Problem Set 7

All parts are due on November 9, 2017 at 11:59PM. 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.csail.mit.edu .

Problem 7-1. [30 points] **Weighted Shortest Paths**

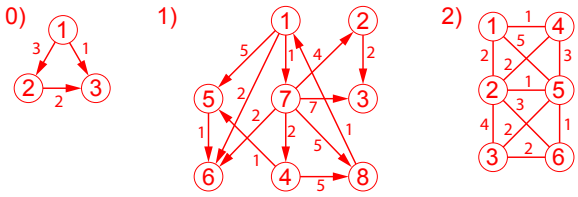
(a) [20 points] **Dijkstra Practice:** Below are representions for three weighted graphs. Each representation is a list of triples. The first two items in the triple are integer labels of two vertices connected by an edge, while the third item is a positive numerical weight of the edge. The first two graphs are directed, with a triple (a, b, w) representing a directed edge from vertex v_a to v_b, while the last graph is undirected. For each of graphs G_1 and G_2, perform Dijkstra's algorithm starting from vertex v_1 by doing the following:

1. Draw the weighted graph
2. List one possible order that edges could be first touched by Dijkstra
3. List the shortest path distance $\delta(v_i)$ from v_1 to each vertex $v_i \in V$

For example, a list of edges in graph G_0 that could be first touched by Dijkstra is $(\{e_1, e_2\}, e_3)$ where the first two edges could be touched in either order, with shortest path distances $(\delta(v_1), \delta(v_2), \delta(v_3)) = (0, 3, 1)$.

```
G_0 = [(1, 2, 3),  G_1 = [(1, 5, 5),  G_2 = [(1, 2, 2),    e_1
       (1, 3, 1),         (1, 6, 2),         (1, 4, 1),    e_2
       (2, 3, 2)]         (1, 7, 1),         (1, 5, 5),    e_3
                          (2, 3, 2),         (2, 3, 4),    e_4
                          (4, 5, 1),         (2, 4, 2),    e_5
                          (4, 8, 5),         (2, 5, 1),    e_6
                          (5, 6, 1),         (2, 6, 3),    e_7
                          (7, 2, 4),         (3, 5, 2),    e_8
                          (7, 3, 7),         (3, 6, 2),    e_9
                          (7, 4, 2),         (4, 5, 3),    e_10
                          (7, 6, 2),         (5, 6, 1)]    e_11
                          (7, 8, 5),                       e_12
                          (8, 1, 1)]                       e_13
```

Solution: Drawings:



Dijkstra edge touching order:

- $edge\_order(G_1) = (\{e_1, e_2, e_3\}, \{e_8, e_9, e_{10}, e_{11}, e_{12}\}, \{e_5, e_6\}, e_7, e_4, e_{13})$
- $edge\_order(G_2) = (\{e_1, e_2, e_3\}, \{e_5, e_{10}\}, \{e_4, e_6, e_7\}, \{e_8, e_{11}\}, e_9)$

Shortest path distance by vertex:

- $(\delta(v_1), \delta(v_2), \delta(v_3), \delta(v_4), \delta(v_5), \delta(v_6), \delta(v_7), \delta(v_8)) = (0, 5, 7, 3, 4, 2, 1, 6)$
- $(\delta(v_1), \delta(v_2), \delta(v_3), \delta(v_4), \delta(v_5), \delta(v_6)) = (0, 2, 5, 1, 3, 4)$

Rubric:

- 10 points each for G_1 and G_2
- 2 points per drawing
- 4 points per edge relaxation ordering
- 4 points per shortest path listing
- Partial credit may be awarded

(b) [10 points] **Minimal Destination Paths:** Let $G = (V, E, w)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, containing negative weights but no negative weight cycles. A **destination path** to vertex $v \in V$ is a path with smallest weight starting from any vertex $u \in V$ that terminates at vertex v. Note that vertex u may be v, resulting in a 0 weight destination path. A destination path is **minimal** if its weight is less than or equal to the weight of any other destination path terminating at v. Describe an algorithm to compute the weight of a minimal destination path for every vertex in V, all in $O(|V||E|)$ time.

Solution:

Add vertex s to G to construct a new graph G', connecting s to every vertex in V with a zero weight edge. Run Bellman-Ford to find the weight of a shortest path from s to each vertex in $O(|V||E|)$. The weight of a shortest path from s to vertex v is identical to the wieght of the minimum destination path to v. If it were not, then path between them of smaller weight could be used to construct a smaller path satisfying the other.

Rubric:

- 10 points for a correct $O(|V||E|)$ algorithm
- Partial credit may be awarded

Problem 7-2. [30 points] **Consulting**

For each of the following scenerios, provide the fastest algorithm you can think of to solve your client's problem.

(a) [10 points] **Downhill Skiing:** Warbler is a ski resort bidding to be the location for next year's downhill ski competition. The resort contains numerous downhill trails weaving down the mountain, with each trail reachable from the lodge at the peak. Warbler has collected trail reviews from customers, and has compiled an average rating for each trail between -5 and 5. The steering committee for the competition has scheduled a visit to evaluate the resort. They will have time for one evaluation downhill ski tour starting at the peak. For the tour, Warbler wants to choose a path of downhill trails beginning at the peak that maximizes the sum of trail ratings along the path. Describe an algorithm to quickly find such a path.

Solution: Construct directed graph $G = (V, E)$ where vertices are locations where downhill ski trails start, end, or diverge, and edges represent downhill ski trails. Each route is downhill, so G will be acyclic. Weight each edge as the negative average rating of the trail associated with the edge. Use DFS to topologically sort the intersections. Initialize each vertex v with an upper bound on the minimum weight path $\delta(v)$ from the peak, i.e. 0 for the peak and $-\infty$ for all other vertices. Relax edges in order based on the topological sort order of their strating vertex. As discussed in lecture, this algorithm computes shortest weighted paths in $O(|V| + |E|)$ time. Among all shortest paths from the peak, choose the path that has the lowest weight, i.e. the path that has the largest sum of trial ratings along the path.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

(b) [10 points] **Driving Options:** 6006LE Maps is a popular application that helps users navigate from one location to another. Given a starting location and an ending location, 6006LE currently provides users with the shortest driving route between two locations. However, 6006LE would like to give their users an additional option to choose from; specifically, they would also like to provide the second shortest route. Describe an algorithm to find both the shortest and second shortest routes between a given pair of cities, so that 6006LE may quickly return them to their customer. You may assume that every road supports traffic in both directions.

Solution:

There is some ambiguity in the wording of this problem. In particular, what does a 'second shortest route' mean? When given vague specifications, one may need to assume something reasonable in order to proceed. Does a route mean a simple path, or a walk allowing vertices or edges to be traversed more than once? Either is a valid

interpretation, so we have provided two solutions. Note that for graphs with positive weights, shortest walks are shortest paths, as cycles can always be shortcut to produce a shorter walk.

Construct undirected graph $G = (V, E)$ where vertices are connections between roads, and edges are roads. Weight each road based on the length of the road, which will be a positive number. Because weights are positive, we may apply Dijkstra to compute the shortest path between a pair of cities in $O(|V| \log |V| + |E|)$ time using a Fibonacci Heap for the priority queue. If we assume the road network is planar (which is reasonable for a road network), the number of edges will be linear in the number of vertices $|E| = O(|V|)$, so you may choose to use a binary heap to implement the priority queue to acheive a $O(|V| \log |V|)$ runtime bound.

Paths: To find a second shortest path, visiting each intersection at most once, we could use a brute force algorithm: any second shortest path cannot use every edge from the shortest path, so remove an edge contained in the shortest path from the graph, and find the shortest path in the new graph using Dijkstra. Doing this for each edge in the shortest path containing at most $|V| - 1$ edges yields a $O(|V|(|V| \log |V| + |E|))$ time algorithm. If we store parent pointers during Dijkstra, we can follow them back to the source to return a requested path.

Walks: Alternatively, we can compute second shortest walks in $O(|V| \log |V| + |E|)$ time, by performing one pass of a modified Dijkstra on the graph. In addition to maintaing δ representing upperbounds on shortest paths, we also maintain δ_2 representing upperbounds on second shortest walks. In our priority queue, we will store pairs (v, i), for each $v \in V$, and for i equal to 1 and 2. Pair $(v, 1)$ will be keyed by $\delta(v)$, while $(v, 2)$ will be keyed by $\delta_2(v)$. Initialize both $\delta(v)$ and $\delta_2(v)$ to ∞ for each $v \in V$, except for the source vertex s where we initialize both $\delta(s)$ and $\delta_2(s)$ to 0. As with Dijkstra, we will repeatedly remove and process pairs with minimum key. To process a pair $(u, 1)$, for each edge (u, v) outgoing from u, relax $\delta(v)$ normally as in Dijkstra, replace $\delta(v)$ with $\delta(u) + w(u, v)$ if it is smaller, or replace $\delta_2(v)$ with $\delta(u) + w(u, v)$ if it is strictly between $\delta(v)$ and $\delta_2(v)$. Also, if $\delta(v)$ was updated, replace $\delta_2(u)$ with the old value of $\delta(v)$. Similarly, to process a pair $(u, 2)$, for each edge (u, v) outgoing from u, replace $\delta_2(v)$ with the smaller of $\delta_2(u) + w(u, v)$ and $\delta(u) + w(u, v)$ that is also larger than $\delta(v)$ and smaller than $\delta_2(v)$. When updating δ and δ_2 values, also update the corresponding keys in the priority queue.

This algorithm finds shortest paths lengths δ identically to Dijkstra, but also finds second shortest walk lengths δ_2 because it is last updated exactly second time that Dijkstra reaches the vertex from a shortest or second shortest walk. To return the walks, we also store parent pointers π and π_2. Since δ and δ_2 will be distinct when the vertex is reachable, we can determine the next parent on the second shortest walk by comparing whether $\delta_2(v)$ equals $w(\pi(v), v) + \delta(\pi(v))$ or $w(\pi(v), v) + \delta_2(\pi(v))$. This modified Dijkstra processes each vertex at most twice and each edge at most four times, so by using a Ficonacci Heap for the priority queue, the algorithm runs in

$O(|V| \log |V| + |E|)$ time.

Rubric:

- 10 points for a correct algorithm (in either interpretation)
- Any correct algorithm that is $O(|V|^3)$ should receive full points
- Partial credit may be awarded

(c) [10 points] **Currency Exchange:** Travelwhy is a currency exchange company that buys and sells currencies at different prices, with a possibly different fixed rate every day. For instance, today they might purchase $\$1$ US Dollar from you and give you $€0.8$ Euros in return, though they might also purchase $€1$ Euro from you in exchange for only $\$1.10$ US Dollars. In this case, we would say the exchange rate from US Dollars to Euros is 0.8, while the exchange rate from Euros to US Dollars is 1.1. Travelwhy is concerned that the daily prices automatically generated by their computer might allow someone to make money off of them, simply by buying and selling the right sequence of currencies on their exchange. Describe an algorithm to check if there's a way to make money on their exchange on a given day.

Solution:

The problem is asking whether there exists a sequence of successive exchange rates whose product is more than one. We note that the logarithm of a product of exchange rates is equal to the sum of the logarithms of the rates. Construct graph G with vertices on currencies, with a directed edge from currancy a to currency b with weight $-\log r$ when there is a conversion from a to b at rate r. Thus, if there is a negative weight cycle in G, there will be a cyclic sequence of conversion rates whose product is positive.

Here's an algorithm: run $|V|$ rounds of Bellman-Ford from an arbitrary currency c in the graph in $O(|V||E|)$ time. If any edge is relaxed in the final round, a negative cycle exists, thus there is a way to make money on the exchange. However there's a subtlety here. This algorithm assumes that all currencies are convertible to every other currency on the exchange; otherwise shortest paths from c may be infinite, and we may never find a negative cycle not reachable from c. It is a reasonable assumption for this problem to assume connectedness (so you will receive full credit for simply applying Bellman-Ford), but you don't need the graph to be connected to solve the problem in $O(|V||E|)$ time. Assume that not all vertices are reachable from c. Run BFS from c to mark all vertices V' and edges E' reachable from c, and run Bellman-Ford on the subgraph (V', E') to find any negative weight cycles within it. This takes $O(|V'||E'|)$ time. Remove V' and E' from the graph and recurse on the remaining subgraph, running BFS then Bellman-Ford from some remaining vertex, checking for negative weight cycles. Do this repeatedly until all vertices and edges in the graph have been processed. Notice that each edge is searched by BFS at most once, and relaxed by Bellman-Ford at most $V' = O(|V|)$ times, so this algorithm also runs in $O(|V||E|)$ time.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

Problem 7-3. [40 points] **Catwidth**

ComCat is a internet service provider that wants to optimize their network to bring you cat videos as fast as they can. Their network is comprised of thousands of servers storing millions of cat videos. Each user is connected to a single server, and servers are connected to each other, though not every pair of servers share a connection. Each direct server-server connection has a maximum bandwidth of cat videos that each may stream to the other; bandwidth may be different for different pairs of servers. Cat videos might need to pass through multiple servers to reach a user. The **catwidth** of a sequence of server-server connections is the smallest bandwith of any server-server connection in the sequence. There are s servers in the network, each labeled with a unique server ID number between 0 and $s - 1$.

(a) [10 points] ComCat wants to direct traffic in the network along routes that maximize catwidth. Given a pair of servers, describe an algorithm to compute a route with the largest catwidth in order to direct cat videos between them.

Solution:

Construct graph G on the servers, with an undirected edge from server s_i to s_j with weight b when s_i and s_j have a conenction between then with bandwith w. Let s_1 and s_2 be the queried pair of servers. We run Dijkstra from s_1 to compute paths that maximize catwidth by modifying the relaxation step. Modify δ to represent a lower bound of catwidth, initializing $\delta[s_1]$ to 0, and $\delta[v]$ to -1 for all other $v \in V$. Store a max priority queue of servers, with server s keyed by $\delta[s]$. The maximum catwidth from u to v will either be the catwidth of an edge from u, or the minimum between the maximum catwidth from u to a neighbor $v_n \in Adj[v]$ and the bandwidth b of the edge from (v_n, v). We modify the relaxation step of Dijkstra to increase the lower bounds δ to satisfy this relation, removing servers from the priority queue when it has maximum δ from among servers not yet processed. Because we have only changed the relaxation step of Dijkstra which may still be computed in $O(1)$ time in terms of previously computed values, the running time of Dijkstra depends on the data structure used for the priority queue. Using a Fibonacci Heap acheives $O(|V| \log |V| + |E|)$ time.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

(b) [5 points] Assume that the number of server-server connections is large compared to the number of servers, i.e. asymptotically quadratic in the number of servers. Discuss the running time of your algorithm; specifically, can you simplify the implimentation of data structures used by your algorithm based on the assumed server-server connection density?

Solution:

A $O(|V| \log |V| + |E|)$ runtime is acheivable by implementing the priority queue in Dijkstra with a Fibonacci heap. When $|E| = O(|V|^2)$, and the running time is quadratic. Dijkstra also runs in $O(|V|^2)$ using an array as your priority queue, which is very simple to implement while remaining asymptotically fast. Note that using a binary heap for the priority queue would result in a $O(|V|^2 \log |V|)$ running time, which is asymptotically slower than using an array.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

(c) [25 points] Implement function `catwidth` that takes as input a list of server-server connection triples (each consisting of two server IDs and the bandwidth between them) and a pair of server IDs, and returns the maximum catwidth of any route between them, assuming the number of server-server connections is asymptotically quadratic. Please submit code for this problem in a single file to alg.csail.mit.edu. ALG will only import your implementation of `catwidth`.

Solution: This part graded by ALG. See code below:

```
def catwidth(n, s1, s2, connections):
    """
    Given a list containing triples (i, j, w) representing the bandwidth w
    between servers i and j, return the catwidth between them.
    Input:
        'n' is the number of servers
        'connections' is a list of triples (i, j, w)
        i, j, 's1', 's2' are server IDs, while w are bandwidth values
        Servers IDs are unique numbers in range [0,n-1]
    Output:
        'cw' is the catwidth between servers i and j
    """
    # Construct adjacency lists
    adj = [[]] * n
    for i, j, w in connections:
        adj[i].append((j, w))
        adj[j].append((i, w))
    # initialize lower bounds on catwidth
    best_cw = [-1 for i in range(n)]
    best_cw[s1] = 0
    # initialize servers remaining to processed
    queue = [i for i in range(n)]
    while len(queue) != 0:
        # remove server with maximum lower bound
        for i in range(len(queue)):
            if best_cw[queue[-1]] < best_cw[queue[i]]:
                queue[-1], queue[i] = queue[i], queue[-1]
        s = queue.pop()
        # relax edges to neighbors
        for v, w in adj[s]:
            if best_cw[s] != 0:
                w = min(w, best_cw[s])
            if best_cw[v] < w and best_cw[v] != 0:
                best_cw[v] = w
    cw = best_cw[s2]
    return cw
```