COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 5

Announced on: Aug 22 (Thurs)

Problems marked with ($\star$) will not be asked in the tutorial quiz.

1. You are given a directed graph $G = (V, E)$ in the adjacency list format. You need to return the reverse graph G^{rev} where all edges have been flipped. The graph G^{rev} should also be represented using adjacency lists. Design a linear time algorithm for this problem.

   Notation: We denote the adjacency list of graph G as `adj`. The entry of `adj` corresponding to a vertex v is denoted by `adj[v]`. This entry consists of a singly linked list of vertices u such that (v, u) is an edge in the graph G. The adjacency list of the reverse graph G^{rev} is denoted by `adj'`.

   Algorithm for constructing G^{rev}: The input consists of the adjacency list representation `adj` of original graph G. The desired output is the adjacency list representation `adj'` of original graph G^{rev}.

   ```
   for v in V:
       for u in adj[v]:
           append v to adj'[u]
   ```

   We will now argue the correctness of this algorithm. To show that, we need to prove that an edge (u, v) exists in G iff the edge (v, u) exists in G^{rev}. The following claims establish these requirements.
   Claim 1: If (v, u) is an edge in G then (u, v) is an edge in G^{rev}.

   If (v, u) is an edge in G then u must be a member of `adj[v]`. This means that we must iterate over it in the for loop, so v gets added to `adj'[u]` according to the code, which means (u, v) is an edge in G^{rev}

   Claim 2: If (u, v) is an edge in G^{rev} then (v, u) is an edge in G.

   If (u, v) is in G^{rev} then we must have added v to `adj'[u]` in the statement in the inner loop. This means that u must be in `adj[v]`, hence (v, u) is an edge in G.

   The running time of this algorithm is $O(|E| + |V|)$, as we iterate over every edge of every vertex once.

2. Design a linear-time algorithm to check if a given directed graph is strongly connected.

Informal Idea: Use Kosaraju's algorithm as discussed in class to find SCC's and check whether all the vertices in the graph belong to the same SCC.

Time Complexity: The total time complexity is the sum of the time needed for Kosaraju's algorithm and the time required to check membership in the same SCC.

$$O(|E| + |V|) \text{ (Kosaraju's algorithm)} + \mathcal{O}(|V|) \text{ (checking membership in SCC)}$$

$$\implies O(|E| + |V|)$$

3. Consider a directed graph $G = (V, E)$ with nonnegative edge lengths and a starting vertex s. Define the *bottleneck* of a path to be the maximum length of one of its edges (as opposed to the sum of the lengths of its edges). Show how to modify Dijkstra's algorithm to compute, for each vertex $v \in V$, the smallest bottleneck of any $s - v$ path.

Informal Idea: To modify Dijkstra's algorithm to compute the smallest bottleneck of any path from a source s to each vertex, we track the *maximum* edge weight (bottleneck) encountered along a path rather than the *sum* of the edge weights. Starting from s, we initialize the bottleneck for s as 0 and for all other vertices as $+\infty$. As we explore a neighbor (say v) of a vertex u, we update the bottleneck for v as the maximum of the current bottleneck of u and the weight of the edge from u to v. If this new bottleneck is smaller than the current known bottleneck for v, we update it and continue. We will show that this algorithm ensures that, at the end, we have the smallest bottleneck path to each vertex.

ALGORITHM 1: Modified Dijkstra's Algorithm for Bottleneck Paths

Input: Graph $G = (V, E)$, source vertex s
Output: Array *bottleneck*, where $bottleneck[v]$ is the smallest bottleneck value of any $s - v$ path

```
1  Initialize bottleneck[s] = 0 and bottleneck[v] = ∞ for all v ≠ s;
2  Priority queue PQ;
3  PQ.push(0, s) // Push source with bottleneck 0
4  while PQ is not empty do
5      (bottleneck_u, u) = PQ.pop() // Extract vertex with smallest bottleneck
6      foreach neighbor v of u do
7          Let w_uv be the weight of edge (u, v);
8          new_bottleneck = max(bottleneck_u, w_uv);
9          if new_bottleneck < bottleneck[v] then
10             bottleneck[v] = new_bottleneck;
11             PQ.push(new_bottleneck, v);
12         end
13     end
14 end
15 return bottleneck;
```

Proof of Correctness:
Let $\delta(s, v)$ be the smallest bottleneck of any path from s to v.

Claim 1: $bottleneck[v] \geqslant \delta(s, v)$
Proof: This claim is trivial, bottleneck of any path from s to v can never be smaller than the smallest bottleneck.

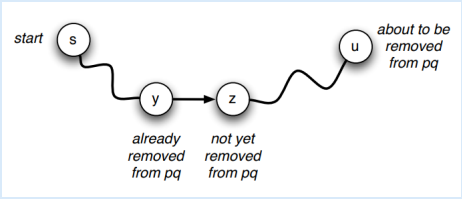


Figure 1: Path with smallest bottleneck from s to u

Note: In the figure, *pq* means priority queue.

Claim 2: The following invariant holds: in the while loop, when v is removed from the priority queue, $bottleneck[v] \leqslant \delta(s, v)$.
If this claim is true, then both the claim implies that $bottleneck[v] = \delta(s, v)$.

Proof: This is a proof by contradiction. Suppose the claim is false there is some node for which algorithm computes the wrong answer. In particular, let u be the first node removed from the priority queue such that $bottleneck[u] > \delta(s, u)$.

There is some path from s to u with smallest bottleneck. Let's call this path P. It's shown

in the figure above. Let's consider the moment in the execution of above algorithm when node u is being removed from the priority queue.
Let y be a node that lies on the path P and has already been removed. Let z be also on the path P but not yet removed. (Note: it's fair to assume that such nodes exist. For instance, we can set y = s and z = u.)
We make the following observations:

a) $bottleneck[y] = \delta(s, y)$: This is true because v is defined as the first node removed where $bottleneck[u] > \delta(s, u)$ and since y has already been removed it must be the case that $bottleneck[y] \leqslant \delta(s, y)$, and we have claim 1 which says $bottleneck[y] \geqslant \delta(s, y)$.

b) $bottleneck[z] \leqslant max(bottleneck[y], w(y, z))$: This is true because when we removed y from the queue, the for each loop ensures that we visited each neighbor of y, including z, and updated its cost. So if at that moment, $bottleneck[z] > max(bottleneck[y], w(y, z))$, then $bottleneck[z]$ would have been updated to be $bottleneck[z] = max(bottleneck[y], w(y, z))$. And notice that in the algorithm, bottleneck start at inf and can only decrease.

c) $bottleneck[u] \leqslant bottleneck[z]$: This is true because u is about to be removed and z is defined as a node not yet removed from the queue (so therefore its bottleneck must be not less than u's).

d) $\delta(s, z) = max(\delta(s, y), w(y, z))$ and $\delta(s, z) = max(\delta(s, z), \delta(z, u))$: These claims are true because these node lie on P which is defined to be the path with smallest bottleneck.

Putting together the above claims we have

$$bottleneck[u] \leqslant bottleneck[z] \leqslant max(bottleneck[y], w(y, z)) = \delta(s, z)$$

$$\delta(s, z) \leqslant max(\delta(s, z), \delta(z, u)) = \delta(s, u)$$

This contradicts our supposition that $bottleneck[u] > \delta(s, u)$. Therefore, our supposition is false and there can be no node such that $bottleneck[u] > \delta(s, u)$.

4. Consider a directed, weighted graph G where all edge weights are positive. You have one *voucher* that lets you traverse the edge of your choice for free (by changing its weight to zero). Design an $\mathcal{O}((|E| + |V|) \cdot \log |V|)$ time algorithm to find a minimum weight path between two vertices s and t using the voucher.

We solve the problem by constructing another graph $G' = (V', E')$. Essentially, G' contains two copies of G, say G_1 and G_2, with additional edges of zero weight going vertex u_1 of G_1 to v_2 of G_2 if the edge (u, v) exists in G. More formally:

V' contains two vertices, v_1 and v_2, for every vertex in $v \in V$. Lets call vertices $\{v_1\}$ vertices of type 1 and $\{v_2\}$ vertices of type 2. The edge set E' is the smallest set which satisfies the following criteria:

$$(u \xrightarrow{w} v) \in E \implies (u_1 \xrightarrow{w} v_1) \in E'$$
$$(u \xrightarrow{w} v) \in E \implies (u_2 \xrightarrow{w} v_2) \in E'$$
$$(u \xrightarrow{w} v) \in E \implies (u_1 \xrightarrow{0} v_2) \in E'$$

Lets call these edges of type 1, 2 and 3 respectively.

Claim 1: For a path from s_1 to t_2 in G', we cross exactly one edge of type 3.

Note that there are no edges going from vertices of type 2 to type 1, so there is no way of going back to type 1 once you are at a vertex of type 2. Furthermore the only way to go from a vertex of type 1 to one of type 2 is to go across an edge of type 3. Now because our path starts from s_1 and ends up at t_2, we need to cross an edge of type 3, and once we have crossed that edge we cannot cross another as we can no longer go back to a vertex of type 1.

Claim 2: Every path from s_1 to t_2 in G' corresponds to a path in G where we use one voucher which has the same length, and vice-versa.

Let the path in G' be $v_1^0, v_1^1, v_1^2, \ldots, v_1^a, v_2^{a+1}, \ldots, v_2^{a+b}, v_2^{a+b+1}$. Here $v_1^0 = s_1$ and $v_2^{a+b+1} = t_2$. We create a path-with-voucher in G as $v^0, v^1, v^2, \ldots, v^a, v^{a+1}, \ldots, v^{a+b}, v^{a+b+1}$ with the voucher being used on the edge $v^a \to v^{a+1}$. As edges of type 1 and 2 in G' have the same weights as edges in G and edges of type 3 have weight 0, the paths have the same lengths by construction. Similarly, we can also construct a path in G' for a path-with-voucher path in G.

To solve the problem we can simply run dijkstra on G' from s_1 to t_2. The time complexity of this algorithm is $\mathcal{O}((|E'| + |V'|) \cdot \log |V'|)$. As $|E'| = 3|E|$ and $|V'| = 2|V|$, we get the complexity as $\mathcal{O}((|E| + |V|) \cdot \log |V|)$

5. You are given a directed graph $G = (V, E)$ with a source vertex s and target vertex t. Each edge $e \in E$ has an associated probability $p_e \in [0, 1]$ of failure, and each edge fails independently of all others. Design an algorithm to find the most reliable path from s to t, that is, the one whose probability of failure is the least. Note that a path fails if any of the edges in the path fails. You may assume any operation with real numbers is $\mathcal{O}(1)$ time.

Informal Idea: The problem reduces to finding the maximum product $s - t$ path in G. We simplify this by taking logarithms of weights, which converts the products to

additions and then, normal Dijkstra can be applied to find shortest $s-t$ path.
We solve the problem using Dijkstra. Let an edge $e \in E$ have the weight $-\ln(1-p_e)$, with $-\ln(0)$ defined as ∞. Note that all of the weights are positive as $p_e \in [0,1]$.

Claim 1: A path with weights $w_1, w_2, \ldots, w_n$ has probability $e^{-\sum w_i}$ of not failing.

Note that the probability $p_1, p_2, \ldots, p_n$ corresponding to $w_1, w_2, \ldots, w_n$ are related by $(1-p_i) = e^{-w_i}$. Also the probability that all of the edges don't fail is $(1-p_1)(1-p_2)\ \ldots\ (1-p_n) = e^{-w_1}e^{-w_2}\ldots e^{-w_n} = e^{-\sum w_i}$.

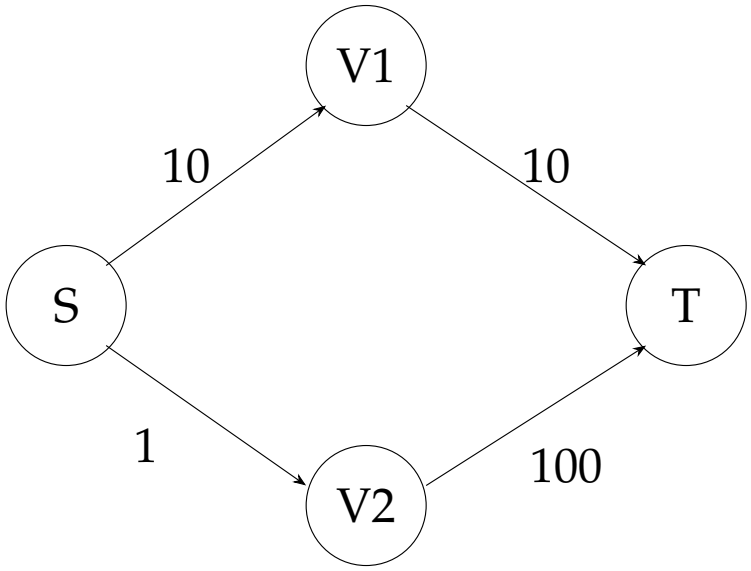
Now to maximize the probability of not failing, we need to minimize $\sum w_i$. This can be done by finding the shortest weighted path in the constructed graph using Dijkstra.

- Written "I do not know how to approach this problem" - 0.6 points
- Solution 1:
  - Taking logarithm of weights and finding the shortest $s-t$ path using Dijkstra - 2 points
  - Showing that shortest path when log of weights are taken corresponds to the maximum product path in the original setting - 1 point
  - Solution 2:
    * Running Dijkstra w.r.t multiplication of weights correctly - 1.5 points
    * Proving the high-level correctness of Dijkstra in case of multiplication of weights (lying in range $[0,1]$) to find the maximum product path from s to t - 1.5 points

6. Is it possible to solve the single-source *longest* path problem by changing minimum to maximum in Dijkstra's algorithm? If so, prove the correctness of your algorithm. If not, provide a counterexample.

No. It is not possible to solve the single-course longest path problem by changing minimum to maximum in Dijkstra's algorithm.

Counterexample: Consider the following graph:



The run of the modified Dijkstra (from S to T) on this graph would be:

```
1. Frontier = {S: 0}
2. Process S, Frontier = {V1: 10, V2: 1}
3. Process V1, Frontier = {V2:1, T: 20}
4. Process T, reached destination
```

As you can see, we get the path $S \to V1 \to T$, but the correct path is $S \to V2 \to T$ as it has a length of 101, while the one we got has a length of 20
Note: The longest path problem is NP-Hard and hence, no polynomial-time algorithm is known to solve it till now.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Claiming that Dijkstra does not solve longest path problem - 1 point
  - Correct counterexample - 2 points

7. Consider a directed graph in which the only negative edges are those that leave s; all other edges are positive. Can Dijkstra's algorithm, starting at s, correctly compute the length of shortest paths to all other vertices? Prove your answer.

If the graph has a negative length cycle then Dijkstra will give the wrong answer, as a "shortest path" doesn't exist in a graph with a negative cycle (as you

can go around that cycle again and again), while Dijkstra can only give us simple paths.

If we assume that no negative length cycle exists, Dijkstra will give the correct solution. To prove this statement, lets construct a modified graph G' with only positive weight edges. Let w_{min} be the minimum weight of all of the edges exiting s in G. G' is constructed to be the same as G, but the edges exiting s have $-w_{min}$ added to their weights. This way all edges in G' have positive weights.

Claim 1: Every simple path in G' from s to any node $v \neq s$ has length $-w_{min}$ more than the same path in G.

This is true because there is always exactly one edge exiting s in a simple path from s to v. If not then the path isn't simple because there will be a cycle from s to s. As the weights of all other edges are the same, the length of the path is just the difference of the weights in the one edge exiting s, which is $-w_{min}$.

Claim 2: Every run of Dijkstra on G will correspond to a valid run of Dijkstra on G', with just the distances of nodes other than s being $-w_{min}$ less.

We can prove this claim by induction on the iterations of Dijkstra:
Hypothesis: The distances of every vertex (other than s) in the run of Dijkstra on G from s are exactly $-w_{min}$ less than the distances calculated in the run on G'.

Base case: For the first node added, this claim is true as we would take the minimum length outgoing edge from s which are the same in both G and G'. The difference between the distances would be exactly $-w_{min}$.

Induction case: Lets assume that at an arbitrary iteration of the algorithm the distances calculated are exactly $-w_{min}$ less than the ones in G'. For the next iteration we would take the minimum distance vertex from the frontier. There are two cases:

a) We take an edge $s \rightarrow v$: In this case the argument is the same as the base case.

b) We take an edge $u \rightarrow v$ where $u \neq s$. Here the distance calculated for u would be exactly $-w_{min}$ less than for u in G'. As the weights for $u \rightarrow v$ in both graphs are the same, the distance for v would also be $-w_{min}$ less.

Hence we prove that the run for Dijkstra on G would calculate the distances as exactly $-w_{min}$ less than in G', which proves that the distances would be correct because of claim 1.

Note:

- Written "I do not know how to approach this problem" - 0.6 points
- Solution 1:

- Correct counterexample for Dijkstra's algorithm failing - 3 points

- Solution 2: Assuming no negative weight cycles.
  - Claiming that there will exist shortest paths without any cycles, i.e., shortest paths are *simple* - 1 point
  - Claiming that every *simple* path from s to any vertex v contains exactly 1 outgoing edge of s - 1 point
  - Claiming that adding a positive constant to all the negative edges (to make them positive) does not affect the shortest paths - 0.5 points
  - Claiming that run of Dijkstra on $G^{'}$ corresponds to a run of Dijkstra on G - 0.5 points

8. ($\star$) Alice and Bob are taking a road trip from City X to City Y. They decide to switch the driving duties at each rest stop they visit; however, because Alice has a better sense of direction than Bob, she should be driving both when they depart and when they arrive (to navigate the city streets).

   Given a weighted undirected graph $G = (V, E, w)$ with positive edge weights, where the vertices represent rest stops and the edges represent routes between rest stops, devise a polynomial-time algorithm to find a route (if possible) of minimum distance between City X and City Y such that Alice and Bob alternate edges and Alice drives the first and last edge.

   Informal Idea: We can easily infer that a shortest path with odd number of edges from City X to City Y satisfy the required condition.
   We solve the problem by constructing another graph $G' = (V', E')$ where V' contains two copies v_1, v_2 of every vertex $v \in V$ and if $(u, v) \in E$ with weight $w(u, v)$ then (u_1, v_2) and $(v_1, u_2) \in E'$ with weight $w(u, v)$. The Graph G' is a bipartite graph with vertex sets $V_1 = \{v_1 : v \in V\}$ and $V_2 = \{v_2 : v \in V\}$.
   Now using Dijkstra check whether there is a shortest path from City X_1 to City Y_2 in graph G'.

   Claim: There exists a shortest path of length w with odd number of edges from City X to City Y in graph G if and only if there exists a shortest path of length w from City X_1 to City Y_2 in graph G'.
   Proof: ($\Rightarrow$) Suppose there exists a shortest path of length w with odd number of edges from City X to City Y in graph G. Let this path P be $X - u - \cdots - v - Y$. By construction, every edge $(u, v) \in E \implies (u_1, v_2)$ and $(u_2, v_1) \in E'$. Since number of edges in path P is odd and it starts from $X_1 \in V_1$ then the corresponding path in G' would be $X_1 - u_2 - \cdots - v_1 - Y_2$. This corresponding path will be a shortest path otherwise it would mean that path P is not a shortest path.
   ($\Leftarrow$) Suppose there exists a shortest path of length w from City X_1 to City Y_2 in

graph G'. Since $X_1 \in V_1$ and $Y_2 \in V_2$ then number of edges in the shortest length path will be odd because of the bipartite property. Also by construction (u_1, v_2) or $(u_2, v_1) \in E' \implies (u, v) \in E$ with same corresponding weight. This path will be a shortest path from X to Y else it would mean that our assumption was wrong. Hence, there is a shortest path of length w with odd number of edges from City X to City Y in graph G.

Time Complexity: The time complexity of this algorithm is $\mathcal{O}(|E'| + |V'| \cdot \log |V'|)$. As $|E'| = 2|E|$ and $|V'| = 2|V|$, we get the complexity as $\mathcal{O}(|E| + |V| \cdot \log |V|)$.

9. ($\star$) A *tournament* is a directed graph $G = (V, E)$ such that for all $u, v \in V$, exactly one of (u, v) or (v, u) is in E.
   a) Show that every tournament has a *Hamiltonian path*, i.e., a path that visits every vertex exactly once.
   b) Design a polynomial-time algorithm to find this path.

   a) **Informal Idea:** We can inductively find a Hamiltonian path on a *tournament* of $n + 1$ vertices from a *tournament* of n vertices.

   Proof. We will use *induction*.

   Inductive Hypothesis- $P(n)$**:** Any *tournament* with $n \geqslant 1$ vertices contains a *Hamiltonian path*

   Base Case - $P(1)$**:** It can be easily seen that a graph with just one vertex contains a Hamiltonian path, which is just that vertex itself. □

   Inductive Step- $P(n) \implies P(n+1)$**:**

   - Consider a *tournament* $G = (V, E)$ such that $|V| = n + 1$. We need to prove that G has a Hamiltonian path.
   - Consider an arbitrary $v \in V$. Delete v from G and call this graph G', i.e., $G' = G \backslash v$. It can be seen that G' is also a *tournament*.
   - Since $P(n)$ is true, G' contains a Hamiltonian path P. Assume that P contains the vertices $(v_1, v_2, \ldots, v_n)$ in the given order, i.e., $P = v_1 \to v_2 \to \cdots \to v_n$. Note that the vertices can always be renumbered for this to hold.
   - Since G' is a *tournament*, either $(v_i, v) \in E$ or $(v, v_i) \in E$ for all $1 \leqslant i \leqslant n$. If $(v_n, v) \in E$, then (v_n, v) appended to P is a Hamiltonian path in G' and hence, we are done. Similarly, we are also done if $(v, v_1) \in E$.
   - So, we assume $(v, v_1) \notin E$ and $(v_n, v) \notin E$. As G' is a *tournament*, we get that $(v_1, v) \in E$ and $(v, v_n) \in E$.

- Due to the above claim, we get that there exists an index j such that $(v_j, v) \in E$ and $(v, v_{j+1}) \in E$.
- So, we get that the path $P = v_1 \to v_2 \to \dots v_j \to v \to v_{j+1} \to \cdots \to v_n$ is a Hamiltonian path in $G^{'}$.

Hence, Proved.

b) An algorithm for finding a Hamiltonian path is immediate from the inductive step of existence proof in the previous part: Remove an arbitrary vertex v from G, find a Hamiltonian path in the graph $G \backslash v$ and append v at an appropriate position, as mentioned in the existence proof.

Try it yourself: What is the time complexity of the above algorithm if you find appropriate position for v naively? Can you do better, by finding the appropriate position for v more smartly?