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

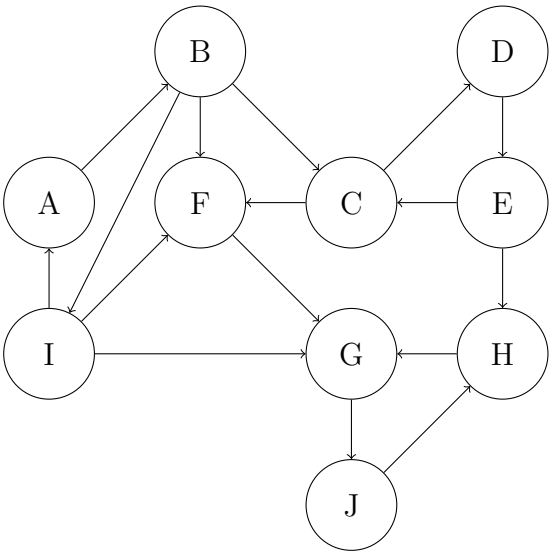
Problem Set 3: Graph Algorithms

Jinseo Lee 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.
A(1,20) B(2,19) C(3,16) D(4,13) E(5,12) F(14,15) G(7,10) H(6,11) I(17,18) J(8,9)

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.
Pre and post labels of each vertex of the reversed graph is: A(1,6) B(3,4) C(7,12) D(9,10) E(8,11) F(13,14) G(15,20) H(16,19) J(17,18). Sorting them according to decreasing order of post level is as followed: G H J F C E D A I B. Running DFS on the original graph using the order above, the SCCs are found in the following order: $[G, J, H], [F], [C, D, E], [A, B, I]$

b.) Label all the sink components and all the source components.
The sink components are the ones with the smallest post levels, and the source components are the ones with the largest post levels.
sink: $[G, J, H]$,
source: $[A, B, I]$

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 the whole graph strongly connected, any vertex must be able to reach any other

vertex in the graph, meaning that the connection between the sink components and the source components are necessary. Since there are one sink component and one source component, only one edge that connects source and sink is required. Connecting J node and I node (J towards I) could be one of the possible ways to make the whole graph strongly connected.

d.) Topologically sort the metagraph using the technique described in class. List the components in that order.
In order to topologically sort the metagraph, you need to sort the SCCs according to the decreasing order of their post levels. SCC containing $[A, B, I]$ has the largest post levels, $[C, D, E]$ comes next, $[F]$ next, and $[G, H, J]$ comes last. Therefore the answer is $[A, B, I], [C, D, E], [F], [G, H, J]$.

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:

   The algorithm I have came up with utilizes the breath-first-search method so that it traverses through the graph without missing any vertices. First, pick a random vertex in a graph and assign it to L. Next, traverse the graph and find the vertices adjacent to the chosen vertex in the previous step, assigning them into another group, R. From now on, the algorithm performs the identical process, which assigns R to the vertices adjacent to any vertex in L, and vice versa, until it traverses the entire graph. During its process, at any point if one vertex is assigned to both L and R, it terminates the traverse and return False. If it successfully traverses the entire graph, it returns L and R.

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

   Solution:

   My algorithm is correct for any input as it traverses through the graph using BFS, ensuring that every single vertex in the given graph has been taken into account. It assigns the chosen vertex to a certain set, L, and another set R to its adjacent vertices (vice versa) and constantly examines whether any vertex is assigned to both groups, which invalidates the bi-partition of the graph. If the algorithm finishes traversing the entire graph without encountering adjacent vertices of the same set, it means all edges properly connect vertices from set L to set R, confirming that the graph is bipartite. The algorithm also works for the graph containing only one vertex as it assigns the vertex to L, and R remains as an empty set as the vertex belongs to only L but not R. Also, it produces the correct output even with the disconnected graph as BFS will explore the disconnected vertex, which is of course not yet visited.

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

   Solution:

   BFS traversal visits each vertex and each edge just once as the vertex will be dequeued once visited. Assigning the vertex to a certain set (L or R) takes constant time, and

checking for the conflict depends on the number of edges, which is O($|E|$). Therefore, the runtime is proportional to the number vertices, V, and the edges, E, of the graph. The runtime is: O($|V| + |E|$). $(O(1) + O(|E|) + O(|V| + |E|) = O(|V| + |E|))$.

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:

   The algorithm I have came up with adds the edge e' that connects random vertices u and v and identifies the edges and their weights in the new cycle formed by the edge, uv, using a Depth-First-Search. Detecting the cycle and finding the unnecessary edge could be done in one DFS run on T and one loop of comparison: Starting at the vertex u, DFS will be performed while storing not only the visited vertex but also their edge and the corresponding weights (if edge uU, u to an arbitrary vertex U, weighs 1, DFS will mark U as visited and store it as (uU,1)). During this search, I can track the path between u and v. Once v is found, the path along with the newly added edge e' forms the cycle. If in any case the weight of uv is smaller than that of the heaviest edge in the cycle, the heaviest edge is removed from the cycle and e' is added. Then, the algorithm returns the new MST from G'. If e' is equal or heavier than the heaviest weight in the cycle, the algorithm just returns the T from G as an output.

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

   Solution:

   Given the instruction, we know that the graph is connected and acyclic, demonstrating that adding e' to G will always yield a cycle as T already connects all vertices with exactly $|V| - 1$ edges. In other words, e' will always create a cycle in the MST because it introduces a second path between two vertices that are already connected. Therefore, we only need to run DFS to see if any edges in the cycle could be replaced by e'. Specifically, it does not have to traverse all edges in G but only the edges in the cycle (which are of course all in T except for e') with newly added e'. After the execution, the algorithm guarantees an updated MST as it returns the correct MST if e' replaces any of the existing edges in MST, and original MST if e' cannot replace any edges due to its weight. Therefore, my algorithm preserves essential properties of MST, which are minimum weight and connectivity, validating the correctness of my approach.

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

Solution:

Adding e' to G takes constant O(1) time. Then, it traverses the cycle formed by e' connected to the vertices in T using DFS and identifies the heaviest edge, which as a whole takes O(V) time. (It is upper-bounded by O(V-1) because DFS is done on the cycle that contains at most V - 1 edges). Lastly, comparing the weights and updating the MST (if necessary) takes constant O(1) time. Therefore, the overall runtime 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:

   The algorithm I have came up with utilizes Dijkstra's algorithm to find the shortest path between the two vertices while passing through v_0 under any circumstances. To achieve this, the algorithm will first run the Dijkstra's algorithm on v_0 to find the shortest paths of v_0 to every vertices. Then, the algorithm reverses the graph, keeping the weight intact but direction of each vertices. Once the graph has been reversed, it runs Dijkstra's algorithm once again on v_0 to find the shortest paths from all edges and v_0. The algorithm will thereafter compute the addition of the shortest path of each vertex to v_0, derived from the second run of Dijkstra's, and the shortest path from v_0 to each vertex, derived from the first run of Dijkstra's. Specifically, let u be a starting vertex and v be the destination vertex. Then, the equation for the shortest path from u to v_0 to v is as followed: (u to v_0) + (v_0 to v).

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

   Solution:

   Given the instruction, we know that the graph is strongly connected with no negative edges, which foster the ideal conditions for Dijkstra's algorithm as the path between any two vertices in graph is guaranteed. Therefore, the shortest path between any vertex u and v in G passing v_0 could be interpreted as the succession of the shortest path from u to v_0 and the shortest path from v_0 to v. Since all edges possess the positive weight, Dijkstra's algorithm identifies those two shortest paths essential to the computation without a hitch. Specifically, it reverses the graph and run Dijkstra's just once more on v_0 to efficiently find the optimal path from any vertex to v_0 rather than running Dijkstra's on every vertex. Since the two shortest path has been identified and proven to be accurate by the execution of Dijkstra's, (u to v_0) + (v_0 to v) correctly calculates the shortest path between u and v, and the algorithm as a whole produces the shortest paths between all pair of nodes in the graph.

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

   Solution:

   The algorithm runs the first Dijkstra's on v_0 without any modification (traversing the whole graph to find every possible shortest paths), which takes $O(Elog(V))$ runtime.

Reversing the graph depends on the number of edges in the graph, which takes O(E) time. Finally, it runs the Dijkstra's once again on the reversed graph, without any modification on the algorithm, which also takes $O(Elog(V))$ runtime. Therefore the overall runtime is $O(Elog(V))$ as $O(Elog(V)) + O(E) + O(Elog(V)) = O(Elog(V))$.

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:

   My algorithm will first run Dijkstra's on my current location, p, until it traverses through G and find all the shortest paths to every cities including my friend's starting location, q. Once finished traversing, now it runs Dijkstra's algorithm again on my friend's starting location, q, to find the shortest paths between q and all cities. Running two Dijkstra's, each vertices (cities) will have two values, both of which represents the minimum distance from p and q, respectively. Then, my algorithm will calculate the maximum distance among the two values the cities possess, and then determine which cities have the minimum value among those maximum distances; Cities with the minimum value of max(dist(p,t),dist(q,t)) will be the optimal places to meet my friend.

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

   Solution:

   Given the instruction, we know that every edges in G is bidirectional as it is possible to drive to the city A to city B only and only if it is possible to drive to city A from city B (in general). We also know that the weight of those edges will be equal to or bigger than 0 as the distance between the two cities cannot be a negative value. Therefore, we can derive the shortest path from one city to the rest of the cities by running Dijkstra's on a source node to all other nodes with non-negative weights. In other words, we calculate the exact shortest travel time from both of my cities and friend's to any other city by running Dijkstra's both on p and q. Hence, by minimizing max(dist(p,t),dist(q,t)), we can ensure that the meeting point is as fair as possible, minimizing the maximum travel time for both travelers.

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

Solution:

Running Dijkstra's algorithm without any modifications takes $O(Elog(V))$ times. We run it twice, which makes no difference in the overall runtime ($O(2 * Elog(V)) = O(Elog(V))$). Looping through the vertices to find the minimum value takes extra O(V) time, but is dominated by O(Elog(V)). Therefore, the runtime is O(Elog(V)).