CS 170 Homework 4

Due **2020-02-17, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

2 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.

3 Finding Clusters

We are given a directed graph $G = (V, E)$, where $V = \{1, \dots, n\}$, 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 $m(1), \dots, m(n)$ 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.

Alternative:

Solution 2

Main Idea Again observe that all the nodes in the same strongly connected component (SCC) will end up with the same value, simply by the definition of SCC. Thus we can turn graph G into the *dag* G' of G's SCC, and the problem gets reduced to compute those values for the *dag* G'. This motivates the following algorithm:

1. Compute the SCC in G using the standard algorithm. (i.e.: First run DFS on G^R and then run DFS on G, processing the vertices of G in order of decreasing *post* found in DFS of G^R.)

2. Compute the *dag* G' of G's SCC. Each vertex in G' is labelled with the minimum index in its corresponding SCC in G.

3. Label nodes with in G' procedure defined in pseudocode

4. Assign all the nodes in SCC j in G with $m[v]$, where v is the node in G' corresponding to SCC j in G.

Pseudocode

```
procedure DFS($G'$)
    for all nodes $v$ in $G'$ do
        $visited[v]$=NIL
        $m[v]$=label of $v$
    for all nodes $v$ in $G'$ do
        if not $visited[v]$ then
            Explore($G', v$)

procedure EXPLORE($G', v$)
    $visited[v]$=TRUE

    for all nodes $u$ such that $(v, u) \in E'$ do
        Explore($G', u$)
        if thenm$[u] < m[v]$
            $m[v] = m[u]$
```

We can that this algorithm is actually in effect nearly the same as Solution 1. The key difference however, is that this algorithm first identifies each SCC, and then determines connectivity between SCC's updating labels as it goes. This is different from Solution 1, where we don't ever actually pick out SCC's – rather, we pick out clumps of SCC's based on their ability to reach a particular vertex.

Proof of Correctness We know that if we have two vertices a and b in the same SCC, any vertex a can reach must also be reachable from b and vice versa, since by definition of an SCC there must be a path from a to b and a path from b to a. Thus it is sufficient to run the algorithm defined in the pseudocode on the DAG of the SCC's and relabel the vertices in the original graph appropriately.

We can prove the correctness of our algorithm on the DAG of the SCC's as follows. We know that if a vertex is a sink, the smallest vertex it can reach is itself. Assume our algorithm can correctly label any vertex at most k steps away from a sink in the DFS tree. The label of a node at most $k+1$ steps away from a sink must be the minimum of its own label, and the labels of its children in the DFS tree, as the smallest vertex it can reach is either itself or a vertex that one of its children can reach. Each of its children is at most k away from a sink, so we can compute their labels. After computing their labels, computing the min of their labels and the vertex itself is trivial.

Every vertex in a DAG is some finite number of steps away from a sink vertex, so by induction, we can correctly compute the label for every vertex in our DAG.

Runtime Analysis Each step above takes time $O(|V|+|E|)$, and hence the whole algorithm takes time $O(|V|+|E|)$.

4 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 DIJKSTRA(G, v) which finds shortest paths starting from v, returning two arrays `dist` and `prev` as described in the textbook.

```
$\texttt{dist}_{\text{from}}, \texttt{prev}_{\text{from}} \leftarrow$ DIJKSTRA$(G, v_0)$
$G' \leftarrow$ Reverse all edge directions of $G$
$\texttt{dist}_{\text{to}}, \texttt{prev}_{\text{to}} \leftarrow$ 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 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 $|V|$.

5 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 intiution is that we want to build a new graph G' such that we can apply Dijkstra's algorithm on G' to solve the problem. Roughly speaking, 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 0th copy and h in the kth copy.

The details of the algorithm are as follows. 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 $r_i = (u_i \to v_i) \in R_i$, 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:

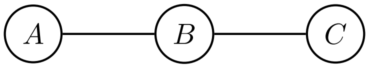
Assume there is a valid path p in G that is shorter than the one produced by this algorithm. Consider the equivalent path p' in G' formed by modifying the path to go to the next copy of G whenever an edge of R is crossed. Since p is valid, p' must go from h in G_0 to h in G_k. But then p' would be a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

6 Graph Game

Given an undirected, unweighted graph G, with each node v having a value $\ell(v) \geq 0$, consider the following game.

1. All nodes are initially *unmarked* and your score is 0.
2. Choose an unmarked node u. Let $M(u)$ be the *marked* neighbours of u. Add $\sum_{v \in M(u)} \ell(v)$ to your score. Then mark u.
3. Repeat the last step for as many turns as you like, or until all the nodes are marked.

For instance, suppose we had the graph:



with $\ell(A) = 3$, $\ell(B) = 2$, $\ell(C) = 3$. Then, an optimal strategy is to mark A then C then B giving you a score of $0 + 0 + 6$. We can check that no other order will give us a better score.

(a) Is the following statement true or false? **There is always an optimal strategy that marks all the nodes.** Briefly justify your answer.

(b) Give a greedy algorithm to find the order which gives the best score. Describe your algorithm, prove its optimality, and analyse its running time.

(c) Now suppose that $\ell(v)$ can be negative. Give an example where your algorithm fails.

(d) Your friend suggests the following modified algorithm: delete all v with $\ell(v) < 0$, then run your greedy algorithm on the resulting graph. Give an example where this algorithm fails.

Solution:

(a) Marking a node can only ever increase your score, since all values are positive.

(b) Sorting the nodes by value largest-first (breaking ties arbitrarily), and taking them in that order will lead to an optimal solution. We show this by an exchange argument. Suppose that we have an ordering $v_1, \ldots, v_n$ which is not sorted by decreasing value, and let i be such that $\ell(v_i) < \ell(v_{i+1})$. Note first that if we swap the order of these nodes, only the scores that we obtain in steps i and $i + 1$ could change, since in all other steps the set of marked nodes is unaffected. There are two cases.

   (a) Case 1: v_i and v_{i+1} do not have an edge between them. Then the score when marking v_i is not affected by whether v_{i+1} is marked, and vice versa.

   (b) Case 2: π_i and π_{i+1} do have an edge between them. Then if we swap them, v_{i+1}'s score decreases by $\ell(v_i)$, and v_i's score increases by $\ell(v_{i+1})$. Since $\ell(v_{i+1}) > \ell(v_i)$, the total score increases.

   Thus, by the exchange argument, sorting the nodes will give us an optimal solution. The running time of this algorithm is $O(|V| \log |V|)$, since we just sort the vertices.

(c) We can take the example graph and set $\ell(A) = 1$, $\ell(B) = -1$, $\ell(C) = -2$. Then the greedy algorithm gives A, B, C with value 0. The optimum is A, B with value 1.

(d) Again we take the example graph and set $\ell(A) = \ell(C) = 3$, $\ell(B) = -1$. The modified algorithm gives A, C which has value 0. The optimum is A, C, B with value 6.