**CS 3510 Section B: Design & Analysis of Algorithms** August 30th 2024

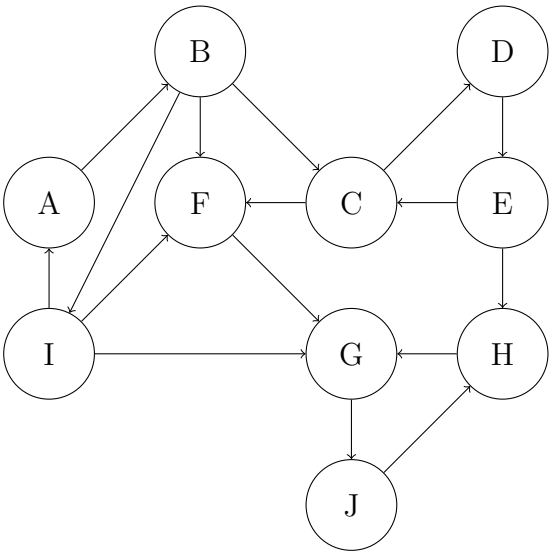
# Problem Set 3: Graph Algorithms

YOUR NAME HERE Due: September 21th 2024

- 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.
- Unless a question explicitly states that no work is required to be shown, you must provide an explanation or justification for your answer

## Problem 1: Graph

(20 points)




### Part A

For this part, in the event of a tie, choose the node that is alphabetically first.

a.) Run DFS on this graph and list the pre and post labels of each vertex.

**Answer:**

$$\text{Previsit Time: } A:1, B:2, C:3, D:4, E:5, H:6, G:7, J:8, F:14, I:17$$

$$\text{Postvisit Time: } J:9, G:10, H:11, E:12, D:13, F:15, C:16, I:18, B:19, A:20$$

### Part B

For this part, we run the SCC algorithm on the above graph. When running DFS on the reverse graph, use the alphabetical order of vertices.

a.) Write the strongly connected components in the order you found them. **Answer**:

$$A, B, I$$

$$C, D, E$$

$$F$$

$$G, H, J$$

b.) Label all the sink components and all the source components.

$$\text{Sink Components} : A, B, I$$

$$\text{Source Components} : G, H, J$$

c.) Give the minimum number of edges that must be added to the graph above to make it strongly connected. Justify your answer. To make it strongly connected, connect sink component to source. Add any edge from G,H,J to A,B,I to make a cycle. Make a connection from F to C,D,E. C,D,E needs to be connected to some other strong component. Thus there needs to be at 3 edges.

d.) Topologically sort the metagraph using the technique described in class. List the components in that order.

$$A, B, I, C, D, E, F, H, G, J$$

## Problem 2: Bipartite Graph

(20 points)

We define a graph $G = (V, E)$ as *bipartite* if the vertex set $V$ can be partitioned into two disjoint sets $L$ and $R$, such that no edge exists between any two vertices in $L$, and no edge exists between any two vertices in $R$.

Design an algorithm that determines whether a given graph is bipartite. If the graph is bipartite, your algorithm should return the sets $L$ and $R$ as part of the solution.

1. Describe (in words) your algorithm (to a degree where someone could implement it if given your answer).

   **Solution:** Initialize an array where each vertex $v$ has the value $-1$ meaning that it doesn't have a mark yet. Choose to start at a random vertex and mark this random vertex $v$ with 0. Perform BFS starting from this node and visit all neighbors, vertex $u$. For every adjacent vertex $u$, if it's not marked, place the opposite value of the vertex $v$. If the adjacent matrix has the same color as the original vertex, then return false because it's not bipartite. As the question implies, a bipartite could have nodes that are not connected with each other (disconnected), thus it's important to iterate through the whole vertex set. If it's not marked then start with another BFS with this vertex. The output will result in two marked vertex (0 or 1).

2. Argue why your algorithm is correct, meaning given any input, why does your algorithm produce the correct outputs? Justify your answer.

   **Solution:** With this solution, it's implied that all vertices marked with 1 are only connected with vertices that is marked with 0. To ensure this, the algorithm returns false whenever there's adjacent vertices with the same color. This algorithm also ensures not to skip over any nodes as this could result in a non-bipartite graph. By ensuring every vertex is check and marked opposite of it's neighbors/adjacent vertices, the algorithm will cover cases where there is disconnected graphs to ensure this algorithm works.

3. Describe the recurrence and runtime of your algorithm. Justify your answer.

   **Solution:** As each vertex and edge is processed once in BFS the time complexity is $O(|V| + |E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges. This is because BFS uses a set to keep track of iterated vertices to ensure that it doesn't expand on a vertex again. The frontier that BFS generates will need to visit all edges to cover the whole graph. When the graph is disconnected, the algorithm still runs in tbe same complexity because we have a loop ensuring each vertex is visited even if disconnected.

## Problem 3: MST Update

(20 points)

Let $G = (V, E)$ be a connected graph with a weight function $w : E \to \mathbb{R}$, and let $T$ be a minimum spanning tree (MST) for $G$. Suppose a new edge $e'$ is added to the graph, where $e' \notin E$.

Design an algorithm to efficiently compute the new minimum spanning tree for the updated graph $G' = G \cup \{e'\}$.

1. Describe (in words) your algorithm (to a degree where someone could implement it if given your answer).

   **Solution:** I will first add the edge e' where $e' = (v, u, \text{weight})$ to the MST, T. I will traverse the new minimal spanning tree (MST) with DFS. DFS will begin at a given node and check for a cycle. If a graph has a cycle, it will have a back edge. A cycle can be defined $v_0 \to ... \to v_k \to v_0$. Let $v_i$ be the first traversed vertex in this cycle. The $v_i \to v_{i+1}$ will be traversed next and $v_{i-1} \to v_i$ will be the back edge. This will be tracked with a marked set and t marked set where t marked represents the parent. Iterate through all the edges in the cycle and remove the highest weight that is not the weight of e'. By replacing the heaviest edge in the cycle with e', it reduce the total weight of the MST. If e' is greater than the heaviest edge in the original MST, don't add e'.

2. Argue why your algorithm is correct, meaning given any input, why does your algorithm produce the correct outputs? Justify your answer.

   **Solution:** DFS is best for traversing the MST as it will find the cycle. When an edge is added to MST, a cycle is created. DFS will trace the path using a set to ensure that it doesn't repeat its search. It recursively explores each vertex marking as it goes to its neighbors. DFS also is used to find the maximum weight. It will initialize a variable to store the max weight. It will recursively update the weight by comparing it. When it finishes, the max weight is the highest weight. DFS ensures that all paths from the start vertex will be explored.

3. Describe the recurrence and runtime of your algorithm. Justify your answer.

   **Solution:** Adding an edge to the MST is $O(1)$ time compleixty. To identify a cycle using DFS is $O(|V|)$ where $|V|$ is the number of edges. Finding the maximum weight edge in the cycle is also $O(|V|)$ where $|V|$ is the number of edges. Replacing the edge is $O(1)$ time complexity. Thus total time complexity is $O(V)$

## Problem 4: APSP with Shared Node

(20 points)

You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights, along with a particular node $v_0 \in V$. Provide an efficient algorithm for finding the shortest paths between all pairs of nodes, with the restriction that these paths must all pass through $v_0$.

1. Describe (in words) your algorithm (to a degree where someone could implement it if given your answer).

   **Solution:** Given that our algorithm wants to find **all pairs of nodes** efficiently such any pairs can be asked for, I want to generate an algorithm such that it processes the graph G. I will use Floyd-Warshall's algorithm which computes the shortest path between all pairs of vertices in weighted graph like this one. This algorithm is more efficient than Djisktras as you wouldn't need a source node. This is because we are not given an input pairs but only the graph G. I will initialize matrix $distance[i][j]$ to capture the shortest distance from node $i$ to node $j$. Now, I have to consider some particular node $v_0$. For this, I will initialize another matrix $sharedNodeDistance[i][j]$ where it contains the shortest path from node $i$ to node $j$ but passes through node $v_0$. This can be calculated using $sharedNodeDistance[i][j] = distance[i][v_0] + distance[v_0][j]$. The matrix holds the shortest distance between any pairs that you want and it will travel through node $v_0$.

2. Argue why your algorithm is correct, meaning given any input, why does your algorithm produce the correct outputs? Justify your answer.

   **Solution:** Floyd Warshall's algorithm is going to guarantee that the shortest path is calculated by considering all the weights between two vertices. We create the second matrix in order to get the shortest distance sum of the distance from node $i$ to node $v_0$ and node $v_0$ to node $j$. Since the distance matrix is guaranteed to find the shortest distance, it should work for the sharedNodeDistance matrix as well. Since the Floyd Warshall algorithm guarantees the correctness of the shortest distance, combining the sum of the shortest distance should be correct as well.

3. Describe the recurrence and runtime of your algorithm. Justify your answer.

   **Solution:** Ed Discussion gave the assumption that there is no input pairs but only the graph G and the specified vertex $v_0$ that all paths must go through.

   Floyd Warhsall's algorithm uses three nested loops iterating over all vertices, so the runtime for the algorithm is $O(|V^3|)$ where $|V|$ is the number of vertices in the graph $G$. In a dense graph, this algorithm works the best as it will always run in $O|V^3|$ time.

## Problem 5: Elementary School Friends

(20 points)

You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cities that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where exactly should you meet?

You are given a weighted graph $G = (V, E)$, where the vertices $V$ represent cities and the edges $E$ represent roads that directly connect cities. Each edge $e$ has a weight $w(e)$ equal to the time required to travel between the two cities. You are also given a vertex $p$, representing your starting location, and a vertex $q$, representing your friend's starting location.

Describe and analyze an algorithm to find the target vertex $t$ that allows you and your friend to meet as quickly as possible.

1. Describe (in words) your algorithm (to a degree where someone could implement it if given your answer).

   **Solution:** Given the vertex $p$ and $q$, use Dijkstra's algorithm starting using the source node $p$. This will find the shortest distance to all cities $V$. To store these values the array, $arrayP$ where $arrayP[V]$ gives the shortest path from node $p$ to any arbitrary vertex $v$. Use Dijkstra's algorithm using $q$ as the source node to compute the shortest path to every other city $v$. To store these values the array, $arrayQ$ where $arrayQ[V]$ gives the shortest path from node $q$ to any arbitrary vertex $v$. For each city in the set $V$ (denoted as $v \in V$), calculate the combined distance/time using the formula $Meet(v) = arrayP[v] + arrayQ[v]$. The smallest value $t$ given from $Meet(v)$ will be the city they meet up in.

2. Argue why your algorithm is correct, meaning given any input, why does your algorithm produce the correct outputs? Justify your answer.

   **Solution:** Dijkstra algorithm will efficiently find the shortest paths from the source node to every other node $|V| - 1$. By summing the shortest paths from $p$ and $q$ to every city $v$ in the set $V$ we will calculate the minimized $meet(v)$ to find the city $v$ they will meet up in. The reason why the algorithm works is becasue Dijkstra's is guarenteed to find the distances where the arrayP and arrayQ is accurate.

3. Describe the recurrence and runtime of your algorithm. Justify your answer.

   **Solution:** Dijkstra's algorithm has a known runtime of $O((|V| + |E|) \log |V|)$ where V is the number of vertices and E is the number of edges in the graph G. I ran Dijkstra algorithm twice, therefore the time complexity is $2 \cdot O((|V| + |E|) \log |V|)$. Additionally, calculating the sum between each vertices takes $O(|V|)$. Therefore the runtime of this algorithm is $O((|V| + |E|) \log |V| + |V|)$. Multiplicative constants and

constants are removed as log dominates thus the total runtime is

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