# CS 228: Introduction to Data Structures
# Lecture 38

## Detecting Cycles

Recall that a (simple) ***cycle*** in a graph G is a sequence of nodes that starts and ends at the same node such that (1) no repetition of nodes is allowed, other than the repetition of the first and last node, and (2) for every two nodes u and v in the sequence, there is an edge (u,v) in G.

Graph G is ***acyclic*** if it contains no cycles. Testing whether G is acyclic turns out to be equivalent to testing whether DFS yields a back edge. We consider undirected and directed graphs separately.

For *undirected graphs* we can show that every edge in a DFS is either a tree edge or a back edge — there are no forward edges or cross edges. Further, it is not hard to see that as soon as we detect a back edge (u, v), we have found a cycle: this cycle consists of the path from v to u (which exists, because v is an ancestor of u in the DFS forest) followed by the edge (u,v). Thus, we can test whether an undirected graph is acyclic in O(V + E) time.

For *directed graphs*, DFS may yield forward or cross edges. Nevertheless, observe first that if a DFS produces a back edge (u, v), G must have a cycle. The reasoning is the same as for undirected graphs: vertex v is an ancestor of vertex u in the DFS forest, so there is a path from v to u, and the back edge (u, v) completes a cycle.

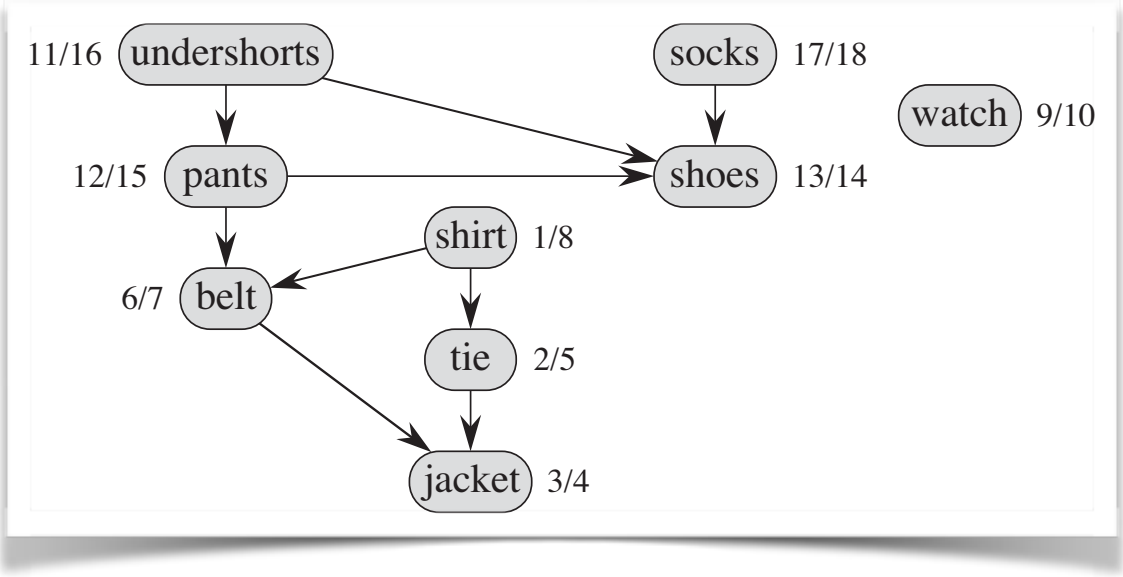
On the other hand, suppose that G contains a directed cycle C. Will a depth-first search of G yield a back edge? The answer is "yes". Let v be the first vertex to be discovered in C, and let (u, v) be the preceding edge in C. At time v is discovered, the vertices of C form a path of white vertices from v to u. By the White-Path Theorem, node u becomes a descendant of v in the DFS forest. Therefore, (u, v) is a back edge.
To summarize:

**Fact.** A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

It is easy to detect back edges during DFS. Thus, we can test if G is acyclic in O(V + E) time.

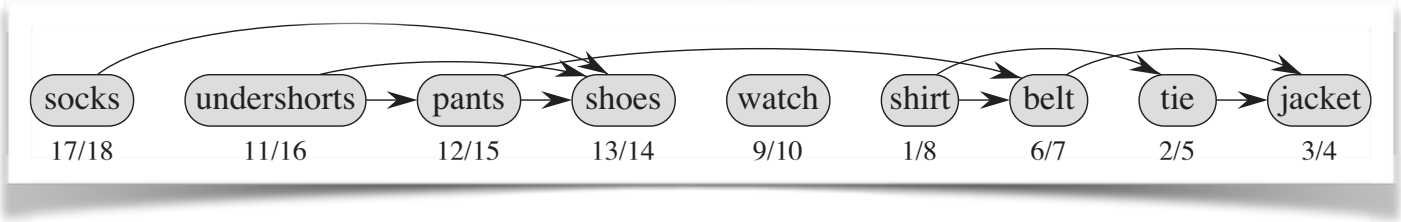
## Topological Sort

For brevity, let us refer to a directed acyclic graph as a ***DAG***. One of the many applications of DAGs is to model




***precedences*** among events. This is illustrated in the next figure[1], which shows the set of precedence relationships that Professor Bumstead faces when getting dressed in the morning. The professor must don certain garments before others (e.g., socks before shoes). Other items may be put on in any order (e.g., socks and pants). A directed edge (u, v) in this DAG indicates that garment u must be donned before garment v.

[1] This example is taken from CLRS. The numbers next to the nodes are called discovery/finishing times. They indicate the time at which a node was first encountered (i.e., went from white to grey) and the time it was finished (went from grey to black). Discovery and finish times have several applications, but we will not cover them in this class.




A ***topological sort*** of a DAG G is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before in the ordering. We can depict a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. The next figure shows a topological sort of the above graph, drawn in this way.

In this case, a topological sort of the DAG gives an order for Professor Bumstead to get dressed.
A DAG may have multiple topological sorts. For instance, here is another topological sort for the graph of the previous page:

| watch, shirt, socks, undershorts, pants, shoes, tie, belt, jacket |
|---|

Note that, for a graph G to have a topological sort, it **must** be a DAG: if G contains a cycle, then no linear ordering is possible.

The following observation gives us a way to use DFS to topologically sort a DAG.

**Fact.** Suppose u, v are any two distinct nodes in a DAG G. If G contains an edge from u to v, then DFS finishes v before u.

An example that illustrates this observation is posted on Blackboard. The observation leads to the following algorithm for finding a topological sort of a DAG.

TOPOLOGICALSORT(G)

1. call `dfs(G)`
2. as each node is finished[2], insert it onto the front of a linked list
3. **return** the linked list of nodes

TOPOLOGICALSORT takes time O(V + E), since depth-first search takes O(V + E) time and it takes O(1) time to insert each of the V vertices onto the front of the linked list.

## Other Applications of DFS

DFS has many other applications. Here are three.

[2] Recall that a node is said to be *finished* immediately after `dfsVisit()` colors it black.

- Finding ***strongly connected components*** in directed graphs.

- Finding ***biconnected components*** in undirected graphs — roughly, these are subgraphs that remain connected even after an edge is removed.

- Testing ***planarity*** — can the G be drawn on the plane so that no two edges cross?

## Weighted Graphs

A ***weighted graph*** is a graph in which each edge is labeled with a numerical weight. A weight might express the distance between two nodes, the cost of moving from one to the other, the resistance between two points in an electrical circuit, or many other things.

It is not hard to modify the graph representations that we have seen to handle weighted edges.

- In an ***adjacency matrix***, we can store each weight in the matrix. Edges missing from the graph can be represented by a special number like `Integer.MIN_VALUE` or `Integer.MAX_VALUE` (this does have the cost of declaring that number invalid as an edge weight, though).

- In an ***adjacency list***, recall that each edge is represented by a list entry containing an endpoint of the edge. We can simply enlarge each list entry to include a weight, in addition to the reference to the endpoint.

- The ***`HashMap`/`HashSet`*** representation is modified similarly to the adjacency list representation: each entry in the `HashSet` of a node u is contains not only a neighbor v of u, but also the weight the weight of edge (u,v).

## Shortest Paths

We have seen that breadth-first search allows us to find the shortest paths from a specified start node s to all other nodes, if the length of a path is measured by the number of edges in it. This way of measuring path length is OK when each edge is as good as any other. This is not always the case, though.

- Suppose the graph represents a map; nodes are intersections, edges are road segments, and edge lengths are distances between intersections. You want to find the shortest route from your house to a supermarket.

- Suppose the nodes of the graph are airports, the edges are possible flights, and the length of an edge is the cost of taking that flight. Your problem would then be to find the cheapest flight from, e.g., DSM to MLE[3].

Note that these graphs may be directed. For example, there may be one-way roads, or flights in one direction might have different costs than those the other way.

We will assume that all edge lengths are positive. This is often but not always the case. For instance, it is conceivable that an airline could pay people to take certain routes, so that the lengths of those edges in the airport graph might be negative. We will pretend this never happens. It makes the algorithms a lot easier.

Although we are often interested in finding the distance from a starting node s to an ending node t, it turns out to be just as easy (in fact, even easier) to compute, once and for all, the distances from s to all other nodes. This is known as the ***single source shortest path problem*** (s is the ***source***). On Wednesday, we will study ***Dijkstra's algorithm***, which solves this problem efficiently.

[3] Ibrahim Nasir International Airport in the Maldives.