Tutorial 7: Graph algorithms

1 DAGs

Give an $O(n+m)$-time time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of paths from s to t in G.

1.1 Solution

Topologically order G. This linearizes the vertices, such that each edge moves from left to right. We take the set of vertices between s and t and edges between these vertices. Suppose w.l.o.g., that the order of this part is $s = v_1, ..., v_k = t$. Let's call this subgraph of G as $G_{s...t}$. Construction of $G_{s...t}$ takes $O(n + m)$ time.

Next we run a DP algorithm that finds number of paths to t from any other node in $G_{s..t}$ (you can also run a similar DP algorithm that finds the number of paths from s to any other node in $G_{s...t}$).

Algorithm 1: NUMPATHS($G_{s...t}$)

1 // recall k is the # nodes in $G_{s...t}$, so at most n.
2 // nP[i] below is the number of paths to t from v_i.
3 $nP[k]$ is a solution array initialized to 0;
4 **for** $i = t$-1...1 **do**
5   **for** $(v_i,\ v_j) \in G_{s...t}$ **do**
6   np[i] += np[j];
7 // $s = v_1$, so we return np[1]
8 return np[1];

The DP algorithms correctness follows from the recurrence that the number of paths from v_i to t is the sum of the number of paths from any v_j to t that v_i has an edge to:

$$np[i] = \sum_{(v_i, v_j)} np[j]$$

Runtime is $O(n + m)$ because: (i) line 4 loops through each node, so incurs $O(n)$ cost; and (ii) across lines 4-6, for each vertex v_i we do $out - deg(v_i)$ many summations, which is at most $O(m)$.

2 SCCs

Reachability: Let $G = (V, E)$ be a directed graph in which each vertex $u \in V$ has a unique ID (assigned arbitrarily). For each vertex $u \in V$, let $R(u) = \{v \in V : u \rightsquigarrow v\}$ be the set of vertices that u has a path to, i.e., that are reachable from u. Define $min(u)$ to be the ID of the minimum-ID vertex in that u can reach. Give an $O(n+m)$ time algorithm that computes $min(u)$ for all vertices $u \in V$.

Hint: Use a linear-time SCC decomposition algorithm (even if you have not yet covered this algorithm in your sessions, you can assume its existence and use it as a subroutine).

2.1 Solution

Algorithm 2: MinID($G(V, E)$)

Find the SCCs of G using Kosaraju's algorithm;
Let $SCC_1, ..., SCC_k$ be k sets storing the vertices in each SCCs.
$G^{SCC}(V^{SCC}, E^{SCC})$: Construct the graph of SCCs; i.e., each SCC is a node and we keep the edges between SCCs.
Topologically sort G^{SCC}
W.l.o.g. let $scc_1, ..., scc_k$ be the vertices in V^{SCC} in topologically sorted order
Let minSCC[k] be an array of size k
Initialize minSCC[i] = minimum ID in SCC_i
for $i = scc_k ... scc_1$ **do**
    for $(scc_i,\ scc_j) \in E^{SCC}$ **do**
        minSCC[i] = min(minSCC[i], minSCC[j]);
// Construct the output min array: an array of n integers
min[n];
for $u \in V$ **do**
    min[u] = minSCC(j) //where $u \in SCC_j$
return min;

We decompose into SCCs using Kosaraju's algorithm. Notice that all vertices that are in the same SCC will get the same min value because they reach exactly the same set of vertices in G. So we can compute a min value for each SCC in a $minSCC$ array and then $min(u)$ of a $u \in SCC_j$ is simply the $minSCC(j)$.

To compute $minSCC$ we first construct the "SCC graph of G", which is a DAG. Call this graph $G^{SCC}(V^{SCC}, E^{SCC})$. We topologically sort G^{SCC}. Then we initialize the $minSCC$ of each SCC_i to the minimum ID across the vertices in SCC_i. Notice that for sink SCC's this correctly assigns their minSCC values (since sink SCC's cannot reach other SCCs). Then we have a DP algorithm that is very similar to the DP algorithm in question 1 for computing the number of (s, t) paths: in reverse topological order, we set the $minSCC(i)$ to the be minimum of its current value and $minSCC(j)$ for any outgoing edge $(SCC_i, SCC_j) \in E^{SCC}$. This recurrence is true because min ID that an SCC_i can reach is either the minimum ID in SCC_i or the minimum ID of some of SCC_j that SCC_i has an edge to (so can reach to).

The runtime of the entire algorithm is $O(n+m)$ because: (i) running Kosaraju's algorithm takes linear time; (ii) constructing G^{SCC} takes linear time; (iii) topologically sorting and and the DP algorithm take linear time in the size of G^{SCC} which will be smaller than G because we collapse all vertices SCCs into a single node and remove all the edges within each SCC.