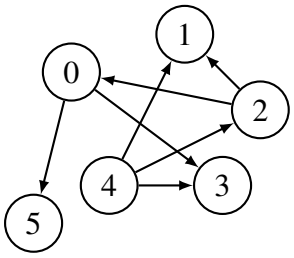
Problem Set 6

All parts are due on October 25, 2019 at 6PM. 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. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

Problem 6-1. [10 points] **Topological Training**

Please answer the following questions about the unweighted directed graph G below, whose vertex set is $\{0, 1, 2, 3, 4, 5\}$.



(a) [5 points] State a topological ordering of G. Then state and **justify** the number of distinct topological orderings of G.

(b) [5 points] State a single directed edge that could be added to G to construct a simple[1] graph with no topological ordering. Then state and **justify** the number of distinct single edges that could be added to G to construct a simple graph with no topological ordering.

Solution: (a) Vertices 4 and 2 must be the first and second vertex in any topological order, since there is a directed path from 4 through 2 to every other vertex in the graph. For the remaining four vertices, the ordering of 0, 3, and 5 are independent from 1. Vertices 0, 3, and 5 have two possible orderings: $(0, 3, 5)$ and $(0, 5, 3)$, while 1 can be placed in one of four positions relative to either ordering. Thus there are 8 distinct topological orders of G. Specifically:

$$(4, 2, 1, 0, 3, 5) \quad (4, 2, 0, 1, 3, 5) \quad (4, 2, 0, 3, 1, 5) \quad (4, 2, 0, 3, 5, 1)$$
$$(4, 2, 1, 0, 5, 3) \quad (4, 2, 0, 1, 5, 3) \quad (4, 2, 0, 5, 1, 3) \quad (4, 2, 0, 5, 3, 1)$$

Rubric:

- 1 point for a correct topological ordering of G
- 2 points for stating the correct number of topological orderings of G
- 2 points for justification of number of topological orderings of G
- Partial credit may be awarded

[1] A simple graph has no self-loops (i.e., each edge connects two different vertices) and has no multi-edges (i.e., there can be at most one directed edge from vertex a to vertex b, though a directed edge from b to a may exist).

Solution: (b) A topological ordering will not exist, if and only if the resulting graph contains a cycle. All vertices are reachable from 4, so we can add any edge from vertices 0, 1, 2, 3, or 5 to 4 to create a cycle. All vertices except 4 are reachable from 2, so we can add any edge from vertices 0, 1, 3, or 5 to create a cycle. Vertices 3 and 5 are reachable from 0, so we can add either edge to 0. No vertices are reachable from 1, 3, or 5, so no edge can be added to them to construct a cycle. Thus there are **eleven** such directed edges, specifically:

$$\{(0,4),(1,4),(2,4),(3,4),(5,4)(0,2),(1,2),(3,2),(5,2),(3,0),(5,0)\}.$$

Rubric:

- 1 point for an edge that would create a cycle when added to G
- 2 points for stating the correct number of such edges
- 2 points for justification of number of such edges
- Partial credit may be awarded

Problem 6-2. [10 points] **DFS Relaxation**

Given an unweighted undirected graph $G = (V, E)$ and vertex $s \in V$, consider the following algorithm (DFS Relaxation):

- Initialize $d'[s] = 0$ and $d'[v] = \infty$ for all $v \in (V - \{s\})$
- Run some depth-first search (DFS) from s (there can be multiple valid ways to DFS)
- Whenever the search **first visits** a vertex v from a vertex u set $d'[v] = 1 + d'[u]$

(a) [4 points] In the special case when G is a **tree**, prove that any DFS Relaxation from a vertex s in the tree correctly computes shortest path distances from s, i.e., $d'[v] = \delta(s, v)$ for every $v \in V$.

Solution: Note that $d'[v]$ will be the length of some simple path from s to v, in particular, the length of the path in the DFS tree. In a tree, there is a unique simple path between any pair of vertices, as two distinct simple paths could be used to construct a cycle. Hence $d[v]$ is the length of the only simple path from s to v, which is $d(s, v)$.

Rubric:

- 4 points for a correct proof
- Partial credit may be awarded

(b) [2 points] State an undirected graph $G = (V, E)$ on three vertices for which DFS Relaxation from some vertex $s \in V$ **incorrectly** computes the shortest path distance to some vertex, i.e., $d'[v] \neq \delta(s, v)$ for some $v \in V$.

Solution: The only simple graph on three vertices that has this property is the complete graph on three vertices (i.e., a three cycle). If the vertices are $\{s, a, b\}$, then any

DFS from s will travel first to one of a or b and then the other. Whichever is visited second will be assigned a d value of 2 while the shortest path distance from s is 1.

Rubric:

- 2 points for description of a three vertex cycle

(c) [4 points] An unweighted graph $G = (V, E)$ **fails spectacularly** if every DFS Relaxation on G (from any vertex $s \in V$, using any valid depth-first search edge visitation order) results in some $d'[v]$ for which $d'[v] \geq \frac{1}{2}|V| \cdot \delta(s, v)$, i.e., $d'[v]$ is at least a factor of $\frac{1}{2}|V|$ away from the true shortest path distance from s. Given any integer $k > 3$, describe an undirected graph on k vertices that fails spectacularly.

Solution: There are many solutions here, but the simplest is to connect the k vertices into a simple cycle using k edges. Beginning at arbitrary vertex s, any DFS will explore edges going in one direction around the cycle until reaching back to s. Thus, while the shortest path length from s to each neighbors of s is 1, one of the neighbors v will be assigned distance $d[v] = k - 1$. This graph fails spectacularly because $k + 1 \geq \frac{1}{2}k \cdot 1$ for all $k > 3$.

Rubric:

- 4 points for description of a graph on k vertices which fails spectacularly
- Partial credit may be awarded

Problem 6-3. [10 points] **Never Return**

Bimsa is a young lioness whose evil uncle has just banished her from the land of Honor Stone, proclaiming: "Run away, Bimsa. Run away, and never return." Bimsa has knowledge of all landmarks and trails in and around Honor Stone. For each landmark, she knows its x and y coordinates and whether it is inside or outside of Honor Stone. For each trail, she knows its positive integer length and the two landmarks it directly connects. Each landmark connects to at most five trails, each trail may be traversed in either direction, and every landmark can be reached along trails from the **gorge**, the landmark where Bimsa is now. Bimsa wants to leave Honor Stone quickly, while only traversing trails in a way that **never returns**: traversing a trail from landmark a to landmark b never returns if the distance[2] from a to the gorge is strictly smaller than the distance from b to the gorge. If there are n landmarks in Honor Stone, describe an $O(n)$-time algorithm to determine a shortest route that never returns, from the gorge to any landmark outside of Honor Stone.

Solution: The 'never return' constraint ensures that we can only visit landmarks in strictly increasing distance from the gorge, so if we construct a graph on landmarks with an edge for each trail only in a direction which increases distance from the gorge, this graph will be acyclic. Let $d(a)$ denote the **squared** Euclidean distance from the gorge to landmark a (squared distances are integers that are computable in $O(1)$ time). Construct a graph G with a vertex for each landmark either in Honor Stone or neighboring a landmark in Honor Stone. Then for each trail directly connecting landmarks a and b, add either directed edge (a, b) if $d(a) < d(b)$, directed edge (b, a) if

[2]We mean Euclidean distance between landmark (x_1, y_1) and landmark (x_2, y_2), i.e., $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

$d(a) > d(b)$, or no edge if $d(a) = d(b)$. Since there are at most $O(n)$ landmarks in Honor Stone or neighboring a landmark in Honor Stone, and each landmark is connected to at most a constant number of trails, there are at most $O(n)$ trials, so this graph can be constructed in $O(n)$ time. Then every directed path in G from the gorge satisfies the 'never return' condition, so we just need to find any shortest route in G from the gorge to any vertex corresponding to a landmark that is outside Honor Stone. Solve single source shortest paths from the gorge (storing parent pointers to allow paths to be reconstructed), and return the shortest path to any vertex corresponding to a landmark that is outside Honor Stone. Since G is a DAG, we can solve single single source shortest paths using DAG relaxation in $O(n)$ time as desired.

Rubric:

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

Problem 6-4. [15 points] **DigBuild**

Software company Jomang is developing **DigBuild**, a new 3D voxel-based exploration game. The game world features n collectable block types, each identified with a unique integer from 1 to n. Some of these block types can be converted into other block types: a conversion (d_1, b_1, d_2, b_2) allows d_1 blocks of type b_1 to be converted into d_2 blocks of type b_2; each conversion is animated in the game by repeatedly dividing blocks in half, then recombining them, so d_1 and d_2 are always constrained to be integer powers of two. Jomang wants to randomly generate allowable game conversions to increase replayability, but wants to disallow sets of game conversions through which players can generate infinite numbers of blocks via conversion. Describe an $O(n^3)$-time algorithm to determine whether a given a set of $\lfloor \frac{1}{5}n^2 \rfloor$ conversions should be disallowed. Assume that a starting world contains D blocks of each type, where D is the product of all d_i appearing in any conversion. Given positive integer x, assume that its bit-length, $\log_2 x$, can be computed in $O(1)$ time.

Solution: Construct a graph G with a vertex for every block type and a directed edge for every conversion (d_1, b_1, d_2, b_2), specifically the edge from b_1 to b_2 with weight $\lg(d_1) - \lg(d_2)$. Then the negative weight of any directed path from b_s to b_t in G corresponds to the base-2 logarithm of the fractional increase of the amount of blocks of type b_t that would result by performing each conversion along the path from some amount of block of type b_s (since $\lg \prod_i \frac{d_{i,2}}{d_{i,1}} = \sum_i (\lg(d_{i,2}) - \lg(d_{i,1}))$). Thus, given a sufficiently large number of starting blocks, there exists a negative-weight simple cycle in G if and only if a user could convert block types along these conversions to generate an unbounded amount of blocks of any block type reachable from the cycle. Since D upper bounds the base-2 logarithm of the sum of all edge weights in G, any simple cycle has absolute weight less than $\lg D$, so D blocks of each type will allow a block type on a negative-weight simple cycle to

be converted around the cycle, without ever running out of blocks during the conversion. We can detect whether G contains a negative-weight simple cycle using Bellman-Ford in $O(|V||E|)$ time. Since G has n vertices and $\Theta(n^2)$ edges, Bellman-Ford runs in $O(n^3)$ time, as desired.

Rubric:

- 3 points for description of graph
- 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
- 5 points if correct algorithm is efficient, i.e., $O(n^3)$
- Partial credit may be awarded

Problem 6-5. [15 points] **Tipsy Tannister**

Lyrion Tannister is a greedy, drunken nobleman residing in Prince's Pier, the capital of Easteros, who wants to travel to the northern town of Summerrise. Lyrion has a map depicting all towns and roads in Easteros, where each road allows direct travel between a pair of towns, and each town is connected to at most seven roads. Each road is marked with the non-negative number of gold pieces he will be able to collect in taxes by traveling along that road in either direction (this number may be zero). Every town has a tavern, and Lyrion has marked each town with the positive number of gold pieces he would spend if he were to drink there. If Lyrion ever runs out of money on his trip, he will just leave a debt marker[3] indicating the number of gold pieces he owes. After leaving the tavern in Prince's Pier with no gold and zero debt, Lyrion's policy will be to drink at the tavern of every third town he visits along his route until reaching Summerrise. Given Lyrion's map depicting the n towns in Easteros, describe an $O(n^2)$-time algorithm to compute the maximum amount of gold he can have (minus debts incurred) upon arriving in Summerrise, traveling from Prince's Pier along a route that follows his drinking policy.

Solution: Given Lyrion's drinking policy, it would be helpful to know at any given time how many towns ago Lyrion last visited a tavern. To keep track of this information, we will make a graph that has three vertices for each town: one vertex associated with arriving at the town having drunk at a town i towns before, either 1, 2, or 3 towns before. Construct a graph G with a vertex $v_{t,i}$ for each town t and for each $i \in \{1, 2, 3\}$. Then for each road connecting town a to town b, add directed edges $(v_{a,i}, v_{b,(i \bmod 3)+1})$ and $(v_{b,i}, v_{a,(i \bmod 3)+1})$ for each $i \in \{1, 2, 3\}$:

- when $i \in \{1, 2\}$, Lyrion does not drink in the town he is coming from, so weight the edge with the negative of the amount of gold he would collect in taxes on that road, while
- when $i = 3$, Lyrion drinks in the town he is coming from, so weight the edge with the amount he will spend at that town's tavarn, minus the amount of gold he would collect in taxes on that road.

[3]The marker will be readily accepted because Tannisters always pay their debts.

Let p and s denote the vertices associated with Prince's Pier and Summerrise respectively, and remove all outgoing edges from the three vertices associated with s (since Lyrion stops as soon as he arrives in Summerrise). If w_p is the cost of drinking at the tavarn in Prince's Pier, then the weight of any directed path in G from vertex $v_{p,3}$ will be w_p less than the amount of taxes minus expenses that Lyrion will have by traversing along the roads corresponding to edges in the path while following his drinking policy. G has $O(n)$ vertices and $O(n)$ edges (since each vertex is connected to at most a constant number of edges), so this graph can be built in $O(n)$ time. Then, since edges in G may have positive or negative weights, we can use Bellman-Ford to compute the weight of the shortest path from p to s in G in $O(n^2)$ time, which will be the negative of the maximum amount of gold (or debt) he can have upon arriving in Summerrise. If Bellman-Ford returns a weight of $-\infty$ for $d(p, s)$, there is no finite upper bound on the amount Lyrion can arrive with, while if Bellman-Ford returns ∞ for $d(p, s)$, Summerrise cannot be reached along roads, so Lyrion should just stay home.

Rubric:

- 3 points for description of graph
- 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
- 5 points if correct algorithm is efficient, i.e., $O(n^2)$
- Partial credit may be awarded

Problem 6-6. [40 points] **Cloud Computing**

Azrosoft Micure is a cloud computing company where users can upload **computing jobs** to be executed remotely. The company has a large number of identical cloud computers available to run code, many more than the number of pieces of code in any single job. Any piece of code may be run on any available computer at any time. Each computing job consists of a code list and a dependency list.

A **code list** C is an array of code pairs $(f, t) \in C$, where string f is the file name of a piece of code, and t is the positive integer number of microseconds needed for that code to complete when assigned to a cloud computer. Assume file names are short and can be read in $O(1)$ time.

A **dependency list** D is an array of dependency pairs $(f_1, f_2) \in D$, where f_1 and f_2 are distinct file names that appear in C. A dependency pair (f_1, f_2) indicates that the piece of code named f_1 must be completed before the piece of code named f_2 can begin. Assume that every file name exists in some dependency pair.

(a) [5 points] A job (C, D) can be **completed** if every piece of code in C can be completed while respecting the dependencies in D. Given job (C, D), describe an $O(|D|)$-time algorithm to decide whether the job can be completed.

Solution: Construct a graph G with a vertex for each piece of code in C and a directed edge from f_1 to f_2 for each dependency pair $(f_1, f_2) \in D$. Also add an auxiliary node

s and add a directed edge (s, f) to every other vertex. A job can be completed as long as G does not contain any cycle. We can do this by running a depth-first search (DFS) from s, checking whether the reverse order of DFS finishing times is a topological order (i.e., check to make sure no edge in the graph goes against this order). Since $|D| \geq |C|$, G has size $O(|D|)$ and DFS can be be run in $O(|D|)$ time. Checking each edge against this order can also be done in $O(1)$ time per edge, so this algorithm takes $O(|D|)$ time as desired.

Rubric:

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

(b) [10 points] Azrosoft Micure wants to know how fast they can complete a given job. Given a job (C, D), describe an $O(|D|)$-time algorithm to determine the minimum number of microseconds that would be needed to complete the job (or return that the job cannot be completed).

Solution: We can use the same graph as in part (a), adding weights corresponding to running times of pieces of code. Let $t(f)$ denote the time for code f to complete. Then for each edge (a, f) in G, weight it by $-t(f)$. Then the length of the shortest path from s to any other vertex in G will be equal to longest time for any piece of code to be completed. If G has a cycle, we can break as in (a); otherwise, G is a DAG, and we can run DAG Relaxation to compute the shortest path in $O(|D|)$ time.

Rubric:

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

(c) [25 points] Write a Python function `min_time(C, D)` that implements your algorithm from (b). You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
def dfs(Adj, s, parent = None, order = None):    # code from R10
    if parent is None:
        parent = [None for v in Adj]
        parent[s] = s
        order = []
    for v in Adj[s]:
        if parent[v] is None:
            parent[v] = s
            dfs(Adj, v, parent, order)
    order.append(s)
    return parent, order

def topo_shortest_paths(Adj, w, s):              # code from R11
    _, order = dfs(Adj, s)
    order.reverse()
    d = [float('inf') for _ in Adj]
    parent = [None for _ in Adj]
    d[s], parent[s] = 0, s
    for u in order:
        for v in Adj[u]:
            if d[v] > d[u] + w(u, v):
                d[v] = d[u] + w(u, v)
                parent[v] = u
    return d, parent

def min_time(C, D):
    n = len(C)
    file_idx, file_time = {}, {}
    for i in range(n):                           # label files from 1 to n
        f, t = C[i]
        file_idx[f], file_time[i + 1] = i + 1, t
    Adj, w = [[] for _ in range(n + 1)], {}      # construct dependency graph
    for f1, f2 in D:
        i1, i2 = file_idx[f1], file_idx[f2]
        Adj[i1].append(i2)
        w[(i1, i2)] = -file_time[i2]
    Adj[0] = list(range(1, n + 1))               # add supernode
    for i in range(1, n + 1):
        w[(0, i)] = -file_time[i]
    _, order = dfs(Adj, 0)                       # check for cycles
    order.reverse()
    rank = [None] * (n + 1)
    for i in range(n + 1):
        rank[order[i]] = i
    for v1 in range(n + 1):
        for v2 in Adj[v1]:
            if rank[v1] > rank[v2]:
                return None
    dist, _ = topo_shortest_paths(Adj, lambda u,v: w[(u,v)], 0)
    return -min(dist)                            # ^ compute min time
```