v) > dist(u) + w(u—>v)
                dist(v) = dist(u) + w(u—>v)
                pred(v) = u

    #check for edge we can still relax ——> negative cycle
    for all edges u—>v:
        if dist(v) > dist(u) + w(u—>v):
            #negative cycle
            add v, u to set C
            while pred(u) is not v:
                add pred(u) to C
                u = pred(u)
            return reverse(C)

    #else no negative cycle
    #create new graph G' consisting of vertices from G
    for vertex u in G:
        if u != s:
            add edge pred(u) —> u in G'␣with␣weight␣w(pred(u)—>u)␣from␣G
␣␣␣␣return␣G'
```

This algorithm uses Bellman Ford, (2*|V| loops to ensure predecessor pointers are all set to create a cycle) which is $\Theta(VE)$, then loops over edges which is $\Theta(E)$, and possibly then follows predecessor pointers to compute a path of vertices which is $\Theta(V)$ when we use a dictionary or constant time lookup (like a set in python) for checking if pred(u) is in L. Thus if there is a negative cycle, the runtime is still $\Theta(VE)$. Otherwise we must create a new graph of vertices, and for each vertex, possibly add an edge. This costs $\Theta(V)$ to create the graph. Looping over vertices and checking to find edge pred(u)->u if it exists costs $\Theta(VE)$ since we loop over vertices and possibly have to check every edge to find the pred(u)->u. Overall, the runtime is still $\Theta(VE)$ ■

(5) (b) Describe and analyze a modification of Bellman-Ford that computes the correct shortest path distances from s to every other vertex of the input graph, even if the graph contains negative cycles. Specifically, if any walk from s to v contains a negative cycle, your algorithm should end with dist(v) $= \infty$; otherwise, dist(v) should contain the length of the shortest path from s to v. The modified algorithm should still run in $O(VE)$ time.

Solution:

```
def ShortestPathsWithNegativeCycles(s):
    pred(s) = NULL
    for all vertices v:
        dist(v) = inf
    dist(s) = 0
    for i = 1 to V—1:
        for all edges u—>v:
            if dist(v) > dist(u) + w(u—>v)
                dist(v) = dist(u) + w(u—>v)
                pred(v) = u

    #check for edge we can still relax ——> negative cycle
    L = []
    for all edges u—>v:
        if dist(v) > dist(u) + w(u—>v):
            #node is part of negative cycle
            add v to list L

    #For each node we need to find all nodes it is connected to and set to inf.
    for v in L:
        run DFS(v) or BFS(v) and mark visited nodes with dist(v) = inf
```

This algorithm uses Bellman Ford initially, which is $\Theta(VE)$. Then we loop over edges and add vertices to a set L. This costs $\Theta(E)$. Then for each vertex marked as part of a negative cycle, we run DFS/BFS on it to find what it is connected to and mark distance as ∞. The worst case would be if we must run a search from each vertex, but since we mark vertices as visited, the runtime is a linear search, thus still bounded by $\Theta(VE)$. Since Bellman Ford correctly computes the distance for vertices without a negative cycle on the path, we have set dist() correctly. Thus the overall runtime is still $\Theta(VE)$ ■

Question 2: Bus schedules..

(This is question 24 in the textbook.)

You are given a directed graph G that represents bus travel in a city. For an edge $u \to v$, $\ell(u,v)$ is the amount of time it takes for the bus to travel from bus stop u to bus stop v. You are also given two numbers for each stop to determine the schedule: $f(u,v)$ is the first time (in minutes past 12 noon) that the bus leaves from stop u towards stop v, and $\Delta(u,v)$ is the time between such buses. In other words, a bus leaves from stop u to stop v at $f(u,v)+k\cdot\Delta(u,v)$ for $k \geq 0$, and arrives at stop v at times $f(u,v)+k\cdot\Delta(u,v)+\ell(u,v)$. Assume that $\ell(u,v), f(u,v)$, and $\Delta(u,v)$ are all positive numbers.

You are given a graph G and the values ℓ, f, and Δ for every edge in G. You are also given the starting stop s and the destination stop t, and the starting time t_0. You need to design and analyze an efficient algorithm to determine the earliest time you can arrive from s to t starting at time t_0. Note that you can change buses at a stop if you arrive at a time that is less than or equal to the departure time of the bus. That is:

$$f(u,v)+k_1\cdot\Delta(u,v)+\ell(u,v) \leq f(v,w)+k_2\cdot\Delta(v,w)$$

Solution:

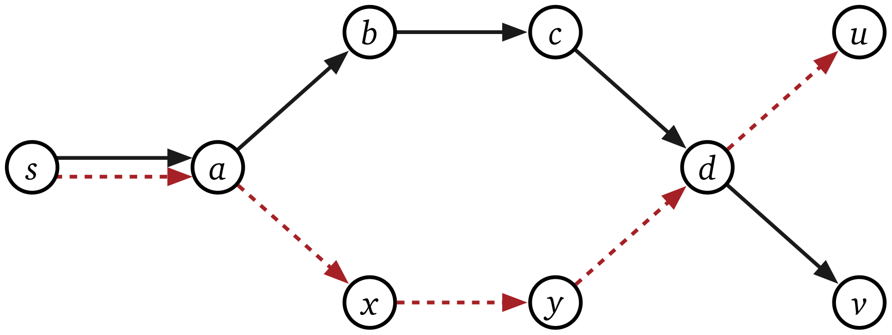
```
from queue import PriorityQueue
from math import ceil

def BusTimes(s,t0):
    """This function should be invoked as BusTimes(s,t0). The time to get to t will then be in dist[t].
    For simplicity we assume that vertices in V are numbered and, by extension, that edges in E are
    identified as tuples of numbers (u,v). We also assume that there are global lists 'f', 'delta', and 'l'
    that are indexed by tuples of vertices. All times are assumed to be minutes past 12 noon."""
    # set up distances and predecessors
    dist = {v: inf for v in V}
    pred = {v: None for v in V}
    dist[s] = t0
    pq = PriorityQueue()
    for v in V:
        pq.put((v,dist[v]))
    while(!pq.empty()):
        (u,dist[u]) = pq.get()
        for (u,v) in E[u]:
            # Solve f[(u,v)]+k*delta[(u,v)]=dist[u] for k, then ceiling that
            number_of_elapsed_deltas = (dist[u]—f[(u,v)])/delta[(u,v)]
            if(number_of_elapsed_deltas < 0): # the first bus time to v is after our arrival at u
              number_of_elapsed_deltas = 0
            time_of_next_bus = f[(u,v)]+ceil(number_of_elapsed_deltas)*delta[(u,v)]
            if(time_of_next_bus+l[(u,v)] < dist[v]):
                dist[v] = time_of_next_bus+l[(u,v)]
                pred[v] = u
                pq.decreaseKey(v,dist[v])
```

This algorithm is an adaptation of Dijkstra's algorithm in the case where it is guaranteed that all edges are of non-negative length. At each vertex u, for each edge $\overrightarrow{uv}$ out of that vertex, we must consider not just the time taking the bus from u to v, but also the time required to wait for the next bus upon arriving at u. Because the next bus arrives every $\Delta(u,v)$ minutes (after $f(u,v)$, that is), it suffices to calculate the number of bus departure intervals that have already passed so that we can identify the time of the next bus to depart from u to v.

These arithmetic operations are run each time for each edge and can be considered essentially constant in cost, leaving the overall time complexity of this algorithm as that of Dijkstra's ($\Theta(E+V\log(V))$). ■

Question 3: Internet paths ....

On the Internet, routing paths follow a local preference rule: each Internet service provider (ISP) makes a local decision between possible routes based on its preferences, usually guided by complex business decisions. These preferences may depend not just on the previous hop but the entire path to the destination. For example, in the graph below, there are two paths from s to d. The path that is chosen is based on d's local preference, which can be computed by a function pref(v,p) where v is the identity of the graph node where the path terminates, p is a path from s to v. The function returns a number such that pref$(v,p_a) >$ pref(v,p_b) means that path p_a is preferred to path p_b. In the graph below, to decide which path to use from s to d you would need to compare pref$(d,(s,a,b,c,d))$ and pref$(d,(s,a,x,y,d))$.



Note that each ISP (i.e., nodes in the graph) chooses only which of the incoming edges to use for its path. Therefore, if d chooses the red path (s,a,x,y,d) then the only path available to u is (s,a,x,y,d,u). In other words, the only decision to be made in this graph is at d, and therefore the set of paths taken through the graph can be made into a preferred path tree.

(6) (a) Design and analyze an efficient algorithm for computing the preferred path tree according to a given pref function. You are given a directed, unweighted graph $G = (V,E)$ as input and a source node s. You also have access to a local preference function pref. In this part, you should only use the pref function to decide between paths that are of equal length. You should assume that preftakes time proportional to the size of its arguments (i.e., $O(|p|)$, the length of the path).

Your result should be a tree T that satisfies the following properties:

- The path from s to v in the preferred path tree T, path(T,s,v), is a shortest path from s to v
- If $v \to u$ is an edge in T and $w \to u$ is another incoming edge to u, then either:
  - path(T,s,w) is longer than path(T,s,v), or
  - pref$(u,$ path$(T,s,u))$ is higher than pref$(u,$ path$(T,s,w) + w \to u)$

Solution: Since the input graph is unweighted, we can use BFS to construct the shortest path tree. The only difference is that when visiting vertex u, an edge $u \to v$ should be relaxed only if either $d(u)+1 < d(v)$, or $d(u)+1 = d(v)$ and $pref(v, path(T,s,u) + u \to v) > pref(v, path(T,s,v))$.

```
def q3a(G,s,pref):
    """ G is a directed graph, stored as a dictionary from nodes to a list of neighbors
    pref is  a function that takes a node and a path and returns a number
    """
    for all vertices v:
        dist(v) = inf
        path(v) = []
    dist(s) = 0
    q = Queue()
    q.push(s)
    while(!q.empty()):
        u = q.dequeue()
```

```
for every edge u —> v:
    if d(u) + 1 < d(v):
        dist(v) = d(u) + 1
        path(v) = path(u) + v
        q.enqueue(v)
    elif d(u) + 1 == d(v) and pref(v,path(v)) < pref(v,path(u)+v):
        path(v) = path(u) + v
        q.enqueue(v)
```

Since each vertex is added to the queue at most the number of times as its in-degree, and each call to pref(v,p) takes $O(V)$ time, the total running time of this algorithm is $O(VE^2)$.

■

(4) (b) Design an algorithm for computing the preferred path tree where longer paths may be preferred. Again, you are given a directed, unweighted graph G, a source node s, and access to the pref function. The only constraint is that you cannot select paths that would result in a loop.) Your algorithm should return a preferred path tree T with the following property: if $v \to u$ is an edge in T and $w \to u$ is another incoming edge to u then either:

- $\text{pref}(u, \text{path}(T, s, u)) > \text{pref}(u, \text{path}(T, s, w) + w \to u)$, or
- $\text{path}(T, s, w)$ traverses u (i.e., adding $w \to u$ to the tree would create a loop)

You do not have to analyze the runtime complexity of this algorithm.

Solution:

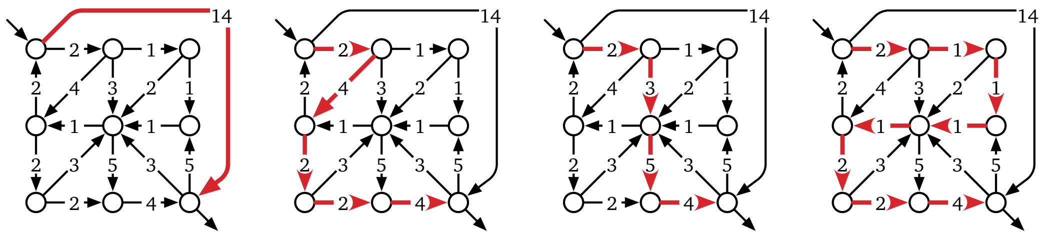
```
def q3b(G, s, pref):
    """ G is a directed graph, stored as a dictionary from nodes to a list of neighbors
    pref is  a function that takes a node and a path and returns a number
    """
    paths = {}
    changed = True
    paths[s] = [s]
    while changed:
        changed = False
        for v, nlist  in G.keys():
            # iterate over edges v—>u
            for u in nlist:
                if  u == s: # skip source
                    continue
                if  u not in paths: # no route to u yet
                    continue
                if  (v not in paths or # no route to v yet
                    (v not in paths[u] and # no loop created
                     pref(v, paths[u]+[u]) > pref(v, path[v]))): # this path is preferred
                    paths[v] = paths[u]+[u] # adopt this path
                    changed = True
```

■

Solved Problem

Question 4: 4 . . . .

Although we typically speak of "the" shortest path between two nodes, a single graph could contain several minimum-length paths with the same endpoints.



Four (of many) equal-length shortest paths.

Describe and analyze an algorithm to determine the <u>number</u> of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that all necessary arithmetic operations can be performed in $O(1)$ time.

[Hint: Compute shortest path distances from s to every other vertex. Throw away all edges that cannot be part of a shortest path from s to another vertex. What's left?]

Solution: We start by computing shortest-path distances $\underline{dist}(v)$ from s to v, for every vertex v, using Dijkstra's algorithm. Call an edge $u \to v$ **<u>tight</u>** if $\underline{dist}(u) + w(u \to v) = \underline{dist}(v)$. Every edge in a shortest path from s to t must be tight. Conversely, every path from s to t that uses only tight edges has total length $\underline{dist}(t)$ and is therefore a shortest path!

Let H be the subgraph of all tight edges in G. We can easily construct H in $O(V + E)$ time. Because all edge weights are positive, H is a directed acyclic graph. It remains only to count the number of paths from s to t in H.

For any vertex v, let $\underline{PathsToT}(v)$ denote the number of paths in H from v to t; we need to compute $\underline{PathsToT}(s)$. This function satisfies the following simple recurrence:

$$\underline{PathsToT}(v) = \begin{cases} 1 & \text{if } v = t \\ \displaystyle\sum_{v \to w} \underline{PathsToT}(w) & \text{otherwise} \end{cases}$$

In particular, if v is a sink but $v \neq t$ (and thus there are no paths from v to t), this recurrence correctly gives us $\underline{PathsToT}(v) = \sum \varnothing = 0$.

We can memoize this function into the graph itself, storing each value $\underline{PathsToT}(v)$ at the corresponding vertex v. Since each subproblem depends only on its successors in H, we can compute $\underline{PathsToT}(v)$ for all vertices v by considering the vertices in reverse topological order, or equivalently, by performing a depth-first search of H starting at s. The resulting algorithm runs in $O(V + E)$ time.

The overall running time of the algorithm is dominated by Dijkstra's algorithm in the preprocessing phase, which runs in <u>$\boldsymbol{O(E \log V)}$ **time**</u>. ■

Rubric: 10 points = 5 points for reduction to counting paths in a dag + 5 points for the path-counting algorithm (standard dynamic programming rubric)