COSC 302: Analysis of Algorithms — Spring 2018
Prof. Darren Strash
Colgate University

Problem Set 6 — Graph Traversals
Due by 4:30pm Friday, March 30, 2018 as a single pdf via Moodle (either generated via LaTeX, or concatenated photos of your work). Late assignments are not accepted.

This is an *individual* assignment: collaboration (such as discussing problems and brainstorming ideas for solving them) on this assignment is highly encouraged, but the work you submit must be your own. Give information only as a tutor would: ask questions so that your classmate is able to figure out the answer for themselves. It is unacceptable to share any artifacts, such as code and/or write-ups for this assignment. If you work with someone in close collaboration, you must mention your collaborator on your assignment.

Suggested practice problems, from CLRS: 22.1-1 through 22.1-5; 22.1-6 (challenge); 22.2-3; 22.2-6; 22.-3-6; 22.3-8; 22.3-11; 22.4-2; 22.4-5; 22.5-3; 22.5-4

1. **(From problem set 5; previous exam question)** Let $A[1..n]$ be an array of non-integers taken from some set K of size $k > 1$. *(Note: For this problem, you are not given the set K or k; this is only to illustrate that there are k distinct non-integer numbers. We only have access to elements through A. Further, note that k may be small or large: from constant to even larger than n.)*

   (a) Describe an algorithm that sorts A in expected time $O(n + k \lg k)$, and describe why it has this running time.

   (b) What is the worst-case running time of your algorithm? Justify your answer.

   Solution: First, we build a hash table of size $2n$, where collisions are resolved with hashing. We will store $\min\{k, n\}$ unique elements in this table, and therefore the load factor will be constant. We further assume we have a constant-time hash function h that maps elements in A to a slot of the hash table, and we make the simple uniform hashing assumption about h. Now on to the algorithm:

   (a) We begin by de-duplicating the data set by inserting all n elements into a hash table. However, with each element in the hash table, we store a count indicating how many occurrences of that value are in A. Thus, we insert $\min\{k, n\}$ of these elements in the hash table. This takes $\Theta(n+\min\{k, n\})$ expected time, or $\Theta(n+n\cdot\min\{k, n\})$ worst-case time.

   (b) Next, we traverse over the elements in the hash table in $\Theta(n)$ time, and insert them, along with their counts into a red-black tree in time $\min\{k, n\} \lg\{\min\{k, n\}\}$ time. We then perform an inorder traversal of the tree in $\min\{k, n\}$ time, copying over all elements to the output array in $\Theta(n)$ time.

   The expected running time for this procedure is $\Theta(n + \min\{k, n\} + \min\{k, n\} \lg \min\{k, n\}) = O(n + k \lg k)$. However, the worst-case time is $\Theta(n + n \cdot \min\{k, n\} + \min\{k, n\} \lg \min\{k, n\}) = \Theta(n + n \cdot \min\{k, n\})$.

2. Let $G = (V, E)$ be an n-vertex undirected graph consisting of *cops* and *robbers* as vertices. You are given two facts about the graph:

   (a) G is connected, and

   (b) each edge is incident to exactly one cop and one robber. (That is, no edge is incident to two cops, and no edge is incident to two robbers.)

   Suppose we know that Dave $\in V$ is a cop. Give an efficient algorithm to distinguish all cops from all robbers.

   Solution: Perform BFS from Dave and compute the distance that each vertex is from Dave. Vertices at distance 1 must be robbers (as these are neighbors of a cop, Dave). Vertices at distance 2 must be cops, as they are neighbors of the robbers at distance 1. In general, a vertex u is a robber if it has distance $\delta(s, u) \mod 2 = 1$ from Dave, and a cop otherwise. Since BFS evaluates vertices in order by distance (and can be augmented to compute distance for each vertex) we can compute this property for each vertex throughout BFS, thus identifying each vertex as either being a cop or a robber.

3. Problem 22.2-5 from CLRS.

   Solution:

   (a) During breadth-first search, values $d[u]$ represent the distance from s in the graph G. The distance in G never changes, therefore, by correctness of breadth-first search $d[u]$ will always be $\delta(s, u)$, independent of the order in which neighbors are evaluated.

   (b) See Figure 1. Suppose we are evaluating w's neighbors x and t. If we first enqueue t, then we will add edge (t, u) as a tree edge during breadth-first search. However, if x is enqueued before t, then edge (x, y) will be added as a tree edge. Therefore, depending on the order we evaluate vertices, we can get different breadth-first trees.

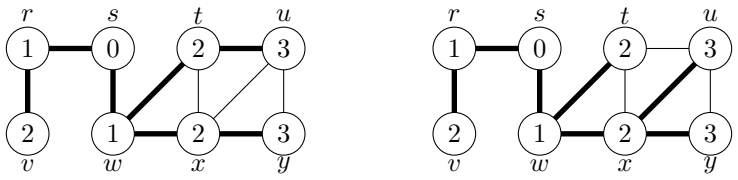

Figure 1: Two different breadth-first trees.

4. Problem 22.5-1 from CLRS. In addition to stating your answer, also (formally) prove its correctness.

   Solution: The number of strongly connected components can decrease or stay the same, but never increase. And, in fact, the number of strongly connected components can decrease to one.

   Claim 1. *Let $G = (V, E)$ be a directed graph with k strongly connected components. Let (u, v) be an edge such that $u, v \in V$, but $(u, v) \notin E$, and let $G' = (V, E \cup \{(u, v)\}$. Then G' has between 1 and k strongly connected components.*

   Proof. Let $C_1, C_2, \ldots, C_k \subseteq V$ be the strongly connected components of G. There are two cases, either

   (a) Vertices u and v are in the same strongly connected component C_i.
   Then G' has k strongly connected components, since it is already the cases that there exists a path from u to v and v to u since C_i is strongly connected.
   Firstly, the number of strongly connected components cannot increase. Since if there exists a path from x and y in G' then the addition of edge (u, v) does not remove connectivity between vertices.

   (b) Vertices u and v are in different strongly connected components C_i and C_j. Then either:

      i. There is a path from u to v in G.

      Then adding edge (u, v) does not change the number of strongly connected components and there remain k strongly connected components in G'. We can see this as follows.

      Let x and y be two vertices connected after the addition of (u, v). Let π_{xy} be a path between them in G'. If π_{xy} does not contain edge (u, v) then it could not have been formed by adding edge (u, v). Suppose then that it contains (u, v). Then, since there was a path ρ_{uv} between u and v before the addition of edge (u, v), we can "splice out" edge (u, v) from π_{xy} and substitute ρ_{uv} and thus there was already a path from x to y before adding (u, v). Therefore, since no new vertices are connected via a path, the strongly connected components remain the same.

      ii. There is a path from v to u in G. Then then the number of strongly connected components in G' is at most $k - 1$. We see this as follows.

      Before adding u and v there was no path from u to v in G. Since in G' there exist paths from u to v and v to u, vertices u and v are now in the same strongly connected component. That is, $C_i \cup C_j$ are contained in the same strongly connected component in G'. Note that adding this edge can reduce the number of strongly connected components to 1. In particular, if the graph G^{SCC} is a single path $C_1, \ldots, C_k$ and $u \in C_k$ and $v \in C_1$, then paths exist between all pairs of nodes in G'.

iii. There is neither a path from u and v nor from v to u in G.

Then adding edge u, v does not change the number of strongly connected components. We see this as follows.

Let x and y be two vertices connected such that there exists path π_{xy} and π_{yx} after the addition of (u, v). If neither path contains edge (u, v) then there was a path from x to y and y to x before the addition of (u, v). Otherwise, suppose (u, v) is is on some path from x to y or y to x. Let π_{xy} be a path between them in G'. If π_{xy} does not contain edge (u, v) then it could not have been formed by adding edge (u, v). Suppose then that it contains (u, v). Then, since there was a path ρ_{uv} between u and v before the addition of edge (u, v), we can "splice out" edge (u, v) from π_{xy} and substitute ρ_{uv} and thus there was already a path from x to y before adding (u, v). Therefore, since no new vertices are connected via a path, the strongly connected components remain the same.

□

5. Problem 22.5-7 from CLRS.

Solution: We begin by computing the weakly connected components of G, by forming $G \cup G^T$ and running BFS to determine if $G \cup G^T$ is connected. If not, there are multiple weakly connected components in G, and there exist two vertices in different weakly connected components such that there is not path from u to v or from v to u (i.e., the graph is not semiconnected). We then compute the strongly connected components of G, giving us a dag G^{SCC}. If G^{SCC} consists of a single vertex then G is semiconnected, as there is a path between all pairs of vertices. If not, we compute a topological ordering of G^{SCC}, giving us an ordering of the vertices. We now refer to the vertices as $v_1, v_2, v_3, \ldots, v_k$ where the subscript refers to the vertex's position in topological order.

Claim 2. *Graph G is semiconnected if and only if, in a topological ordering of G^{SCC}, edge (v_i, v_{i+1}) exists for all $i = 1..n-1$.*

Proof. ($\Rightarrow$): Proof by contrapositive. Suppose there exists a pair (v_i, v_{i+1}) in G^{SCC} that are not connected by an edge. Then there is no path from v_i to v_{i+1} in G^{SCC}, since all edges go from v_l to v_m where v_l comes before v_m in the topological ordering. Any edge leaving v_i then would be after v_{i+1}, and so would remaining edges on paths from v_i. Let C_i and C_{i+1} be the corresponding strongly connected components in G. Let u be some vertex in C_i and v be some vertex in C_{i+1}. Since there is no path from v_i to v_{i+1} in G^{SCC}, then there is no path from u in C_i to v in C_{i+1} in G.

($\Leftarrow$): Suppose edge (v_i, v_{i+1}) is in G^{SCC} for all $i = 1..n-1$. Let v_l and v_m be any two vertices in G^{SCC}, and assume without loss of generality that v_l comes before v_m in the topological ordering. Then there is a path from v_l to v_m in G^{SCC}, the path: $v_l, v_{l+1}, \ldots, v_{m-1}, v_m$. Let C_l and C_m be the corresponding strongly connected components of G, and let $u \in C_l$ and $v \in C_m$. Then there is a path from u to v in G. Since this is true for all u and v in any two strongly connected components. Then there is a path between all pairs u and v and G is semiconnected. □

Therefore, for each v_i, we check its neighborhood to see if it has a neighbor v_{i+1}, returning true if this is true for every vertex v_i for $i = 1..n-1$.

All steps of the algorithm take $O(V + E)$ time.