CMSC 451: Lecture 3
Cycles and Strong Components

Directed Acyclic Graphs: A *directed acyclic graph*, or *DAG*, is a directed graph that has no cycles (see Fig. 1). DAGs arise in many applications where there are precedence or ordering constraints. For example, if there are a series of tasks to be performed, and certain tasks must precede other tasks (e.g., in construction you have to build the walls before you install the windows). In general a *precedence constraint graph* is a DAG in which vertices are tasks and the edge (u, v) means that task u must be completed before task v begins.

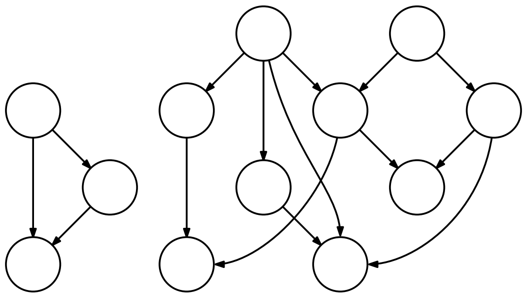

Fig. 1: Directed acyclic graph (DAG).

It is easy to see that every DAG must have at least one vertex with no incoming edges, and at least one vertex with no outgoing edges. A vertex with no incoming edges (only outgoing) is called a *source* and a vertex with no outgoing edges (only incoming) is called a *sink*.

Acylicity Testing: Let us consider the problem of determining whether a digraph is acyclic. We are given a directed graph $G = (V, E)$, and we with to determine whether G contains a cycle. If so, G is not a DAG.

We will present a simple algorithm based on DFS. Recall that in addition to tree edges, a DFS forest contains three other types of edges, back edges (which go to a vertex's ancestor), forward edges (which go to a vertex's descendant), and cross edges (everything else). Observe that if the DFS forest of G contains at least one back edge, then G has a cycle. This is easy to see. If (u, v) is a back edge, then there is a path in the tree from the ancestor v to the descendant u, and the back edge from u to v completes the cycle. The following lemma shows that this condition is not only sufficient, but necessary.

Claim: If a digraph G has a cycle, then *any* DFS forest of G (i.e., no matter what order the vertices are visited) has a *back edge.*

Proof: The proof is based on a very simple observation about the various edge types and finish times. Recall from the Parenthesis Lemma (from the previous lecture) that if u is an ancestor of v then we have $[d[v], f[v]] \subset [d[u], f[u]]$. It follows that if (u, v) is a tree edge or forward edge then $f[u] > f[v]$. Also, observe that (u, v) is a cross edge, it must be u was discovered after v was finished (for otherwise, u would have made a DFSvisit call on v, implying that this would be a tree edge). Therefore $d[u] > f[v]$. Since a vertex cannot finish until after it was discovered, we have $f[u] > f[v]$.

In summary, for all these three edge types (tree, forward, and cross), the finish time of the origin is strictly greater than the finish time of the destination. It follows directly that it is impossible to complete a cycle from any combination of just these three edge types. Only for back edges do we have $f[u] < f[v]$, and therefore in order to form a cycle we need to have *at least one* back edge. Therefore, if a graph G has a cycle, in any DFS forest of G there must be at least one back edge.

The above theorem implies that in order to determine whether a graph G has a cycle, it suffices to test whether it has a back edge. How do we know whether an edge is a back edge. The proof of the above theorem provides an easy way. We can first apply DFS to G, and we then run through the edges, checking whether $d[u] > d[v]$. Can we do this on the fly as DFS is running? The answer is yes. Observe that a back edge goes from a vertex u to an ancestor v. Such an ancestor must have been discovered, but not yet finished. For the other non-tree edge types, the destination v will have already finished. The main DFS function is the same, only DFSvisit needs to be updated.

Determining whether a graph has a cycle

```
DFSvisit(u) {                               // perform a DFS search at u
    mark[u] = discovered                    // u has been discovered
    d[u] = ++time
    for each (v in Adj(u)) {
        if (mark[v] == undiscovered) {      // undiscovered neighbor?
            pred[v] = u
            DFSvisit(v)                     // ...visit it
        }
        else if (mark[v] != finished) {     // equivalent to f[u] <= f[v]
            output "cycle found!" and terminate DFS
        }
    }
    mark[u] = finished                      // we're done with u
    f[u] = ++time
}
```

Topological Sorting: A *topological sorting* (or *topological ordering*) of a DAG is a linear ordering of the vertices of the DAG such that for each edge (u, v), u appears before v in the ordering. Note that in general, there may be many valid orderings for a given DAG. We will present a simple algorithm based on DFS. (Kleinberg and Tardos present a different algorithm. We have elected this approach as an illustration of DFS.)

Recall our earlier comments on the nature of DFS edge types and discover/finish times. After running any DFS on a graph, if (u, v) is a tree, forward, or cross edge, then the finish time of u is greater than the finish time of v. Since a DAG is acyclic, there can be no back edges, which implies that *every* edge goes from vertex a higher finish time to one of lower finish times. Thus, in order to produce a topological ordering of the vertices it suffices to output the vertices in *reverse* order of finish times. To do this we run a (stripped down) DFS. As each vertex is finished, we push it onto a stack. (Thus, the later a vertex finishes, the closer it is to the top of the stack.) Popping the elements off the stack yields the final topological order.

Topological Sort via DFS

```
topSort(G) {
    for each (u in V) mark[u] = undiscovered    // initialize
    S = empty stack
    for each (u in V)
        if (mark[u] == undiscovered) topVisit(u)
    while (S is nonempty) output pop(S)         // pop stack for final ordering
}

topVisit(u) {                                   // start a search at u
    mark[u] = discovered                        // mark u visited
    for each (v in Adj(u))
        if (mark[v] == undiscovered)
            topVisit(v)                         // visit v (pushing it on stack)
    push u onto S                               // push u when finished
}
```

Observe that the structure is essentially the same as the generic DFS procedure given in the previous lecture, but we only include the elements of DFS that are needed for this application. As with standard DFS, the running time is $O(n + m)$ (recalling that $n = |V|$ and $m = |E|$).

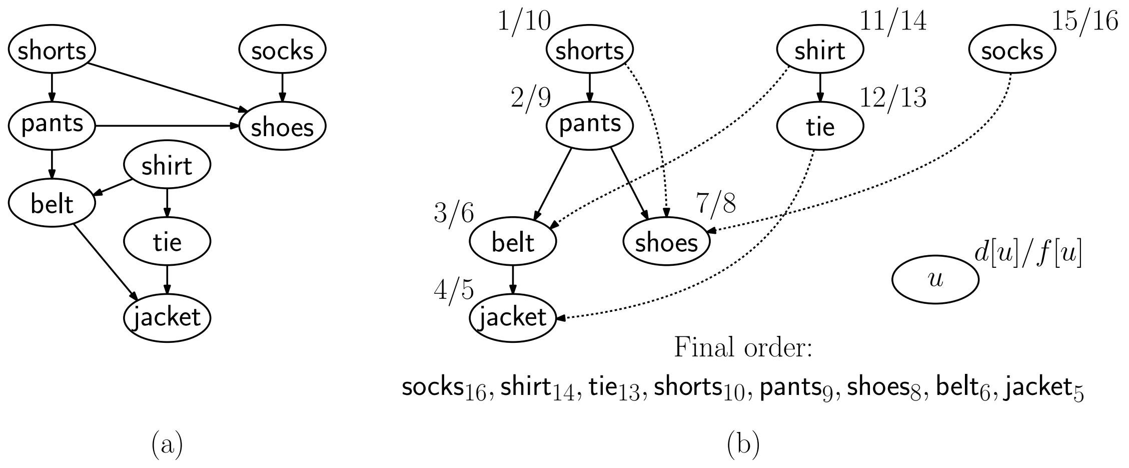

Fig. 2: Topological ordering example.

As an example we consider the DAG showed in the Fig. 2(a), which shows the precedence constraints for a person deciding in what order to get dressed. In the example we show the discovery/finish times, but the algorithm does not need them. Note that there are many different possible DFS's of the same graph, and each one corresponds to a potentially different, but still valid, topological ordering. (A question worth pondering is whether every possible topological ordering arises from some DFS search.)

Longest Path in a DAG: (Optional) Here is a short exercise to test your understanding of DFS. Suppose that you are given a DAG $G = (V, E)$, where each vertex $u \in V$ is to be thought of as a task that takes $\mathsf{time}[u]$ time units to perform. Each edge (u, v) of the DAG represent precedence constraints, meaning that task u must be completed before task v is started. The question is, assuming the maximum degree of parallelism is allowed, what is the minimum amount of time needed to complete all the tasks? This is just maximum sum of the time values along any path in the DAG.

We can solve this in $O(n + m)$ time through DFS. The trick is to associate each vertex u of the DAG with the maximum length any path that emanates from this vertex. We denote this quantity by $\mathsf{maxTime}[u]$. When we first encounter a vertex u in the DFS visit procedure, which we rename LongPathVisit, we initialize $\mathsf{maxTime}[u] = \mathsf{time}[u]$. For each adjacent vertex v, we invoke $\mathsf{LongPathVisit}(v)$ if v has not yet been discovered. We let maxLength to be the maximum maxTime of all u's neighbors, and we set $\mathsf{maxTime}[u] = \mathsf{maxTime} + \mathsf{time}[u]$. As in standard DFS, the main program invokes $\mathsf{LongPathVisit}(u)$ for all undiscovered vertices u. $\mathsf{LongPathVisit}(u)$ is given in the code-block below.

Longest Path via DFS

```
LongPathVisit(u) {                                        // start a search at u
   mark[u] = discovered                                   // mark u visited
   maxTime[u] = time[u]                                   // initialize max time for u
   for each (v in Adj(u)) {
       if (mark[v] == undiscovered) LongPathVisit(v)      // process v if undiscovered
       maxTime[u] = max(maxTime[u], maxTime[v] + time[u]) // update our max time
   }
}
```

An example is shown in Fig. 3. Each vertex's maxTime value is the sum of its own time and the maxTime values of its neighbors.

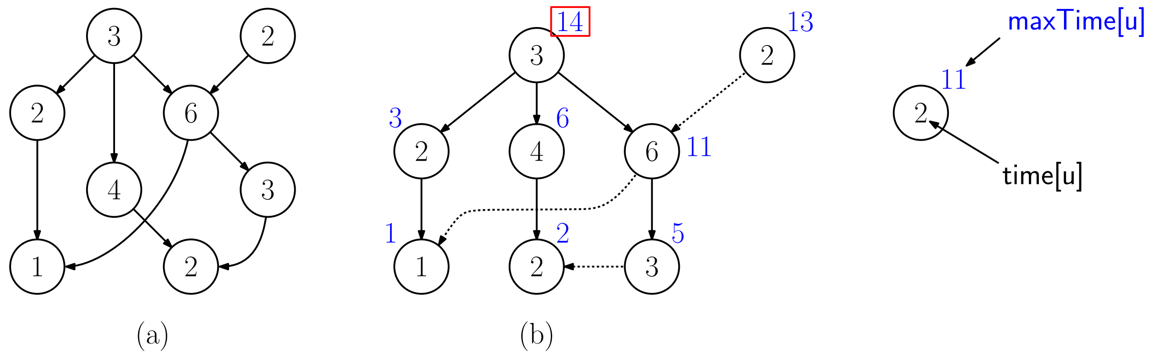

Fig. 3: Longest path in a DAG. Each vertex u is labeled with $\mathsf{time}[u]$ and $\mathsf{maxTime}[u]$.

Because the graph is acyclic, every edge (u, v) goes from u to a vertex v whose finish time is greater than u's. Therefore, $\mathsf{maxTime}[v]$ is fixed before it is accessed by u. The longest path in the entire DAG is the largest value of $\mathsf{maxTime}[u]$ among all vertices u. The *critical tasks* are those that lie on the longest path. How would you compute these?

Can this be used to compute the longest simple path in a digraph with cycles? The answer is no, but you should think about why it does not work.

Strong Components: (The following material applies *only* to directed graphs!)

A digraph $G = (V, E)$ is said to be *strongly connected* if for every vertex u and v there is a path from u to v and from v to u. It is easy that this *mutual reachabilitys* relation between vertices is an equivalence relation. This implies that it partitions V into equivalence class, called the *strong components* (or *strongly-connected components*) of G (see Fig. 4(a) and (b)).

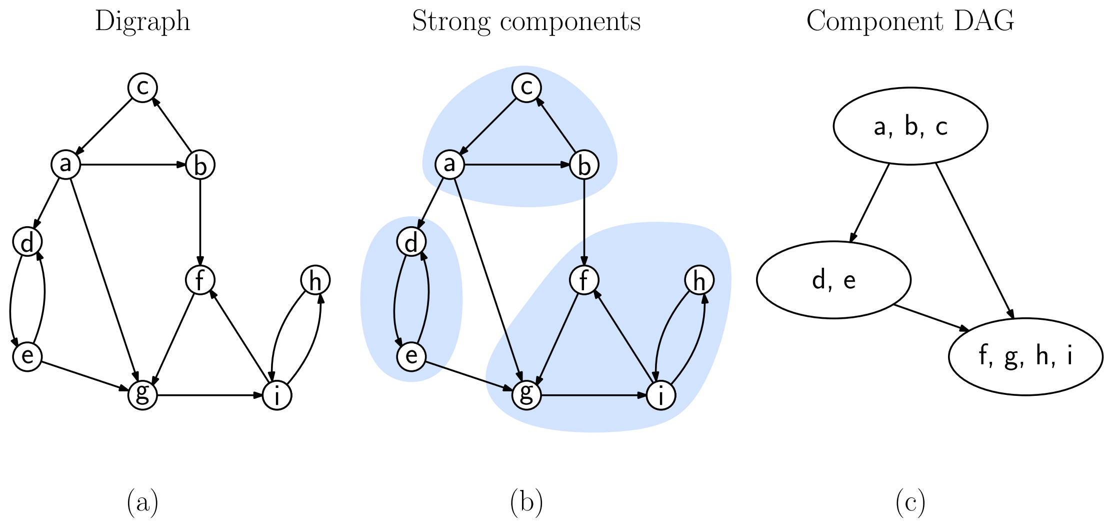

Fig. 4: Strong components and the component DAG.

If the vertices within each strong component are collapsed into a single vertex, the resulting digraph is called the *component digraph* (see Fig. 4(c)). It is easy to see that the component digraph must be acyclic (since if a number of components could be joined in a cycle, they would collapse into a single larger strong component). Therefore, this graph is usually called the *component DAG*.

There exists an $O(n + m)$-time DFS algorithm for computing strong components. It is based on the following lemma.

Claim 1: If DFSvisit is started at a vertex u, it will terminate precisely when all the vertices reachable from u have been visited.

Proof: This follows from the exhaustive nature of DFS. (Note that some of these vertices may have been reached by earlier calls to DFSvisit.)

Claim 2: If C and C' are two strong components, and there is an edge from a vertex in C to a vertex in C', then the highest finish time in C is bigger than the highest finish time in C'.

Proof: There are two cases depending on whether the DFS first encounters a vertex from C or C'. If it first encounters a vertex u in C, then by Claim 1 the DFS will visit all the vertices of both C and C' before returning to u. Therefore, u will have the highest finish time of every vertex in $C \cup C'$. If it first visits a vertex in C', then the DFS will get stuck in C' (since it is not possible to reach anything in C). It follows that all the vertices of C will have higher discovery times than those of C', which further implies that they will have higher finish times as well.

Claim 3: The vertex that receives the highest finish time in a DFS must lie in a source vertex of the component DAG. (Recall that a vertex in a DAG is called a source if it has no incoming edges.)

Claim 3 is equivalent to saying that the strong components can be linearly arranged in decreasing order of their highest finish times. By doing so, every edge in the component DAG will go from an earlier component in the linear order to a later one. How can we exploit this to obtain an efficient algorithm to find the strong components.

Claim 3 allows us to identify a vertex in some *source* of the component DAG. Unfortunately, this is not all that useful. What *would* be useful is to identify a vertex in a *sink* of the component DAG. If we could do this, we could start a DFS at this vertex, with the knowledge (by Claim 2) that no other strong components would be visited. We could then delete all these vertices (or equivalently, mark them as visited), and repeat the process. Eventually, all the strong components will be identified, each one arising as a separate subtree of the DFS forest.

So how to we convert an algorithm that identifies sources to one that identifies sinks in the component DAG? The trick is to reverse all the edges of G. Let G^R denote the directed graph that has the same vertex set as G, but every edge (u, v) is replaced by its reverse (v, u). Note that the strong components of G^R are the same as G, but the direction of edges in the component DAG have all be *reversed*. Thus, the sources in the component DAG of G^R are sinks in the component DAG of G. This leads to the following (insanely clever) algorithm for computing strong components.

(1) Given G, compute G^R (see Fig. 5(a)). (Note: This is a small programming exercise, which involves a simple traversal of G's adjacency list. It can be done in $O(n + m)$ time.)

(2) Run DFS(G^R) and label each vertex of G with the finish time of the corresponding vertex of G^R (see Fig. 5(b)).

(3) Sort the vertices of G in decreasing order of finish times. (Note: Since finish times are integers in the range $[1, 2n]$, this can be done in $O(n)$ time through Bucket Sort.)

(4) Run DFS(G), but in the outermost loop, whenever we need to find a new vertex to start DFSvisit, take the vertex with the highest finish time (using the above sorted order).

(5) Each subtree of the DFS forest will be a strong component (see Fig. 5(c)).

The correctness of the above algorithm follows from the remarks made earlier. The running time is $O(n + m)$, dominated by the time to compute G^R and the times for the two DFS's.

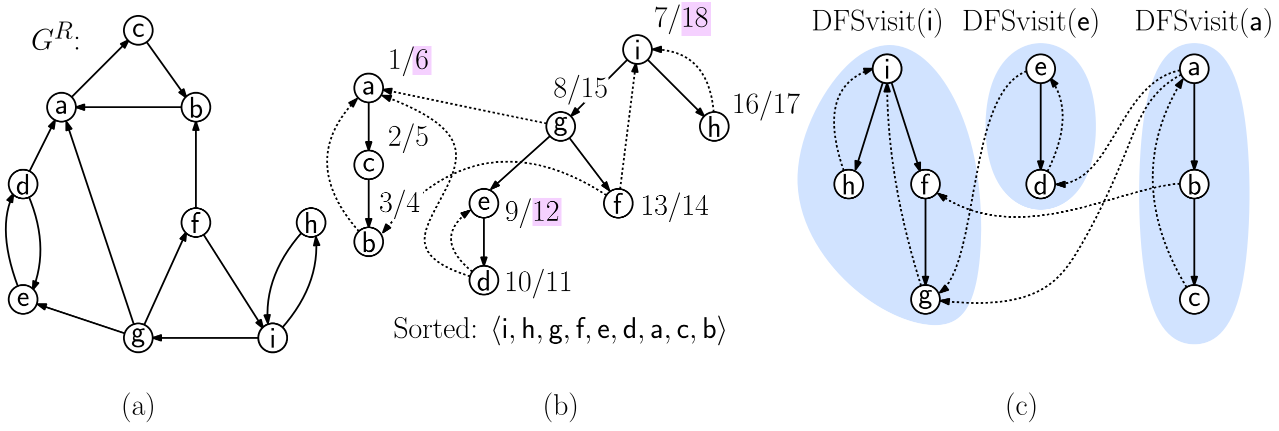

Fig. 5: Strong components and DFS.