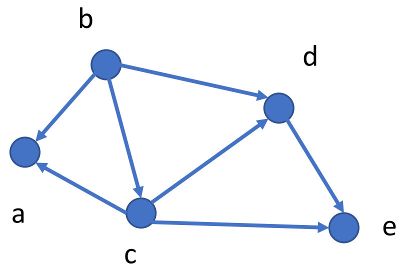
Topological Sort of a Directed Acyclic Graph

These notes contain some material that was discussed in class and that is not in the book in the same form.

If $G = (V, E)$ is a directed graph, we say that an ordered list L of the vertices V is a *topological sort* of the graph G if, for every edge $(u, v) \in E$, the vertex u comes before the vertex v in L.

For example, the list b, c, a, d, e is a topological sort of the graph below.



The list a, b, c, d, e is not a topological sort of the graph above, because the edges (b, a) and (c, a) do not satisfy the required property.

If the vertices of a graph represent tasks, and each edge (u, v) represents the information that task u must be completed before task v can be started, then a topological sort is a possible order in which to perform the tasks so that all the precedence constraints are satisfied.

A *cycle* in a directed graph $G = (V, E)$ is a sequence of vertices $v_1, \ldots, v_k, v_1$ such that all the edges (v_i, v_{i+1}) are in E for $i = 1, \ldots, k - 1$, and the edge (v_k, v_1) is also in E. Equivalently, a cycle is a path $v_1, v_2, \ldots, v_k, v_1$ such that the first and the last vertex of the path are the same.

Theorem 1 *If a directed graph G contains a cycle, then G cannot have a topological sort.*

PROOF: Suppose toward a contradiction that $G = (V, E)$ is a directed graph, that $v_1, \ldots, v_k, v_1$ is a cycle in G and that L is a topological sort in G. Let v be the first vertex of the cycle to appear in L. Then v has a predecessor u in the cycle such that $(u, v) \in E$ (the predecessor of v_1 is v_k, and the predecessor of v_{i+1} is v_i for $i = 1, \ldots, k-1$); but this is a contradiction because (u, v) is an edge and u comes after v in L. $\square$

Now we want to argue that the converse also holds.

Theorem 2 *If a directed graph G does not contain any cycle, then G has a topological sort.*

Note on terminology: a directed graph that does not contain any cycles is called a *directed acyclic graph*, abbreviated *DAG*.

To prove Theorem 2, we describe an algorithm for finding a topological sort, and we will argue that it works on all graphs that are acyclic.

Given a graph $G = (V, E)$, the algorithm is:

```
• $L$ = empty list
• while $V$ is not empty:
    – let $v$ be a vertex with zero incoming edges
    – $L = L + v$
    – remove $v$ from $V$, and remove all the edges incident on $v$ from $E$
• return $L$
```

The idea of the algorithm is that if a vertex has no incoming edges, then it is correct to put it at the beginning of a topological sort, because this will satisfy all edges incident on the vertex. Having done that, we can remove the vertex and the edges incident on it from the graph, and continue with the rest of the graph.

If the algorithm terminates, then it does terminate with a valid topological sort.

The only way the algorithm fails to terminate is if, at some point, the algorithm is left with a graph in which every vertex has at least one incoming edge.

Here we make the following observation.

Lemma 3 *If $G = (V, E)$ is a directed graph in which every graph has at least one incoming edge, then G contains a cycle.*

PROOF: Consider any vertex v in V, and call it v_{n+1}, where $n = |V|$. We know that there is at least one edge (x, v_{n+1}) that goes into v_{n+1}. Call the other endpoint of this edge v_n. Now v_n has at least one incoming edge, call the other endpoint of this edge v_{n-1} and so on. Now we have a sequence of vertices

$$v_1, v_2, \ldots, v_n, v_{n+1}$$

such that for all $i = 1, \ldots, n$ the edge (v_i, v_{i+1}) is in E. By the pigeonhole principle, there must be some repeated vertex in the sequence, so for some $i < j$ we must have $v_i = v_j$. Now the sequence

$$v_i, v_{i+1}, \ldots, v_j$$

is a cycle. $\square$

Applying the lemma to each phase of the algorithm, we have that if the initial graph is acyclic then all the graphs that we obtain after deleting vertices and edges are also acyclic, and so we can always find a vertex of indegree zero. This proves that the algorithm always finds a topological sort when given an undirected acyclic graph and so, in particular, all undirected acyclic graphs admit a topological sort, proving Theorem 2.

Now we bring the discussion back to DFS, and we prove that if G is a directed acyclic graph, and L is a list of vertices in the reverse order in which *explore*$(\cdot)$ completes its execution, then L is a topological sort of G. This means that we can find a topological sort in the same running time $O(|V| + |E|)$ of DFS, and this is also teaches us something about DFS that will later have other applications.

We refer to the following version of DFS in which we keep a list of nodes whose *explore*$(\cdot)$ procedure has terminated:

```
Global variable: Boolean array visited
Global variable: list L of vertices

explore(G,v)
  visited[v] = True
  for each node w such that (v,w) is an edge of G:
    if not visited[w]:
      explore(G,w)
  L = v + L

DFS(G)
  initialize visited[v] = False for each vertex v of G
  L = empty list
  for each vertex v of G:
    if not visited[v]:
      explore(G,v)
```

We want to say that if G is acyclic, then, at the end of the execution of $DFS(G)$, the list L is a topological sort of G. This is equivalent to the following statement.

Theorem 4 *Let $G = (V, E)$ be an acyclic directed graph. If we run $DFS(G)$, then, for every edge $(u, v) \in E$ we have that the call to $explore(G, u)$ terminates after the call to $explore(G, v)$.*

PROOF: Consider the state of the algorithm (meaning, which vertices are visited and which recursive calls are on hold) when $explore(G, u)$ is called. At that point, each vertex x can be in one of three states:

- not yet visited
- visited, but not yet on the list L, because the call to $explore(G, x)$ is on hold
- visited and on the list L

Note that if a vertex x is of the second type, then there is a path from x to u, because when $explore(G, x)$ makes a recursive call, it calls $explore(G, y)$ for some vertex y such that (x, y) exists, and then $explore(G, y)$ can only make calls of the type $explore(G, w)$ for some vertex w such that (y, w) is an edge, and so on, meaning that all the call to $explore(G, \cdot)$ that are on hold when $explore(G, u)$ is called involve vertices from which u is reachable.

There can, however, be no path from v to u, otherwise there would be a cycle in G. This means that when $explore(G, u)$ is called, the vertex v is either not yet visited or is already on the list L.

If v is not visited, then $explore(G, v)$ will be called inside the execution of $explore(G, u)$, and so $explore(G, u)$ is on hold during the execution of $explore(G, v)$, and $explore(G, u)$ terminates after $explore(G, v)$ terminates.

If v is already on the list, then it means that $explore(G, v)$ has already terminated at the time that $explore(G, u)$ is called, so certainly $explore(G, u)$ terminates after $explore(G, v)$ terminates. $\square$