CSC263 Fall 2017 - Tutorial Week 9

Roleen Nunes
Department of Computer Science
University of Toronto

Depth-first Search & Topological Sort

Depth-first Search

- Searches "deeper" in the graph whenever possible
- Each vertex of the graph is initially white, is grayed when discovered in the search, and is blackened when it is finished
- DFS search also timestamps each vertex with discovered time and finished time

Algorithm

```
DFS(G)
1. for each vertex u ∈ G.V
2.     u.color = WHITE
3.     u.π = NIL
4. time = 0
5. for each vertex u ∈ G.V
6.     if u.color == WHITE
7.         DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1.  time = time + 1    // white vertex u just discovered
2.  u.d = time
3.  u.color = GRAY
4.  for each v ∈ G.Adj[u]        // explore edge (u, v)
5.      if v.color == WHITE
6.          v.π = u
7.          DFS-VISIT(G, v)
8.  u.color = BLACK        // blacken u; it is finished
9.  time = time + 1
10. u.f = time
```

Example

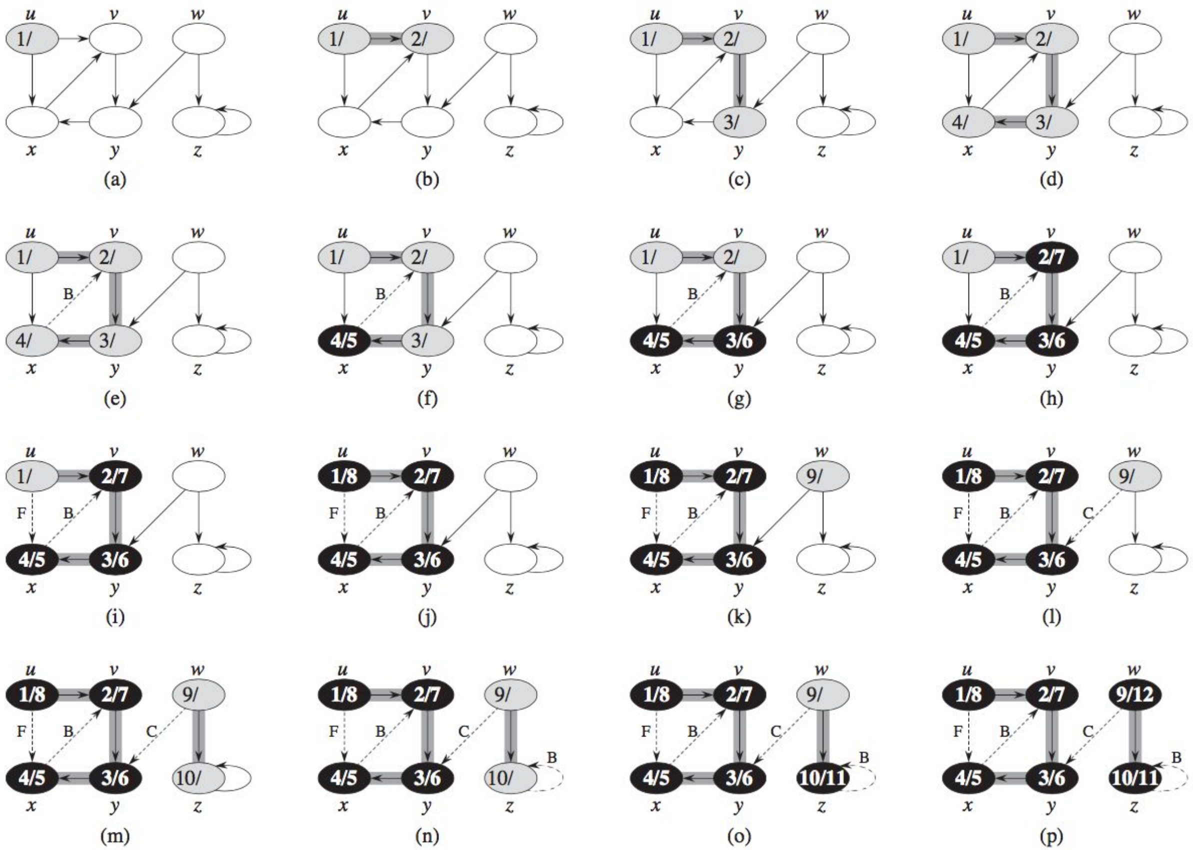

Image Source: CLRS 22.3 (Page 605)

Types of edges

1. Tree edges:

Edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v)

2. Back edges:

Edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree

3. Forward edges:

Nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree

4. Cross edges:

All other edges

CLRS 22.3-5

Show that edge (u, v) is

a. a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$,

b. a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, and

c. a cross edge if and only if $v.d < v.f < u.d < u.f$

CLRS 22.3-5 Solution

a. Since we have that $u.d < v.d$, we know that we have first explored u before v. This rules out back edges and rules out the possibility that v is on a tree that has been explored before exploring u's tree. Also, since we return from v before returning from u, we know that it can't be on a tree that was explored after exploring u. So, This rules out it being a cross edge. Leaving us with the only possibilities of being a tree edge or forward edge.

To show the other direction, suppose that (u, v) is a tree or forward edge. In that case, since v occurs further down the tree from u, we know that we have to explored u before v, this means that $u.d < v.d$. Also, since we have to of finished v before coming back up the tree, we have that $v.f < u.f$. The last inequality to show is that $v.d < v.f$ which is trivial.

CLRS 22.3-5 Solution continued

b. By similar reasoning to part a, we have that we must have v being an ancestor of u on the DFS tree. This means that the only type of edge that could go from u to v is a back edge.

To show the other direction, suppose that (u, v) is a back edge. This means that we have that v is above u on the DFS tree. This is the same as the second direction of part a where the roles of u and v are reversed. This means that the inequalities follow for the same reasons.

CLRS 22.3-5 Solution continued

c. Since we have that $v.f < u.d$, we know that either v is a descendant of u or it comes on some branch that is explored before u. Similarly, since $v.d < u.d$, we either have that u is a descendant of v or it comes on some branch that gets explored before u. Putting these together, we see that it isn't possible for both to be descendants of each other. So, we must have that v comes on a branch before u, So, we have that u is a cross edge.

To See the other direction, suppose that (u, v) is a cross edge. This means that we have explored v at some point before exploring u, otherwise, we would have taken the edge from u to v when exploring u, which would make the edge either a forward edge or a tree edge. Since we explored v first, and the edge is not a back edge, we must of finished exploring v before starting u, so we have the desired inequalities.

CLRS 22.3-8

Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, and if $u.d < v.d$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

CLRS 22.3-8 Solution

Consider a graph with 3 vertices u, v, and w, and with edges $(w, u), (u, w)$, and (w, v). Suppose that DFS first explores w, and that w's adjacency list has u before v. We next discover u. The only adjacent vertex is w, but w is already grey, so u finishes. Since v is not yet a descendant of u and u is finished, v can never be a descendant of u.

Topological Sort

- A topological sort of a of a directed acyclic graph (DAG) is a linear ordering of all its vertices such that if G contains edge (u, v), then u appears before v in the ordering
- We can use DFS to perform topological sort of a DAG

Example

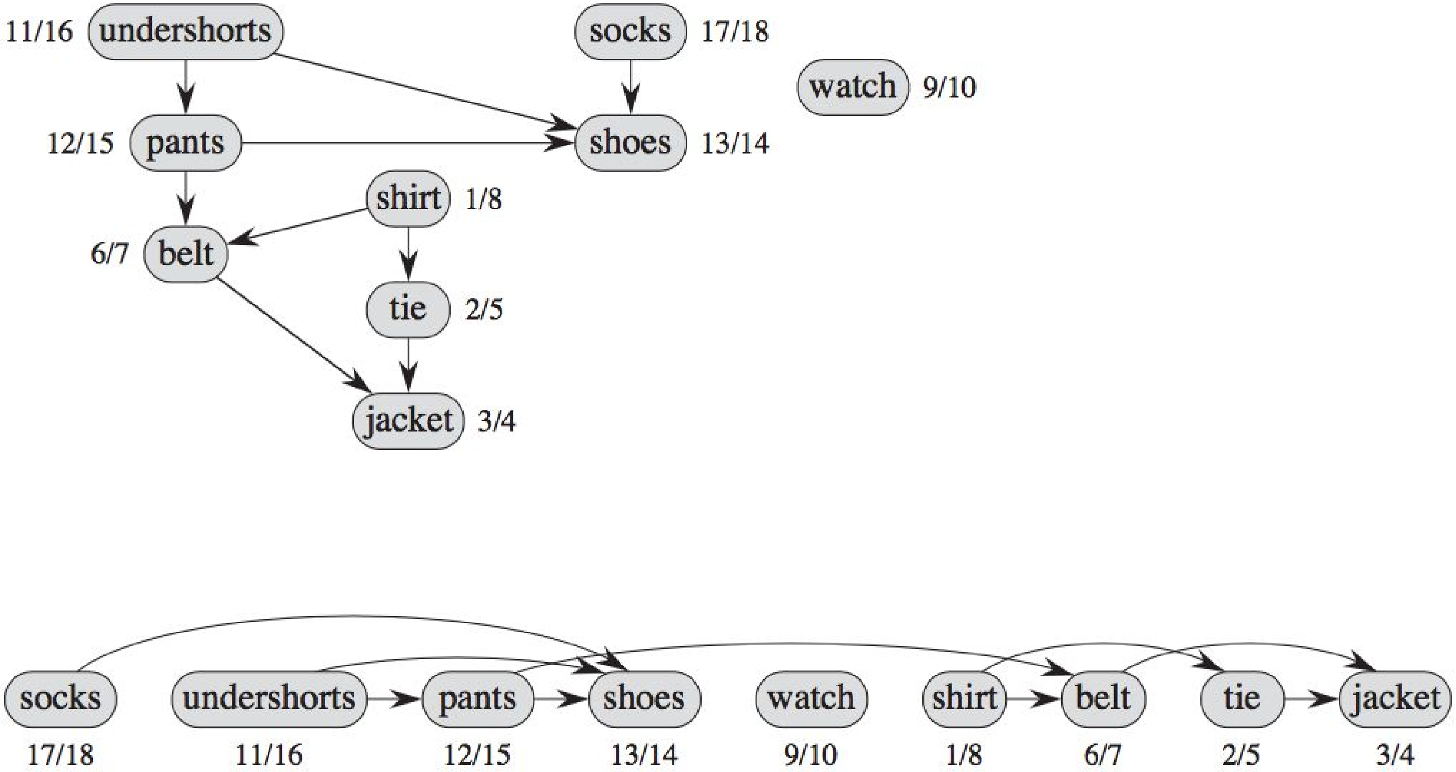

Image source: CLRS 22.4 (Page 613)

Algorithm

TOPOLOGICAL-SORT(G)
1. call DFS(G) to compute the finishing times v.f for each vertex v
2. as each vertex is finished, insert it into the front of a linked list
3. return the linked list of vertices

CLRS 22.4-2

Give a linear-time algorithm that takes as input a directed acyclic graph G = (V, E) and two vertices s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex p to vertex : pov, poryv, posryv, and psryv. (Your algorithm needs only to count the simple paths, not list them.)

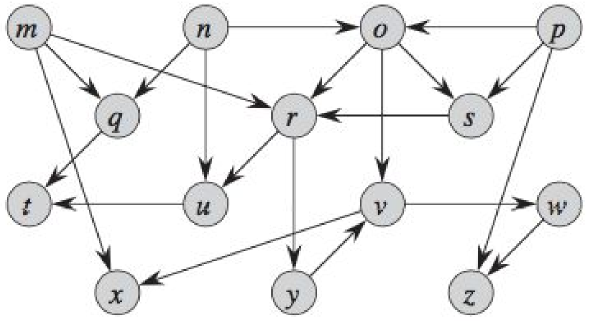

Figure 22.8 A dag for topological sorting.

CLRS 22.4-2 Solution

The algorithm works as follows. The attribute $u.paths$ of node u tells the number of simple paths from u to v, where we assume that v is fixed throughout the entire process. To count the number of paths, we can sum the number of paths which leave from each of u's neighbors. Since we have no cycles, we will never risk adding a partially completed number of paths. Moreover, we can never consider the same edge twice among the recursive calls. Therefore, the total number of executions of the for-loop over all recursive calls is $O(V + E)$. Calling SIMPLE-PATHS(s, t) yields the desired result.

CLRS 22.4-2 Solution continued

Algorithm 6 SIMPLE-PATHS(u,v)

```
if u == v then
    Return 1
else if u.paths ≠ NIL then
    Return u.paths
else
    for each w ∈ Adj[u] do
        u.paths = u.paths + SIMPLE-PATHS(w, v)
    end for
    Return u.paths
end if
```

CLRS 22.4-3

Give an algorithm that determines whether or not a given undirected graph G = (V, E) contains a cycle. Your algorithm should run in O (V) time, independent of |E|.

CLRS 22.4-3

We can't just use a depth first search, since that takes time that could be worst case linear in $|E|$. However we will take great inspiration from DFS, and just terminate early if we end up seeing an edge that goes back to a visited vertex. Then, we should only have to spend a constant amount of time processing each vertex. Suppose we have an acyclic graph, then this algorithm is the usual DFS, however, since it is a forest, we have $|E| \leq |V| - 1$ with equality in the case that it is connected. So, in this case, the runtime of $O(|E| + |V|)$ $O(|V|)$. Now, suppose that the procedure stopped early, this is because it found some edge coming from the currently considered vertex that goes to a vertex that has already been considered. Since all of the edges considered up to this point didn't do that, we know that they formed a forest. So, the number of edges considered is at most the number of vertices considered, which is $O(|V|)$. So, the total runtime is $O(|V|)$.