CS 457, Data Structures and Algorithms I
Sixth Problem Set Solutions

December 7, 2019

For full credit, for any algorithms that you propose, also provide a (high level) justification regarding their worst-case running time bounds, and a brief explanation regarding their correctness. Throughout the problem set n denotes the number of vertices of a graph and m denotes the number of edges.

1. Your textbook defines the *connected components* of undirected graphs and the *strongly connected components* of directed graphs (see pages 1170-1171). Adding edges to a graph increases the connectivity and may lead to a decrease in the number of such components. i) If you add an edge to an undirected graph G, what is the largest possible drop in the number of connected components of G that this can possibly lead to? Similarly, ii) if G is a directed graph and you add a directed edge to it, what is the largest possible drop in the number of strongly connected components of G that this can lead to? Provide both upper and lower bound arguments to support your claims. For instance, if you claim that the largest possible drop is $n/4$, then provide an example where the drop is $n/4$ and an argument why it can be no more than $n/4$ in general.

   Solution:

   In an undirected graph, the largest possible drop in the number of connected components is 1. For a lower bound instance, consider a graph of two disconnected vertices. When we add an edge between them we go from 2 connected components to 1. To see that this is also an upper bound, suppose that we add an edge to some graph G between vertices v_1 and v_2. If v_1 and v_2 were previously in the same connected component, then there existed a path from v_1 to v_2 previously and v_1 was then connected to everything v_2 was connected to (since the graph is undirected, the same reasoning holds for v_2 being connected to everything v_1 was connected to). The addition of the edge has therefore not connected any previously disconnected components so there is no drop in the number of connected components. Now suppose v_1 and v_2 lie in distinct connected components, C_1 and C_2, respectively. The addition of the edge certainly connects components C_1 and C_2. Suppose it also connects another distinct component C_3 to C_1 and C_2. Previously, v_1 did not have a path to any vertex in C_3 by assumption, but the addition of an edge between v_1 and v_2 connected v_1 to C_3. This means that v_2 must have some path to the vertices in C_3, but then C_2 and C_3 were the same connected component, contradicting the assumption that they were distinct. The other case holds by a symmetric argument. Therefore, the total number of connected components can drop by at most one given the addition of a new edge.

   In a directed graph, the number of strongly connected components can drop by at most $n-1$ on a graph of n vertices. For the lower bound, consider a directed path of n vertices and $n-1$ directed edges from v_i to v_{i+1} for all $1 \leq i < n$. Since there are no back edges, each vertex lies in a distinct strongly connected component and there are thus n strongly connected components. If we add a directed edge from v_n to v_1, we form a cycle containing all vertices so they all then lie in the same strongly connected component (and there is thus 1 strongly connected component). This gives a drop of $n-1$ strongly connected components. To see that this is also an upper bound, consider that in any graph each vertex lies in exactly one strongly connected component and there is at least one strongly connected component in any graph. Therefore, on a graph of n vertices there is at most n strongly connected components and at least 1 strongly connected component meaning that the maximum possible drop in strongly connected components on the addition of an edge is $n - 1$.

2. (25 pts) You are given a weighted graph $G = (V, E)$ as well as a minimum spanning tree T of G. Then, a new edge e is added to G and you want to compute a minimum spanning tree for the new graph $G' = (V, E \cup \{e\})$. Provide an algorithm NEW-MST(G, T, e) that computes a minimum spanning tree T' for G' in time $O(n)$. Clearly, T' could be computed from scratch using Prim's or Kruskal's algorithm, or one could use algorithms covered in recitation, but all of these solutions would require time $\omega(n)$ which is not good enough.

   Solution:

   Call the two vertices that the edge added to G connects u and v. Since there are n vertices in the graph and n edges including (u, v) and the original edges of T, the addition of (u, v) must form a cycle. A new minimum spanning tree T' for G' can be found by removing the heaviest edge in this cycle. To see this, we make the observation that if an edge $e \notin E(T) \bigcup \{(u, v)\}$ is in T' then it should have been in T as well. More precisely, we first note that Kruskal's algorithm is capable of generating every MST of a graph by breaking ties in the correct order. Consider rerunning Kruskal's algorithm on G' breaking ties in the same order as was done to generate T in G. When we must decide if (u, v) should be added to T', we first examine if u and v lie in separate sets. If not, we ignore it and would generate the exact same T. If they do lie in separate sets, we would add (u, v) to T and merge the sets containing u and v. At some point in the initial running of Kruskal's algorithm to generate T, we would have merged the two sets containing u and v through the addition of an edge, but now we would ignore this edge. This is the only change in the operation of the algorithm on the two edge sets, so the only possible edges added to T' are those originally in T and (u, v).

   This observation suggests an algorithm to solve this problem. We will look for the cycle containing (u, v) and discard the heaviest edge of this cycle. Fortunately, since the number of edges and vertices are the same, we can examine all of the vertices and all of the edges in T without exceeding our desired running time.

   In the rooted minimum spanning tree (which we would have obtained with Prim's algorithm or we can run breadth-first search in $O(V)$ time just using the vertices and edges in T), u is an ancestor of v, v is an ancestor of u, or they share a common ancestor. Beginning with u, continually follow the parent pointer in the tree until v is found or the root is reached (the parent pointer of the root is null) marking each vertex found. If v is found, all marked vertices are part of the cycle and the exact edges to be examined have been noted. If the root is found, begin the same process of following parent pointers from v until u is found or the first marked node from the first step is found. If u is found then all edges followed from v to u are in the cycle. If a marked node y is found, then all edges from u to y and all edges from v to y are in the cycle. Since we are only examining edges in the tree (which are at most the number of vertices) this process takes $O(n)$. It then remains to examine the weights of each of the edges in the located cycle. If the maximum weight edge in this cycle exceeds the weight of (u, v) remove it and add (u, v) to T', otherwise discard (u, v) (since $T' = T$).

3. (25 pts) You are given a directed graph $G = (V, E)$ along with a function $w : E \to \mathbb{R}_+$ that assigns positive weights to all the edges. This graph may contain directed cycles and your goal is to check if such cycles exist in G and, if they do, to return one with the smallest possible weight. The running time of your algorithm should be $O(n^3)$ or $O(n^2 m)$ (whichever one you prefer). You can use any algorithm from Chapter 24 as a subroutine.

   Solution:

   We first make the observation that we can view all cycles (and indeed the smallest weight cycle) as beginning at some vertex u and passing through others before arriving at some vertex v which has an edge leading back to u. This observation suggests the algorithm. For every vertex v in the graph, run Dijkstra's algorithm beginning at v and then pass over each other vertex w. We now know the shortest paths from v to all w. We then add the cost of a back edge from w to v if it exists, otherwise we "add infinity". This gives us the smallest weight cycle beginning at v passing to w (for each w) and returning to v. The smallest weight cycle beginning and ending at v is the one which has the minimum sum of distance to w plus the cost of the back edge from w to v. We then select the minimum of the smallest weight cycles beginning and ending at each vertex v and return it as the smallest weight cycle in the graph. We are running a $O(n^2)$ algorithm (Dijkstra's algorithm) in addition to doing $O(n)$ extra processing (checking all other vertices to find the minimum weight cycle) for each vertex. Therefore we do $n \cdot (O(n) + O(n^2)) = n \cdot O(n^2) = O(n^3)$ total work, satisfying the required running time.