CS/ECE 374 Spring 2025 Heng An Cheng (hacheng2@illinois.edu)

Homework 9 Problem 1

Solution:

(a) Idea

We first reduce the problem to a Directed Acyclic Graph (DAG) by computing the Strongly Connected Components (SCCs) of G. Within each SCC, all nodes can reach each other, so we can collapse each SCC into a single "meta-node" in a new DAG, where each meta-node's value is the minimum $b(v)$ among its constituent vertices.

Then, we propagate these minimum values along the DAG in topological order. For each SCC, we keep track of the smallest b-value that can reach it, updating from its predecessors in the DAG.

Algorithm Steps

1 Compute the SCCs of G using Kosaraju's or Tarjan's algorithm.

2 Build the meta-graph G_{meta} where each node represents an SCC.

3 For each SCC C, compute $minB(C) = \min\{b(v) \mid v \in C\}$.

4 Topologically sort the DAG G_{meta}.

5 For each SCC in topological order:

- For each neighbor C' of C, update $minB(C') = \min(minB(C'), minB(C))$.

6 For each original node u, set $a(u) = minB(C_u)$, where C_u is the SCC containing u.

Pseudocode

Algorithm 1 Compute $a(u)$ For all $u \in V$

```
1: Compute SCCs of $G$, label each node $u$ with its SCC id $C_u$
2: for each SCC $C$ do
3:     $minB[C] \leftarrow \min\{b(v) \mid v \in C\}$
4: end for
5: Build DAG $G_{meta}$ of SCCs
6: $order \leftarrow$ TopologicalSort($G_{meta}$)
7: for each SCC $C$ in $order$ do
8:     for each neighbor SCC $C'$ of $C$ do
9:         $minB[C'] \leftarrow \min(minB[C'], minB[C])$
10:    end for
11: end for
12: for each node $u \in V$ do
13:     $a(u) \leftarrow minB[C_u]$
14: end for
```

Time Complexity Analysis

- Computing SCCs: $\mathcal{O}(|V| + |E|)$
- Building the meta-graph: $\mathcal{O}(|V| + |E|)$
- Topological sort: $\mathcal{O}(|V| + |E|)$ (on the DAG)
- Propagating min values: $\mathcal{O}(|V| + |E|)$
- Final assignment to each node: $\mathcal{O}(|V|)$

Total time complexity: $\mathcal{O}(|V| + |E|)$ — linear in the size of the graph.

(b) **Idea**

The idea is to work on the **meta-graph** (or **condensation**) of G:

1 Compute the Strongly Connected Components (SCCs) of G in $O(|V| + |E|)$ time (using Tarjan's or Kosaraju's algorithm).

2 Build the **meta-graph** G' where each vertex represents an SCC of G; an edge (C_i, C_j) exists if there is an edge from some node in C_i to some node in C_j.

3 Since G' is a Directed Acyclic Graph (DAG), we count its sources (nodes with no incoming edges) and sinks (nodes with no outgoing edges).

It is known that to make a DAG strongly connected, one needs to add at least

$$\max(\text{number of sources}, \text{number of sinks})$$

edges. Thus, if $\max(\text{sources}, \text{sinks}) > 1$, then one edge is insufficient. Conversely, if there is exactly one source and one sink (i.e., $\max(\text{sources}, \text{sinks}) = 1$), one edge can be added (from the unique sink to the unique source) to make G strongly connected.

Pseudocode

Algorithm 2 Check if One Edge is Sufficient For Strong Connectivity

```
Input: Directed graph $G = (V, E)$
Output: True if adding at most one edge makes $G$ strongly connected, otherwise False

Compute SCCs of $G$ using Tarjan's or Kosaraju's algorithm.
Build the meta-graph $G'$ where each node represents an SCC.
Initialize sourceCount and sinkCount to 0.
for each node $C$ in $G'$ do
    if $C$ has no incoming edges then
        sourceCount $\leftarrow$ sourceCount $+1$
    end if
    if $C$ has no outgoing edges then
        sinkCount $\leftarrow$ sinkCount $+1$
    end if
end for
if $\max($sourceCount, sinkCount$) = 1$ then
    return True                    ▷ One edge is enough (add edge from sink to source)
else
    return False                   ▷ More than one edge required
end if
```

Time Complexity Analysis

- **Computing SCCs:** $O(|V| + |E|)$.
- **Building the meta-graph G':** $O(|V| + |E|)$ (each edge is inspected once to determine inter-SCC connections).
- **Counting sources and sinks:** $O(|V|)$ on the meta-graph, noting that the number of SCCs is at most $|V|$.

Overall time complexity: $O(|V| + |E|)$.

(c) Idea

Because G may contain cycles, a natural idea is to reduce the problem to a Directed Acyclic Graph (DAG) via **strongly connected components (SCCs)**. The key observation is that when a walk enters an SCC, it is possible to collect all rewards on the edges inside that SCC. This is true because:

- Within an SCC the vertices are mutually reachable.
- All edge rewards are nonnegative.
- Even if some extra (zero-reward) traversals are needed, a careful walk can eventually cover every edge in the SCC exactly once (collecting its reward) before leaving.

Thus, we proceed with the following two main steps:

- **Collapse each SCC:** Compute the SCCs of G (in $O(|V|+|E|)$ time) and build the *meta-graph* G' where each node corresponds to an SCC. Since every vertex within an SCC is mutually reachable, if the walk can enter an SCC then it can collect the total reward of that component.
- **Dynamic Programming on the DAG:**
  - For each SCC C, define
$$R_{\text{in}}(C) = \sum_{\substack{e=(u,v)\\u,v\in C}} p(e)$$
    i.e. the total reward from all edges that lie completely inside C.
  - In the meta-graph G' (which is a DAG), for every edge $e=(u,v)$ in the original graph that crosses from C to C' (with $C \neq C'$), the reward $p(e)$ is collected at the moment of transition.

  Then define a dynamic programming value $dp(C)$ as the maximum total reward collected on a walk that starts in the SCC containing s and ends in the SCC C. When transitioning from one SCC to the next, add both the inter-SCC edge reward and the total internal reward of the target SCC.

Algorithm Details

Let SCC(s) denote the SCC containing the start node s. We perform the following steps:

1 **Compute SCCs:** Use Tarjan's or Kosaraju's algorithm to compute the SCCs of G.
Time: $O(|V|+|E|)$.

2 **Compute Internal Rewards:** For each SCC C, compute
$$R_{\text{in}}(C) = \sum_{\substack{e=(u,v)\\u,v\in C}} p(e)$$
This can be done by scanning each edge once.
Time: $O(|E|)$.

3 **Build the Meta-graph G':** Construct a DAG $G'=(\mathcal{C},E')$ where each node represents an SCC C, and add an edge from C to C' if there exists an edge $e=(u,v)$ in G with $u\in C$ and $v \in C'$ (note $C\neq C'$). Each such edge e is associated with a reward $p(e)$.
Time: $O(|V|+|E|)$.

4 **Dynamic Programming on G':**

   i. Topologically sort G'.
   Time: $O(|\mathcal{C}|+|E'|) \le O(|V|+|E|)$.

   ii. Initialize the dp value for each SCC C:
$$dp(C) = -\infty \quad \text{for all } C \neq \text{SCC}(s)$$
   and set
$$dp(\text{SCC}(s)) = R_{\text{in}}(\text{SCC}(s)).$$

iii. Process the nodes in topological order. For each SCC C and for each edge from C to a neighbor C' in G', consider every edge e from some $u \in C$ to $v \in C'$. Then update:

$$dp(C') \leftarrow \max\{dp(C'),\ dp(C) + p(e) + R_{\text{in}}(C')\}.$$

(Note that if there are multiple edges from C to C', they provide different amounts of reward. In a single walk you can only take one transition from C to C', so we maximize over all possibilities.)

5 **Return the Answer:** The maximum reward that can be collected is

$$\max_{C \in \mathcal{C}} dp(C),$$

where the maximum is taken over all SCCs reachable from SCC(s).

Pseudocode

Algorithm 3 Maximum Reward Walk Starting at s

```
1: Input: Directed graph $G = (V, E)$, edge rewards $p(e) \geq 0$, start node $s$.
2: Output: Maximum reward collectible on a walk starting at $s$.
3: Compute the SCC decomposition of $G$, labeling each vertex with its SCC id.
4: for each SCC $C$ do
5:     $R_{\text{in}}(C) \leftarrow 0$
6: end for
7: for each edge $e = (u, v)$ in $G$ do SCC($u$) = SCC($v$)
8:     $R_{\text{in}}(\text{SCC}(u)) \leftarrow R_{\text{in}}(\text{SCC}(u)) + p(e)$
9: end for
10: Build the meta-graph $G' = (\mathcal{C}, E')$:
11: for each edge $e = (u, v)$ in $G$ do SCC($u$) $\neq$ SCC($v$)
12:     Add edge from $C = \text{SCC}(u)$ to $C' = \text{SCC}(v)$ with reward $p(e)$ to $E'$
13: end for
14: Topologically sort $G'$ to obtain ordering $C_1, C_2, \ldots, C_k$.
15: for each SCC $C \in \mathcal{C}$ do
16:     $dp(C) \leftarrow -\infty$
17: end for
18: Let $C_s = \text{SCC}(s)$
19: $dp(C_s) \leftarrow R_{\text{in}}(C_s)$
20: for each SCC $C$ in topological order do
21:     for each outgoing edge $e = (C, C')$ with reward $p(e)$ do
22:         $dp(C') \leftarrow \max\{dp(C'),\ dp(C) + p(e) + R_{\text{in}}(C')\}$
23:     end for
24: end for
25: return $\max_{C \in \mathcal{C}} dp(C)$
```

Time Complexity Analysis

- Computing the SCCs: $O(|V| + |E|)$.
- Computing internal rewards: $O(|E|)$.

- Building the meta-graph: $O(|V| + |E|)$ (each edge is processed once).
- Topological sorting: $O(|\mathcal{C}| + |E'|) \leq O(|V| + |E|)$.
- Dynamic programming on the DAG: $O(|\mathcal{C}| + |E'|) \leq O(|V| + |E|)$.

Overall Time Complexity: $O(|V| + |E|)$.

■

CS/ECE 374 Spring 2025 Heng An Cheng (hacheng2@illinois.edu)

Homework 9 Problem 2

Solution:

(a) Example: 2-Norm vs. Standard Shortest Path

Consider the following directed graph with source s and target t:

Edge	$\ell(e)$
$s \to a$	4
$a \to t$	5
$s \to b$	7
$b \to t$	1

Standard Shortest Paths:

- Path $P_1 : s \to a \to t$ has standard length $4 + 5 = 9$.
- Path $P_2 : s \to b \to t$ has standard length $7 + 1 = 8$.

Path P_2 has a shortest path.

2-Norm Calculation: For $k = 2$, we compute:

- For P_1:
$$L_2(P_1) = \sqrt{4^2 + 5^2} = \sqrt{16 + 25} = \sqrt{41} \approx 6.40.$$
- For P_2:
$$L_2(P_2) = \sqrt{7^2 + 1^2} = \sqrt{49 + 1} = \sqrt{50} \approx 7.07.$$

Thus, while the standard shortest path favors P_2, the 2-norm metric favors P_1 (with a value of approximately 6.40) over P_2 (approximately 7.07).

Idea

The key observation is that for every edge $e \in E$, we can define a new weight

$$w(e) = \ell(e)^k.$$

Then, for any path P, the sum of the new weights is

$$\sum_{e \in P} w(e) = \sum_{e \in P} \ell(e)^k,$$

and minimizing this sum is equivalent to minimizing $L_k(P)$ (since raising to the power $1/k$ preserves the order).

Algorithm Description

Thus, to compute the shortest k-norm path from a source vertex s to a target vertex t, we can:

1 Replace each edge weight $\ell(e)$ with $\ell(e)^k$.

2 Run Dijkstra's algorithm on the modified graph starting at s.

3 The resulting shortest path minimizes $\sum_{e \in P} \ell(e)^k$, and therefore, P is also the shortest path under the k-norm.

Pseudo-code

Algorithm 4 Compute Shortest k-Norm Path from s to t

```
procedure ShortestKNormPath($G = (V, E)$, $\ell$, source $s$, target $t$, parameter $k$)
    for each edge $e \in E$ do
        $w(e) \leftarrow \ell(e)^k$
    end for
    $P \leftarrow$ Dijkstra($G = (V, E)$ with weights $w$, source $s$, target $t$)
    return $P$
end procedure
```

Time Complexity Analysis

- The transformation (computing $w(e) = \ell(e)^k$ for all edges) takes $\mathcal{O}(|E|)$ time.
- Dijkstra's algorithm runs in $\mathcal{O}((|V| + |E|) \log |V|)$ time when using a binary heap.

Thus, **the overall time complexity is**

$$\mathcal{O}((|V| + |E|) \log |V|).$$

(b) **Idea**

We adapt Dijkstra's algorithm by redefining the "distance" from s to a node v as the minimum possible **maximum edge weight** on any path from s to v.

Instead of summing edge weights, we update the bottleneck of a path using:

$$\text{bottleneck}(s, u) = \min\left(\text{bottleneck}(s, u), \max(\text{bottleneck}(s, v), \ell(v, u))\right).$$

Modified Dijkstra's Algorithm for Bottleneck Paths

Algorithm 5 BottleneckShortestPaths($G = (V, E)$, ℓ, source s)

```
1: for all $v \in V$ do
2:     $B[v] \leftarrow \infty$                ▷ Initialize bottleneck distance
3: end for
4: $B[s] \leftarrow 0$
5: $Q \leftarrow$ priority queue ordered by $B[v]$
6: while $Q$ is not empty do
7:     $v \leftarrow$ extract node from $Q$ with minimum $B[v]$
8:     for all neighbors $u$ of $v$ do
9:         $b \leftarrow \max(B[v], \ell(v, u))$
10:        if $b < B[u]$ then
11:            $B[u] \leftarrow b$
12:            update $u$ in $Q$
13:        end if
14:    end for
15: end while
16: return $B$
```

Explanation

The algorithm maintains for each node v the minimum bottleneck value $B[v]$ among all paths from s to v. During the relaxation step, we consider how adding edge (v, u) to the path affects the bottleneck — i.e., it becomes the maximum of the bottleneck so far and the new edge's weight. If this value is better than the previous value for u, we update it.

Time Complexity

- Initialization: $\mathcal{O}(|V|)$
- Each edge is relaxed at most once, with heap operations (priority queue updates): $\mathcal{O}(|E| \log |V|)$
- Thus, **total time complexity is:**

$$\mathcal{O}((|V| + |E|) \log |V|)$$

(c) **Idea**

Reduction to s–t Reachability

Given a parameter $\lambda \geq 0$, define the subgraph

$$G_\lambda = (V, E_\lambda), \quad \text{where } E_\lambda = \{e \in E : \ell(e) \leq \lambda\}.$$

Observe that there is an s–t path in G_λ if and only if there exists an s–t path in G whose bottleneck is at most λ. Hence, by deciding reachability in G_λ (using breadth-first search or depth-first search), we can determine whether the bottleneck distance from s to t is at most λ.

Binary Search Strategy

Let $\ell_{\min} = \min\{\ell(e) : e \in E\}$ and $\ell_{\max} = \max\{\ell(e) : e \in E\}$. The bottleneck distance $d_{BN}(s,t)$ must lie in the interval $[\ell_{\min}, \ell_{\max}]$. The strategy is:

1 Set $l = \ell_{\min}$ and $r = \ell_{\max}$.

2 While $l < r$, set $\lambda = \lfloor (l+r)/2 \rfloor$.

3 Construct G_λ and perform a reachability test from s to t.

4 If t is reachable in G_λ, then a valid path exists with bottleneck at most λ, and we update $r \leftarrow \lambda$.

5 Otherwise, update $l \leftarrow \lambda + 1$.

When the binary search terminates, $l = r$ will be the bottleneck distance.

Pseudocode

Algorithm 6 BottleneckShortestPath($G = (V,E)$, ℓ, source s, target t)

```
1:  $\ell_{\min} \leftarrow \min\{\ell(e) : e \in E\}$, $\ell_{\max} \leftarrow \max\{\ell(e) : e \in E\}$
2:  $l \leftarrow \ell_{\min}$,   $r \leftarrow \ell_{\max}$
3:  while $l < r$ do
4:      $\lambda \leftarrow \lfloor (l+r)/2 \rfloor$
5:      Construct $G_\lambda = (V, \{e \in E : \ell(e) \leq \lambda\})$
6:      if there is a path from $s$ to $t$ in $G_\lambda$ (using BFS or DFS) then
7:          $r \leftarrow \lambda$
8:      else
9:          $l \leftarrow \lambda + 1$
10:     end if
11: end while
12: return $l$                              ▷ Bottleneck distance from $s$ to $t$
```

Time Complexity

- **Reachability Test:** Each BFS/DFS in G_λ takes $\mathcal{O}(|V| + |E|)$ time.
- **Binary Search Iterations:** The range of edge values is between $\ell_{\min}$ and $\ell_{\max}$. If edge lengths are integers, the number of iterations is $\mathcal{O}(\log(\ell_{\max} - \ell_{\min}))$. If the edges are real numbers, one can perform binary search over the sorted list of edge weights, leading to $\mathcal{O}(\log |E|)$ iterations.

Thus, **the overall time complexity is:**

$$\mathcal{O}(\log |E| \cdot (|V| + |E|)).$$

(d) **Idea**

Since probabilities multiply along a path, it is natural to transform the problem using logarithms. Define for each edge e

$$w(e) = -\log(1 - p(e)).$$

Then, for a path P, we have:

$$\sum_{e\in P} w(e) = -\sum_{e\in P} \log(1 - p(e)) = -\log\left(\prod_{e\in P}(1 - p(e))\right).$$

Minimizing this sum is equivalent to maximizing

$$\prod_{e\in P}(1 - p(e)).$$

Thus, we can apply Dijkstra's algorithm to find the path from s to t with the minimum total weight, where the weight of an edge is defined as $w(e) = -\log(1 - p(e))$.

Modified Dijkstra's Algorithm

We now describe the algorithm in detail.

Algorithm 7 MaximumSuccessPath($G = (V, E)$, probabilities $p(e)$, source s, target t)

```
 1: for all $v \in V$ do
 2:     $d[v] \leftarrow \infty$                    ▷ Initialize distance estimates
 3: end for
 4: $d[s] \leftarrow 0$
 5: Initialize a priority queue $Q$ with $(s, d[s])$
 6: while $Q \neq \emptyset$ do
 7:     $v \leftarrow$ EXTRACT-MIN($Q$)
 8:     for all edges $(v, u) \in E$ do
 9:         $w \leftarrow -\log(1 - p(v, u))$
10:         if $d[v] + w < d[u]$ then
11:             $d[u] \leftarrow d[v] + w$
12:             DECREASE-KEY($Q, u, d[u]$)
13:         end if
14:     end for
15: end while
16: return $d[t]$       ▷ Minimum sum of $-\log(1 - p(e))$, so success probability is
                        $\exp(-d[t])$.
```

Explanation

- Each edge e has a weight $w(e) = -\log(1-p(e))$. Since $0 \le 1-p(e) \le 1$, it follows that $w(e) \ge 0$, and the non-negative weight property allows us to use Dijkstra's algorithm.

- The distance $d[v]$ found by Dijkstra's algorithm is the minimal sum

$$d[v] = \min_{P \in \mathcal{P}(s,v)} \sum_{e \in P} w(e) = -\log\left(\max_{P \in \mathcal{P}(s,v)} \prod_{e \in P} (1 - p(e))\right).$$

- Therefore, the maximum success probability for an s–t path is given by:

$$\exp(-d[t]).$$

Time Complexity Analysis

- **Weight Computation:** For each edge, computing $w(e) = -\log(1 - p(e))$ requires constant time, so the total time for this step is $\mathcal{O}(|E|)$.
- **Dijkstra's Algorithm:** Running Dijkstra on the graph with nonnegative weights takes $\mathcal{O}((|V| + |E|)\log|V|)$ time.

Thus, **the overall time complexity is**

$$\mathcal{O}((|V| + |E|)\log|V|).$$

■