CS 124 Data Structures and Algorithms — Spring 2015

Problem Set 2 solutions

1 Problem 1

A directed graph is *nice* if for every pair of distinct vertices u, v, there is at most 1 non-cycling path from u to v (a path is "non-cycling" if it doesn't contain a cycle). Give an efficient algorithm to determine whether an input graph is nice, and prove its correctness.

Solution "Nice" graphs are typically called "singly-connected" in the literature. We claim that G is singly-connected iff for every vertex u, starting a DFS from u only results in tree and back edges. Each DFS takes $O(n+m)$ time, and we run $O(n)$ of them, for a total running time of $O(n(n+m))$. For correctness, we prove both directions.

singly-connected $\implies$ only tree and back edges

We prove the contrapositive (namely that $\neg$(only tree and back edges) implies $\neg$(singly-connected)). Suppose that in our DFS starting from u, we have a forward edge from v to w. Then we have a path from v to w given by that forward edge, and a different path consisting only of tree edges from the DFS (since by definition of a forward edge, w is a descendant of v). This contradicts the fact that our graph is singly-connected. Similarly, if we have a cross edge from v to w, then we have a path from u to w consisting only of tree edges via the DFS, and a different path by appending the cross edge to the path of tree edges from u to v. This contradicts that our graph is singly-connected. Therefore, we can only have tree and back edges.

singly-connected $\impliedby$ only tree and back edges

We again prove the contrapositive. Suppose that our graph is not singly-connected. Then there must be at least one pair of vertices with two distinct paths between them. Let p be the shortest path (break ties arbirarily) for which there exists an alternate non-cycling path, and say p goes from u to v. Denote the alternate, second-shortest path by p'. Notice that p and p' cannot have the same second-to-last vertex, since we could then just remove the last edge from both paths to make an even shorter path with an alternate route.

Then when we run DFS from u, we produce a path from u to v. If this path is p, then the last edge in p' is a cross edge. We can see this through process of elimination: it's not in the tree, it's not a back edge (or there would be a cycle in p'), and it's not a forward edge (or we could make a shorter path than p). This is a contradiction. If the path is not p, then either p is a single edge (which would be a forward edge) or it has a final edge that would be a cross edge by the previous case. This completes the proof.

It is also possible to achieve a running time of $O(n^2)$ for this problem, although $O(n(n+m))$ solutions will receive full credit. Details of the $O(n^2)$ solution can be found here: `https://www.cs.umd.edu/~samir/grant/khuller99.ps`. The $O(n^2)$ solution only uses concepts taught in lecture and is accessible.

2 Problem 2

You are given an array A of length n containing distinct integer elements. You are also given the promise that, for each index $1 \leq i \leq n$, $A[i]$ is within t positions of where it should be if A had been sorted. Show that A can be sorted in time $O(n \log t)$.

Solution Split the array A into $2t$ arrays $B_0, B_1, \ldots, B_{2t-1}$, where B_j contains every $2t$-th element of A starting at element j. That is,

$$B_j[i] = A[i \cdot 2t + j].$$

By the condition that the every element of A is at most t indices away from being sorted, we know that when A is sorted, $A[i]$ has index at most $i + t$, and $A[i + 2t]$ has index at least $i + t$. Thus, $A[i] \leq A[i + 2t]$ for all i, which tells us that all of the B_j are already sorted. To then sort A, we insert the first elements of all of the B_j into a heap. Then at each step, remove the minimum element from the heap, check which B_j it was from, and insert the next element from that array into the heap (if one exists). We are guaranteed to find the smallest remaining element, since each B_j is sorted, so the smallest element remaining must be the smallest remaining element of some B_j. Then since this takes $O(n)$ insertions, $O(n)$ deletions, and $O(n)$ findmins, and the heap's size is bounded at all times by $2t$, the running time is

$$O(n)O(\log 2t) + O(n)O(\log 2t) + O(n)O(1) = O(n \log t).$$

3 Problem 3

In a directed graph $G = (V, E)$ a *termination* point is a vertex $v \in V$ such that for all $w \in V$, if there exists a path in G from v to w, then there also exists a path from w to v. Describe and analyze an efficient algorithm which receives G in adjacency list representation and outputs a list of all termination points.

Solution First, find the strongly connected components. Then for each SCC, iterate over the vertices. For each vertex u, consider all edges (u, v). If any of these v is in a different SCC than u, we conclude that every vertex in in our current SCC is not a termination point. Otherwise, add all vertices in the SCC to our list of termination points. Equivalently, the termination points are the vertices which are in an SCC whose node in the SCC graph is a sink (i.e. has out-degree 0). The running time to compute the SCC's is $O(n + m)$, and the running time to check for termination points is $O(n + m)$, so the overall running time is $O(n + m)$.

Now we show correctness. If we have a termination point u, then any outgoing edge from its SCC to a vertex v in another SCC gives us a path from u to v. This then implies a path from v to u by the definition of a termination point, so the two SCC's are the same, a contradiction. Hence, no such outgoing edge can exist. On the other hand, if there are no outgoing edges from the SCC containing u to another SCC, then every path from u must end

at a point within the same SCC. Therefore, there is a path back to u, so u is a termination point.

4 Problem 4

You are a merchant, and you want to walk from your home to your shop. You figure that you might as well get a good workout while doing so. Your city contains various roads and intersections, with various lengths of road between each intersection. Each intersection is also elevated at some height above sea level. Your primary goal is to get to your shop using the shortest route possible. However, you want a path in which all road segments you take are initially all inclined upward (the exercise) followed by road segments which are all inclined downward (to cool down). The number of road segments you take inclined upward before you switch to going downward is up to you (you could also choose a route that is entirely upward or downward). Give efficient algorithms to find tshe shortest such paths in the following two cases:

(a) All elevations are distinct.

(b) Some elevations may be the same. You may walk on flat roads during both the uphill and downhill portions of your walk.

Solution

(a) For each intersection x, we find the minimum distance from your home to x via only uphill roads, and the minimum distance from x to your shop via only downhill roads. When we only consider uphill roads, we have a directed acyclic graph, so we can find all shortest paths in $O(n+m)$. Similarly, we find all shortest downhill paths to the shop in $O(n+m)$. Then we iterate over each vertex and add the two values, keeping track of the minimum, which takes $O(n)$ time. Thus, the overall time is $O(n+m)$.

(b) We can use a similar approach, except we now use Dijkstra's to compute the shortest uphill and downhill distances, since we no longer have a DAG. This now runs in $O(m+n\log n)$.

Alternatively, one could build a directed graph G' on $2n$ vertices in the following way. For each vertex x we make two vertices x^0, x^1 where being at x^0 means we are currently still trying to go uphill, and being at x^1 means we've switched to trying to go downhill. Thus for every edge (x, y) in the original graph, if (x, y) goes uphill then we make the edge (x^0, y^0). If it is going downhill then we make the edge we make the edge (x^1, y^1). If they are the same altitude, then we create both edges. We also make the edge (x^0, x^1) of length 0 for every x (so that we can always transition to going downhill from now on, at zero cost). If h is the home location and s is the shop, then we want the shortest path from h^0 to y^1.

If the elevations are distinct, G' is a DAG and we can do this in $O(n+m)$ time using Lecture Notes 4 (n is the number of intersections, and m is the number of road segments

connecting intersections). If elevations are not distinct, Dijkstra's algorithm can be used to yield time $O(m + n \log n)$ (e.g. if using Fibonacci heaps), or $O(m \log n)$ time using binary heaps.

5 Problem 5

Solve "Problem A - Snake" on the programming server; see the "Problem Sets" part of the course web page for the link.

Solution We can solve this problem using a completely unmodified breadth-first search (BFS), as seen in class. Define a "snake configuration" to be a tuple of $2k$ numbers $(x_1, y_1, x_2, y_2, \ldots, x_k, y_k)$, where (x_i, y_i) is the location of the ith body piece of the snake. The graph G we do the BFS on is the graph where every possible snake configuration is a vertex. An edge from node u to node v exists if we can get from configuration u to v in one move. The source vertex fed into BFS is simply the starting configuration of the board. We are trying to find the minimum distance to any vertex corresponding to a snake configuration where the snake's head has landed on the apple.

This graph G has some number N of vertices and M of edges. Note $M \leq 4N$, and in fact $M \leq 3N$ in cases where $k > 1$ (the snake can move in four directions at any given time, except back onto its '2' piece of $k > 1$). So, what is N? One upper bound is $N \leq (mn)^k$, but this is actually quite a bad bound given m, n, k in the problem statement. A better bound is as follows: note the head has mn possible positions. Given where the head is, each subsequent body piece has at most 4 possible locations (it must be adjacent to the previous piece). Thus $N \leq mn4^{k-1}$. For $k \leq 9$ and $m, n \leq 10$, this is 6,553,600, which isn't a lot given the time limit. In fact one can give an even better bound: each piece j from '3' onward actually only has 3 possibilities since it must be adjacent to piece $j - 1$ but it also cannot be in the same location as piece $j - 2$. Thus $N \leq 4mn3^{k-2}$, which is at most 874,800. Furthermore, to follow an "edge" in this graph can be implemented to take $O(k)$ time (moving the k pieces of the snake along one direction), and thus BFS takes $O(N + kM) = O(kN)$ time, which is not much here.

In fact, even the $N \leq 4mn3^{k-2}$ bound is far from tight since there are more ways a configuration can fail to be valid which are not being considered here (e.g. the jth piece colliding with the $(j-4)th$ piece for some j). As an exercise you may wish to write a simple recursive procedure which counts how many valid length-k snake configurations exist in a 10×10 grid. If you find the combinatorics of this interesting, you may be interested to read more: check out the article on Wikipedia for a related (but not identical) problem: `http://en.wikipedia.org/wiki/Self-avoiding_walk`.

One implementation detail is that a snake configuration can be maintained as an `int` so that checking whether a snake configuration has been visited already can be checked in constant time by an array lookup. See the solution code for details.