CS 170 HW 4 (Optional; 0 Credits)

Due on 2018-02-18, at 10:00 pm

1 Study Group

List the names and SIDs of the members in your study group.

2 All Roads Lead to Rome

You are the chief trade minister under Emperor Caesar Augustus with the job of directing trade in the ancient world. The Emperor has proclaimed that *all roads lead to (and from) Rome*; that is, all trade must go through Rome. In particular, you are given a strongly connected directed graph $G = (V, E)$ with positive edge weights, and there is a particular node $v_0 \in V$ (Rome). Give an efficient algorithm for finding shortest path between *all pairs of nodes*, with the one restriction that these paths must all pass through v_0 (Rome). Make your algorithm as efficient as you can, perhaps as fast as Dijkstra's algorithm.

(a) Give the efficient algorithm.

[Provide 3 part solution.]

(b) Occasionally, Augustus will ask you for the (smallest) distance between two vertices. You want to do this as quickly as possible, so that Augustus does not have your head.

This is called a *distance query*: Given a pair of vertices (u, v), give the the distance of the shortest path that passes through v_0. Describe how you might store the results such that you require $O(|V|)$ storage and from your data structure you can compute the result in $O(1)$ time. For your answer, a clear description of the data structure and its usage is sufficient.

(c) On the other hand, the traders need to know the paths themselves.

This is called a *path query*: Given a part of vertices (u, v), give the shortest path itself, that passes through v_0. Describe how you might store the results such that you require $O(|V|)$ storage and from your data structure you can compute the result in $O(|V|)$ time. Again, a clear description of the data structure and its usage is sufficient.

Solution:

(a) **Main Idea:**
We want to run an initial computation after which we can compute the shortest distance from any vertex to another quickly. To do that we first run Dijkstra's to find the shortest paths from v_0 (Rome) to all other nodes, then find the shortest paths from all other nodes to v_0. The latter is done by reversing the direction of edges of the graph and running Dijkstra's starting from v_0, since a shortest path from v_0 to u in the reversed graph is a shortest path from u to v_0 in the original graph.

Pseudocode:
Assume we have access to a procedure DIJKSTRA(G, v) finding shortest paths starting from v, which returns two arrays `dist` and `prev` as described in the textbook.

```
1: $\texttt{dist}_{\text{from}}, \texttt{prev}_{\text{from}} \leftarrow$ DIJKSTRA$(G, v_0)$
2: $G' \leftarrow$ Reverse all edge directions of $G$
3: $\texttt{dist}_{\text{to}}, \texttt{prev}_{\text{to}} \leftarrow$ DIJKSTRA$(G', v_0)$
```

Proof of Correctness:
A shortest path from v_0 to u in the reversed graph is a shortest path from u to v_0 in the original graph (with the direction reversed). This is because reversing all edges also reverses the direction of all paths.

The correctness of the full algorithm comes from the correctness of Dijkstra's algorithm and the fact that the shortest path from s to t passing through v_0 is the combination of the shortest path from s to v_0 and v_0 to t.

Runtime:
This algorithm has the same runtime as Dijkstra's, which is $O((|V| + |E|) \log |V|)$ if a binary heap is used for the priority queue.

(b) Using the arrays saved in the above algorithm, to query the shortest path from u to v passing through v_0, return $\texttt{dist}_{\text{to}}[u] + \texttt{dist}_{\text{from}}[v]$, where u, v are used here to mean the corresponding indices of the nodes. This is a constant time operation and the storage is linear in $|V|$.

(c) To query (u, v), first follow the pointers from u to v_0 in the array $\texttt{prev}_{\text{to}}$. This gives the path from u to v_0. Then then follow the pointers from v to v_0 in the array $\texttt{prev}_{\text{from}}$. This gives the reversed sequence of vertices in shortest path from v_0 to v. Return both sequences of vertices with the second sequence reversed. Both the query and storage are linear in $|V|$.

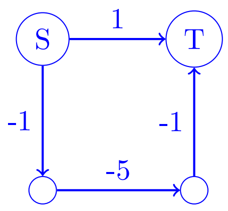
3 Fixing Dijsktra's Algorithm with Negative Weights

Dijkstra's algorithm doesn't work on graphs with negative edge weights. Here is one attempt to fix it:

1. Add a large number M to every edge so that there are no negative weights left.
2. Run Dijkstra to find the shortest path in the new graph.
3. Return the path Dijkstra found, but with the old edge weights (i.e. subtract M from the weight of each edge).

Show that this algorithm doesn't work by finding a graph for which it must give the wrong answer.

Solution: The above algorithm doesn't work when the actual shortest path has more edges than other potential shortest paths. In this case, the paths with more edges have their weights increased more than the path with fewer edges. We can see this in the following counterexample:



The shortest path is "down-right-up" (weight -7). After adding $M = 5$ to each edge, we increase the actual shortest path by fifteen 15. The path "right" only increases by 5 and so the algorithm returns this path as the shortest path.

4 Connectivity vs Strong Connectivity

(a) Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ such that removing v from G gives another connected graph.

(b) Give an example of a strongly connected directed graph $G = (V, E)$ such that, for *every* $v \in V$, removing v from G gives a directed graph that is not strongly connected.

(c) Let $G = (V, E)$ be a connected undirected graph such that G remains connected after removing any vertex. Show that for every pair of vertices u, v where $(u, v) \notin E$ there exist two different u-v paths.

Solution:

(a) Let T be a DFS tree on G, and let v be a leaf of T. Then $T - v$ is a connected graph because any simple path P from u to w $(u, w \neq v)$ in T cannot pass through v (since v has degree 1). Since $T - v$ is a subgraph of $G - v$, $G - v$ is also connected.

(b) A directed cycle of three nodes is an example here (i.e. $V = \{a, b, c\}$ and $E = \{(a, b), (b, c), (c, a)\}$).

(c) Let P be a u-v path in G; then P contains a vertex w which is not u or v. The graph $G - w$ does not contain P, but there exists a path P' connecting u and v since $G - w$ is connected. P' exists in G and is different from P.

5 Disrupting a Network of Spies

Let $G = (V, E)$ denote the "social network" of a group of spies. In other words, G is an undirected graph where each vertex $v \in V$ corresponds to a spy, and we introduce the edge $\{u, v\}$ if spies u and v have had contact with each other. The police would like to determine which spy they should try to capture, to disrupt the coordination of the group of spies as much as possible. More precisely, the goal is to find a single vertex $v \in V$ whose removal from the graph splits the graph into as many different connected components as possible. This problem will walk you through the design of a linear-time algorithm to solve this problem. In other words, the running time will be $O(|V| + |E|)$.

In the following, let $f(v)$ denote the number of connected components in the graph obtained after deleting vertex v from G. Also, assume that initial graph G is connected (before any vertex is deleted), has at least two vertices, and is represented in adjacency list format.

For each part, prove that your answer is correct (some parts are simple enough that the proof can be a brief justification; others will be more involved).

(a) Let T be a tree produced by running DFS on G with root $r \in V$. (In particular, $T = (V, E_T)$ is a spanning tree of G.) Given T, find an efficient way to calculate $f(r)$.

(b) Let $v \in V$ be some vertex that is not the root of T (i.e., $v \neq r$). Suppose further that no descendant of v in T has any non-tree edge (i.e. edge in $E \setminus E_T$) to any ancestor of v in T. How could you calculate $f(v)$ from T in an efficient way?

(c) For $w \in V$, let $D_T(w)$ be the set of descendants of w in T including w itself. For a set $S \subseteq V$, let $N_G(S)$ be the set of *neighbours* of S in G, i.e. $N_G(S) = \{y \in V : \exists x \in S \text{ s.t. } \{x,y\} \in E\}$. We define $\mathsf{up}_T(w) := \min_{y \in N_G(D_T(w))} \mathsf{depth}_T(y)$, i.e. the smallest depth in T of any neighbour in G of any descendant of w in T.

Now suppose v is an arbitrary non-root node in T, with children $w_1, \ldots, w_k$. Describe how to compute $f(v)$ as a function of k, $\mathsf{up}_T(w_1), \ldots, \mathsf{up}_T(w_k)$, and $\mathsf{depth}_T(v)$.

Hint: Think about what happened in part (b); think about what changes when we can have non-tree edges that go up from one of v's descendants to one of v's ancestors; and think about how you can detect it from the information provided.

(d) Design an algorithm which, on input G, T, computes $\mathsf{up}_T(v)$ for all vertices $v \in V$, in linear time.

(e) Given G, describe how to compute $f(v)$ for all vertices $v \in V$, in linear time.

Solution: Lemma: Let T be a DFS tree, and let $u, v \in V$ be such that u is neither a descendant nor an ancestor of v in T. Then there is no edge $\{u, v\} \in E$.

Proof: Suppose that there is an edge in $\{u, v\} \in E$, and suppose that u is visited first in the DFS. Then at some point we leave u without traversing $\{u, v\}$ (else v would be a child of u). But this means that v was visited between entering and leaving u, and so it is a descendant of u. If v was visited first, the same argument shows that v would be an ancestor of u.

(a) $f(r) =$ the number of children of r.

Proof: Let k be the number of children of r, and let $T_1, \ldots, T_k$ be the subtrees of T rooted at those children. If $u \in T_i, v \in T_j$ for $i \neq j$ then the conditions of the lemma hold and so $\{u, v\} \notin E$, and so each T_i is a connected component when we remove r.

(b) $f(v) = 1+$ the number of children of v.

Proof: Consider a partition of V into three sets: A, the ancestors of v, B, the tree rooted at v, and C, the rest of the graph. It suffices to show that there are no edges from $B - v$ to $A \cup C$, since $A \cup C$ is connected; applying the previous subpart to B then gives the result. For every $u \in B - v$, all descendants and ancestors of u lie in $A \cup B$, and so by the lemma there is no edge from u to C. By assumption there are no edges from u to A, and so $B - v$ has no edges to $A \cup C$.

(c) Let N denote the number of children c of v with the property that $\mathsf{up}_T(c) \geq \mathsf{depth}(v)$, i.e., $N = |\{c : c \text{ is a child of } v \text{ and } \mathsf{up}_T(c) \geq \mathsf{depth}(v)\}|$. Then $f(v) = N + 1$.

Proof: If we show that a child c has $\mathsf{up}_T(c) < \mathsf{depth}(v)$ iff c is connected to an ancestor of v by a path that excludes v, then the proof follows directly from (b) because these children are in the same connected component as the root r.

If c is connected to a proper ancestor a of v by a path that excludes v then $\mathsf{up}_T(c) \leq \mathsf{depth}(a) < \mathsf{depth}(v)$. Conversely, if $\mathsf{up}_T(c) < \mathsf{depth}(v)$, then there is an edge from a descendant d of v to some vertex w which has smaller depth than v. Since w cannot be a descendant of d, it must be an ancestor of d by the lemma, and since it has smaller depth than v, it is also an ancestor of v.

(d) By definition, $\mathsf{up}_T(v)$ is the minimum of v's neighbors' depths, and the up_Ts of v's descendants. Formally,

$$\mathsf{up}_T(v) = \min\left(\min\{\mathsf{depth}(w) : \{v, w\} \in E\},\ \min\{\mathsf{up}_T(w) : w \text{ is a child of } v\}\right).$$

We can thus compute $\mathsf{up}_T(v)$ by traversing the DFS tree bottom-up.

For a leaf, $\mathsf{up}_T(v)$ can be computed by minimizing over the depth of all neighboring vertices.

This can be computed in linear-time as each vertex and edge is considered a constant number of times.

(e) This follows immediately from parts (c)–(e). Pick some node as the root. Then compute $\mathsf{up}_T(\cdot)$ and $\mathsf{depth}(\cdot)$ at each node. Then, compute the function defined in part (d).

The running time is $\Theta(|V| + |E|)$. We needed to make three passes over the graph, one to compute $\mathsf{depth}(\cdot)$, one to compute $\mathsf{up}_T(\cdot)$, and a third to compute $f(\cdot)$. In each pass, we process each vertex once, and each edge at each vertex. This is $\Theta(|V| + |E|)$ for each pass, and $\Theta(3|V| + 3|E|) = \Theta(|V| + |E|)$.

Note that in a practical implementation, some of these separate passes could be combined, without affecting the asymptotic complexity of the algorithm.

6 Unique Shortest Path

Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((|V| + |E|)\log |V|)$ time.

Input: An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; starting vertex $s \in V$.

Output: A Boolean array `usp[·]`: for each node u, the entry `usp[`u`]` should be `true` if and only if there is a *unique* shortest path from s to u. (Note: `usp[`s`] = true`.)

[Provide 3 part solution.]

Solution: Main Idea:

Suppose there are two different shortest paths from s to u. These two paths can either share the same last edge (the edge ending at u), or not. If they do, this can be detected by modifying Dijkstra's to detect if a node has been added to the known region previously with the same distance. If not there must be two different shortest paths to u's parent, which can be detected by propagating (for every edge (a, b)) usp(a) to usp(b) if DECREASEKEY(H, v) is called.

Pseudocode:

This can be done by slightly modifying Dijkstra's algorithm. The array usp[·] is initialized to true in the initialization loop. The main loop is modified as follows (lines 7-9 are added):

```
1: while $H$ is not empty do
2:     $u$ = DELETEMIN($H$)
3:     for all $(u, v) \in E$ do
4:         if dist($v$) > dist($u$) + $l(u, v)$ then
5:             dist($v$) = dist($u$) + $l(u, v)$
6:             DECREASEKEY(H, v)
7:             usp[$v$] = usp[$u$]
8:         else if dist($v$) = dist($u$) + $l(u, v)$ then
9:             usp[$v$] = false
```

Proof of Correctness:

By Dijkstra's proof of correctness, this algorithm will identify the shortest paths from the source u to the other vertices. For uniqueness, we consider some vertex v. Let p denote the shortest path determined by Dijkstra's algorithm. If there are multiple shortest paths, then take another path $p' \neq p$ and it will either share the same final edge (w, v) (for some vertex w) as p, or they have different final edges from p. In the former case, there must be multiple shortest paths from u to w. Using an inductive argument, which supposes that usp is already set correctly for all vertices that are closer than v, this will be detected in the first conditional statement when the algorithm explores from w and updates the distance to v by taking edge (w, v), as it will detect that there are multiple shortest paths to w. In the latter case, if the last edges of the two shortest paths are (w_1, v) and (w_2, v), then since edge lengths $l_e > 0$, both w_1 and w_2 must be visited before v, thus the algorithm will detect the existence of multiple shortest paths with the second conditional statement. The base case is true because there is a unique shortest path from the source s to itself.

Runtime:

The runtime analysis follows that of Dijkstra's, and will run in the required time when a binary heap is used for the priority queue.