Topological Sort on a Directed Acyclic Graph

Module: Graphs

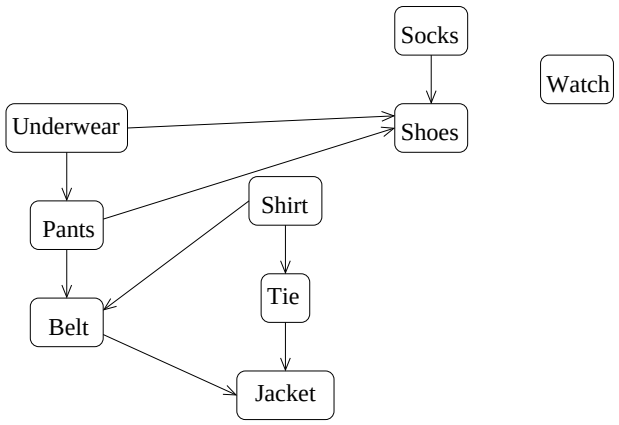
1 Overview

Today we talk about *topological order* and *topological sort*, which is defined on *directed* graphs and corresponds to ordering the vertices in such a way such that the edges go "forward". If we think of directed edges as representing precedence, i.e. the start point must come before the endpoint, then sorting the vertices in topological order assures that all the edges are "fullfilled". Topological order only exists on *directed acyclic graphs*, or DAGs.

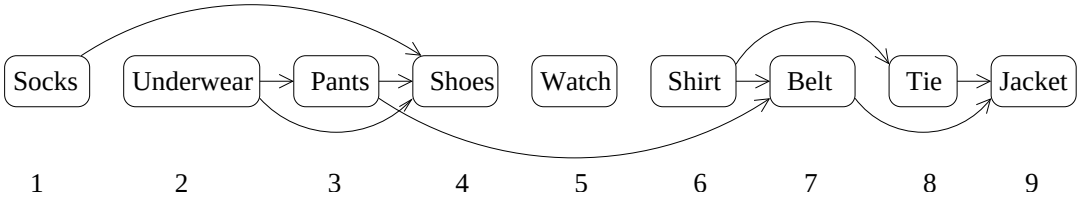
Topological order is a very useful concept on DAGs. Many problems admit simpler and faster solutions on this class of graphs, and the solutions involve in one way or another the fact that a DAG can be topologically sorted. For example, we'll see that computing shortest paths on DAGs can be done in linear time; also the problem of computing longest paths, which is known to be NP-complete on general graphs (think: exponential), can be solved in linear time on DAGs.

2 Topological Order: The problem

- Let $G = (V, E)$ be a directed graph
- We think of a directed edge (u, v) as meaning that u has to come before v—thus an edge defines a precedence relation.
- Example: The directed graph below represents constraints of getting dressed. Here an edge/arrow implies "must come before"



- A topological sorting of a G is a linear ordering of vertices V such that for any edge $(u, v) \in E$, vertex u appears before v in this ordering.
- Put differently, a topological order is an order of the vertices that "satisfies" all the edges.
- For e.g., on the graph above, a topological order corresponds to an order in which we can get dressed. One possible topological order is the following:



- A graph may have many different topological orders; we want to compute one of them

3 Does a tpopological order always exist?

- If the graph has a cycle, a topological order cannot exist. Imagine the simplest cycle, consisting of two edges: (a, b) and (b, a). A topological order, if it existed, would have a come before b and a come before a. This is not possible.
- Now consider we have a graph without cycles; this is usually referred to as a DAG (directed acyclic graph). Does any DAG have a topological order?
- The answer is YES. We'll prove this below indirectly by showing that the algorithm always gives a valid ordering when run on any DAG.

4 Algorithms for topologically sorting a DAG

We'll see two algorithm for computing a topological order, both of which run in linear time in the size of the graph, or $O(V + E)$.

4.1 Topologically sorting a DAG via repeated sink-elimination

Idea: Intuitively, in order to sort topologically, we want to start with a vertex that has no incoming edges. This suggests the following algorithm:

TopologicalSort(G is a DAG)

- $L = [\ ]$ //L is a list that will hold the vertices of G in topological order
- while there are remaining edges
  - find a vertex with no incoming edges, append it to L and delete all its outgoing edges from G

Why does this work? Clearly, a vertex without incoming edges can be first on topological order. But, is such a vertex guaranteed to exist? What about inside the loop? As we delete vertices from the graph and repeat, are we always able to find a vertex with no incoming edges, or is it possible that we get stuck in the loop?

Claim: A directed acyclic graph always contains a vertex of indegree zero.

Proof: We will do a proof by contradiction. Assume, by contradiction, that *every* vertex in G has at least one incoming edge. Consider an arbitrary vertex v; it must have an incoming edge, call it (v_1, v); v_1 must have an incoming edge, (v_2, v_1); v_2 must have an incoming edge, call it (v_3, v_2); and so on. Since every vertex has an incoming edges, we can do this forever. But, since the graph is finite, this means at some point we must hit the same vertex again. This means we found a path through edges form a vertex back to itself $\Longrightarrow$ cycle. But, our graph is a DAG, so we got a contradiction. Therefore there must exist at least one vertex with no incoming edges.

The claim tells us that initially the graph is guaranteed to have a vertex with indegree zero. We add this vertex to the order, and remove all its outgoing edges from G. The remaining graph is still a DAG, so by the claim again, it is guaranteed to have a vertex of indegree zero. The next iteration of the loop will find this vertex, add it to the order, delete its edges. And so on. Using the claim we can conclude that the algorithm above will run until G has no more edges, and then will stop. So given any DAG, the algorithm computes a topological order for it. It follows that any DAG has a topological order.

Analysis: A straightforward implementation of the algorithm above spends $O(V)$ to find a vertex with no incoming edges; this results in $O(V^2)$ total time.

The algorithm can be implemented to run in $O(V+E)$ time with a small modification. Instead of computing indegrees of vertices every time, looking for a vertex with indegree 0, we'll compute indegree of vertices once at the beginning, and update them along the way as we delete edges. You can find the algorithm in the slides:

- Create a list L with all vertices u such that in-degree[u] =0
- Repeat |V| times:
  - delete a vertex u from L, and output it as the next vertex in top order
  - For each edge (u,v):
    - indegree[v]- -
    - if indegree[v]==0: add v to list L

Since G is a DAG and by the lemma it has at least one vertex of indegree 0, it follows that L contains at least one vertex initially. As we delete edges from G and update the indegrees, we catch any vertex whose indegree drops to 0. L can never run empty and the algorithm finishes.

4.2 Topologically sorting a DAG via DFS

TopologicalSort(graph G):

1. Call DFS(G) to compute start and finish times for all vertices in G.
2. Return the list of vertices in reverse order of their finish times (That is, the vertex finished last will be first in the topological order, and so on).

Why does this work? Suppose DFS(G) is run on DAG G. We would like to show that the vertex finished last by DFS(G) cannot have any incoming edges. We can prove something more general:

Claim: Suppose DFS(G) is run on DAG G. For any edge (u, v), the finish time of u is larger than the finish time of v; in other words, u is finished after v.

Proof: Consider running DFS(G), and exploring an arbitrary edge (u, v). When this edge is explored, there are three options for the status of v:

- v is UNVISITED: If v is UNVISITED, DFS(G) will continue to v, and thus v becomes a descendant of u in the DFS tree, and it is finished before u is finished. Therefore u finishes after v.
- v is IN-PROGRESS: If this were true, v would be an ancestor of u and (u, v) would reach back to v, meaning a cycle would exist. Impossible. So v cannot be IN-PROGRESS.
- v is ALL-DONE: If v is ALL-DONE, it's already been finished and since u is now IN-PROGRESS, it will be finished after v.

Thus for any edge (u, v) we have that $finish[u] > finish[v]$.

If we sort the vertices decreasingly by finish time, for any edge (u, v), we know that $finish[u] > finish[v]$ by the lemma, so u will ahead of v in this order. So it is a valid topological order.

Analysis: Computing topological order via DFS runs in $O(V + E)$ time.