CS 3510: Design & Analysis of Algorithms (Sections A & B)

Homework 6: Graph Algorithms

Professors Dana Randall and Gerandy Brito **Due: 11/3/2022 11:59pm**

Name: Anthony Wong

1.) (20 points each) Let G be an undirected graph, and let s and t be distinct vertices of G. Each edge in G is assigned one of two colors, `white` or `gold`. (Hint: For this question, you may modify the graph and make a new graph as long as the solution is still for the original graph).

(a.) Design an algorithm that determines if there is a path from s to t with edges of only one color (that is, a path containing either `white` edges only or `gold` edges only).

Solution:

To solve this problem, we can use DFS as our method of exploring the undirected graph G with a visited set V. The DFS algorithm itself needs to be edited slightly however. Instead of just recursing and passing in the next node, we also need to pass in the next node to recurse to and the edge that came before the node we care recursing from. We can determine this edge by defining edges as the node it came from to the node it connects to, as well as its color. That means at each node we encounter, we can find the edge we just came from and pass it to the next node in the search. To make sure we find a path with only white or gold edges, we just need to make sure that the previous edge is the same as any edge we select as the next to recurse to (and the next node is not in the visited set v). If any path results in a color change, we simply stop recursing. This means, we will get all the single color paths starting from node s. If we ever recurse to t, we simply return true as we've found a single color path.

(b.) Design an algorithm that determines if there is a path from s to t such that all `white` edges appear before all `gold` edges in the path.

Solution:

To solve this problem, we do something similar to part a with DFS and a visited set v. We also define the recursion just like in part a, where instead of just recursing and passing in the next node, we also need to pass in the next node to recurse to and the edge that came before the node we care recursing from. We can determine this edge by defining edges as the node it came from to the node it connects to, as well as its color. That means at each node we encounter, we can find the edge we just came from and pass it to the next node in the search. To ensure we find paths where white edges come before gold edges, we must check that we never go from a gold edge to a white edge. This means checking this conditional every time we chose what to recurse to. If any path results in a gold edge changing to a white edge, we simply stop recursing. As a result, we will get all paths that where white edges come before all gold edges starting from node s. If we ever recurse to t

starting from s, we simply return true as we've found a path that fits the conditions.

2.) (20 points each) 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 algorithm.

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

Solution:

If we frame this problem into graph theory, we first need to describe the intersections as vertices and the roads connecting the intersections as edges. With this description, we can then form a directed graph (given one-way streets) G to describe the scenario. Once we have constructed this graph, we can say that if there is a way to drive legally from any intersection to any other intersection, our graph G then itself is a strongly connected component. This can be done in linear time because of the SCC algorithm we have defined in lecture. We first find the reverse of graph G which has a runtime of $O(m)$ where m is the number of edges in graph G. We then run DFS on the reverse graph which has a runtime of $O(n+m)$ where n is the number of vertices in graph G and m is again the number of edges. The last step is to run DFS on the normal graph G which has a runtime of $O(n+m)$. To check this condition that the graph G is itself a SCC, we can edit the DFS algorithm described in lecture for the specifically the DFS call on the normal graph G. We know that if we reach the outer loop within DFS (not Explore) and count more than one occurrence of a non-visited node, the claim that the mayor makes is false. If we only have one occurrence of a non-visited node (the first call), the claim is true. When you combine the components of the SCC algorithm and their runtimes, you get a runtime of $O(2n+3m) = O(n+m)$ which is linear.

(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:

This problem is similar to part a in the way that we frame the question. We again first need to describe locations as vertices and the roads connecting them as edges. With this description, we can then form a directed graph (given one-way streets) G to describe the scenario. Here we are checking if any intersection you can reach from the town hall has a path back to the town hall. When you frame this as a graph theory question, you want to check if the set of vertices that are reachable from the town hall is equal to the strongly connected component set that contains the town hall itself. To get the set of vertices that are reachable from the town hall, we can call DFS starting at the town hall and edit the Explore algorithm such that is keeps track and returns the nodes it visits. This would have a runtime of $O(n+m)$ where n is the number of vertices in G and m is the number of edges in G. We then need to get the strongly connected component containing the town hall. To do this we run the SCC algorithm from lecture. As described in part a, We first find the reverse of graph G which has a runtime of $O(m)$. We then run DFS on the reverse graph which has a runtime of $O(n+m)$ where n is the number of vertices in graph G. The last step is to run DFS on the normal graph G which has a runtime of $O(n+m)$. If we again use the Explore algorithm that keep tracks of the nodes it sees, we then can construct the set of SCCs. The last step is then to compare the set of reachable vertices to the SCC containing the town hall. If there is any discrepancy between them, the claim is false, else the claim is true. The runtime of this portion of the algorithm is $O(n)$. IF we combine all these elements and their runtimes, you get a runtime of $O(3n+3m) = O(n+m)$ which is linear.

3.) (20 points) There is a network of roads $G = (V, E)$ connecting a set of cities V. Each road in E has an associated positive length ℓ_e. There is a proposal to add one new road to this network, and there is a list E' of pairs of cities between which the new road can be built. Each such potential road $e' \in E'$ has an associated length. As a designer for the public works department you are asked to determine the road $e' \in E'$ whose addition to the existing network G would result in the maximum decrease in the driving distance between two fixed cities s and t in the network. Give an efficient algorithm for solving this problem.

Solution:

To do this, we can utilize Dijkstra's Algorithm with a Fibonacci heap which can compute minimum distances between nodes in a graph with a runtime of $O(|E| + |V|\log(V))$. To determine the maximum decrease in distance between two fixed cities s and t when adding an edge e' from E', we can add each edge from the set E' one by one into graph G and run Dijkstra's on the new graph each time. If we keep track of the minimum driving distance between s and t and the corresponding edge e' that resulted in that minimum distance, at the end we will have the edge e' that results in the maximum decrease in driving distance between s and t. Combining the components of this algorithm, we get a runtime of $O((|E| + |V|\log(V))|E'|))$.