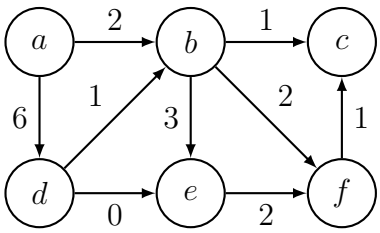
# Problem Set 6

Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct.

---

**Problem 6-1.** [15 points] **Dijkstra Practice**

Consider weighted graph $G = (V, E, w)$ below, which is acyclic and has nonnegative edge weights.




**(a)** Run both DAG Relaxation and Dijkstra on $G$ from vertex $a$. Each algorithm tries to relax every edge $(u, v) \in E$, but does so in a different order. For each algorithm, write down a list of all edges in **relaxation order**: the order the algorithm tries to relax them. If there is ambiguity on which vertex or outgoing adjacency to process next, process them in **alphabetical order**.

**(b)** List $\delta(a, v)$ for each $v \in V$ (via either algorithm).

**Solution:** Relaxation order is unique for both algorithms.

DAG Relaxation: $[(a,b), (a,d), (d,b), (d,e), (b,c), (b,e), (b,f), (e,f), (f,c)]$

Dijkstra: $[(a,b), (a,d), (b,c), (b,e), (b,f), (f,c), (e,f), (d,b), (d,e)]$

| $v$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|---|---|---|---|---|---|---|
| $\delta(a, v)$ | 0 | 2 | 3 | 6 | 5 | 4 |

**Rubric:**

- 5 points for each relaxation order (2)
- 5 points for shortest path distances
- -1 point for each mistake (addition, omission, inversion)
- Minimum 0 points for each of three parts

**Problem 6-2.** [10 points] **Short Circuits**

Given a weighted directed graph $G = (V, E, w)$ and vertex $s \in V$, with the property that, for every vertex $v \in V$, some minimum-weight path from $s$ to $v$ traverses at most $k$ edges, describe an algorithm to find the shortest-path weight from $s$ to each $v \in V$ in $O(|V| + k|E|)$ time.

**Solution:** Since this graph may contain cycles and negative weights, at first it seems we do not know how to do better than Bellman-Ford. Fortunately, the problem gives us an additional restriction: that a shortest path to each vertex traverses few edges. Since layer $k'$ of the duplicated

graph $G'$ of Bellman-Ford corresponds to shortest paths to that vertex using at most $k'$ edges, our approach will be to modify Bellman-Ford to only construct the duplicated graph layers incrementally, relax edges into the new layers online to compute $k'$-edge shortest paths, and then stop when $k'$-edge shortest path distance do not change from layer to layer.

Specifically, rather than creating the duplicated graph $G'$ all at once and then running DAG Relaxation, we begin with $G'$ only containing the first layer ($v_0$ for $v \in V$) and $\delta$ initialized with first-layer distances $\delta(s_0, v_0) = \infty$ for $v \in V$, except for $\delta(s_0, s_0) = 0$.

Then for $i$ starting at 0, assume for induction that $G'$ contains layers 0 to $i$ and $\delta(s_0, v_j)$ have been computed for all $v \in V$ and $j \in \{0, \ldots, i\}$, append layer $i + 1$ by adding vertices $v_{i+1}$ for $v \in V$ and associated edges from layer $i$. This new layer cannot effect distances in previous layers (since the new layer comes later in the topological order. So, continue DAG Relaxation by relaxing the new edges out of the vertices in layer $i$, which correctly computes $\delta(s_0, v_{i+1})$ for all $v \in V$.

Since a shortest path from $s$ to each vertex uses at most $k$ edges, the distance $\delta(s, v)$ equals the $k'$-edge distance $\delta_{k'}(s, v) = \delta(s_0, v_{k'})$ for every $k' \geq k$. Thus, as soon as $\delta(s_0, v_{k'}) = \delta(s_0, v_{k'+1})$ for all $v \in V$, then $k' = k$ and we can terminate by outputting $\delta(s, v) = \delta(s_0, v_k)$ for each $v \in V$.

Since every vertex is reachable from $s$, $|V| = O(|E|)$, and since we only construct $k + 1$ layers of $G'$ before finding two equal layers, and each layer takes at most $O(|E|)$ time to process, this algorithm runs in $O(k|E|)$ time (which is also $O(|V| + k|E|)$). The $|V|$ was originally provided for this problem to allow the input to include disconnected graph, where an initial single-source reachability algorithm could restrict to the subset of the graph reachable from $s$; but this term is no longer relevant for connected input graphs.

**Rubric:**

- 6 points for description of a correct algorithm that runs in $O(k|E|)$ time
- 2 points for a correct argument of correctness
- 2 points for a correct argument of running time
- Partial credit may be awarded
- Max 4 points for a correct $O(|V||E|)$-time algorithm

**Problem 6-3.** [20 points] **Dynamite Detonation**

Superspy Bames Jond is fleeing the alpine mountain lair of Silvertoe, the evil tycoon. Bames has stolen a pair of skis and a trail map listing the mountain's clearings and slopes ($n$ in total), and she wants to ski from the clearing $L$ by the lair to a clearing $S$ where a snowmobile awaits.

- Each **clearing** $c_i \in C$ has an integer **elevation** $e_i$ above sea level.
- Each **slope** $(c_i, c_j, \ell_{ij})$ connects a pair of clearings $c_i$ and $c_j$ with a **monotonic** trail (strictly decreasing or increasing in elevation) with positive integer length $\ell_{ij}$. Bames doesn't have time to ski uphill, so she will only traverse slopes so as to decrease her elevation.

- On her way up the mountain, Bames laced the mountain with **dynamite** at known locations $C_D \subset C$. Detonating the dynamite will immediately change the elevation of clearings $c_i \in C_D$ by a known amount, from $e_i$ to a lower elevation $e'_i < e_i$. The **detonator** exists at clearing $D \in C$ (where there is no dynamite). If she reaches clearing $D$, she can choose to detonate the dynamite before continuing on.

Given Bames' map and dynamite data, describe an $O(n)$-time algorithm to find the minimum distance she must ski to reach the snowmobile (possibly detonating the dynamite along the way).

**Solution:** We can model the mountain as a DAG of trails that Bames may ski downhill, both before and after possible detonation. Construct a graph $G_1$ with a vertex for every clearing and a directed edge for every slope pointing to the lower clearing; remove any slopes that do not change height along the way (since Bames will not traverse the slope in either direction). Construct a second graph $G_2$ in the same way as $G_1$, except using the modified heights $e'_i$ that represents the state of the mountain after the dynamite explodes. In both graphs, we weight edges by the length $l_{ij}$ of the corresponding slope. Connect $G_1$ and $G_2$ into a single graph $G$ by adding one directed edge from node $D_1$ in graph $G_1$ to node $D_2$ in graph $G_2$ with weight $0$ (taking this edge represents blowing up the dynamite); and add a supernode $T$ that has incoming edges of zero weight from nodes $S_1$ in graph $G_1$ and $S_2$ in graph $G_2$ (representing reaching $S$ by either not detonating or detonating the dynamite respectively).

We can now run DAG Relaxation from node $L_1$ in graph $G$ to the supernode $T$; the shortest such path is the minimum distance to get from $L$ to $S$, with or without detonation, while respecting that Bames only decreases her elevation. Graph $G$ has $O(n)$ vertices and edges, so DAG Relaxation runs in $O(n)$ time, as desired.

**Rubric:**

- 6 points for construction of a graph that can be used to solve the problem
- 3 points for description of a correct algorithm
- 2 points for a correct argument of correctness
- 3 points for a correct argument of running time
- 6 points if correct algorithm is efficient, i.e., $O(n)$-time
- Partial credit may be awarded

**Problem 6-4.** [10 points] **Conservative Cycles**

In Wentonian physics, a force field acts on the motion of a particle by requiring a certain positive or negative amount of **work** to move along any path. A force field is **conservative** if the total sum of work is zero **along every closed loop**. Given a discrete force field represented by $n$ possible particle locations $\ell_i$ and $O(n)$ possible particle transitions[1], where transition $(\ell_i, \ell_j, w_{ij})$ moves a particle along a **directed** path from location $\ell_i$ to location $\ell_j$ requiring (positive or negative) integer work $w_{ij}$, describe an $O(n^2)$-time algorithm to determine whether the force field is conservative.

[1]Note that a transition from $\ell_i$ to $\ell_j$ does not imply the existence of a transition from $\ell_j$ to $\ell_i$.

**Solution:** Construct a graph $G$ with a vertex for each particle location $\ell_i$ and a directed edge $(\ell_i, \ell_j)$ of weight $w_{ij}$ for each transition $(\ell_i, \ell_j, w_{ij})$. We can refute that the force field is conservative by finding a cycle in $G$ that has non-zero weight. Add a supernode $s$ connected to every $v \in V$ with a directed zero-weight edge $(s, v)$ (which does not add any new cycles to the graph). Then, run Bellman-Ford from $s$ to determine whether there are any negative-weight cycles; then negate all edges weights and run Bellman-Ford from $s$ again to determine whether there are any negative-weight cycles (positive-weight cycles in the original graph). If Bellman-Ford detects either type of cycle, we can return that the force is not conservative (since we have found a cycle with non-zero weight); otherwise no non-zero weight cycles exist, so we can return that the field is conservative. Since we only add one node and $O(|n|)$ edges to a graph with $n$ vertices and $O(n)$ edges, Bellman-Ford will take $O(n^2)$ time, as desired.

Note that it is possible to solve this problem in $O(n)$ time using algorithms we have not learned in this class, but this is beyond the scope of this class. Namely:

- identify the strongly connected components (SCCs) of $G$ in $O(n)$ time via a modified DFS;
- use DFS within each SSC to assign shortest-path length potentials to each vertex;
- and then check each edge of each SCC against these potentials in $O(n)$ time.

**Rubric:**

- 3 points for description of a correct algorithm
- 2 points for a correct argument of correctness
- 2 points for a correct argument of running time
- 3 points if correct algorithm is efficient, i.e., $O(n^2)$
- Partial credit may be awarded

**Problem 6-5.** [20 points] **SparkPlug Derby**

LightQueen McNing is a race car that wants to drive to California to compete in a road race: the annual SparkPlug Derby. She has a map depicting:

- the $n$ intersections in the country, where each intersection $x_i$ is marked with its positive integer **elevation** $e_i$ and whether it contains a **gas station**; and
- the $r$ roads connecting pairs of them, where each road $r_j$ is marked with the positive integer $t_j$ denoting the **driving time** it will take LightQueen to drive along it in either direction.

Some intersections connect to many roads, but the average number of roads at any intersection is less than $5$. LightQueen needs to get from Carburetor Falls at intersection $s$ to the SparkPlug Derby race track at intersection $t$, subject to the following conditions:

- LightQueen's gas tank has a positive integer **capacity** $g < n$: it can hold up to $g$ units of gas at a time (starting full). Along the way, she can refill her tank (by any integer amount) at any intersection marked with a gas station. It takes exactly $t_G$ time for her to fill up $1$ unit of gas.

- LightQueen uses gas only when **driving uphill**. Specifically, if she drives on a road from intersection $x_i$ to $x_j$ at elevations $e_i$ and $e_j$ respectively, LightQueen will use exactly $e_j - e_i$ units of gas to travel along it if $e_j > e_i$, and will use zero units of gas otherwise.

Given LightQueen's map, describe an $O(n^2 \log n)$-time algorithm to return a fastest route to the race that keeps a **strictly positive** amount of gas in her tank at all times (if such a route exists).

**Solution:** Create a graph $G$:

- **Vertices**: for each intersection $v$, add a vertex $v_i$ for every $i \in \{1, \ldots, g\}$, where vertex $v_i$ represents being at intersection $v$ with $i$ units of gasoline in her tank; and
- **Edges**: for each bi-directional road $\{u, v\}$ connecting intersections $u$ and $v$ with driving time $t(u, v)$ (let $e(u)$ and $e(v)$ be the elevations of intersections $u$ and $v$ respectively, and without loss of generality assume $e(u) \leq e(v)$), construct the following directed edges:
  1. (Down-hill Road Edges)
     - add a weight-$t(u, v)$ edge $(v_i, u_i)$ for every $i \in \{1, \ldots g\}$ (since traveling in this direction on the road requires no gas).
  2. (Up-hill Road Edges)
     - if $e(u) == e(v)$, add another weight-$t(u, v)$ edge opposite the previous, specifically: $(u_i, v_i)$ for every $i \in \{1, \ldots, g\}$;
     - otherwise $e(v) - e(u) = \ell > 0$, so add a weight-$t(u, v)$ edge $(u_i, v_{i-\ell})$ for every $i \in \{\ell + 1, \ldots, g\}$ (since traveling uphill costs gas, and she is not allowed to ever run out of gas).
  3. (Gas Station Edges)
     - for each gas station, e.g., at intersection $v$, add a weight-$t_G$ edge $(v_i, v_{i+1})$ for every $i \in \{1, \ldots, g - 1\}$ (corresponding to filling up the tank by one unit at the station).

There are $gr$ Down-hill Road Edges, at most $gr$ Up-hill Road Edges, and at most $(g - 1)n$ Gas Station Edges, so graph $G$ has $gn$ vertices and $O(g(r + n))$ edges. This graph has the property that any path from vertex $s_g$ to any vertex $t_i$ for $i \in \{1, \ldots, g\}$ will be a valid route for LightQueen to leave Carburetor Falls with a full tank and reach SparkPlug Derby without ever running out of gas along the way; and the path's weight corresponds to the time it would take her to get there. So solve SSSP from $s_g$ to find a minimum weight path to each $t_i$ for $i \in \{1, \ldots, g\}$, and return a path to one with minimum weight (or return no such path exists). This graph may contain cycles, but has only positive edge weights, so solve SSSP using Dijkstra.

Since the average degree of each intersection is $< 5$ and every road connects two intersections, then $2r < 5n$ and $r = O(n)$. Since $g < n$ by the problem statement, $G$ has $O(n^2)$ vertices and $O(n^2)$ edges, so takes $O(n^2)$ time to construct. Then Dijkstra takes $O(n^2 \log(n^2) + n^2) = O(n^2 \log n)$ time, while finding the minimum path to any $t_i$ takes $O(n)$ time, leading to $O(n^2 \log n)$ time in total, as desired.

**Rubric:**

- 6 points for construction of a graph that can be used to solve the problem
- 3 points for description of a correct algorithm
- 2 points for a correct argument of correctness
- 3 points for a correct argument of running time
- 6 points if correct algorithm is efficient, i.e., $O(n^2 \log n)$-time
- Partial credit may be awarded

**Problem 6-6.** [25 points] **Johnson's Algorithm**
In this problem, you will implement Johnson's Algorithm to compute all-pairs shortest-path weights in a general weighted directed graph $G$, as described in Lecture 14. The input to your algorithm will be: a positive integer $n$ representing the number of vertices of your graph identified by consecutive integers 0 to $n-1$, and a tuple $S$ of triples, where each triple $(u, v, w)$ corresponds to a directed edge from vertex $u$ to $v$ of weight $w$. Your output should be a length-$n$ tuple $D$ of length-$n$ tuples where $D[u][v] = \delta(u, v)$ for all $u, v \in \{0, \ldots, n-1\}$, except when the input graph contains a **negative-weight cycle** when you should return `None`.

Please implement the `johnson(n, S)` function in the template code provided. Your code template contains working implementations of Bellman–Ford and Dijkstra (using a binary heap as a priority queue) modified from the recitation notes. Note that you will have to construct your own adjacency list and weight function if you would like to use this code. You can download the code template and some test cases from the website.

**Solution:**

```
INF = 99999     # distance magnitudes will not be larger than this number

def johnson(n, S):
    D = [[INF for _ in range(n)] for _ in range(n)]
    Adj = [[] for _ in range(n)]        # construct graph
    W = {}
    for (u, v, w) in S:
        Adj[u].append(v)
        W[(u, v)] = w
    Adj.append([i for i in range(n)])   # run bellman-ford from supernode
    for i in range(n):
        W[(n, i)] = 0
    def wf(u, v):   return W[(u, v)]
    args = bellman_ford(Adj, wf, n)
    if args is None:
        return None
    h, _ = args
    def wf(u, v):   return W[(u, v)] + h[u] - h[v]
    for u in range(n):                  # compute SSSP on reweighted graph
        d, parent = dijkstra(Adj, wf, u)
        for v in range(n):
            if d[v] < INF:
                D[u][v] = d[v] - h[u] + h[v]
    return tuple(tuple(row) for row in D)
```

# 6.006 Introduction to Algorithms

Spring 2020