Problem 1 (20 Points)

Sometimes, we may have more than one shortest path in the graph. We want to determine whether there is more than one different shortest path. Given a graph $G = (V, E)$ with weight $w(e) > 0$, and a start vertex s. Design an efficient algorithm to determine whether there exists more than one shortest path from s to every $u \in V$.

Solution. The main idea here is to mark the length of each path while finding the shortest path: if there exists an path from s to t has the same length with the shortest path, then we find out the uniqueness. Consider the classic Dijkstra algorithm. If the specific shortest path is required, we just maintain an array $prev[v]$ to record the vertex u that explores v. To find out the uniqueness, another array $mark$ is needed. Our algorithm can be divided into three steps:

1. Calculate $dist[u]$, $prev[u]$ and $mark[u]$ by Dijkstra algorithm.
2. Construct shortest path tree G' based on $prev$ where s is the root.
3. Expand $mark$ on G' to find the answers.

After step 1, $mark[u]$ reveals, on the premise that all the paths updated are shortest, whether there exists more than one vertex v that $prev[u] = v$. After step 3, $mark[u]$ simply represents whether there exists more than one shortest path from s to u, namely the answers.

Algorithm 1 Modified Dijkstra

```
Input: Graph $G = (V, E)$ and start vertex $s$
Output: Three arrays $dist$, $prev$ and $mark$
 1: $T \leftarrow \{s\}$, $dist[s] \leftarrow 0$
 2: for $(s, u) \in E$ do
 3:     $dist[u] \leftarrow w(s, u)$
 4:     $prev[u] \leftarrow s$
 5: while $T \neq E$ do
 6:     $u \leftarrow \arg\min_{v \notin T}\{dist[v]\}$
 7:     $T \leftarrow T \cup \{u\}$
 8:     for $(u, v) \in E$ do
 9:         if $dist[v] > dist[u] + w(u, v)$ then
10:             $dist[v] \leftarrow dist[u] + w(u, v)$
11:             $mark[v] \leftarrow 0$, $prev[v] = u$
12:         else if $dist[v] = dist[u] + w(u, v)$ then
13:             $mark[v] \leftarrow 1$
14: return $dist$, $prev$, $mark$
```

e exists more than one shortest path from s to t if and only if $mark[t] = 1$.

Consider its running time. Step 1 is essentially same with Dijkstra.

Algorithm 2 Shortest Path Uniqueness

Input: Graph $G = (V, E)$ and start vertex s

Output: A Boolean array $mark$ as the answers

```
1: function EXPAND($u$)
2:     for $(u, v) \in E'$ do
3:         $mark[v] \leftarrow mark[u]$ or $mark[v]$
4:         EXPAND($v$)
5: DIJKSTRA($G$,$s$)
6: Construct shortest path tree $G' = (V', E')$ based on $prev$
7: EXPAND($s$)
8: return $mark$
```

To prove its correctness, consider an arbitrary vertex t. By algorithm we have found a shortest path from s to t, denoted by $(s \to v_1 \to \cdots \to v_k \to t)$, which can be constructed with the $prev$ array. After step 2, about the $mark$ array, we have following claims:

- **Equivalence**: There exists more than one shortest path from s to t **if and only if** there exists a vertex $u \in \{v_1, v_2, \ldots, v_k, t\}$ that is marked, namely $mark[u] = 1$.
- **Transitivity**: If there exists a path from u to v and there exists more than one shortest path from s to u, then there exists more than one shortest path from s to v.

Above properties promises that after step 3, for any vertex $t \in E$, thertra algorithm. It is $O(|E| + |V| \log |V|)$ with Fibonacci heap optimization. Step 2 and 3 are both linear procedures on the shortest path tree which has no more than $|V|$ vertices and edges. Thus, they are both $O(|V|)$. The total time complexity of this algorithm is $O(|E|+|V| \log |V|)$.

This solution is provided by ***T.A. Panfeng Liu*** *and* ***Stu. Xiangyuan Xue***. □

Problem 2 (20 points)

Given a directed graph $G = (V, E)$, and a reward r_v for all $v \in V$. The revenue of a vertex u is defined as the maximum reward among all vertices reachable from u (including u itself). Design a linear time algorithm to output every vertex's revenue.

Solution. ***Algorithm.*** The description of the algorithm is as follows.

1. Find all the SCCs in G using DFS.
2. Construct a new graph $G' = (V', E')$ by replacing every SCC in G with a single vertex, where the reward for the new vertex is the maximal reward in the SCC.
3. DFS G' to find the revenue for every vertex in G': $revenue[v'] = \max(reward[v'], revenue[v'\text{'s successors}])$.
4. If v' corresponds to a vertex v in G, then $revenue[v] = revenue[v']$. If v' corresponds to an SCC S in G, then the revenue for every vertex in S equals to $revenue[v']$.

Correctness. If G is a DAG, we can easily find the topological order of G using DFS and then update the revenue along this order. If G contains some SCCs, we have the following two observations: the revenues for all vertices in an SCC are the same, and only the maximal reward in an SCC may effect other vertices' revenue. Hence, we can reduce G to a DAG G' by replacing every SCC S in G with a single vertex and set its reward as the maximal reward in S.

Time complexity. The time complexity for the algorithm contains the following parts.

- In the algorithm, the first and third step takes $O(|V| + |E|)$ as the complexity for DFS.
- The second step takes $O(|V| + |E|)$ because constructing G' will modify at most $|V|$ vertices and $|E|$ edges, and calculating the reward for new vertices will at most involves $|V|$ vertices.
- The last step takes $O(|V|)$.

Hence, the overall time complexity is $O(|V| + |E|)$, which is linear in $|V|$ and $|E|$. □

This solution is provided by ***T.A. Xiaolin Bu***.

Problem 3 (25 points)

Given a directed graph $G = (V, E)$, a source vertex $s \in V$ and a destination vertex $t \in V$. Design an efficient algorithm to determine whether there is a path from s to t containing every vertex in V? (Vertices and edges can appear in the path more than once.)

Solution. See Algorithm 3.

Algorithm 3 Every-vertex Path

```
1: $G' = (V', E') \leftarrow$ the SCC graph of $G$.
2: Set each edge's weight in $E'$ to be $-1$.
3: $d \leftarrow$ min distance from $SCC(s)$ to $SCC(t)$.        ▷ Using Bellman-Ford
4: if $d = -(|V'| - 1)$ then
5:     return Yes.
6: else
7:     return No.
```

Correctness analysis: By the property of SCC, the problem can be converted to finding a path from $SCC(s)$ to $SCC(t)$ containing every SCC[1]. Since SCC graph (G') has no cycle, it is equivalent to find a path with length $|V'| - 1$. There is no path that can be longer than $|V'| - 1$, so it is equivalent to finding the longest path from $SCC(s)$ to $SCC(t)$. By assigning each edge's weight to be -1, the problem is transformed to finding shortest path with negative weights (and no negative cycles), so Bellman-Ford can be used.

Time complexity: Finding SCC: $O(|V| + |E|)$. Bellman-Ford: $O(|V||E|)$. Total: $O(|V||E|)$.

This solution is provided by ***T.A. Kangrui Mao****.* □

[1] This is the essential point of this problem. There is a topological-order approach that deals with this point, and it is better than $O(|V||E|)$.

Problem 4 (35 points)

Let $G = (V, E)$ be a directed acyclic graph (DAG). Suppose that G is not strongly connected, and you are allowed to add extra edges into G. What is the minimum number of extra edges to make G strongly connected? In the following questions, we let $H(G)$ and $T(G)$ denote the set of head vertices (no incoming edges) and tail vertices (no outgoing edges). Notice that an isolated vertex is both a head and a tail. Moreover, if all vertices in $T(G)$ can be reached by every vertex in $H(G)$, then we call the graph G is *fully reachable*.

(a) (5 points) Discuss the minimum number of extra edges we need to make a fully reachable graph strongly connected, and prove your claim

(b) (10 points) If G is not fully reachable, prove that we can always find an new edge e, such that after adding e, $G\ \ = (V, E \cup e)$ is still a DAG, and we have $|H(G\ \ )| = |H(G)|-1, |T(G\ \ )| = |T(G)|-1$

(c) (10 points) Design an algorithm to find the minimum number of extra edges to make G strongly connected.

(d) (10 points) If the given graph G is not required to be a DAG, design an algorithm to find the minimum number of extra edges to make G strongly connected.

Solution. **(a)** We need at least $\max\{|T(G)|, |H(G)|\}$ edges. This is because, if we want to make G strongly connected, we must add an incoming edge for each vertex in $H(G)$ and an outgoing edge for each vertex in $T(G)$. Please note that each added incoming (outgoing) edge is different from each other. Thus, we at least need $\max\{|T(G)|, |H(G)|\}$ extra edges to make this graph strongly connected.

On the other hand, we can make G strongly connected by just adding $\max\{|T(G)|, |H(G)|\}$ edges. If $|T(G)| \geq |H(G)|$, then there exists an onto (denote it by f) from $T(G)$ to $H(G)$. Then, for each $v \in T(G)$, we construct an edge starting from v and terminating at $f(v)$. It is not hard to verify current G is strongly connected. If $|T(G)| < |H(G)|$, then there exists an onto (denote it by g) from $H(G)$ to $T(G)$. For each $v \in T(G)$, we construct an edge starting from v and terminating at $g^{-1}(v)$. G is also strongly connected now.

(b) Since G is not fully reachable now, there exists vertex $u \in T(G)$ which can not be reached by some vertex $v \in H(G)$. Let $e = (u, v)$. Now we prove adding e will not cause any cycle. Otherwise, it means that there is already a path from v to u, which contradicts that G is not fully reachable.

(c) To begin with, we prove the minimum number of extra edges is still $\max\{|T(G)|, |H(G)|\}$ even if G is not fully reachable. The same reason can be used to verify this number is necessary. Now we show how to construct these edges. According to (b), we can keep adding edges without loss of the property of DAG until G is fully reachable or $|H(G)| = 0$ ($|T(G)| = 0$). In order to distinguish from the initial graph, we denote current graph by G'. For the first termination case, according to (a), we can make G' strongly connected by adding $\max\{H(G'), T(G)\}$ edges. Hence, the total number of added edges is

$$\max\{|H(G)|, |T(G)|\} - \max\{|H(G')|, |T(G')|\} + \max\{|H(G')|, |T(G')|\} = \max\{|H(G)|, |T(G)|\}.$$

For the second termination case, it can not happen since a DAG must have at least one head and one tail.

Next, we need to calculate the size of $H(G)$ and $T(G)$. To achieve this, we just need to traverse the edge set and record if each vertex has incoming edges and outgoing edges. Finally, we output the maximum number of $|H(G)|$ and

$|V(G)|$. The time complexity is $O(|V|+|E|)$ and the correctness can be verified by our previous explanation.

(d) We first construct a hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{E})$, where $\mathcal{V}$ is the set of all the strongly connected components of G and $(V_1,V_2)\in\mathcal{E}$ if there is a path from V_1 to V_2. Now $\mathcal{H}$ is a DAG. The Same as previous statement, the minimum number of extra edges to make $\mathcal{H}$ strongly connected is $\max\{|T(\mathcal{H})|,|H(\mathcal{H})|\}$. For each added edge (V_1,V_2), we can just construct an edge starting from an arbitrary vertex in V_1 and ending at an arbitrary vertex in V_2 in the original graph. Hence, we can use $\max\{|T(\mathcal{H})|,|H(\mathcal{H})|\}$ extra edges to make G strongly connected.

The pseudo codes are shown in Algorithm .

Algorithm 4 Minimum number of extra edges for any graph

```
1: Let $G=(V,E)$ be the input graph;
2: Run DFS to get all the strongly connected components $V_1,V_2,\dots,V_k$;
3: Denote $\mathcal{V}=\{V_1,V_2,\dots,V_k\}$;
4: Run DFS from $V_1$ and find all the strongly connected components can be reachable by $V_1$, and construct directed
   edges from $V_1$ to these vertex sets. Then remove these sets and continue this operation on the remaining strongly
   connected components;
5: Run DFS from $V_1$ and find all the strongly connected components can reach $V_1$, and construct directed edges
   from these sets to $V_1$. Then remove these sets and continue this operation on the remaining strongly connected
   components;
6: Run the algorithm in $(3)$ on graph $\mathcal{H}$
```

Time Complexity. For constructing strongly connected components, the time complexity is $O(|V|+|E|)$. For constructing the edges in $\mathcal{H}$, since we just run DFS twice on the original graph. The total time complexity is also $O(|V|+|E|)$. For running algorithm of (3), the time complexity is $O(|\mathcal{V}|+|\mathcal{E}|)<O(|V|+|E|)$. Therefore, the total time complexity is just $O(|V|+|E|)$.

Correctness. Correctness of this algorithm can be verified by our first paragraph's statement.

*This solution is provided by **Jiaxin Song**.* □