CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 16

October 31, 2019

Instructor: Sepehr Assadi

1 Topological Sorting

Recall that the topological sorting problem is as follows:

Problem 1 (Topological Sorting). Given a directed graph $G(V, E)$ either output a topological ordering of G if G is a DAG or output that G is *not* a DAG. A topological ordering is an ordering of vertices $v_1 \prec v_2 \prec \ldots \prec v_n$ such that for any edge (u, v), we have $u \prec v$ (note that $u \prec v$ does not mean that u, v necessarily appear consecutively in the order, only that u appears somewhere before v).

We proposed the following simple algorithm for topological sorting: we pick a vertex with in-degree equal to 0 and place it in the beginning of the ordering; remove all its outgoing edges from the graph, and repeat. Also if at any point we can no longer find a vertex of in-degree 0, we return that the graph is not a DAG.

We now go over how to implement this algorithm efficiently and prove its correctness.

Algorithm:

1. Let $D[1:n]$ be an array initialized to 0 and O be an empty linked-list for the output ordering.
2. Go over all vertices v and for any $u \in N(v)$, increase $D[u]$ by one. (At the end of this step, $D[v]$ denotes the *in-degree* of v for all $v \in V$).
3. Insert every vertex v with $D[v] = 0$ into a queue Q.
4. While Q is not empty:
   (a) Let v be the first vertex of Q and dequeue (remove) this vertex from Q.
   (b) Add v to the end of the linked-list O.
   (c) For $u \in N(v)$:
      i. Reduce $D[u]$ by one. If $D[u] = 0$ insert u to Q.
5. If $|O| < n$, output G is not a DAG; otherwise output O as a topological ordering of G.

Proof of Correctness: Suppose first that $|O| = n$: we prove O is a topological ordering of G in this case:

(1) Consider any edge (u, v) in G. For the vertex v to be added to O, it should first join the queue Q; for this to happen, we should have $D[v] = 0$ at some point.

(2) For vertex v to have $D[v] = 0$, we should have 'removed' edge (u, v) first: in other words, as long as u is not dequeued, and so we reduce $D[v]$ by one, $D[v] > 0$ and so v will not join Q. (Here, we are also using the fact that at the beginning $D[v]$ was equal to in-degree of v and we only reduce $D[v]$ by one whenever we add one of the in-neighbors of v to Q).

(3) This implies that u should have joined O before v, thus appearing before v in the ordering. This implies that if $|O| = n$ (and hence all vertices appear in the ordering), O would be a topological ordering of G.

Now suppose that $|O| < n$: we prove that G is not a DAG:

(1) Consider the time step where Q is empty but not all vertices have been output to O. This means that not all vertices have joined Q.

(2) Let S be the set of all vertices that never joined Q. By definition of the algorithm, we know that in-degree of each of these vertices *from other vertices in* S is more than 0 (if it was 0, then that vertex v should have $D[v] = 0$ and so joined Q because we removed the in-degree of all vertices out of S).

(3) This means that there is a directed cycle in G between some of the vertices of S: simply start some vertex in S and go to its in-coming edge, and continue like this; since every vertex has in-degree more than 0, we should end up in a loop eventually (this process cannot go forever as S has finite size). This implies G is not a DAG.

Runtime analysis: The runtime of this algorithm is $O(n+m)$: The first and third line before the for-loop takes $O(n)$ and the second line takes $O(n+m)$ (we go over each edge only once). Also, the while-loop takes at most $O(n+m)$ time because we consider each vertex at most once in the while-loop and when considering each vertex, we go over its out-degree; since sum of the out-degrees is $O(m)$, we obtain the $O(n+m)$ bound.

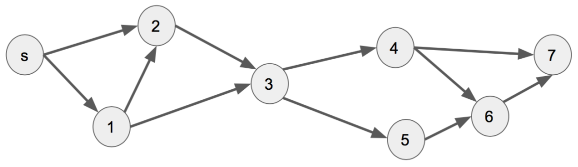
2 Dynamic Programming on DAGs: Longest Path Problem

Because DAGs do not have any cycle, we can use dynamic programming as an effective technique for solving problems over them (do you see why lack of cycles is very helpful? If not, we will get to this point later in this lecture itself).

Perhaps one of the most important problems we can solve efficiently over DAGs is to find a *longest* path from a source vertex to other vertices[1]. We now study this problem.

Problem 2 (Longest Path in DAGs). Given a directed acyclic graph $G(V, E)$ and a source vertex s, output the length of a *longest* path from s to every other vertex ('undefined' if s cannot reach that vertex).

Example: The correct answer for the longest path problem in the following DAG is $[0, 1, 2, 3, 4, 4, 5, 6]$ (compare this with the *shortest* path distances which are $[0, 1, 1, 2, 3, 3, 4, 4]$).



Algorithm: Similar to any other dynamic programming algorithm, we start with a recursive formula:

Specification: For every vertex $v \in V$:

- *Long*(v): denote the length of the longest path starting from s and ending in v. By convention, we define *Long*$(s) = 0$ and *Long*$(v) =$ 'undefined' if v is not reachable from s.

[1]We will (hopefully) see by the end of the semester that this problem does not seem to have an efficient algorithm on general graphs that have cycles, quite opposite of the *shortest* path problem.

The output answer is the array that contains $\mathit{Long}(v)$ for every $v \in V$.

Recursive formula: For every vertex $v \in V$:

$$\mathit{Long}(v) = \begin{cases} 0 & \text{if } v = s \\ \text{`undefined'} & \text{if } v \text{ is any source other than } s \text{ .} \\ \max_{u \in V} \{\mathit{Long}(u) + 1 \mid u \text{ is an in-neighbor of } v\} & \text{otherwise} \end{cases}$$

(the last condition can alternatively be interpreted as taking a maximum of $\mathit{Long}(u) + 1$ for every u that have an edge to v).

In this formula, we use the convention that if $\mathit{Long}(u) =$'undefined', we will consider $\mathit{Long}(u)+1$ in the recursive formula as 'undefined', and when we take the maximum in that line, 'undefined' will always be considered than any number returned (if $\mathit{Long}(u) =$ 'undefined' for all in-neighbors u of v, then $\mathit{Long}(v) =$'undefined' as well).

Let us prove the correctness of this recursive formula.

- The base case for $\mathit{Long}(s) = 0$ is true by definition of the problem. Similarly, if v is a source, no other vertex can reach v and hence distance of s to v is 'undefined'.
- Consider any vertex $v \in V$ which not a source. The longest path starting from s to v should go through one of the in-neighbors of v first, say w, and then take the edge (w, v) to reach v. Moreover, the path from s to w should be the longest s-w path (otherwise by switching it to a longer path we increase the length of the s-v path also). Hence, length of that path should be $\mathit{Long}(w)$. By taking maximum over all in-neighbors of v, we will set $\mathit{Long}(v) = \mathit{Long}(w) + 1$ which is correct.

Important Remark: Let us pause for a second and examine something: where did we use the fact that G is a DAG at all in the above argument? Did we just designed a formula that work on every graph G?

This is a subtle point and is closely related to the notion of a *recursive* formula: we can only consider $\mathit{Long}(v)$ a truly recursive formula if when computing value of $\mathit{Long}(v)$ from $\mathit{Long}(u)$ and so on and so forth, we never *loop* back to computing $\mathit{Long}(v)$ again; in other words, there should not exists any chain of recursive calls where $\mathit{Long}(v)$ is updated from itself! during this chain. So, for us to be able to call this formula truly recursive, we should ensure that there is a way to compute the values of $\mathit{Long}(v)$ in an order so that whenever we want to compute v, all in-neighbors u of v have already been assigned a correct value $\mathit{Long}(u)$.

Now does this ordering sound familiar at this point? Such an ordering is simply a topological ordering of the vertices! So the only way we can indeed prove the correctness of this formula is to use a topological ordering of vertices. But this can only happen if our underlying graph is a DAG.

Runtime: Runtime of this algorithm would then be $O(n + m)$ because there are n subproblems and each subproblem v takes time proportional to in-degree of v; since the total sum of in-degrees is equal to m, the total runtime is $O(n + m)$.

Graph Reductions for Dynamic Programming

We can model many dynamic programming problems (including many of the ones we studied) as a longest path problem over a DAG using proper reductions. Consider the following example.

Longest Increasing Subsequence (LIS) : Recall the definition of the LIS problem:

Problem 3. The longest increasing subsequence problem (LIS for short) is defined as follows:

- **Input:** An input array $A[1 : n]$ of distinct integers of length n.

- **Output:** The length of the longest *increasing* subsequence of array A, i.e., the length of longest B such that (1) B is a subsequence of A, and (2) B is increasing, i.e., $B[1] < B[2] < B[3] < \cdots$.

We show how to solve this problem by reducing it to the longest path problem over a DAG.

Reduction: We create a new graph $G(V, E)$ with vertices $V = \{1, \ldots, n\} \cup \{s\}$ and connect vertex i to j if $i < j$ and $A[i] < A[j]$ to get the set of edges. We also have a vertex s which is connected to every other vertex of G.

We can run *any* algorithm (including the one we just saw) to find the longest path from s to all other vertices and return the maximum distance from s to any vertex (so length of the longest path starting from s) as the answer.

Proof of Correctness: We first have to argue that G is a DAG. It is easy since we can pick the ordering $s \prec 1 \prec 2 \ldots \prec n$ as a topological ordering of G: the condition of the first part of the edges ensures that this is indeed a correct topological ordering and hence G should be a DAG.

We then need to argue that the answer to both problems is the same. We do so by showing that there is a *one-to-one correspondence* between paths leaving s in G and increasing subsequences in A. Consider any path $s, v_1, \ldots, v_k$; by definition of edges we know $A[v_1] < A[v_2] < \ldots < A[v_k]$ and $v_1 < \ldots < v_k$, so this is indeed an increasing subsequence in A. The same exact proof also shows that any increasing subsequence maps to a unique path in G. Hence, the longest increasing subsequence of A corresponds to the longest path starting from s and vice versa, proving the correctness.

Runtime Analysis: Graph G has $n + 1$ vertices and $m = O(n^2)$ edges. Since we can find the longest path in G in $O(n + m) = O(n^2)$ time, we can solve LIS in $O(n^2)$ time as well.

Another example of a problem which can be solved by reducing it to the longest path problem in DAGs is Problem 2 of Homework 3, namely, the box stacking problem (you are encouraged to do this on your own as a practice).

3 Minimum Spanning Trees

We switch gear now and consider *minimum spanning trees* of *weighted* graphs. Let $G(V, E)$ be an undirected *connected* graph with weights $w_{e_1}, \ldots, w_{e_m}$ on edges $e_1, \ldots, e_m$ of E (namely, each edge e has a weight w_e). Recall that a spanning tree of a connected subgraph G is any subgraph of G which is a tree – a tree itself is a connected subgraph which has no cycle[2]. We have,

Problem 4 (**Minimum Spanning Tree (MST)**)**.** The minimum spanning tree problem (MST for short) is defined as follows:

- **Input:** An undirected connected graph $G(V, E)$ with weights over the edges.
- **Output:** A spanning tree of G with minimum total weights of edges in the tree.

We study algorithms for this problem in the next lecture.

[2]Any connected graph has at least one spanning tree; simply greedily remove any edge e from G which is part of some cycle: this 'breaks' the cycle without making G disconnected; continue until G has no cycle and hence is a tree.