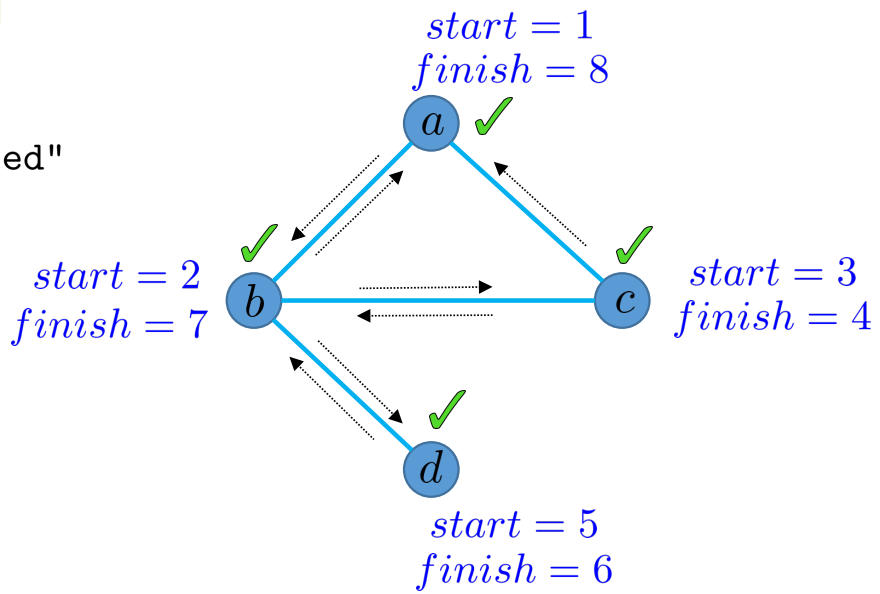
Algorithms

Depth First Search (DFS)

```
T = {}; // T is the DFS tree
time = 0

DFS(u):
    visit u and mark it "visited"

    time += 1
    u.start = time

    for each edge (u,v):
        if v is not "visited":
            T = T ∪ {(u,v)}
            DFS(v)

    time += 1
    u.finish = time
```

Execution of DFS(a):



DFS Tree. Edges through which we discover new vertices.

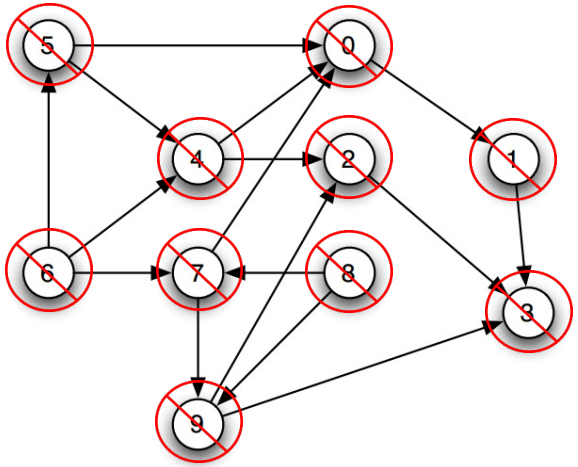
The order of visiting the vertices is not unique since the neighbors of a vertex can be process in any order.

Topological Sorting: Algorithm

At least one

Observation. If there are no directed cycles, there must be a vertex with no incoming edges.

We can safely make such a vertex the first vertex in our ordering.



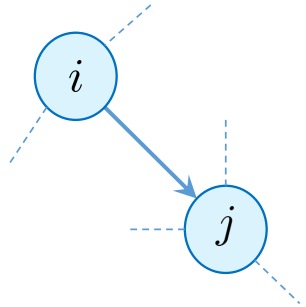
Exercise. *How do we implement this algorithm so that it runs in $O(m+n)$ time?*

Topological Sorting: Algorithm

Let's do DFS until all vertices are visited.

```
for each vertex v:
    if v is not "visited":
      DFS(v)
```

If there is a directed edge from i to j, what can we say about the finish times of DFS at i and j?



Two cases:

1. i is visited before j
2. j is visited before i

In other words, in DFS, i is always visited before j

Observation. In both cases DFS at j finishes before DFS at i.

$\Longrightarrow$ The reverse order of finish times gives a feasible order!

Topological Sorting: Algorithm

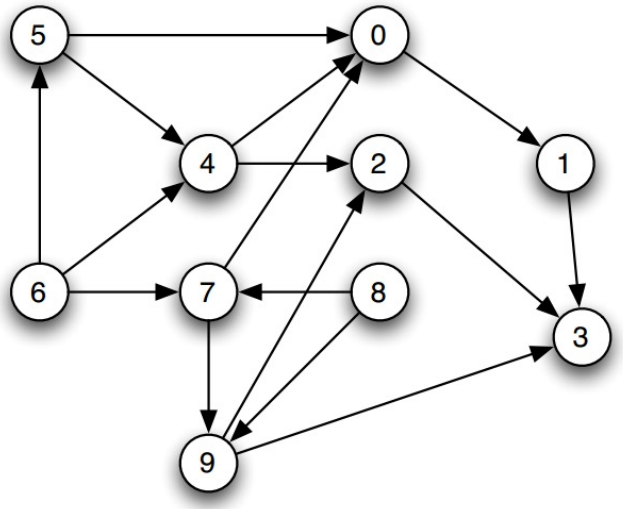
```
time = 0
```
can be removed

Makes sure all are visited

```
for each vertex v:
    if v is not "visited":
        DFS(v)

DFS(u):
    visit u and mark it "visited"

    time += 1
    u.start = time
```
can be removed

Visits the nodes that are connected to the starting node

```
    for each edge u->v:
        if v is not "visited":
            DFS(v)

    time += 1
    u.finish = time
```
can be removed

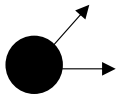


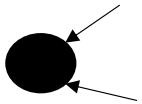
Say we start with u=6, then nodes are visited in this order:
6,4,0,1,3,2,5,7,9,8
And finish in this order
3,1,0,2,4,5,9,7,6,8
8,6,7,9,5,4,2,0,1,3

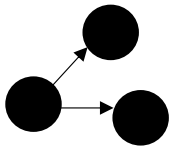
Tasks can be executed in the reverse order of finish times.

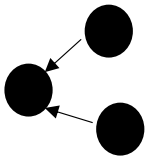
DIRECTED GRAPHS

- What if the directed graph has cycles?
- Definitions:

Source node 

Sink node 

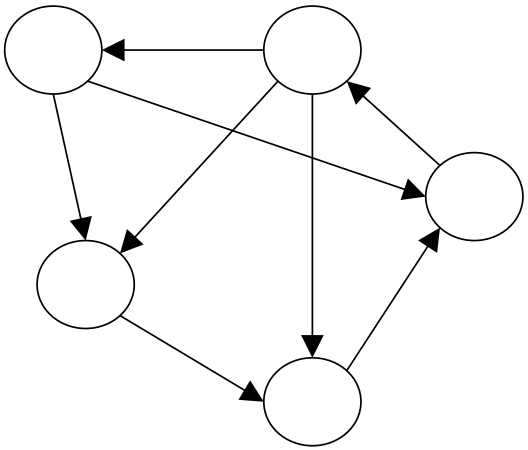
G 

G^R 

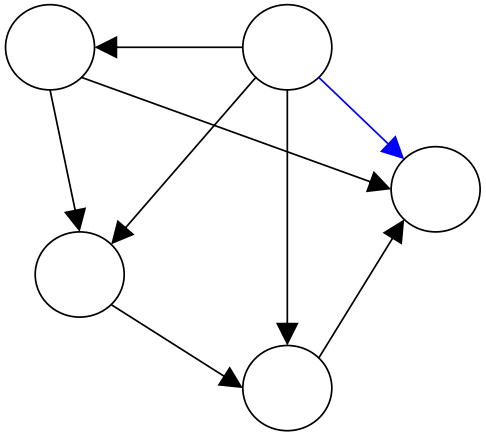
Strong Connectivity

In a directed graph G, u and v are strongly connected if there is a directed path from u to v and a directed path from v to u.

A graph is strongly connected if each pair of vertices are strongly connected.



strongly connected



not strongly connected

Strong Connectivity

Given a graph G in adjacency list representation, how do you check if it is strongly connected?

Idea. Consider any vertex v.

If the graph is strongly connected:

(i) all other vertices should be reacheable from v

(ii) v should be reacheable from all other vertices

If any vertex v is connected to all other vertices, then the graph is strongly connected

Observation. It suffices to verify this for any single vertex v.

How do we check these for a vertex v?

v connected to all and all connected to v

(i) : DFS on G

(ii) : DFS on G^R obtained from G by reversing all edges.

Strong Connectivity

Algorithm.

Do a DFS from any vertex v.

If all vertices are **not** visited:

report that the graph is *not* strongly connected

} $O(m+n)$ *time*

Obtain G^R from G by reversing all edges in G. ← *How? Time?* $O(m+n)$

Do a DFS on G^R starting at v.

If all vertices are **not** visited:

report that the graph is *not* strongly connected

Report that the graph *is* strongly connected.

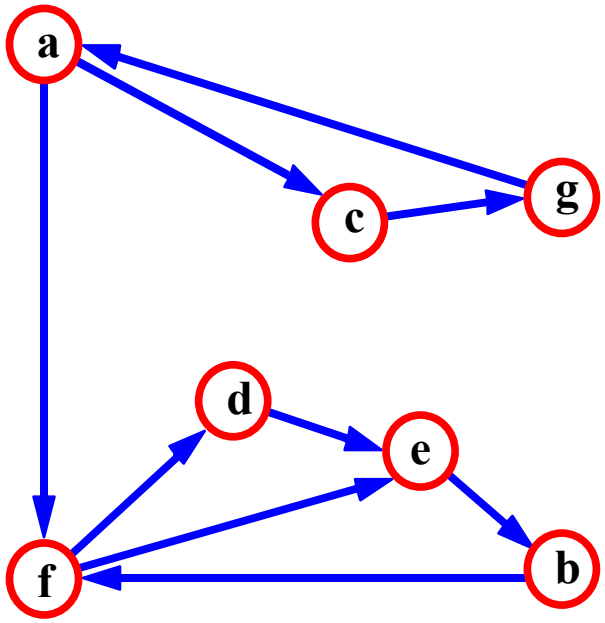
} $O(m+n)$ *time*

Complexity?

Total time. $O(m+n)$ where $m = \#\text{vertices}$, $n = \#\text{edges}$ in G.

Strong Connectivity

What can you say about components?



{ a , c , g }

{ f , d , e , b }

A **strongly connected component** of G is maximal subset of the vertices so that each pair of vertices in the subset are strongly connected.

By maximal we mean that there is no superset of S which is also satisfies the property.

What are the strongly connected components in the above graph?

Strong Connectivity

Q: Can two strongly connected components share a vertex?

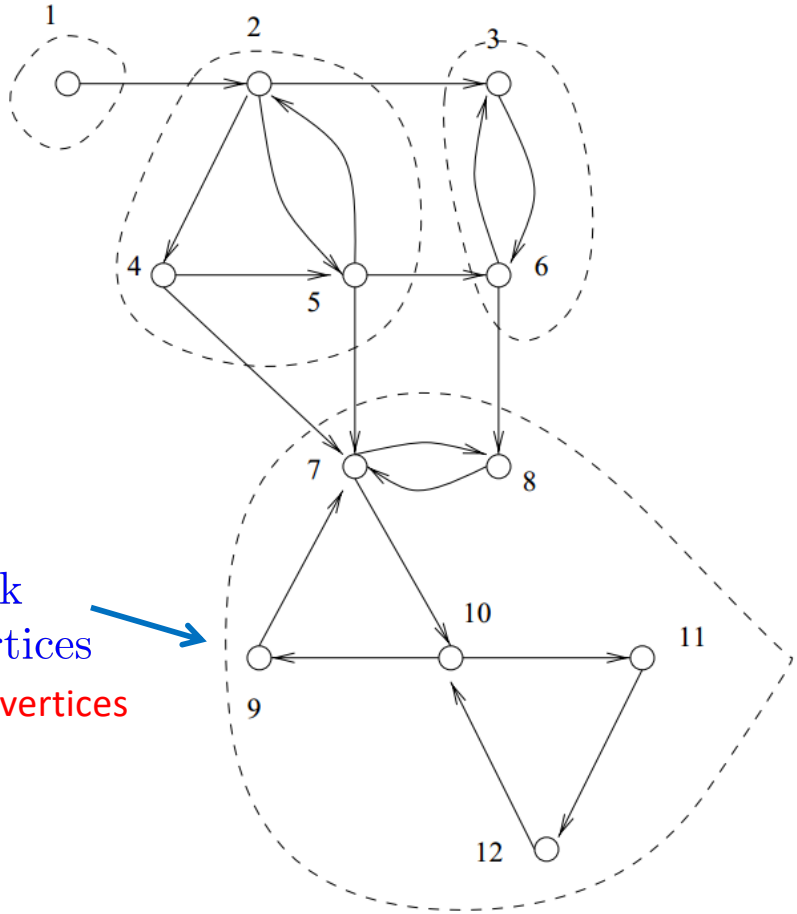
No, since then all vertices in either component are reacheable from each other.

How do you compute the strongly connected components of a given directed graph?

Input: A directed graph in adjacency list representation

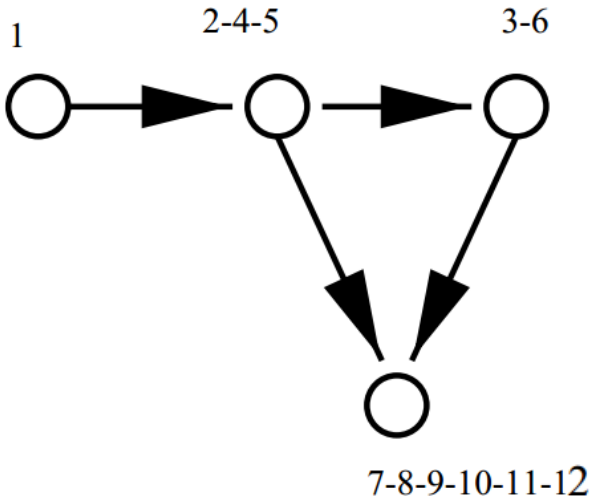
Output: Partition of the vertices into sets so that each set defines a strongly connected component.

Strong Connectivity



Sink Vertices

No outgoing vertices



directed graph on the strongly connected components

This graph is acyclic. Why?

There is always a sink vertix is DAG

Which vertices do we visit if we do a dfs from one of the sink vertices?
We see exactly the vertices in the strongly connected component of that vertex.

Strong Connectivity

Idea: Find a sink vertex v.

Do DFS from v to figure out vertices reacheable from v.
This gives us one strongly connected component.

Repeat the procedure on the rest of the graph.

Question: How to find a sink vertex?

Strong Connectivity

It is hard to figure out a sink vertex directly.

But, it is easy to find a *source* vertex, i.e. a vertex from a strongly connected component with no incoming edges from other components.

Claim: The node with the largest finish time is a source vertex.

Proof:

Let C and C' be two strongly connected components and suppose that there is an edge from a node in C to a node in C'.

Two cases: either C is visited first by the DFS or C' is visted first.

In either case, the node of C that is visited first is the last one to finish among the nodes in C and C'.

Strong Connectivity

Proof(continued)

Let v be the first vertex to be visited among all vertices in C and C'.

If v lies in C', then a DFS at v visits all vertices in C' but since there is no path from C' to C, no vertex in C is visited until we finish with all vertices in C'.

If v lies in C, then the DFS at v eventually visits all vertices in C' and we finish DFS at all vertices at C' before finishing the DFS at v.

So, among the vertices in C and C' the vertex with the largest finish time lies in C.

This means that the vertex with the largest finish time among all vertices in the graph is a source vertex. That is, it lies in a component with no incoming edge from other components.

■

Strong Connectivity

Ok, but how do we find a sink vertex?

Find a source vertex in G^R!

Note that the connected components in G and G^R are the same!

So, to find a sink node

Do a DFS on the G^R and record the finish times.

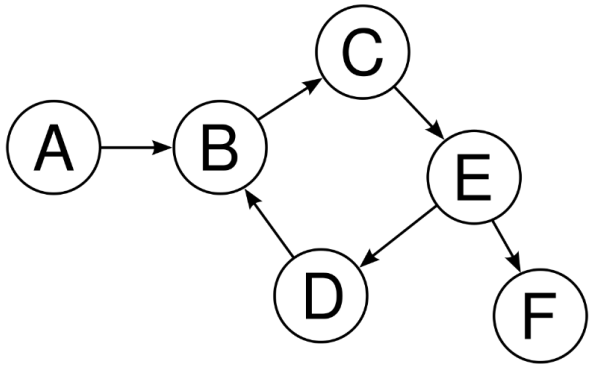
Let $v_1, v_2, \cdots, v_n$ be the ordering of the vertices with decreasing finish times. Then we start with v_1

Homework:

- Prove that v_1 is a sink node
- Continue the algorithm and find complexity (answer on next slides)

Strong Connectivity

Consider the reversed graph below. Do a DFS on the graph and record start and finish time. Assume we start at E.



G^R

DFS(E) will execute first:

Node	start	Finish
E	1	10
D	2	7
B	3	6
C	4	5
F	8	9

DFS(A) will then execute

A	10	11

The time order is: A, E, F, D, B, C

Note that A is a source node in G^R and sink node in G.

Strong Connectivity

Algorithm:

Do a DFS on the G^R and record the finish times.

Let $v_1, v_2, \cdots, v_n$ be the ordering of the vertices with decreasing finish times.

This ordering can be obtained using a stack.

```
for (i = 1; i <= n; ++i) {
    if(v_i is not visited){
        DFS(v_i);
    }
}
```

This DFS is in the original graph G

The vertices visited by this DFS form a strongly connected component.