Problems

You may talk with your fellow CS161-ers about the problems. However:

- Please follow our Collaboration Policy on the course website.
- Try the problems on your own before collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

1. (6 pt.) In this exercise we’ll see how to adapt pseudocode for Dijkstra’s algorithm to return shortest paths (in addition to shortest distances). One way to do that is shown in the pseudocode below:

   ```plaintext
   Dijkstra_st_path(G, s, t):
   for all v in V, set d[v] = Infinity
   for all v in V, set p[v] = None
   // we will use the information p[v] to reconstruct the path at the end.
   d[s] = 0
   F = V
   while F isn’t empty:
     x = a vertex v in F so that d[v] is minimal
     for y in x.outgoing_neighbors:
       d[y] = min( d[y], d[x] + weight(x,y) )
       if d[y] was changed in the previous line, set p[y] = x
     F.remove(x)
   // use the information in p to reconstruct the shortest path:
   path = [t]
   current = t
   while current != s:
     current = p[current]
     add current to the front of the path
   return path, d[t]
   ```

Step through `Dijkstra_st_path(G, s, t)` on the graph `G` shown below. Complete the table below (on the next page) to show what the arrays `d` and `p` are at each step of the algorithm, and indicate what path is returned and what its cost is.

Graph, table, and more problems on next page...
We are expecting: The following things: The table below filled out, AND the shortest path and its cost that the algorithm returns. No justification is required.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>When entering the first while loop for the first time, the state is:</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Immediately after we remove the first element from F, the state is:</td>
<td>0</td>
<td>3</td>
<td>∞</td>
<td>9</td>
<td>None</td>
<td>s</td>
<td>None</td>
<td>s</td>
</tr>
<tr>
<td>Immediately after we remove the second element from F, the state is:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediately after we remove the third element from F, the state is:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediately after we remove the fourth element from F, the state is:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. (2 pt.) Your friend offers the following way to patch up Dijkstra’s algorithm to deal with negative edge weights. Let G be a weighted graph, and let \(w^*\) be the smallest weight that appears in G. (Notice that \(w^*\) may be negative). Consider a graph \(G' = (V, E')\) with the same vertices, and so that \(E'\) is as follows: for every edge \(e \in E\) with weight \(w\), there is an edge \(e' \in E'\) with weight \(w - w^*\). Now all of the weights in \(G'\) are non-negative, so we can apply Dijkstra’s algorithm to that:

\[
\text{modifiedDijkstra}(G, s, t):
\]
\[
\text{Construct } G' \text{ from } G \text{ as above.}
\]
\[
\text{return Dijkstra_st_path}(G', s, t)
\]

Prove or disprove: Your friend’s approach will always correctly return a shortest path between \(s\) and \(t\) if it exists.

[We are expecting: Either a formal proof or a (small) counterexample, where “small” means no more than 5 vertices].
3. (12 pt.) How to do it?

(a) (4 pt.) The **diameter** of a strongly connected directed graph $G = (V, E)$ is the largest distance between any two vertices in $V$. That is,

$$\text{diam}(G) = \max_{u,v \in V} \text{dist}(u,v),$$

where as usual the distance $\text{dist}(u,v)$ between vertices $u, v \in V$ is the cost of a shortest directed path from $u$ to $v$; in an unweighted graph, it is the number of edges on a shortest directed path from $u$ to $v$.

Suppose that $G = (V, E)$ is an unweighted strongly connected directed graph on $n$ vertices with $m$ edges. Design an algorithm that finds the diameter of $G$ in time $O(nm)$.

[**We are expecting:** A clear English description of your algorithm (pseudocode is not necessary, but you may include it if it helps clarify your algorithm).]

(b) (4 pt.) Let $G = (V, E)$ be a directed weighted graph on $n$ vertices with $m$ edges, and let $s \in V$ be a vertex. Suppose there are no negative-weight cycles, and only the edges going out of $s$ may have negative weights. From lecture, we learned that Bellman-Ford can solve the single-source shortest path problem on any graph (without negative cycles). Design an algorithm that runs in time $O(m + n \log n)$ that computes the shortest path distances from source $s$ to all other nodes on this graph, and briefly justify why it is expected to work properly on this particular kind of graph.

[**We are expecting:** A clear English description of your algorithm (pseudocode is not necessary, but you may include it if it helps clarify your algorithm), AND an informal justification of why your algorithm is correct.]

(c) (4 pt.) Let $G = (V, E)$ be a directed weighted graph on $n$ vertices with $m$ edges, where all the edge weights are either 1, 2, or 3. Let $s \in V$ be a vertex. Design an algorithm that runs in time $O(n + m)$ that computes the shortest path distances from source $s$ to all other nodes on this graph, and briefly justify the runtime.

[**We are expecting:** A clear English description of your algorithm (pseudocode is not necessary, but you may include it if it helps clarify your algorithm), AND an informal justification that the running time is $O(n + m)$.)

---

1For this definition, it is okay if the “distance” is negative if there are negative edge weights in $G$; assume that there are no negative cycles in $G$. 

---

More problems on next page...
4. **(12 pt.) [Messenger Owls.]** The postal services of a certain town are conducted exclusively via messenger owls. Each messenger owl is in charge of the route between two particular owl post offices, and each owl is only able to carry up to a certain weight of letters during any flight. This network of owl routes is modeled by a connected, undirected graph $G$ with $n$ vertices and $m$ edges (see below), where vertices represent the owl post offices in this town, and each edge represents the route that one of the owls is able to provide. Each edge is labeled with the maximum weight (in grams, let’s say) that the owl associated with that edge is able to carry. In other words, a letter of weight $x$ can only be delivered across edges whose label is $\geq x$.

For example, in the graph below, a letter with weight $x = 8$ could get from $v_2$ to $v_4$ (either by $v_2 \rightarrow v_0 \rightarrow v_1 \rightarrow v_4$ or by $v_2 \rightarrow v_3 \rightarrow v_0 \rightarrow v_1 \rightarrow v_4$). However, a letter of weight $x = 15$ could not get from $v_2$ to $v_4$.

The graph is stored in the adjacency-list format we discussed in class. More precisely, $G$ has vertices $v_0, \ldots, v_{n-1}$ and is stored as an array $V$ of length $n$, so that $V[i]$ is a pointer to the head of a linked list $N_i$ which stores integers. An integer $j \in \{0, \ldots, n-1\}$ is in $N_i$ if and only if there is an edge between the vertices $v_i$ and $v_j$ in $G$. Feel free to iterate over $N_i$ in pseudocode by writing for $j$ in $N_i$, or something similar, rather than work with linked list pointers.

You have access to a function $\text{maxLetterWeight}$ which runs in time $O(1)$ so that if $\{v_i, v_j\}$ is an edge in $G$, then $\text{maxLetterWeight}(i,j)$ returns the maximum weight of any letter than can travel between $v_i$ and $v_j$. (Notice that $\text{maxLetterWeight}(i,j) = \text{maxLetterWeight}(j,i)$ since the graph is $G$ undirected). If $\{v_i, v_j\}$ is not an edge in $G$, then you have no guarantee about what $\text{maxLetterWeight}(i,j)$ returns.

(a) **(6 pt.)** Design a deterministic algorithm which takes as input $G$ in the format above, integers $s, t \in \{0, \ldots, n-1\}$, and a desired letter weight $x > 0$; the algorithm should return True if there is a route from $v_s$ to $v_t$ that a letter of weight $x$ could be delivered across, or False if no such path exists.

(For example, in the example above we have $s = 2$ and $t = 4$. Your algorithm should return True if $0 < x \leq 10$ and False if $x > 10$).

Your algorithm should run in time $O(n + m)$ (i.e. $O(m)$ since the graph is connected). You may use any algorithm we have seen in class as a subroutine, as long as your inputs to these subroutines are clearly stated. You are allowed to say something like “Run BFS on $G$ starting from $x$. If $y$ is discovered during BFS: do $z$.” (no need to be describe these black boxes at a very low level).

**Note:** In your pseudocode, make sure you use the adjacency-list format for $G$ described above. For example, your pseudocode should not say something like “iterate over all edges in the graph.” Instead it should more explicitly show how to do that with the format described. (As mentioned in class, we will generally not be so pedantic about this in the future, but one point of this problem is to make sure you understand how the adjacency-list format works).

**[We are expecting:]** Pseudocode AND an English description of your algorithm, and a short justification of the running time. You should make sure to use the adjacency-list representation of $G$ described above in your pseudocode. You can use any algorithms we have seen from class as
a subroutine, but if you significantly modify them make sure to be precise about how this interacts with the adjacency-list representation.]

(b) (6 pt.) Design a deterministic algorithm which takes as input \( G \) in the format above and integers \( s, t \in \{0, \ldots, n - 1\} \); the algorithm should return the largest real number \( x \) so that there exists a route from \( v_s \) to \( v_t \) which accommodates a letter of weight \( x \). If \( s = t \), your algorithm may return \( \infty \). Your algorithm should run in time \( O((n + m)\log m) \). You may use any algorithm we have seen in class as a subroutine.

**Note:** Don’t assume that you know anything about the maximum letter weights (edge labels) ahead of time. (e.g., do not underestimate these owls; the maximum letter weights are not necessarily bounded integers).

**Note:** The same note about pseudocode holds as in part (a).

[HINT: A runtime of \( O((n + m)\log m) \) could result from performing part (a) \( O(\log m) \) times.]

[We are expecting: Pseudocode AND and English description of your algorithm, and a short justification of the running time. You should make sure to use the adjacency-list representation of \( G \) described above in your pseudocode. You can use any algorithms we have seen from class as a subroutine, but if you significantly modify them make sure to be precise about how this interacts with the adjacency-list representation.]
5. (12 pt.) (Wake up, Sheeple!) You arrive on an island with $n$ sheep. The sheep have developed a pretty sophisticated society, and have a social media platform called Baaaahtter (it’s like Twitter but for sheep). Some sheep follow other sheep on this platform. Being sheep, they believe and repeat anything that they hear. That is, they will re-post anything that any sheep they are following said. We can represent this by a graph, where $(a) \rightarrow (b)$ means that $(b)$ will re-post anything that $(a)$ posted. For example, if the social dynamics on the island were:

![Diagram of sheep and arrows]

then Sherman the Sheep follows Sugar the Sheep, and Sugar follows both Shakira and Shifra, and so on. This means that Sherman will re-post anything that Sugar posts, Sugar will re-post anything by Shifra and Shakira, and so on. (If there is a cycle then each sheep will only re-post a post once).

For the parts below, let $G$ denote this directed, unweighted graph on the $n$ sheep. Let $m$ denote the number of edges in $G$.

(a) (3 pt.) Call a sheep an influencer if anything that they post eventually gets re-posted by every other sheep on the island. In the example above, both Shifra and Shakira are influencers. 

Prove that all influencers are in the same strongly connected component of $G$, and every sheep in that component is an influencer.

[We are expecting: A short but rigorous proof.]

(b) (6 pt.) Suppose that there is at least one influencer. Give an algorithm that runs in time $O(n+m)$ and finds an influencer. You may use any algorithm we have seen in class as a subroutine.

[We are expecting: The following things:
- Pseudocode OR a very clear English description of your algorithm
- an informal justification that your algorithm is correct
- an informal justification that the running time is $O(n + m)$
You may use any statement we have proved in class without re-proving it.]

(c) (3 pt.) Suppose that you don’t know whether or not there is an influencer. Give an algorithm that runs in time $O(n + m)$ and either returns an influencer or returns no influencer. You may use any algorithm we have seen from class as a subroutine, and you may also use your algorithm from part (b) as a subroutine.

[We are expecting: The following things:
- Pseudocode OR a very clear English description of your algorithm
- an informal justification that your algorithm is correct
- an informal justification that the running time is $O(n + m)$
You may use any statement we have proved in class without re-proving it.]

More “problems” on next page...
Feedback
There’s no “correct” answer here, and it is completely anonymous.

6. (1 pt.) Please fill out the following feedback form, which asks about your thoughts on the course concepts we’ve covered so far and quizzes:

https://forms.gle/HndT6vZp3JGh2YJC9

Did you fill out the survey?
[We are expecting: The answer “yes.”]
Bonus Problem

This extra bonus question is entirely optional, so please feel free to completely ignore this section if you’ve got other things on your plate. And remember, bonus points do not directly boost a specific assignment score. Instead, they may be applied to your overall grade after the curves are determined.

7. (1 BONUS pt.) [Mystery Alphabet.] Given a list of \( n \) words which have been sorted lexicographically by some arbitrary alphabet of length \( k \), design an algorithm that returns the lexicographic order of the characters in the alphabet, in time \( O(k + n) \). For example, if you were given the list of words

\[
["za", "cba", "caa", "ba"]
\]

The alphabet which sorted these words is: "zcba". It is guaranteed that there is sufficient information provided from the given ordering of the words to derive the order for every character in the alphabet.

[We are expecting: An English description of your algorithm, pseudocode, a convincing justification of correctness, and justification of why the runtime is \( O(k + n) \).]

That’s all the problems we have for you!