CSCI 2300 Fall 2020 Introduction to Algorithms

Exam 1 Solutions

Problem 2: Topological Sort This is False: Consider for example any graph with 2 source nodes.

Problem 3: Warehouse Fire

(a) Form a graph $G = (V, E)$ as follows. There is a node for every cell, and a cell has edges to their neighboring cells (not diagonally) when they are both warehouse cells. Thus $|V| = k^2$ and $|E| < 4k^2 = O(k^2)$. Then, if a fire starts at a certain cell, the cells which will be burned are exactly the connected component of the cell where the fire started. Now run BFS (or DFS) on this graph to find the connected components. Look at the size of each connected component: the maximum size is the maximum amount of cells that can be burned. Running BFS takes linear time in the size of G, and so is finding the component of maximum size. Thus, the total runtime is $O(|V| + |E|) = O(k^2)$.

Note that some students gave an incorrect way of finding connected components, by running BFS from every node separately. This is instead of running BFS from a node to find a connected component, then seeing if any nodes are still unvisited, and then running it from one of the unvisited nodes. Running BFS from each node is not linear time, it is $\Theta(|V|(|V| + |E|))$, which for this graph would be $\Theta(k^4)$.

(b) Create the same graph as in part (a), but add a new node s_0 with edges to all nodes in S. Then run BFS from s_0. The distance from s_0 to any node v is exactly one plus the time it would take the fire to reach it. As before, the runtime of BFS is $O(|V| + |E|) = O(k^2)$. Alternatively, instead of adding a node s_0, we can just change BFS and run it simultaneously starting at all nodes in S instead of starting at a single node, and not taking edges to nodes which were already visited. Note that running BFS from each node in S separately would be too slow: it would take time $\Theta(|S|(|V| + |E|))$, which could be as high as $\Theta(k^4)$ if S is large.

Problem 4: DNA analysis Form a graph $G = (V, E)$ as follows. There is a node for each fragment, and an edge from i to j if i must be located to the left of j. There exists a consistent ordering exactly when this graph is a DAG, and this ordering is a topological sort of this graph. First detect if the graph has any cycles (this can be done by running Kosaraju's algorithm, for example, and then seeing if any SCC has more than one node). If there is a cycle, then there is no consistent ordering. If there is no cycle, form a topological sort of the graph to get the ordering. Both Kosaraju and topological sort are linear time, so the total runtime is $O(|V| + |E|) = O(n^2)$, since there are n nodes and at most n^2 edges.

Problem 5: Evacuation Centers First obtain a DAG of SCC's in linear time using Kosaraju's algorithm. Now consider any sink SCC. We must put at least one evacuation center in this SCC, since there are no edges leaving this SCC and thus nodes in this SCC cannot reach anything outside of it. What happens if we put exactly one evacuation center in each of the sink SCC's? Then this set R obeys the desired property. To see this, first note that all nodes inside an SCC are reachable from each other, so we can just consider each SCC to be a single node, and assume our graph is a DAG. Now, consider an arbitrary node v. If it has edges leaving it, follow an arbitrary one of these edges. Continue following edges which are leaving it until we get to a sink node. By our construction, this node has an evacuation center, so there is a path from v to it. Thus, placing an evacuation center at each sink SCC is enough to guarantee reachability from all nodes in the graph, and it is the smallest such set since we *must* put at least one evacuation center in each sink SCC.

In conclusion, the full algorithm is: (1) Run Kosaraju's algorithm, which takes $O(|V| + |E|)$ time. (2) Put one evacuation center inside each sink SCC. Finding the sink SCCs requires only to see which ones have edges leaving them, and so can also be done in $O(|V| + |E|)$ time.

Problem 6: There and back again First we can run Dijkstra from s, which gives us shortest distances to all other nodes, including nodes in T. Then we form a new graph G' which is the same as G, but with edges reversed, and run Dijsktra on that graph from s. This gives us shortest distances *to* s from all nodes in T. Finally, for each $t \in T$ we add the distance from s to t and from t to s, and take the minimum one. In total, the runtime is $O(|V|^2)$, since it is dominated by the two runs of Dijkstra. Note that running Dijkstra from each node in T would take too long, as that would take $O(|T||V|^2)$ time, which could be as bad as $O(|V|^3)$.