Graphs : DFS Applications : Topological Sort[1]

1 Applications of DFS

We already saw two applications of DFS in the last lecture: the REACHABLE? problem and the CYCLE? problem. In this lecture and the next, we see two more applications of DFS on directed graphs. The first application actually gives a way to "order" all nodes in a *directed acyclic graph* (DAG). This is called the *topological order*. This itself has many applications, which your problem sets explore.

1.1 Topological Ordering of DAGs

Throughout this subsection, G is a directed acyclic graph (DAG). Recall from the previous lecture, this means that if we run DFS on G, there are no back edges. A topological ordering of (the vertices of) a DAG is an ordering σ of the vertices such that for any $i < j$, there is **no** edge from $\sigma[j]$ to $\sigma[i]$. That is, if we write down the vertices from left to right in the σ order, then *all* edges go from left to right. If one thinks of an edge (u, v) as v being "bigger" than u, then the topological ordering is a linearization of the graph according to this (partial) order. Of course, not every pair of vertices may be comparable.

Remark: *Note that the first vertex v in the topological order must have $\deg^-(v) = 0$. There is no vertex to its left to "send" an edge to it. Such vertices are called **sources**. Similarly, the last vertex v in the topological order must have $\deg^+(v) = 0$. There is no vertex to its right to which it can "send" an edge. Such vertices are called **sinks**. Furthermore,* any *source can be the first vertex of a topological order, and any sink can be the last vertex in the topological order.*

TOPOLOGICAL ORDERING
Input: Directed Acyclic Graph G.
Output: A topological ordering of G.

Does a topological order always exist of a DAG? Motivated by the previous remark, let us first ask ourselves, does a DAG always have a source vertex? If you think for a minute you will see that the answer is yes: if a vertex has an edge coming into it from another vertex, and that vertex has an edge coming into it from yet another vertex, and so on, either we will reach a source, or we will reach a cycle. Since the latter is not possible in a DAG, we must have a source vertex. This way we can find the first vertex of a topological order. What about the rest? Well, delete the source vertex (putting it up front); the resulting graph remains a DAG. Rinse and repeat. This way you will get a topological order; in particular, one exists. Indeed, this actually gives an algorithm to find a topological order and if you are careful you can make that algorithm run in $O(n + m)$ time. The following lemma shows how a single DFS can return a topological order almost immediately.

Lemma 1. Consider running DFS in any arbitrary order on the DAG G. Let σ be the ordering of the vertices in *decreasing* order of $\mathsf{last}[v]$. Then, σ is a topological order.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

Proof. To show σ is a topological order, we need to show there is no edge going from right to left. To this end, pick two vertices x and y such that $\sigma[x] < \sigma[y]$. We need to show (y, x) is *not* an edge. Suppose, for contradiction's sake, it is. Now, $\sigma[x] < \sigma[y]$ implies, by our definition, $\mathsf{last}[y] < \mathsf{last}[x]$. The *edge property* now implies that $\mathsf{first}[x] < \mathsf{first}[y]$. Why? If not, that is, if $\mathsf{first}[y] < \mathsf{first}[x]$, then since (y, x) is an edge the edge property would imply $\mathsf{last}[x] < \mathsf{last}[y]$. But the reverse is true. Therefore, $\mathsf{first}[x] < \mathsf{first}[y]$. In sum, we have $\mathsf{first}[x] < \mathsf{first}[y] < \mathsf{last}[y] < \mathsf{last}[x]$. The Nested Interval Property now tells us y must be a descendant of x, that is, (y, x) is a back-edge. This contradicts that G is acyclic. □

1: **procedure** TOPORDER(G): ▷ *G is assumed to be a DAG*
2: Run DFS(G) for any arbitrary order of the vertices.
3: Return the decreasing order of $\mathsf{last}[v]$'s. ▷ *Can use Count-Sort, or can return them as* last*'s are being assigned.*

Theorem 1. TOPORDER finds the topological order of any DAG G in $O(n + m)$ time.

The topological order is super powerful as it allows us to run dynamic programming to solve many problems on DAGs (which may be hard to solve on general directed graphs). The reason, as alluded to in an earlier lecture on DPs, is that the topological ordering is an "ordering" (albeit partial) and we can think of "last" element in our ordering and ask questions like "if they are in our solution". Let us give an example by showing how to find ***longest*** paths in DAGs in $O(n + m)$ time. To contrast this, no one knows *any* polynomial time algorithm (let alone linear time) for finding the longest path in a general directed graph[2].

LONGEST PATH
Input: DAG G, costs $c(e)$ on every edge which can be arbitrary real numbers, source vertex s.
Output: Find longest path from s to every vertex $v \in V$.

We start with a topological order σ of G. Let $s = \sigma[i]$, that is, in the ith position in σ. Now consider any vertex $v = \sigma[j]$. If $j < i$, there can be *no* path from s to v, and so we answer $\perp$. Otherwise, the longest path p from s to v must pass through a penultimate vertex $\sigma[k]$ with $k < j$ where $(\sigma[k], \sigma[j])$ is an edge. Furthermore, the path from s to $\sigma[k]$ must be the longest path to $\sigma[k]$ as well (do you see why? Proof coming shortly). Therefore, the remaining edges of p (without the $(\sigma[k], \sigma[j])$ edge) can be found in the smaller problem where we consider the graph only up to $\sigma[k]$. This is our recursive substructure, and leads to the following dynamic programming algorithm which we write in our usual six-step way.

a. ***Definition.*** Let $\mathcal{P}_j$ denote all paths from $\sigma[i]$ to $\sigma[j]$. Note this could be empty.

$$\mathsf{longest}[j] := \max_{p \in \mathcal{P}_j} c(p)$$

where $c(p)$ is the *sum* of the costs of edges in p.

b. ***Base Case.*** For all $j < i$, $\mathsf{longest}[j] = \perp$. It will be useful to think of $\perp$ as $-\infty$; the longest path is of length $-\infty$ means there is no path.

$\mathsf{longest}[i] = 0$, the longest path from s to itself is of length 0, ***because there are no cycles in*** G***.***

[2]For those who know the jargon, the longest path problem on general directed graphs is NP-hard; more on NP-hardness (probably) later in the course

c. ***Recurrence.*** For $j > i$,

$$\mathsf{longest}[j] = \max_{\ell < j : (\sigma_\ell, \sigma_j) \in E \text{ and } \mathsf{longest}[\ell] \neq \perp} \left(\mathsf{longest}[\ell] + c(\sigma_\ell, \sigma_j) \right)$$

If there is *no* such edge to take max over, then $\mathsf{longest}[j] = \perp$.

d. ***Proof.***

($\leq$). If there is no path from σ_i to σ_j, that is if $\mathcal{P}_j = \emptyset$, then $\mathsf{longest}[j] = \perp$, and it doesn't matter what the RHS is (here, the mnemonic of $\perp = -\infty$ is useful). So, suppose $\mathcal{P}_j$ is not empty, and fix a path $p \in \mathcal{P}_j$ with $c(p) = \mathsf{longest}[j]$. Let this path $p = (\sigma_i = x_0, x_1, x_2, \ldots, x_t = \sigma_j)$. Since $j > i$, the vertex x_1 is well-defined (it could be σ_j).

Consider the vertex x_{t-1}, the penultimate vertex in this path. Let ℓ be such that $x_{t-1} = \sigma[\ell]$; by the definition of topological order, $\ell < j$ (note, ℓ could be i). Also, $p' := (\sigma_i = x_0, \ldots, x_{t-1} = \sigma_\ell)$ is a path from σ_i to σ_ℓ. Therefore, $c(p') \leq \mathsf{longest}[\ell]$. And, $c(p) = c(p') + c(\sigma_\ell, \sigma_j)$. Putting together, we get

$$\mathsf{longest}[j] = c(p) = c(p') + c(\sigma_\ell, \sigma_j) \leq \mathsf{longest}[\ell] + c(\sigma_\ell, \sigma_j)$$

and thus, the LHS $\leq$ the RHS which maximizes over all such ℓ's.

($\geq$). Fix ***arbitrarily*** any $\ell < j$ such that $\mathsf{longest}[\ell] \neq \perp$ and (σ_ℓ, σ_j) is an edge. Since $\mathsf{longest}[\ell] \neq \perp$, there is a path $p \in \mathcal{P}_\ell$ with $c(p) = \mathsf{longest}[\ell]$. Consider the path $p' = p \circ (\sigma_\ell, \sigma_j)$. This is a path from σ_i to σ_j, that is, $p' \in \mathcal{P}_j$. So,

$$\mathsf{longest}[j] \geq c(p') = c(p) + c(\sigma_\ell, \sigma_j) = \mathsf{longest}[\ell] + c(\sigma_\ell, \sigma_j)$$

Since ℓ was picked arbitrarily among the set the RHS is maximizing over, we get that the LHS is at least the RHS.

e. ***Pseudocode.***

```
1:  procedure LongestPath(G, s, c): ▷ G is a DAG.
2:      Run Topological Order on G to obtain σ.
3:      ▷ All edges of G are of the form (σ_i, σ_j) with i < j.
4:      Let i be such that s = σ[i].
5:      Initialize longest[v] = ⊥ and parent[v] = ⊥. ▷ parent is used for recovery: see later
6:      Set longest[σ[i]] = 0 and parent[σ[i]] = ⊥.
7:      Set longest[σ[j]] = parent[σ[j]] = ⊥ for j < i.
8:      for j = i + 1 to n do:
9:          Set longest[σ[j]] = max_{ℓ<j:(σ_ℓ,σ_j)∈E and longest[σ[ℓ]]≠⊥} (longest[σ[ℓ]] + c(σ_ℓ, σ_j)).
10:         ▷ If there is nothing to "max" over, then longest[σ[j]] remains unchanged at ⊥.
11:         If longest[σ[j]] ≠ ⊥, set parent[σ[j]] = σ[ℓ], which maximizes the above term.
12:         ▷ Observe 1: longest[σ[j]] = longest[parent[σ[j]]] + c(parent[σ[j]], σ[j]).
13:         ▷ Observe 2: parent[v] if not ⊥ comes before v in the topological order.
14:         ▷ Observe 3: For all v ≠ s : longest[v] ≠ ⊥ implies parent[v] ≠ ⊥.
```

The above pseudocode returns the *length* of the longest paths from s to every vertex. How about the paths itself? Well, the parent data-structure allows one to recover paths on query. So suppose we want the longest path to a particular vertex v (and not all of them), then we would find $(v, \mathsf{parent}[v], \mathsf{parent}[\mathsf{parent}[v]], \ldots)$ till we reach s, and then reverse it. This is just a "slick" way of doing the while-loop. I provide this below, but in the future I will use the above notation.

1: **procedure** RECOVERLONGESTPATH(G, s, c; longest, parent; v): ▷ *Find the longest path from s to particular vertex v.*
2: ▷ *Assumes you have obtained* longest$[v]$ *by running* LONGESTPATH
3: Let j be such that $\sigma[j] = v$.
4: **if** longest$[\sigma[j]] = \perp$ **then**:
5: **return** No path from s to v.
6: $w \to \sigma[j]$; $p = [\sigma[j]]$.
7: ▷ **Invariant:** $c(p) + \mathsf{longest}[w] = \mathsf{longest}[\sigma[j]]$
8: **while** parent$[w] \neq \perp$ **do**:
9: Append parent$[w]$ to the front of p. ▷ *This increases the cost of p by $c(\mathsf{parent}[w], w)$.*
10: ▷ *The invariant is maintained by the observation in Line 12*
11: $w \leftarrow \mathsf{parent}[w]$
12: ▷ *At this point, we assert $w = s$. This is because s is the* **only** *vertex whose* longest$[w]$ *is not $\perp$ but* parent$[w]$ *is $\perp$. Since* longest$[w]$ *was defined by* parent$[w] = \perp$*, we get this.*
13: ▷ *Why does the while loop end? Because* parent$[w]$*, if not $\perp$, comes* before *w in the Topological Order.*

f. ***Running Time.*** Line 9 in LONGESTPATH takes time $O(\deg^-(\sigma_j))$, the in-degree of σ_j. Every other step in the jth for loop takes $O(1)$ time. Thus, the total time taken by the for loop is $\sum_{j=i+1}^{n} \left(O(1) + O(\deg^-(\sigma_j)\right) = O(n+m)$

Theorem 2. LONGESTPATH can be used to find the longest path in a DAG in $O(n+m)$ time to any vertex v from s.