CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

●

Homework 3

Due on Tuesday, Februrary 14th, 2017 at 11:59am

●

Solutions by

Michael Fan

26697596

In collaboration with

Julie Han, George Wu

Problem 1: Dijkstra's Durability

★★ Level

For the following claims, answer yes or no and provide justification. Consider an arbitrary graph G with positive edge weights. Shortest path means least cost path.

a. Suppose we want to know the distance from a node s to a node t_1, so we run Dijkstra's algorithm. Now suppose we also want to know the distance from s to another node t_2. Do we need to run the algorithm again?

> **Solution** No - Dijkstra's algorithm will explore and return the shortest paths to all pairs (u, v) where u is the start and v is any node reachable from u

b. Suppose we know the shortest path from a node s to another node t. Is the shortest path (the sequence of nodes, not the total cost) always the same if we add k to all edge weights, where k is an arbitrary positive number?

> **Solution** The shortest path may change because paths with more edges will increase more in length. Fr example, take G with $V = \{A, B, C\}$ and E $= V = \{(A, B, 1), (A, C, 3), (B, C, 1)\}$. Then the shortest path from A to C is $\{A, B, C\}$ with length 2. But if we add 2 to each edge length, then path $\{A, C\}$ is now the shortest path with length 5 in comparison to the original path which is now length 6.

c. Now answer the same question, except we multiply by k instead of adding k.

> **Solution** The shortest oath will not change. Assume for the sake of contradiction that there exists a path p_1 from u to v of length l_1 and p_2 of length l_2 which is longer than p_1, but after all edge lengths are multiplied by k, the altered p_2' is now shorter than p_1'. Every edge in the graph was multiplied by k so $p_1' = kp_1$ and $p_2' = kp_2$. We claim that $p_2' < p_1' \equiv kp_2 < kp_1 \equiv p_2 < p_1$ which is a contradiction.

Problem 2: Biconnected Components

★★★★★ Level

Let $G = (V, E)$ be a connected undirected graph. For any two edges $e, e' \in E$, we say that $e \sim e'$ if either $e = e'$ or there is a (simple) cycle containing both e and e'. We say a set of edges C is a *biconnected component* if there is some edge $e \in E$ such that $C = \{e' \in E \mid e \sim e'\}$.

a. Show that two distinct biconnected components cannot have any edges in common. (Note: That this is equivalent to showing that if $e' \sim e_1$ and $e' \sim e_2$.)

Solution Let B and C be two distinct biconnected components defined by $B - \{e' \mid e_1 \sim e'\}$ and $C = \{e' \mid e_2 \sim s'\}$, where e_1 and e_2 are distinct edges. Suppose B and C share edge f in common. Then $f \sim s_1$ and $f \sim e_2$. But this implies $e_1 \sim e_2$.

To see why, note that it is clearly true if $f = e_1$ or $f = e_2$. Otherwise, there is a simple cycle containing f and e_1 and a simple cycle containing f and e_2. The union of these two simple cycles contains a simple cycle that has both e_1 and e_2 as edges. (Convince yourself why. Drawing some visuals may help.)

By a similar argument, if $e' \sim e_1$, then $e' \sim e_2$ because $e_1 \sim e_2$. Likewise, if $e' \sim e_2$, then $e' \sim e_1$. Thus, $B = C$, contradicting the fact that they are distinct sets.

b. Associate with each biconnected component all the vertices that are endpoints of its edges. Show that the vertex sets corresponding to two different biconnected components are either disjoint or intersect in a single vertex. Such a vertex is a *separating vertex*.

Note that a separating vertex, if removed, would disconnect the graph. We will now walk you through how you can use DFS to identify the biconnected components and separating vertices of a graph in linear time. Consider a DFS tree of G.

Solution Let $B = \{e'|e_1 \sim e'\}$ and $C = \{e'|e_2 \sim e'\}$ be 2 distinct biconnected components. If the vertex sets of B and C are disjoint, then they cannot share any edges so they are trivially distinct biconnected components. Similarly, if the vertex sets only share 1 vertex, then B and C still share 0 edges which means they are distinct. Now, consider the case where B and C share 2 vertices. Let $V_B = \{v_1, v_2, ..., v_n\}$ be the set of vertices included in the edges in B and $V_C = \{v_n + 1, ..., v_k, v_1, v_2\}$. By the definition of a biconnected component, v_1 and v_2 are connected by a path of edges that belong to the same cycle in the edge set of B. But, this is also true for C and the path in B has to be included in the edge set of C as a consequence of v_1 and v_2 being in the same cycle. This is true for the path in C appearing in B as well. Thus, we have shown that B and C share edges. But in part (a), we have shown that distinct biconnected components cannot share edges.

c. Show that the root of the DFS tree is a separating vertex if and only if it has more than one child in the tree.

Solution We first show that if the root has more than one child in the tree then it is implied that the root is a separating vertex. We will show this via contraposition and prove that if the root is not a separating vertex, it has less than or equal to 1 child. If the root has 0 children, then it is trivially not a separating vertex. If it has 1 child, then upon removal of the root, the child becomes the new root and the rest of the graph is untouched. Hence, it is not a separating vertex. If it has 2 children, then upon removal of the root, the 2 children are no longer guaranteed to be connected and neither are their descendants to each other. Thus, we have proven that if the root is not a separating vertex, then it has less than or equal to one child.
We will now show that if the root is a separating index, then it has more than 1 child. With the same logic as above, if the root has less than or equal to 1 child, then the removal of the root will leave the rest of the graph connected and is thus not a separating index. However, as shown above, if the root has 2 or more children, then the removal of the root removes the connection of the children through the root and leave no guarantee that they are connected at all. Thus, we have shown equality from both

Solution (cont.)

sides.

d. Show that a non-root vertex v of the DFS tree is a separating vertex if and only if it has a child v' none of whose descendants (including itself) has a backedge to a proper ancestor of v.

For each vertex u define pre(u) to be the pre-visit time of u. For two vertices u and w, w is a **backcestor** of u if and only if the following two conditions both hold:

1. w is an ancestor of u.
2. $\exists$ (back) edge (u, w) OR $\exists$ (back) edge (v, w), where v is some descendant of u in the DFS tree.

Define low(u) to be the minimum possible value of pre(w), where w is a backcestor of u.

Another way of stating the result of (d) is that a non-root vertex u is a separating vertex if and only if pre$(u) \leqslant$ low(v) for any child v of u.

Solution We will first show that if v is a separating vertex, then v' has no descendant with a backedge to an ancestor of v. We will prove the contraposition that if all children v' have descendants with a backedge to an ancestor of v, then v is not a separating vertex. We remove v. But all children of v can navigate to the graph before v because they have backedges connecting to ancestors of v.
We now show the other direction. Assume for the sake of contradiction that v' has no descendant with a backedge to an ancestor of v but v is not a separating vertex. We remove v. But now, the children of v can no longer navigate to the graph before v.

e. Give an algorithm that computes all separating vertices and biconnected components of a graph in linear time.

Hint: Think of how to compute low *in linear time with DFS. Use* low *to identify separating vertices and run an additional DFS with an extra stack of edges to remove biconnected components one at a time.*

No need for a full 4-part solution. Just give a clear description of the algorithm, in plain English or psuedocode, and running time analysis.

Solution We first go over an algorithm for calculating low in linear time. We run dfs as usual except when we reach each vertex, we keep track of the lowest *pre* value of all neighbors of the vertex. low(current vertex) is then the minimum of the lowest *pre* value of the children of the current vertex, pre(current vertex) and the low values of all the children of the current vertex. The only nontrivial part is passing the low values of the children to the parents, but this can be done when the children are popped off the queue. Thus, this algorithm runs with the same time as dfs which is linear.
We now give an algorithm that computes all separating vertices and biconnected components of a graph in linear time. We know that a non-root node u is a separating vertex iff pre$(u) \leq$ low(v) for any child v of u. This is checked while computing the low array as described above (linear time). Hence, we have discovered how to compute all separating vertices in linear time.
We now describe an algorithm for returning biconnected components. If u is a separating vertex and v is a child such that pre(u) ¡ low(v), then everything that follows v as the root are in different biconnected components as the ancestors or other descendants of u by the definition of a separating vertex (because the graph is disconnected upon its removal, it effectively separates biconnected components because no edges can be shared through the separating vertex). Thus, we mark v with a counter variable to show the beginning of a new biconnected component. We now consider the subgraph stemming from v as it may contain multiple biconnected components. We push all edges onto a stack via a traversal using dfs. This process is run recursively whenever we meet a separating vertex. We increment the counter, mark the child, and recurse. Upon considering the original dfs at v, we realized that we had popped the subtree of v from the stack of edges. We know to stop and consider at the marks. The marks denote different biconnected components. Back at v, the problem of multiple biconnected components after v has been solved: they would have been popped before we returned to v.

Problem 3: Alternative Factory

★★★ Level

You have the unfortunate predicament that you frequently find yourself inexplicably trapped within factories, a different one every time. Each factory consists of many small platforms, connected by a network of conveyor belts. Being an amateur surveyor, you can accurately calculate the time it takes to ride each conveyor belt with a quick glance. Some platforms have a button on them. Whenever you press one of these buttons, ALL conveyor belts in the factory reverse direction. A factory has one exit; you want to get there as quickly as possible. Give an efficient algorithm in $\mathcal{O}((|V| + |E|) \log |V|)$ time, to find the fastest way out of the factory.

You only need to give a main idea and running time analysis. Be sure your main idea clearly and fully describes your algorithm.

For convenience, you can treat the factory as a graph, with the platforms being vertexes and the conveyor belts being directed edges. You know before running your algorithm the platform s at which you start, the location of the exit t, and the location of all buttons (think of this as a list of button locations). You can determine the time it takes to traverse the conveyor belt between any two platforms u and v via the function $\ell(u, v)$.

Notes in case you want to pick at the details:

- It takes a negligible amount of time to get from one end of a platform to the other.
- The time it takes to press a button is also negligible.
- You can't run up a conveyor belt the wrong way.
- You may assume $\ell(u, v) = \ell(v, u)$.
- Be warned that you should throw out your sense of spatial reasoning. You have seen all kinds of setups of conveyor belt configurations, sometimes with thousands of conveyor belts connected to a single platform.

Hint: Try to construct a new graph that encodes the state of the conveyor belts. Perhaps your new graph will have twice as many vertexes as the original.

Solution I construct a graph of the exact setup of the conveyor belts with a vertex at every platform and directed edges indicating the direction that the conveyor belts move in. The lengths of the directed edges represent the time it takes to move from the source vertex to the sink vertex linked by the edge. I construct a second graph that is identical to the first except that the directed edges all point in opposite directions (but still to the same vertices). I then find the vertices that represent platforms with buttons in the first graph and connect them with the corresponding vertices in the second graph with undirected/bidirectional edges. These edges have length 0. This model represents the entire search space - I start in the first graph, but whenever I get to a platform with a button, I have the option of moving to the second graph which represents the same space except that all the conveyor belts move in opposite directions. It takes 0 time to press the button/reverse the belts, hence length 0 links connecting the platforms with buttons. Note that in this graph, I have 2 end states, one in either subgraph in the same location (as long as I get to t, I don't care which direction the conveyor belts go in). Then, I simply run Dijkstra's algorithm starting from s in the first subgraph. If the algorithm terminates without hitting t, there is no path from s to t. It takes $\Theta(2(|V| + |E|)) = \Theta(|V| + |E|)$ time to construct the dual graphs if $|V|$ and $|E|$ are taken to be the sizes of the original vertex/edge set sizes from the first subgraph. Running Dijkstra's algorithm will take $O((2|V| + 2|E|) \log |2V|) = O((|V| + |E|) \log |V|)$. Hence, the overall runtime of our algorithm is $\Theta(|V| + |E|) + O((|V| + |E|) \log |V|) = O((|V| + |E|) \log |V|)$

Problem 4: Shortest Path Oracle

★★★ Level

Given a connected, directed graph $G = (V, E)$ with positive edge weights, and vertices s and t, you want to find the shortest path from s to t in a graph in $\mathcal{O}(|E| + |V|)$ time. Normally, this wouldn?t be possible; however, you have consulted the Order Oracle, who gave you a list L of the V vertices in increasing order of the shortest-path distance from s.

In other words, $L[1]$ is s, $L[2]$ is the closest other vertex to s, $L[|V|]$ is the farthest vertex from s, etc. Design an algorithm ShortestPath(G, s, t, L) which computes the shortest $s \to t$ path in $\mathcal{O}(|E| + |V|)$ time.

Solution
Overview:
Run Dijkstra's algorithm except replace the heap/priority queue with the Oracle's List

Pseudocode:

```
function shortestPath(G(V, E), s, t, L):     for all u in V:
        dist(u) = Int.MAX
        prev(u) = nil
    dist(s) = 0
    while L is not empty:
        u = L.pop()
        for all edges (u, v) in E:
            if dist(v) ¿ dist(u) + length(u, v):
                dist(v) = dist(u) + length(u, v)
                prev(v) = u
```

Proof of Correctness:
The list L guarantees shortest length, so we know for a fact that we are popping elements in the right order. Dijkstra's algorithm guarantees that each node along the path is the closest possible option to the start while contributing to the shortest path. The rest of the algorithm is literally Dijkstra's algorithm and is thus correct.

Runtime analysis:
Iterating through L: $|V|$
Iterating through edges: $|E|$
Runtime: $O(|V| + |E|)$ (Dijkstra's with constant time pops from heap)

Problem 5: Archipelago Adventure

★★★★★ Level

As a tourist visiting an island chain, you can drive around by car, but to get between islands you have to take the ferry (more formally, our graph is $G = (V, R \cup F)$. where V is our set of locations (vertices), R is the set of roads, and F is the set of ferry routes). A road is specified as an **undirected** edge $(u, v, C_{u,v})$ that connects the two points u, v with a **non-negative** (time) cost $C_{u,v}$. A ferry is specified as a **directed** edge $(u, v, C_{u,v})$ which connects point u to point v on another island (one way) with cost $C_{u,v}$. Some of the ferries are actually magical fairies so their costs **can possibly be negative**! Each ferry only runs one way and the routes are such that if there is a ferry from u to v, then there is no sequence of roads and ferries that lead back from v to u.

Assume we have the the adjacency list representation of roads and fairies (two separate data structures).

You are finished with your vacation so you want to find the shortest (least cost) route from your current location s to the airport t . Design an efficient algorithm to compute the shortest path from s to t (given V, R, F, s and t as inputs). Your running time should be $\mathcal{O}((|V| + |E|) \log |V|)$, where $E = R \cup F$.

Hint 1: Notice that the directed (ferry) edges are between different islands, each of which forms a separate (strongly) connected component.

Hint 2: If you are stuck, try thinking about the following questions (no credit for answering).

- *Consider a directed graph with no negative cycles in which the only negative edges are those that leave some node s; all other edges are positive. Will Dijkstra's algorithm, started at s, work correctly on such a graph? Find a counterexample or prove your answer.*
- *Now consider the case in which the only negative edges are those that leave v, a vertex different from the start node s. Given two vertices s, t, find an efficient algorithm to compute the shortest path from s to t (should have the same asymptotic running time as Dijkstra's algorithm).*

Solution

Overview:

We first collapse all island nodes into SCCs. We then push the negative edge weights in a way that does not affect the relative path lengths along the graph until we reach a sink SCC at which we can ignore the negative weights. We then have a regular graph with all positive weights and simply run Dijkstra's algorithm to solve the problem. leaving us with a graph with only positive weights, so we can run a simple Dijkstra's algorithm to find the shortest path from s to t.

Pseudocode:

```
function sccDijkstra(islands V, roads R, ferry routes F, s, t):
    collapse V into a linearized DAG of SCCs //algorithm shown in class
    for each SCC in topological order:
        if there is negative edge among incoming ferry edges:
            let minEdgeWeight = |min(incoming edge weights)|
            for each incoming edge:
                add minEdgeWeight to the edge weight
            for each outgoing edge: //notice how this does nothing at the last SCC bc it's a sink
                subtract minEdgeWeight from the edge weight
    run dijkstra's. //it'll work and i can sleep now :')
```

Proof of correctness:

After we take care of the negative values, this problem reduces to a simple run of Dijkstra's algorithm which is proven to be correct. The algorithm for collapsing nodes into a DAG of SCCs was proven to be correct in class. I will now prove the final part that the process of propagating/removing negative edge weights is correct. The key is that by adding the absolute value of the most negative incoming edge to all incoming edges and then subtracting it from all outgoing edges, the ferry path through the SCC has not changed in weight. However, there are the edge cases of the source SCC and sink SCCs to consider which do not have incoming edges or outgoing edges, respectively. Source SCCs do not have incoming edges and hence

Solution (cont.)

the paths through them are trivially undisturbed. In the case of sink SCCs, all incoming edges are adjusted by the same constant, and thus there is no relative change done. The shortest path to reach the sink SCC remains the shortest, and vice versa. There may be multiple sinks, but we only care about the one that contains t, so it doesn't matter that the relative ranking of weights between different sinks may change.

Runtime analysis:
Making the linearized the DAG of SCCs and pushing along the negative weights are done in linear time: $\Theta(|V| + |R| + |F|)$.
Running Dijkstra's algorithm on the altered graph of all positive weights takes $O((\log |V|)(|V| + |R| + |F|)) = O(\log |V|(|V| + |E|))$ as usual with $|V|$ vertices and $|E|$ edges.
Thus, the runtime is $\Theta(|V| + |R| + |F|) + O(\log |V|(|V| + |E|)) = O(\log |V|(|V| + |E|))$.

Optional Redemption File

Submit your *redemption file* for Homework 2 here. If you looked at the solutions and took notes on what you learned or what you got wrong, include them here.