**Problem Session 7**

**Problem 7-1. Dijkstra Practice**

(a) Run Dijkstra on the following graph from vertex $s$ to every vertex in $V = \{a, b, c, d, e, f, g, h, s\}$. Write down (1) the minimum-weight path weight $\delta(s, v)$ for each vertex $v \in V$, and (2) the order that vertices are removed from Dijkstra’s queue.

```
     s --- 8 --- a --- 0 --- b
       |    7    1    0    1    3
      c --- 4 --- d --- 2 --- e
       0    1    2    1
     f --- 0 --- g --- 2 --- h
```

**Solution:**

<table>
<thead>
<tr>
<th>Vertex $v$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
<th>$h$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta(s, v)$</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>$\infty$</td>
<td>12</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Removal order</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

Nodes are processed in increasing order of $\delta(s, v)$, which for this graph is unique. The processing order is not always unique; if two vertices have the same shortest path distance, it is possible that Dijkstra could process them in either order, depending on how the implementation of Dijkstra breaks ties in the priority queue. While $a$ and $d$ have the same shortest path distance from $s$, $d$ is only reachable through $a$, so $a$ must be processed first.

(b) Change the weight of edge $(g, c)$ to $-6$. Identify a vertex $v$ for which running Dijkstra from $s$ as in part (a) would change the shortest path estimate to $v$ at a time when $v$ is not in Dijkstra’s queue.

**Solution:** If the weight of edge $(g, c)$ were changed to $-6$, running Dijkstra’s would remove vertices from the queue in the same order as in part (a). But then when $g$ is removed from the queue, $c$ would have already been popped from the queue and would change the shortest path estimate to $c$ from 7 to 6, so $c$ has the requested property.
Problem 7-2. Weighted Graph Radius

In a weighted directed graph \( G = (V, E) \), the weighted eccentricity \( \epsilon(u) \) of a vertex \( u \in V \) is the shortest weighted distance to its farthest vertex \( v \), i.e., \( \epsilon(u) = \max\{\delta(u, v) \mid v \in V\} \). The weighted radius \( R(G) \) of a weighted directed graph \( G = (V, E) \) is the smallest eccentricity of any vertex, i.e., \( R(G) = \min\{\epsilon(u) \mid u \in V\} \). Given a weighted directed graph \( G \), where edge weights may be positive or negative but \( G \) contains no negative-weight cycle, describe an \( O(|V|^3) \)-time algorithm to determine the graph’s weighted radius (compare with Problem 2 from PS5).

**Solution:** Use Johnson’s algorithm to compute all-pairs shortest paths distances \( \delta(a, b) \) for every ordered distinct pair of vertices \( (a, b) \) in \( O(|V||E| + |V|^2 \log |V|) = O(|V|^3) \) time (none of which are \(-\infty\) since the graph contains no negative-weight cycles). Then, simply compute \( \epsilon(u) \) directly for each \( u \in V \), by computing a maximum in \( O(|V|) \) time per vertex and \( O(|V|^2) \) time in total, and then find the smallest \( \epsilon(u) \) for \( u \in V \) in \( O(|V|) \) time. This algorithm is correct because we are performing the requested computation directly and Johnson’s computes every \( \delta(a, b) \) correctly for any graph not containing negative-weight cycles. The running time for this algorithm is bounded by Johnson’s, so runs in \( O(|V|^3) \) in total.

The Floyd-Warshall algorithm could also be used in combination with Bellman-Ford, but since we have not yet talked about this algorithm, students would need to fully describe and analyze Floyd-Warshall in order to use it.

Problem 7-3. Under Games

Atniss Keverdeen is a rebel spy who has been assigned to go on a reconnaissance mission to the mansion of the tyrannical dictator, President Rain. To limit exposure, she has decided to travel via an underground sewer network. Every pipe is marked with its positive integer length, while some junctions are marked as containing identical motion sensors, any of which will be able to sense Atniss if her distance to that sensor (as measured along pipes in the sewer network) is too close. Unfortunately, Atniss does not know the sensitivity of the sensors. Describe an \( O(n \log n) \)-time algorithm to find a path along pipes from a given entrance junction to the junction below President Rain’s mansion that stays as far from motion sensors as possible.

**Solution:** Construct a graph \( G \) with a vertex for every junction in the sewer network and an undirected edge between junction \( a \) and junction \( b \) with weight \( w \) if \( a \) and \( b \) are connected by a pipe with length \( w \). Since each junction connects to at most four pipes, and every junction is reachable from every other junction via the sewer network. Each pipe connects to at most four pipes, and every junction is reachable from every other junction via the sewer network. Every pipe is marked with its positive integer length, while some junctions are marked as containing identical motion sensors, any of which will be able to sense Atniss if her distance to that sensor (as measured along pipes in the sewer network) is too close. Unfortunately, Atniss does not know the sensitivity of the sensors. Describe an \( O(n \log n) \)-time algorithm to find a path along pipes from a given entrance junction to the junction below President Rain’s mansion that stays as far from motion sensors as possible.

To find \( k^* \), first label each junction with its shortest distance to any sensor. To do this, construct a new graph \( G' \) from \( G \) by adding an auxiliary vertex \( x \) and an undirected edge of weight zero from \( x \) to every vertex in \( G \) marked as containing a motion sensor, and then run Dijkstra from \( x \) in \( O(n \log n) \) time. Then \( \delta(x, v) \) corresponds to the shortest distance labels from \( v \) to any sensor as desired. Then we can construct \( G_k \) for any \( k \) in \( O(n) \) time by looping through vertices, removing any vertex \( v \) for which \( \delta(v, k) \leq k \) (also removing any edge adjacent \( v \)). Further, we can check whether \( t \) is reachable from \( s \) in \( G_k \) in \( O(n) \) time.
via breath-first search or depth-first search. We could find \( k^* \) in \( O(n k^*) \) time by simply constructing \( G_k \) for every \( k \in \{1, \ldots, k^* + 1\} \), but we can find \( k^* \) faster via binary search. Sort the vertices by their distance \( \delta(x, v) \) from any sensor in \( O(n \log n) \) time using any optimal comparison sort algorithm (e.g., merge sort), and let \( d_i \) be the \( i^{\text{th}} \) largest distance in this sorted order for \( i \in \{1, \ldots, n\} \), where \( d_1 \) is the smallest and \( d_n \) is the largest. Then construct \( G_{d_i} \) for \( i = \lceil n/2 \rceil \) and check whether \( t \) is reachable from \( s \) in \( G_{d_i} \) in \( O(n) \) time. If it is, recurse for \( i > \lceil n/2 \rceil \), and if not, recurse for \( i < \lceil n/2 \rceil \). The binary search proceeds in \( O(\log n) \) iterations that each take at most \( O(n) \) time until finding the smallest \( d_i = d^* \) such that \( t \) is not reachable from \( s \). Then \( t \) is reachable from \( s \) for \( k < d^* \) but not for \( k \geq d^* \), so \( k^* = d^* - 1 \). Finally, we can use either breadth-first search or depth-first search while storing parent pointers to reconstruct some path from \( s \) to \( t \) in \( G_{k^*} \) in \( O(n) \) time. If \( k^* = 0 \), every path from \( s \) to \( t \) goes through a junction containing a sensor, so any path from \( s \) to \( t \) may be returned.

**Problem 7-4. Critter Collection**

Ashley Getem (from PS3) is trying to walk from Trundle Town to Blue Bluff, which are both clearings in the Tanko region. She has a map of all clearings and two-way trails in Tanko. Each of the \( n \) clearings connects to at most five trails, while each trail \( t \) directly connects two clearings and is marked with its length \( \ell_t \) and capacity \( c_t \) of critters living on it, both positive integers. Ashley is a compulsive collector and will collect every critter she comes across by throwing an empty Pocket Sphere at it (which will fill the Pocket Sphere with \( c_t \) critters, since Ashley will use up all of them). Ashley also has \( n \) stores where Ashley can buy empty Pocket Spheres, and deposit full ones. Ashley has more money than she knows what to do with, but her backpack can only hold \( m \) Pocket Spheres at a time. Ashley has a map, describe an \( O(n k \log(nk)) \)-time algorithm to return the shortest route for Ashley to walk from Trundle Town (with a backpack full of empty Pocket Spheres) to Blue Bluff without ever being sad, or return that sadness is unavoidable.

**Solution:** Construct a graph \( G = (V, E) \) with \( k + 1 \) vertices for each clearing, where vertex \( v_{c,i} \) corresponds to being at clearing \( c \) while having \( i \) empty pocket spheres. Then, for each directed pair of clearings \( (a, b) \) for which \( a \) and \( b \) are connected by a trail \( t \), having length \( \ell_t \) and critters, add the following directed edges:

- If clearing \( a \) does not contain a store, add an edge of weight \( \ell_t \) from \( v_{a,i} \) to \( v_{b,i-c_t} \) for every \( i \in \{c_t, \ldots, k\} \), since Ashley will use up \( c_t \) pocket spheres by traversing the trail.
- Otherwise, if clearing \( a \) contains a store, add an edge of weight \( \ell_t \) from \( v_{a,i} \) to \( v_{b,k-c_t} \) for every \( i \in \{0, \ldots, k\} \), since it is never bad for Ashley to completely fill her backpack with empty pocket spheres whenever she leaves a store.

\( G \) has \( (k + 1)n = O(kn) \) vertices and at most \( k \) edges per trail. Since there are at most five trails per clearing, there are at most \( 5kn = O(kn) \) edges in \( G \), so \( G \) has size \( O(nk) \). Let \( s \) be the vertex associated with Trundle Town and \( t \) be the vertex associated with Blue Bluff. Then any path from \( v_{a,i} \) to any vertex \( v_{t,i} \) for \( i \in \{0, \ldots, k\} \) in \( G \) avoids sadness, and every path that avoids sadness corresponds to a path in \( G \). Since the weights in \( G \) are non-negative, we can run Dijkstra, storing parent pointers to reconstruct a shortest route from \( s \) to \( t \) that avoids sadness in \( O(nk \log(nk)) \) time. If the shortest path weight to every \( v_{t,i} \) in \( G \) is infinite, then return that sadness is unavoidable.
Problem 7-5. Shipping Servers

The video streaming service UsTube has decided to relocate across country, and needs to ship their servers by truck from San Francisco, CA to Cambridge, MA. They will pay third-party trucking companies to transport servers from city to city. An intern at UsTube has compiled a list of server names from San Francisco to Cambridge (s_l, t_l, w_l, c_l) where s_l and t_l respectively the starting and ending cities of the trucking route, w_l is the positive integer weight capacity of the truck, and c_l is the positive integer cost of shipping along that route (cost is the same for shipping any weight from 0 to w_l). Note that the existence of a shipping route from s_l to t_l does not imply a shipping route from t_l to s_l. Some of UsTube’s servers are too heavy to fit on any truck, so they will need to transfer them to smaller servers for the move. Assume that it is possible to ship some finite weight from San Francisco to Cambridge via some routes in R. Help UsTube evaluate their shipping options.

(a) (Useful Digression) Given a weighted path \( \pi \), its bottleneck is the minimum weight of any edge along the path. Given a directed graph containing vertices s and t, let \( b(s, t) \) denote the maximum bottleneck of any path from s to t, and let \( I(t) \) denote the set of incoming neighbors of t. Argue that \( b(s, t) \geq \min(b(s, v), w(v, t)) \) for every \( v \in I(t) \), and that \( b(s, t) = \min(b(s, v^\ast), w(v^\ast, t)) \) for at least one \( v^\ast \in I(t) \).

**Solution:** First, we argue that \( b(s, t) \geq \min(b(s, v), w(v, t)) \) for every \( v \in I(t) \). Suppose for contradiction that there exist two vertices \( s \) and \( t \) such that \( b(s, t) < \min(b(s, v), w(v, t)) \). Then there exists some path from \( s \) to \( v \) for which the minimum weight of any edge along the path is strictly greater than \( b(s, t) \). So extending that path along the edge from \( v \) to \( t \) would result in a path from \( s \) to \( t \) for which every edge along the path has weight greater than \( b(s, t) \), i.e., a larger bottleneck, contradicting that \( b(s, t) \) is the maximum bottleneck over all paths.

Next, we argue that \( b(s, t) = \min(b(s, v^\ast), w(v^\ast, t)) \) for at least one \( v^\ast \in I(t) \). Suppose for contradiction that \( b(s, t) > \min(b(s, v), w(v, t)) \) for every \( v \in I(t) \). Then there exists a path \( \pi \) from \( s \) to \( t \) whose bottleneck equals \( b(s, t) \). Let \( v \) be the vertex appearing before \( t \) along the path. The bottleneck of \( \pi \) cannot be the weight of the edge \( (v, t) \) or else \( b(s, t) = w(v, t) \); but the bottleneck can also not occur for some other edge in the path from \( s \) to \( v \) in \( \pi \), or else \( b(s, t) = b(s, v) \), a contradiction.

(b) Assuming that the number of cities appearing in any of the \( n \) trucking routes is less than \( 3\sqrt{n} \), describe an \( O(n) \)-time algorithm to return both: (1) the weight \( w^\ast \) of the largest single server that can be shipped from San Francisco to Cambridge via a sequence of trucking routes, and (2) the minimum cost to ship such a server with weight \( w^\ast \).

**Solution:** Suppose we knew \( w^\ast \). Then we could construct a graph \( G_C = (V_c, E_c) \) with a vertex for every city and a directed edge from vertex \( s_i \) to \( t_i \) weighted by \( c_i \) for every trucking route \( (s_i, t_i, w_i, c_i) \) for which \( w_i \geq w^\ast \), and then run Dijkstra from the vertex corresponding to San Francisco to find the minimum cost of shipping weight \( w^\ast \) to Cambridge. This graph has \( n \) edges and at most \( O(\sqrt{n}) \) vertices (by the problem statement), so Dijkstra runs in \( O(n) \) time, even when a direct access array or hash table is used for Dijkstra’s priority queue.

To find \( w^\ast \), we construct a graph \( G_w = (V_w, E_w) \), similar to \( G_c \), with a vertex for every city and a directed edge from vertex \( s_i \) to \( t_i \) weighted by \( w_i \) for every trucking route \( (s_i, t_i, w_i, c_i) \), and let \( s \) and \( t \) be the vertices associated with San Francisco and Cambridge respectively. The problem is then to determine the maximum bottleneck of any path from \( s \) to \( t \). We can modify

---

1Assume names are ASCII strings that can each be read in constant time.
Dijkstra to compute the bottleneck by replacing shortest path distance estimates with bottleneck estimates $b_s(v)$. Specifically, initialize all bottleneck estimates to zero (since not shipping anything achieves this), except for $b_s(s)$ which we set equal to positive infinity, since we have no limits at the start; and then add them all to a maximum priority queue. Then, repeatedly remove the vertex with largest bottleneck estimate from the priority queue and relax all outgoing edges from it. To relax an edge $(u, v)$ with weight $w_i$, set $b_s(v) = \max(\min(b_s(u), w_i), b_s(v))$ (either a path through $u$ to $v$ improves on the estimate for $b_s(v)$ or it does not).

To prove that this modification to Dijkstra is correct, we can follow an identical proof structure as the one provided in the Lecture 13 notes, proving correctness of normal Dijkstra. Specifically, we can prove the analogue of Claim 2: for every $v \in V_w$, when $v$ is removed from Dijkstra’s queue, $b_s(v) = b(s, v)$. We omit the full proof here. For this problem, students can receive full points for a description of a correct description, without a full proof of correctness.

After the modified Dijkstra has completed, $w^* = b_s(t) = b(s, t)$ by definition of $w^*$. Since we only changed the relaxation step from one constant time operation to another, and follow the same structure as Dijkstra on a graph with $O(n)$ edges and $O(\sqrt{n})$ vertices, this modified Dijkstra also runs in $O(n)$ time (using a direct access array or hash table for Dijkstra’s priority queue).

(e) Write a Python function `ship_server_stats(R, s, t)` that implements your algorithm from (b).
Solution:

```python
def dijkstra(Adj, w, s):
    # modified from R13
    d = [float('inf') for _ in Adj]
    d[s] = 0
    Q = [i for i in range(len(Adj))]  # direct access as priority queue
    while len(Q) > 0:
        u = Q[0]  # find min distance in Q
        for v in Q:
            if d[v] < d[u]:
                u = v
        Q.remove(u)  # remove min
        for v in Adj[u]:  # relax outgoing edges
            d[v] = min(d[v], d[u] + w(u, v))
    return d

def dijkstra_bottleneck(Adj, w, s):
    # modified from R13
    d = [0 for _ in Adj]  # initially vertices not reached
    d[s] = float('inf')  # source has arbitrary capacity
    Q = [i for i in range(len(Adj))]
    while len(Q) > 0:
        u = Q[0]  # find max capacity in Q
        for v in Q:
            if d[v] > d[u]:
                u = v
        Q.remove(u)  # relax outgoing edges
        for v in Adj[u]:
            d[v] = max(d[v], min(d[u], w(u, v)))
    return d

def ship_server_stats(R, s, t):
    n = 0
city_idx = {}  # label cities with integers
    for (_s, _t, _w, _c) in R:
        for city in (_s, _t):
            if city not in city_idx:
                city_idx[city] = n
                n += 1
    Adj = [[] for i in range(n)]  # construct graph
    w_w, w_c = {}, {}
    for (_s, _t, _w, _c) in R:
        si, ti = city_idx[_s], city_idx[_t]
        Adj[si].append(ti)
        w_w[(si, ti)], w_c[(si, ti)] = _w, _c
    for i in range(n):  # find max bottleneck weight
        w = dijkstra_bottleneck(Adj, lambda u,v: w_w[(u,v)], si)[ti]
        for j in Adj[i]:
            if w_w[(i, j)] < w:
                w_c[(i, j)] = float('inf')
    c = dijkstra(Adj, lambda u,v: w_c[(u,v)], si)[ti]  # compute min length
    return w, c
```

Problem Session 7