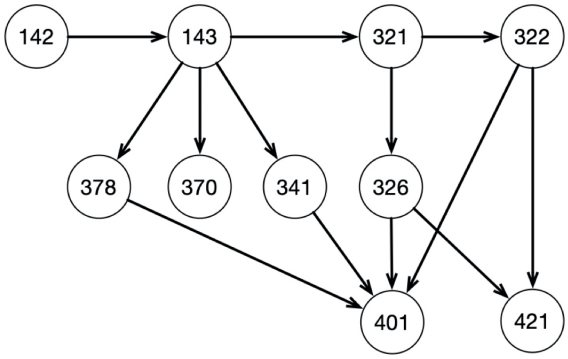
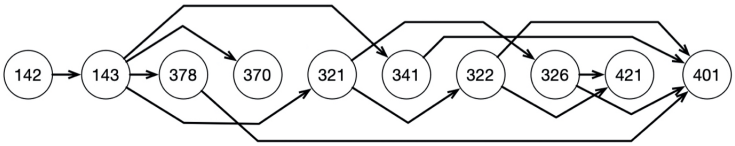
Graph Algorithms 2: Topological sort and Strongly connected components

In this lecture we study algorithms on directed graphs. Our first algorithm is *Topological sort* which is a sorting algorithm on the vertices of a directed graph. Suppose the nodes of a graph represent courses that are required for a specific program, and directed edge (a, b) indicates that course a must be taken before course b. If the requirement is to take all the courses in order to complete the program, then it would be necessary to find an ordering of the courses that respects the prerequisite requirements. Such an ordering is in fact a *Topological sort*: A linear ordering of the courses so that for all edges (a, b) in the graph, course a precedes course b in the ordering:



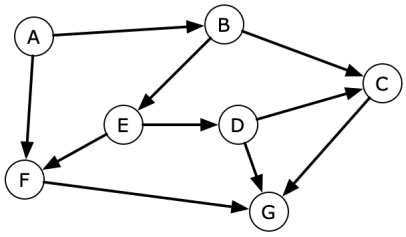
Course prerequisites indicated with directed arrows



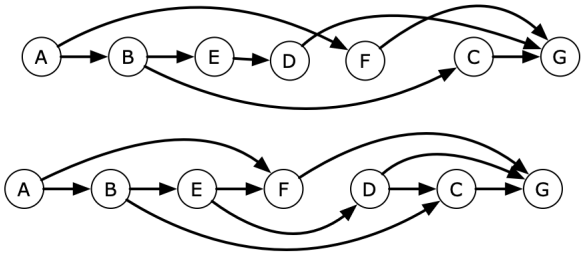
Possible order of courses

1 Topological sort

Formally, a topological sort is a linear ordering of V on the graph $G = (V, E)$ such that for all $(u, v) \in E$, the vertex u appears before v in the ordering. If the graph contains a *cycle*, then no topological ordering is possible. Therefore we consider only **directed acyclic graphs** in this section, commonly referred to as **dags**. Note that a directed graph may have more than one possible toplogical ordering, as in the example below:



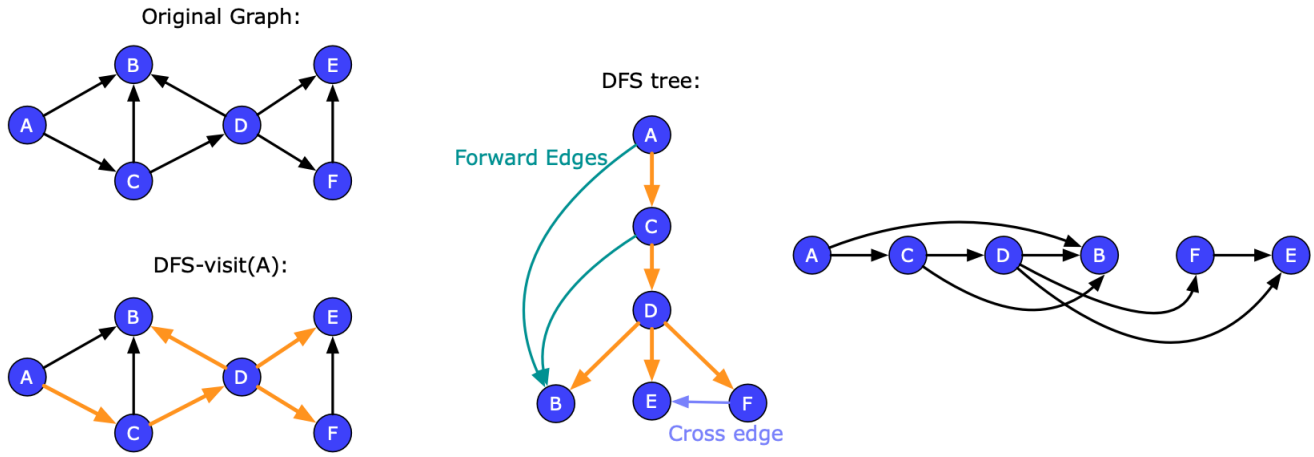
Directed Graph



Two different Topological Orderings

The DFS algorithm from our last lecture is in fact at the heart of finding this topological order. Let's start by looking at an example of a dag and the resulting DFS tree. In the figure below, we show the original graph G, and the result of DFS-visit(A). The DFS tree is drawn on the right, showing not only the tree edges, but also the other edge types of G (back, forward, and cross). By examining the DFS tree and the additional edges, it is relatively easy to identify a topological ordering: A, C, D, B, F, E. How does this ordering relate to how the vertices were discovered by DFS? It is not exactly the order in which they were discovered. Instead, this is in fact the order in which they were *finished*. Notice that vertex e

is the first vertex from which a "backtrack" was made. In other words, the order of the recursive calls was DFS-visit(A), DFS-visit(C), DFS-visit(D), DFS-visit(E), and then DFS-visit(E) terminated because there were not more unvisited neighbors. Next, DFS-visit(D) made a call to DFS-visit(F), which then also terminates because it has no more unvisited neighbors. Therefore the second-last vertex to finish is vertex F.



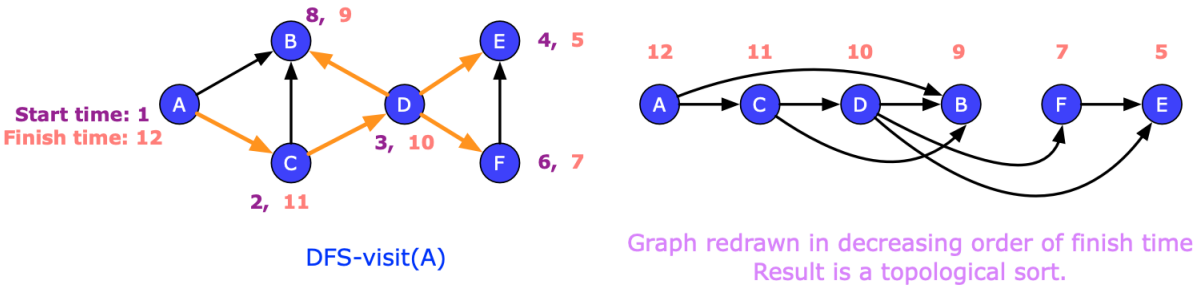
This is in fact the key to selecting a topological ordering from the DFS tree: write the nodes in **decreasing order** of the "time" at which they were finished in the DFS algorithm. It is important to note that the graph may have other topological sorts, but this method produces one of them.

1.1 The use of time-stamps in DFS

DFS can be implemented to keep track of a pair of time-stamps for each vertex. These time-stamps are useful in the Topological sort algorithm and in many other algorithms.

Time stamps in DFS
The "unit" of time is incremented each time a vertex is discovered, or when a vertex terminates DFS-visit(v).

In the example below, we write next to each vertex a pair of *time stamps* that represent when the DFS-visit(A) first discovers a vertex, and when it is finishing processing the neighbors of that vertex.



One confusion regarding DFS time-stamps is that they are often incorrectly associated with"walking" on the edges (probably due to many online animations that associate this walking withe the DFS algorithm). In the example above, vertex A is discovered at time 1, then vertex C, then vertex D, then vertex E. At this point it is "time" 4. Vertex E now terminates its DFS call, and this termination counts as"a step", therefore its termination occurs at step 5. The finish time stamp of vertex E is 5. The next vertex to be discovered is vertex F, which occurs at time step 6. Be careful! The time step is 6 - NOT 7. Often the misconception is that we had to somehow walk over to vertex F, which takes 2 steps. This is incorrect.

Think of the time-stamps as the next event that happens, where events are either discovering or ending a DFS call. Therefore vertex F is discovered next, at step 6, then next it terminates at time 7. The next vertex to be discovered is vertex B, at time 8, it then terminates at time 9, and then finally D terminates at time 10 since it has no more undiscovered neighbors. Finally C terminates at time 11, then A terminates at time 12.

Finally, using the finish times of DFS, we can write the vertices is *decreasing* order of their finish time and the result is a valid topological order, as shown in the example above.

The fact is given below:

Theorem 1. *For a directed acyclic graph G, by ordering the vertices in decreasing order of their finish time in DFS, the result is a topological ordering.*

Proof: We just nee to argue that for any edge $(u, v) \in E$, the finish time of v is less than the finish time of u. At some point during the execution of DFS, the algorithm must visit node u and consider the edge (u, v). There are three distinct cases.

Case 1: If node v is not yet visited, then node v is visited after u, and DFS will completely finish visiting v before it finishes u. This represents the case for edge (c, b) in the above example. When DFS first visits vertex c, the neighbor b is not yet visited. This means that the finish time of c will be after that of b.

Case 2:. If node v is already visited, but not yet finished, then this means there is a path from v to u through some other part of the graph, and the edge (u, v) creates a cycle. This is impossible in a dag.

Case 3: If node v is already visited but is finished, then it's finish time has already been stamped. Therefore since we are still exploring u, it's finish time will be greater than that of v. This is the case with edge (f, e) in the above example. When DFS explores vertex f, its neighbour e is already finished. Therefore the time stamp of $f's$ finish time will be greater than that of e.

1.2 Update DFS algorithm to record time stamps

It remains to show how to update the DFS algorithm so that we can record these time-stamps. We alter the DFS-visit algorithm by adding a global variable called *time*, which is initialized as 0. When all the neighbors of a vertex have been explored, the vertex is "finished" and we time-stamp the attribute $v.finish$ with the current value of *time*

DFS-visit(u)

$time = time + 1$
Mark node u as visited: $u.visited = true$
$u.start = time$
For each $v \in Adj\,[u]$
    If $v.visited = false$
        $v.parent = u$
        $v.distance = u.distance + 1$
        DFS-visit(v)
$time = time + 1$
$u.finish = time$

The topological sort algorithm follows directly from the information stored in each node after the execution of this updated DFS:

Topological-sort(G)

Step 1: Call DFS(G) to compute the finish times.
Step 2: As each vertex is finished in DFS, insert that vertex at the front of the topological ordering.

Runtime:

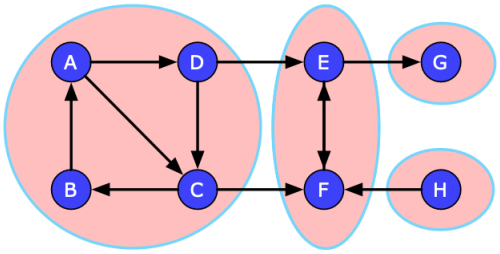
The runtime of DFS is $\Theta(V + E)$, and the extra time needed to simply place the vertices in a linked list in the order that they are finished is $O(1)$ per node. Therefore the overall runtime is $\Theta(V + E)$.

2 Strongly Connected Components

One important notion that characterizes graphs is the notion of *connectivity*. For directed graphs, we can describe connectivity in terms of the existence of directed paths between pairs of vertices.

> A **Strongly connected component**, or SCC, of a graph is a maximal set of vertices such that directed paths exist between every pair of vertices (from both u to v and v to u).

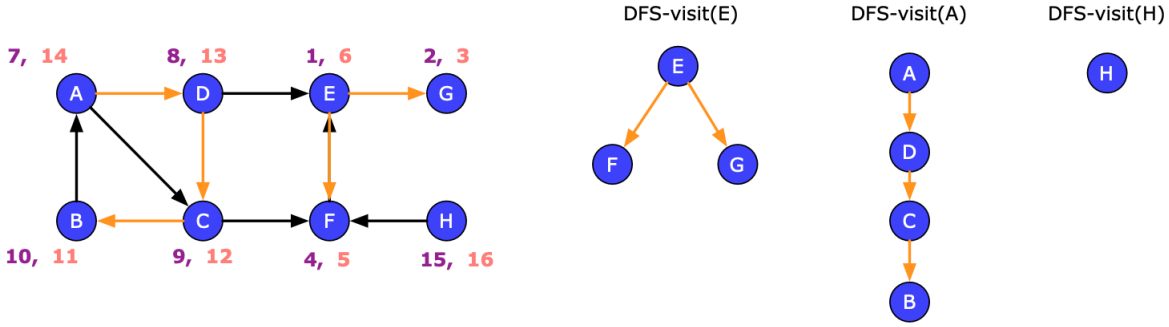
The directed graph below consists of four strongly connected components. Note that each vertex within a strongly connected component can reach every other vertex along a path. Vertex G is in its own strongly connected component, because there is no path to and from another vertex in G. Similarly, vertex H is in its own SCC.



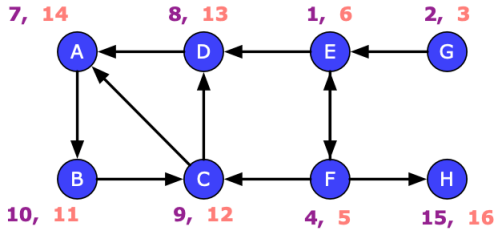
Four strongly connected components

The algorithm that identifies these strongly connected components is again based on DFS. There are three main steps:

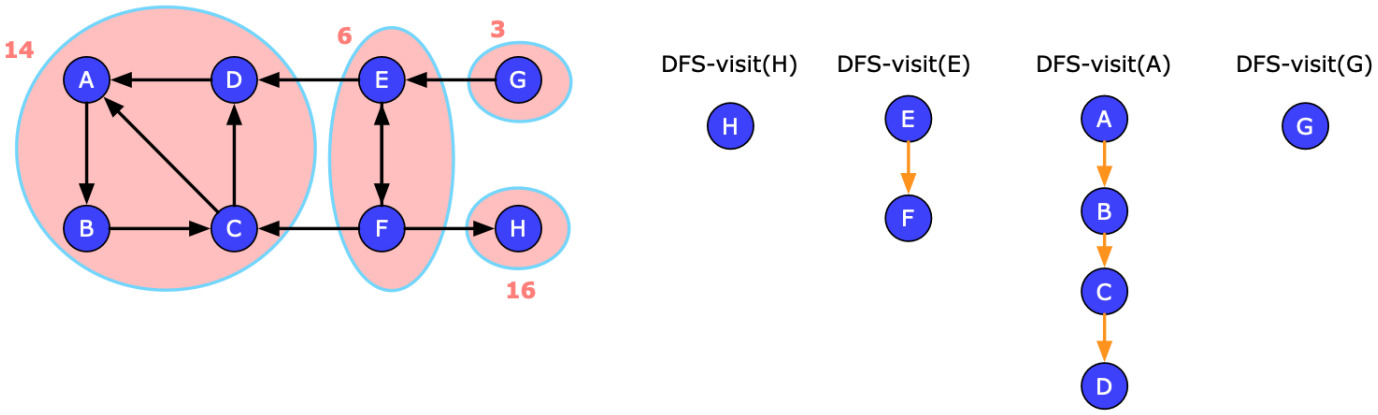
Step 1: The first step is to call DFS(G) and compute the start/finish time stamps. The first vertex that is picked to start DFS can be any of the vertices of the graph. Recall if DFS-visit() terminates before visiting all vertices, it restarts from a new vertex. In the example below, DFS-visit starts on vertex E, visits vertices G, F and then terminates at E. Next, it restarts at A, visits D, C, B and terminates at A. Finally DFS-visit restarts at H and terminates. The corresponding DFS trees are shown on the right, making up the DFS forest.



Step 2: Create a *new* graph which is the result of **reversing** all the edges in G. This graph is called G^T.



Step 3:. Call DFS(G^T). In the main loop of DFS the vertices are considered in **decreasing order of finish time**. In the example below, we start with DFS-visit(H) since it has the highest finish time. The call terminates immediately since H has no neighbors. The next call is to DFS-visit(A) since it has the next largest finish time. The resulting search visits A, B, C, D and then terminates. Next the call is made to DFS-visit(E), with the third largest finish time. Vertices E and F are discovered. Finally, the last call is to DFS-visit(G), which terminates immediately since there are no unvisited neighbors. The corresponding DFS trees are shown on the right. Each DFS tree corresponds to a strongly connected component!



The result is exactly four strongly connected components.

Runtime: The algorithm above runs a DFS twice: once on G on again on G^T. Therefore the total runtime is $\Theta(V + E)$.