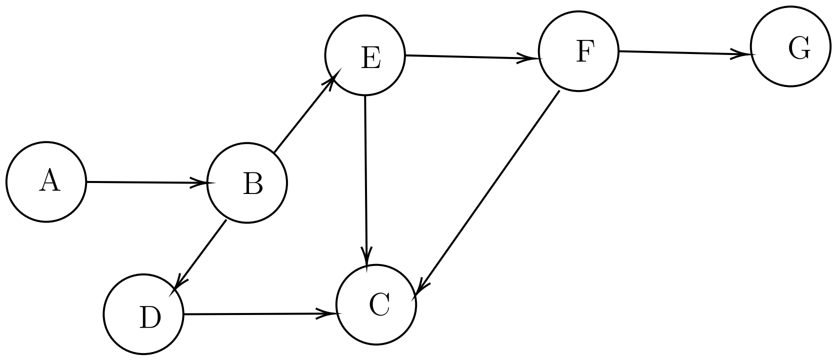
CS 3510 Section C: Design & Analysis of Algorithms September 11th 2023

HW 3: Graphs

YOUR NAME HERE Due: September 18th 2023

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- Your algorithms must be in plain English & mathematical expressions, and the pseudo-code is optional. Pseudo-code, without sufficient explanation, will receive no credit.
- Unless otherwise stated, all logarithms are to base two.
- If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists.

1.) Topological Sort (20 points)



Using the above graph G, answer the following questions.

(a) Why does a topological sorting of G exist?

Solution: G is a directed acyclic graph, so a topological sorting exists by definition.

(b) What are the strongly connected components of G?

Solution: A, B, C, D, E, F, G

(c) Give all valid topological orderings of the nodes if any exist.

Solution:

A, B, D, E, F, C, G
A, B, D, E, F, G, C
A, B, E, D, F, C, G
A, B, E, D, F, G, C
A, B, E, F, D, C, G
A, B, E, F, G, D, C
A, B, E, F, D, G, C

2.) Computopia (20 points)

The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough time to run a linear-time algorithm.

(a) Formulate this problem graph-theoretically, and explain why it can indeed be solved in linear time

Solution: Model the intersections as nodes and one way streets as directed edges in the graph. Now the above problem can be modeled as the problem of determining the number of strongly connected components in the graph.
If mayor's claim is correct there should be only one strongly connected component in the graph. Thus, run DFS on the reversed graph G^R to determine the post order and then run it again this time on the original graph G in by selecting nodes in decreasing order of post numbers. If we get only one connected component the mayor is right! This is a $O(|V| + |E|)$ algorithm since it just makes two runs of DFS which is itself an $O(|V| + |E|)$ algorithm.

(b) Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

Solution: Now she should just run the previous algorithm and check the strongly connected component to which the townhall belongs. If this SCC has an outgoing edge to any of the other SCCs, then you would never be able to come back to the townhall if you took a route that included that edge. Thus, this problem is equivalent to seeing if the townhall is in a sink SCC.
This can be checked in $O(|V| + |E|)$ again by running DFS in the original graph starting at townhall, if in this search you encounter any node outside the SCC to which townhall belongs, then mayor's claim is not true.

3.) Multiple Monkeys (20 points)

Given a set of monkeys $M \subseteq V$ and a set of bananas $B \subseteq V$ in an unweighted, undirected graph G. Find the shortest path in the graph (with respect to the number of edges) from any monkey to any banana in linear time. Formally, find the shortest path in G such that it starts at some monkey $m \in M$ and finishes at any banana $b \in B$.

Solution: This can be solved by doing a multi source BFS - push all nodes in M into the queue, and the first banana node that gets popped from the queue through this BFS is going to be the endpoint of the shortest distance. We can keep track of the path using a data structure to keep track of the ancestor of each node that we traverse during BFS. When we reach a banana node, we loop backwards through this data structure to build the path from target to the source. This takes $\mathcal{O}(|V| + |E|)$ time.

4.) Shortest Paths (20 points)

Assume we have an undirected graph $G = (V, E)$ with positive edge weights and two nodes $s, t \in V$. For each of the problems below, give an efficient algorithm and determine the running time.

(a) Determine the set of all edges that lie on at least one shortest path from s to t.

Solution: *Algorithm:*

(a) Run Dijkstra's algorithm on graph G starting at index s. The algorithm will return a list of shortest distances to each node in the graph from node s.

(b) Again, run the Dijkstra's algorithm on graph G starting at index t. The algorithm will return a list of shortest distances to each node in the graph from node t.

(c) We check the above condition on two different lists: $dist_s$, which returns the distances from 1) and $dist_v$ returns the distances from 2).

```
for E(u,v) in all edges in graph G
  if (dist_s(s,u)+edgeWeight(u,v)==dist_s(s,v)):
    if(dist_t(t,v)+edgeWeight(u,v)==dist_t(t,u)):
        list.append(E(u,v))
```

(d) If the if conditions are true, then we append the edge to the final list. This means that the edge is in the shortest path from s to t.

Running Time:

Since we are running Dijkstra's algorithm twice, the runtime of that would be $\mathcal{O}(2 * (|V| + |E|) \log(|V|))$. In step 3) we are looping over all the edges in the graph which will result in time complexity of $O(|E|)$. Hence, the overall time complexity for the find the set of edges in the shorted path from s to t is $O((|V| + |E|)log(|V|))$.

(b) Count the total number of shortest paths from s to t.

Solution:

Algorithm:

(a) While performing Dijkstra's Algorithm, we can keep a *counter* for every vertex in our new counter list referenced by the vertices in the graph. The initial value of all vertices in this list is 0. The updated value of $counter[start] = 1$.

(b) We will update this list as we check the distances in the Priority Queue.

(c) When the edges are added to the Priority Queue based on their distances from previous nodes, we can keep updating the value of this counter for every vertex in the list.

```
    if distances[u]+weight(u,v) < distances[v]
        counter[v] = counter[u]
    if distances[u]+weight(u,v) == distances[v]
        counter[v] = counter[v]+counter[u]
```

Running Time:
Since we are running Dijkstra's algorithm once, the runtime of that would be $\mathcal{O}((|V|+|E|)\log(|V|))$. Since we are just adding another list to our Dijkstra's implementation, that would not affect the time complexity of the algorithm. Although it would increase the space complexity.

5.) Global Destination (20 points)

Given a directed graph $G = (V, E)$ we want to determine whether it has a *global destination*: a vertex v such that every vertex in V has some path to v. Design an algorithm to determine whether a global destination exists in G, and if so, output it. Your algorithm should have running time $\mathcal{O}(|V| + |E|)$.

Solution: To find whether a global destination vertex v exists in G, we should find a unique sink strongly-connected component such that v is inside it. To approach this, we should first create a directed acyclic graph of the strongly-connected components (SCC) in G, which we can call a meta-graph, with each node representing an SCC of G. We can do this with the algorithm presented in lecture that takes $\mathcal{O}(|V| + |E|)$ time by calculating the reversed graph G^R, finding the source vertex in G^R by using the highest post number, and using that as a sink vertex in G to run `explore` and build up the meta-graph starting from sink SCCs.

If there is simply one node in the meta-graph, then every vertex of G is a global destination since G in its entirety is strongly connected, and we can return an arbitrary vertex in G. Then we have two cases to consider:

1. There is exactly **one** sink node/SCC in the meta-graph.
2. There is more than one sink node/SCC in the meta-graph.

If there are multiple sinks in the meta-graph, it is not possible for all vertices to reach a global destination, as the vertices in one of the sinks cannot reach the vertices in another sink. Therefore we would output "Does not exist."

Therefore the only case where there is a global destination is when there is exactly one sink in the meta-graph, since this means all vertices outside the sink SCC have a path to the sink SCC. In this case we can just return any vertex v in the sink SCC.

To find the sinks in the meta-graph, we can traverse each edge (u, v) of the meta-graph and mark the source SCC u of each edge as not a sink. The unmarked SCCs will be sinks. This will be an $O(|V| + |E|)$ algorithm, and the amount of edges and vertices is significantly less than G since we abstract the SCCs of G as nodes in the meta-graph.

Therefore the total running time of the algorithm is $\mathcal{O}(|V| + |E|) + \mathcal{O}(|V| + |E|) = \mathcal{O}(|V| + |E|)$.