CS 170 Homework 4

Due 2019-09-25, at 10:00 pm

1 Study Group

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

2 Vertex Cut

Let $G = (V, E)$ be an undirected, unweighted graph with $n = |V|$ vertices. The distance between two vertices $u, v \in G$ is the length of the shortest path between them. A vertex cut of $G$ is a subset $S \subseteq V$ such that removing the vertices in $S$ (as well as incident edges) disconnects $G$.

Show that if there exist $u, v \in G$ of distance $d > 1$ from each other, then there exists a vertex cut of size at most $n - 2 + \frac{d}{d-1}$. Assume $G$ is connected.

Solution: For each $k \geq 0$, let $U_k$ be the set of vertices of distance $k$ from $u$. Note that the $U_k$ are disjoint (each vertex has a well-defined distance to $u$), $U_0 = \{u\}$, and $v \in U_d$.

Also notice that for any $k, j$ where $k > j + 1$, there cannot be an edge between a vertex in $U_k$ and a vertex in $U_j$ (if there was, then some vertex in $U_k$ would be of distance only $j + 1 < k$ from $u$). But for every $j$ where $U_j$ and $U_{j+1}$ are nonempty, there must be an edge from a vertex in $U_j$ to $U_{j+1}$ (otherwise $G$ would not be connected). So any nonempty $U_k$ where $k \geq 1$ is a vertex cut (if we remove it, there cannot exist paths from $U_j$ where $j < k$ to $U_i$ where $i > k$).

Consider $U_1, \ldots, U_{d-1}$. These $d-1$ sets of vertices are disjoint, and all together they have at most $n - 2$ vertices (excluding $u$ and $v$). In addition, all of them must be nonempty for a path to exist from $u$ to $v$.

We show that one of these sets must have size at most $\frac{n-2}{d-1}$. If this was not the case, they would all have size $> \frac{n-2}{d-1}$, meaning there would be $> \frac{n-2}{d-1}(d-1) = n - 2$ vertices among them, which is impossible because there are at most $n - 2$ vertices among these sets. (Not everyone is above average).

As one of $U_1, \ldots, U_{d-1}$ has size at most $\frac{n-2}{d-1}$, it is a vertex cut of size $\frac{n-2}{d-1}$.

3 True Source

Design an efficient algorithm that given a directed graph $G$ determines whether there is a vertex $v$ from which every other vertex can be reached. (Hint: first solve this for directed acyclic graphs. Note that running DFS from every single vertex is not efficient.)

Please give a 3-part solution to this problem.

Solution:

We provide two solutions below that both run in linear time (there may be many more).
Solution 1: In directed acyclic graphs, this is easy to check. We just need to see if the number of source nodes (zero indegree) is 1 or more than 1. Certainly if it is more than 1, there is no true source, because one cannot reach either source from the other. But if there is only 1, that source can reach every other vertex, because if \( v \) is any other vertex, if we keep taking one of the incoming edges, starting at \( v \), we have to either reach the source, or see a repeat vertex. But the fact that the graph is acyclic means that we can’t see a repeat vertex, so we have to reach the source. This means that the source can reach any vertex in the graph.

Now for general graphs, we first form the SCCs, and the metagraph. Now if there is only one source SCC, any vertex from it can reach any other vertex in the graph, but if there are more than one source SCCs, there is no single vertex that can reach all vertices.

Finding the SCCs/metagraph can be done in \( O(|V| + |E|) \) time via DFS as seen in the textbook, and counting the number of sources in the metagraph can also be done in \( O(|V| + |E|) \) time by just computing the in-degrees of all vertices using a single scan over the edges.

Solution 2: There is an alternative solution which avoids computing the metagraph altogether. The solution is to run DFS once on \( G \) to form a DFS forest. Now, let \( v \) be the root of the last tree that this run of DFS visited. Run DFS starting from \( v \) to determine if every vertex can be reached from \( v \). If so, output \( v \), if not, output that no true source exists.

It suffices to show no other vertex besides \( v \) can be a true source. In this case, if we determine \( v \) is not a true source, then saying there is no true source is correct. (Of course, if we find \( v \) is a true source, outputting it is also correct)

If there is a true source \( u \), \( v \) can’t reach \( u \) because \( v \) is not a true source. So nothing visited after \( v \) can be a true source, since \( v \) is the last root and thus all vertices visited after \( v \) are reachable from \( v \). But every vertex visited before \( v \) must not be able to reach \( v \), because otherwise the DFS would have taken a path from one of those vertices to \( v \) and thus \( v \) would not be a root in the DFS forest. So nothing visited before \( v \) can be a true source, since nothing visited before \( v \) can reach \( v \). Thus \( v \) is the only candidate for a true source.

Since the algorithm just involves running DFS twice, it runs in linear time.

4 Finding Clusters

We are given a directed graph \( G = (V, E) \), where \( V = \{1, \ldots, n\} \), i.e. the vertices are integers in the range 1 to \( n \). For every vertex \( i \) we would like to compute the value \( m(i) \) defined as follows: \( m(i) \) is the smallest \( j \) such that vertex \( j \) is reachable from vertex \( i \). (As a convention, we assume that \( i \) is reachable from \( i \).) Show that the values \( m(1), \ldots, m(n) \) can be computed in \( O(|V| + |E|) \) time.

Please give a 3-part solution to this problem.

Solution: Let \( G^R \) be the graph \( G \) with its edge directions reversed. The algorithm is as follows.

procedure DFS-CLUSTERS(G)
while there are unvisited nodes in \( G \) do
Run DFS on \( G^R \) starting from the numerically-first unvisited node \( i \)
for \( j \) visited by this DFS do \( m(j) := i \)
To see that this algorithm is correct, note that if a vertex \( i \) is assigned a value then that value is the smallest of the nodes that can reach it in \( G^R \), and every node is assigned a value because the loop does not terminate until this happens. Now observe that the set of vertices reachable by \( i \) in \( G^R \) is the set of vertices which can reach \( i \) in \( G \).

The running time is \( O(|V| + |E|) \) since computing \( G^R \) can be done in linear time (or faster if we use an adjacency matrix!), and we process every vertex and edge exactly once in the DFS.

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 the following, let \( f(v) \) denote the number of connected components in the graph obtained after deleting vertex \( v \) from \( G \). Also, assume that the initial graph \( G \) is connected (before any vertex is deleted), has at least two vertices, and is represented in an 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 neighbors 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 \( \text{up}_T(w) := \min_{y \in N_G(D_T(w))} \text{depth}_T(y) \), i.e. the smallest depth in \( T \) of any neighbor 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 \( k, \text{up}_T(w_1), \ldots, \text{up}_T(w_k) \), and \( \text{depth}_T(v) \).

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 \( \text{up}_T(v) \) 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 $\{u, v\} \in E$, 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 descendant 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 $\{A, B, C\}$, where $A$ is the set of ancestors of $v$, $B$ is the tree rooted at $v$, and $C$ is 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 $B$ 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 $\text{up}_T(c) \geq \text{depth}(v)$, i.e., $N = |\{i : c \text{ is a child of } v \text{ and } \text{up}_T(c) \geq \text{depth}(v)\}|$. Then $f(v) = N + 1$.

**Proof:** If we show that a child $c$ has $\text{up}_T(c) < \text{depth}(v)$ 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 $\text{up}_T(c) \leq \text{depth}(a) < \text{depth}(v)$. Conversely, if $\text{up}_T(c) < \text{depth}(v)$, 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, $\text{up}_T(v)$ is the minimum of $v$'s neighbors’ depths, and the $\text{up}_T$s of $v$’s descendants. Formally,

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

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

For a leaf, $\text{up}_T(v)$ 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 $\text{up}_T(\cdot)$ and $\text{depth}(\cdot)$ at each node. Then, compute the function defined in part (d). The running time is $\Theta(|V| + |E|)$. We need to make three passes over the graph: one to compute $\text{depth}(\cdot)$, one to compute $\text{up}_T(\cdot)$, 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 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).

(a) Give an efficient algorithm for finding shortest paths 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).

Please give a 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 distance of the shortest path from $u$ to $v$ that passes through $v_0$. Describe how you might store the results such that you require $O(|V|)$ storage, and 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 pair of vertices $(u, v)$, give the shortest path from $u$ to $v$ that passes through $v_0$. Describe how you might store the results such that you require $O(|V|)$ storage, and 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 this, 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 directions 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 \( \text{Dijkstra}(G, v) \) which finds shortest paths starting from \( v \), returning two arrays \( \text{dist} \) and \( \text{prev} \) as described in the textbook.

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

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 \( \text{dist}_{\text{to}}[u] + \text{dist}_{\text{from}}[v] \), 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 \( \text{prev}_{\text{to}} \). This gives the path from \( u \) to \( v_0 \). Then then follow the pointers from \( v \) to \( v_0 \) in the array \( \text{prev}_{\text{from}} \). This gives the reversed sequence of vertices in the 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 \( |V| \).

7 The Greatest Roads in America

Arguably, one of the best things to do in America is to take a great American road trip. And in America there are some amazing roads to drive on (think Pacific Crest Highway, Route 66 etc). An intrepid traveler has chosen to set course across America in search of some amazing driving. What is the length of the shortest path that hits at least \( k \) of these amazing roads?

Assume that the roads in America can be expressed as a directed weighted graph \( G = (V, E, d) \), and that our traveler wishes to drive across at least \( k \) roads from the subset \( R \subseteq E \) of “amazing” roads. Furthermore, assume that the traveler starts and ends at her home \( h \in V \). You may also assume that the traveler is fine with repeating roads from \( R \), i.e. the \( k \) roads chosen from \( R \) need not be unique.

Provide a 3-part solution with runtime in terms of \( n = |V| \), \( m = |E| \), \( k \), and \( r = |R| \).
Hint: First consider $k = 1$. How can $G$ be modified so that we can use a “common” algorithm to solve the problem?

**Solution:*** The main idea is that we want to build a new graph $G'$ such that we can apply Dijkstra’s algorithm on $G'$ to solve the problem. We start by creating $k + 1$ copies of the graph $G$, where each copy represents how many “amazing” roads the traveler has crossed. We modify the edges of the new graph so that they cross between the various copies. Then we apply Dijkstra’s to find the distance between $h$ in the $0$th copy and $h$ in the $k$th copy.

**Pseudocode:**
Generate $k + 1$ copies of the graph $G$. Call these copies $G_0 \ldots, G_k$, and let $R_0 \ldots, R_k$ be their respective amazing roads. For each edge $r_i = (u_i \rightarrow v_i) \in R_i$, for $i = 0 \ldots, k - 1$, modify the edge to be between $u_i$ and $v_{i+1}$. Let the entire graph be $G'$. Run Dijkstra’s algorithm on $G'$ starting from $h$ in $G_0$ and ending at $h$ in $G_k$.

**Runtime:**
Since $G'$ includes $k$ copies of $G$, Dijkstra’s algorithm will run in time $O((km + kn) \log (kn))$.

Since $k \leq m$ and $\log m = O(\log n)$, the runtime is $O(k(m + n) \log n)$.

**Correctness:**
Assume there is a valid path $p$ in $G$ that is shorter than the one produced by this algorithm. Consider the equivalent path $p'$ in $G'$ formed by modifying the path to go to the next copy of $G$ whenever an edge of $R$ is crossed. Since $p$ is valid, $p'$ must go from $h$ in $G_0$ to $h$ in $G_k$. But then $p'$ would be a shorter path in $G'$ than the one produced by Dijkstra’s, which is a contradiction.