CS 3510 — Algorithms, Spring 2022

Homework 3A Solutions

1 Restricted Paths

Description. First, we want to compute the distance from v_0 to every other node. This can be done using Dijkstra. Hence, we run `Dijkstra`$(G = (V, E), v_0)$ to get a distance ($dist_{fwd}$) map from $v_0 \to v : \forall v \in V$.

Now, set $E^R = \{(v, u) : (u, v) \in E\}$ which are the reversed edges in a graph and define $G^R = (V, E^R)$ as the reversed edge graph. Run `Dijkstra`$(G^R = (V, E^R), v_0)$ to get a distance ($dist_{bwd}$) map from $v_0 \to u : \forall u \in V$ *in the reversed graph*. This distance map also corresponds to $u \to v_0 : \forall u \in V$ in G.

Hence, for any pair (u, v), the shortest path from $u \to v$ that goes through (u, v) can be given by $dist_{bwd}[u] + dist_{fwd}[v]$.

Justification. This algorithm is correctly as Dijkstra is an algorithm that returns a distance map from a fixed source to every possible destination in the graph from that source. The forward direction is trivially correct as it is a literal application of the algorithm.

Since Dijkstra's algorithm uses a fixed source with variable destinations, then reversing the edges lets us get the shortest paths from variable sources to a fixed destination. This is the exact neccesity for the first half of the path from $u \to v_0$.

Finally, the shortest combined path is also the shortest path from $u \to v_0$ and $v_0 \to v$ when v_0 must be crossed. Hence, computing these two distances seperately and adding them is minimal.

Runtime. This algorithm runs the Dijkstra algorithm twice taking $\mathcal{O}((|V|+|E|) \log |V|)$ runtime and reverses the edges taking $\mathcal{O}(|E|)$ time giving a total runtime of $\mathcal{O}((|V|+|E|) \log |V|)$.

2 Red Blue Paths

Description. We solve this by using a modified breadth first search (although a DFS would work just as well, but the natural iterative implementation of BFS is beneficial). We modify the standard BFS algorithm such that the queue stores the next vertex as well as the color used to get to that vertex. For example, starting at root vertex with red edges to a, b and blue edges to c, d, the queue would look like $[(a, red), (b, red), (c, blue), (d, blue)]$.

When popping elements off the queue, we only look at adjacent edges of a different color. For example, if we pop (x, red) off the queue, then only edges (x, u) (for some u) that are blue are considered. We only consider edges that are of a *different color*. During initialization, the root edge is added onto the stack with `null` in the color parameter (hence, it will traverse both red and blue edges that are adjacent to it).

If we finish the `explore` starting at u without reaching v, return `false`. If we see v during the traversal, return `true`.

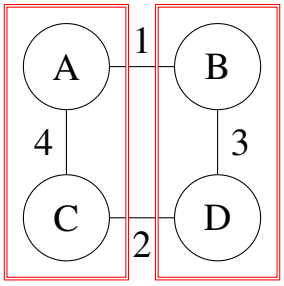
Justification. Since the paths must be alternating red/blue, we can make use of a search algorithm that restrictions on only visiting the next node if the color differs from the edge used to get to the current location. By modifying the `explore` method, this alternating can be achieved by restricting the search space. Since we also allow to start from both colors, this will get both alternating patterns.

Runtime. This is simply a modified BFS which has a linear runtime of $\mathcal{O}|V| + |E|)$ with the modification performing checks on each edge during iteration which takes $\mathcal{O}1$ time maintaining the original BFS runtime.

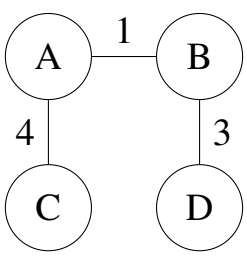
3 Minimum Spanning Tree Algorithms

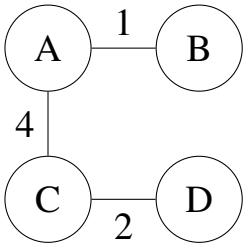
3.1 Divide and Conquer

Solution. This is not a valid solution as we can be ignoring many minimal edges depending on how we split our graph G which is arbitrary. Take an example.



In this case, if we arbitrarily split vertically, then we have our base case on the left and right subgraphs as these are already MSTs for the subgraphs. We then choose the minimum edge connecting these two subgraphs. We can make an arbitrary selection of either the top or bottom edge. This gives a total weight of $1+4+3 = 8$ (see left).





The figure shown to the right is the correct MST, though. It has weight of $1 + 4 + 2 = 7$. Hence, this algorithm does not produce the correct MST.

3.2 Cycle Breaking Strategy

Solution. This strategy is correct, we can proceed to prove it.

Proof. For the sake of contradiction, suppose the heaviest edge, e, in some cycle C is in T (where T is the MST). Clearly, T cannot contain all of C (as T is acyclic). As a result, there is some edge $e' \neq e \in C \notin T$ (some edge in the cycle that is not in T). Since e was maximal, e' must be smaller.

Since T is a tree, removing e causes there to be a disconnect, but since e' is apart of the same cycle C, then we know that removing e and adding e' makes this a tree (no longer disconnected). Hence, we have another spanning tree that is constructed as $T' = (V, (E \setminus \{e\}) \cup \{e'\}$.

Let k be the weight of the MST T. Then, T' has weight given by $k - w(e) + w(e')$. It is clear that $w(e) > w(e') \implies w(e) - w(e') > 0$ meaning that $k - w(e) + w(e') = k - b$ where $b \in \mathbb{R}^{\geq 0}$ (b is some positive number) and therefore this spanning tree is more minimal than T, but this is a contradiction as T was an MST.

Hence, the original assumption that the heaviest edge e was in the MST was incorrect and it is true that the heaviest edge e of a cycle is not in the MST. $\square$

4 Pseudoconnectivity

Description. Run the metagraph algorithm on G which produces another graph G^M where V^M are the SCCs of G and the edges are the edges between SCCs.

Determine whether or not there is a Hamiltonian path between the vertices. Later in the course, we may discover the difficulty in computing Hamiltonian paths, but due to the nature of a metagraph (DAG), we can topologically sort this metagraph and check whether or not the topologically sorted graph has consecutive edges. After topologically sorting the metagraph, we can simply iterate through the sorted vertices list and check whether an edge $(v_i, v_{i+1}) \in E^M$. If for any i, this is not the case, we return false. Otherwise, return `true`.

Justification. This algorithm works because every strongly connected component is also a pseudo connected component. Hence, this question can be simplified to whether or not the metagraph (which is a graph of these SCCs) is pseudoconnected.

This can be solved by determining a Hamiltonian path, as a Hamiltonian path is a path such that it traverses every vertex *exactly once*. This is difficult (NP-complete) to perform generally, but since this graph is a DAG, we can topologically sort it and check consecutive edges. We claim that if there is a consecutive edge between vertices in the topologically sorted metagraph, then there is a Hamiltonian path (path that crosses every vertex once).

Proof of claim. By nature of topologically sorted graphs, there will never be an edge from $v_i \to v_j$ if $i < j$ (there are no edges going back). As a result, if there is no edge from $v_i \to v_{i+1}$, then v_{i+1} will be unreachable as every edge beyond (inclusive) v_{i+2} will not be able to reach v_{i+1}. Hence, there will not be a path that covers every vertex *exactly once* if there are not consecutive edges. □

As a result, if we can show the metagraph is pseudoconnected (by finding a satisfying path), then the entire graph is pseudoconnected.

Runtime. The runtime $\mathcal{O}(|V|+|E|)$ as we are simplying computing the metagraph and performing a topological sort, then checking for an edge path ($\mathcal{O}(|E|)$).