Algorithm Mining

- Algorithms designed for one problem are often usable for a number of other computational tasks, some of which seem unrelated to the original goal

- Today, we are going to look at how to use the depth-first search algorithm to solve a variety of graph problems

Algorithm Mining techniques

- Deeper Analysis: What else does the algorithm already give us?
- Augmentation: What additional information could we glean just by keeping track of the progress of the algorithm?
- Modification: How can we use the same idea to solve new problems in a similar way?
- Reduction: How can we use the algorithm as a black box to solve new problems?

Graph Reachability and DFS

- Graph reachability: Given a directed graph G, and a starting vertex v, return an array that specifies for each vertex u whether u is reachable from v

- Depth-First Search (DFS): An efficient algorithm for Graph reachability
- Breadth-First Search (BFS): Another efficient algorithm for Graph reachability.

DFS as recursion

- procedure explore(G, v)
- Input: graph $G = (V, E)$; node v in V output:
- Output: array visited[u]
- 1. visited[v] = true
- 2. for each edge $(v, u) \in E$ do:
- if not visited[u]: explore(G, u)

Key Points of DFS

- No matter how the recursions are nested, for each vertex u, we only run explore(G, u) ONCE, because after that, it is marked visited. (We need this for termination and efficiency)

- On the other hand, we discover a path to a new destination, we always explore all new vertices reachable

(We need this for correctness, to guarantee that we find ALL the reachable vertices)

DFS as iterative algorithm

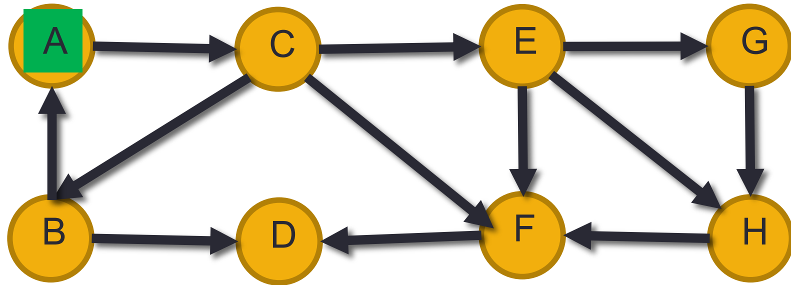
procedure explore(G: directed graph, v: vertex)

Initialize array visited[u] to False
Initialize stack of vertices F, PUSH v; Visited[v]=True;
While F is not empty:
  v=Pop;
  For each neighbor u of v (in reverse order):
    If not visited[u]:
      Push u; visited[u] = True;

Return visited

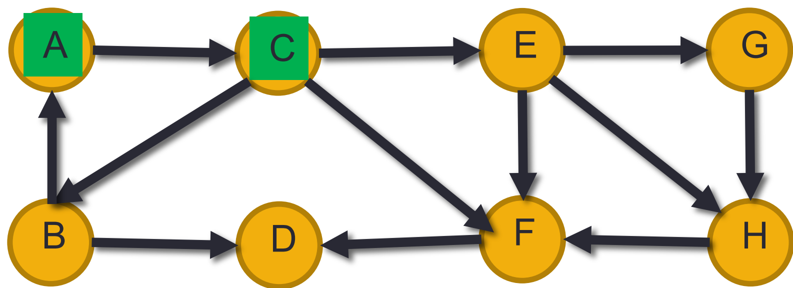
```
procedure explore(G = (V, E), s)
visited(s)=true
for each edge (s, u):
        if not visited(u):
                explore(G, u)
```

DFS on Directed Graphs



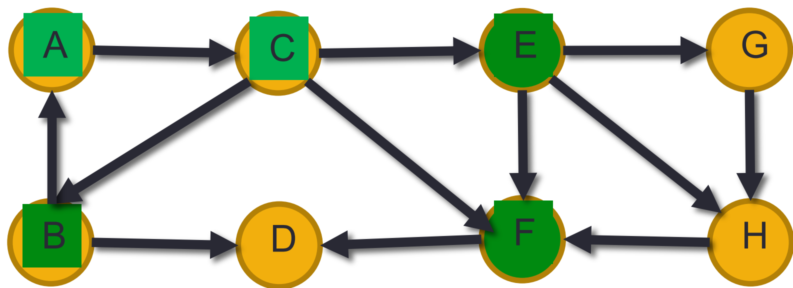
F = A

DFS on Directed Graphs



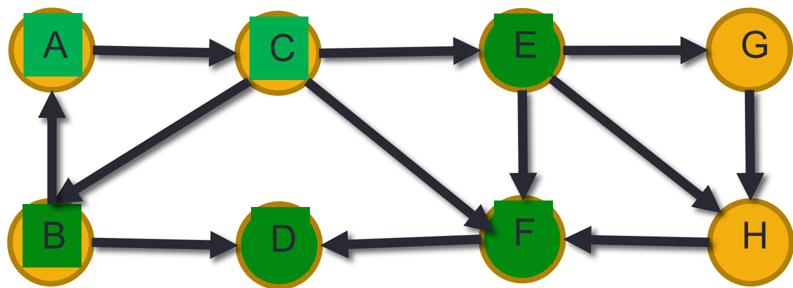
F= A. Pop A. Neighbors of A = (C)
Push C, visited C == True
F= C

DFS on Directed Graphs



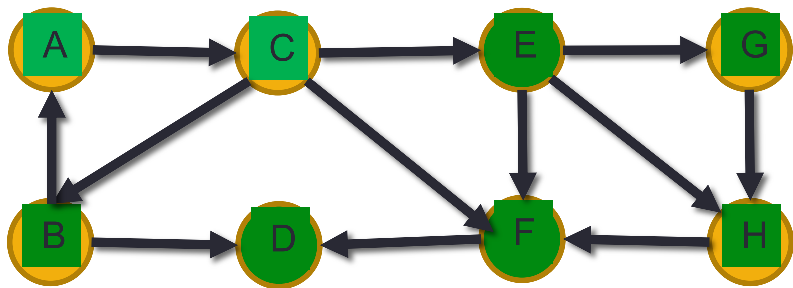
F= C. Pop C. Neighbors of C = (F,E,B)
Push F, Push E, Push B,
F= B, E, F

DFS on Directed Graphs



F= B,E,F. Pop B. Neighbors of B = (D,A)
Push D ,
F= E, F, D

DFS on Directed Graphs



F= E,F, D Pop E. Neighbors of E = (H,G,F)
Push G, H
F= F, D, G, H. Pop, Pop, Pop, Pop

Running time of DFS

```
procedure explore(G = (V, E), s)
visited(s)=true
for each edge (s, u):
        if not visited(u):
                explore(G, u)
```

DFS as iterative algorithm

procedure explore(G: directed graph, v: vertex)

Initialize array visited[u] to False. O(|V|)
Initialize stack of vertices F, PUSH v; Visited[v]=True; O(1)
While F is not empty: done at most |V| times, once per v
v=Pop;
For each neighbor u of v (in reverse order): $O(1 + \deg(v)) = O(|V|)$
If not visited[u]:
Push u; visited[u] == True;
Return visited.

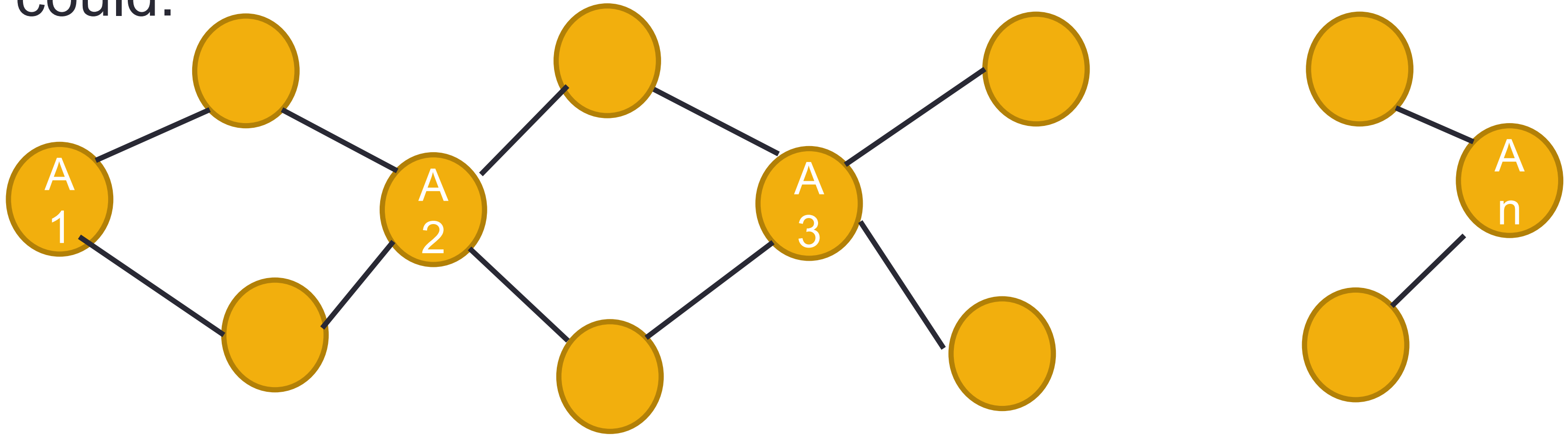
Tighter : Loop runs once for each v, $O(1 + \deg(v))$ time on that loop.
So total time at most : $O(\sum_v 1 + \deg(v)) = O(\,|V| + |E|)$

Complete DFS

- DFS actually just costs O(number of reachable nodes + number of reachable edges). Parts of the graph that weren't found don't cost either.
- So, still in total $O(|V|+|E|)$ time, we can run also keep on running explore from undiscovered vertices, until we've found the whole graph. We usually keep track of which iteration each vertex was discovered in.
- Alternative viewpoint: Add a new vertex with edges to all vertices. Run DFS from the new vertex.

All reachable vertices, not all paths

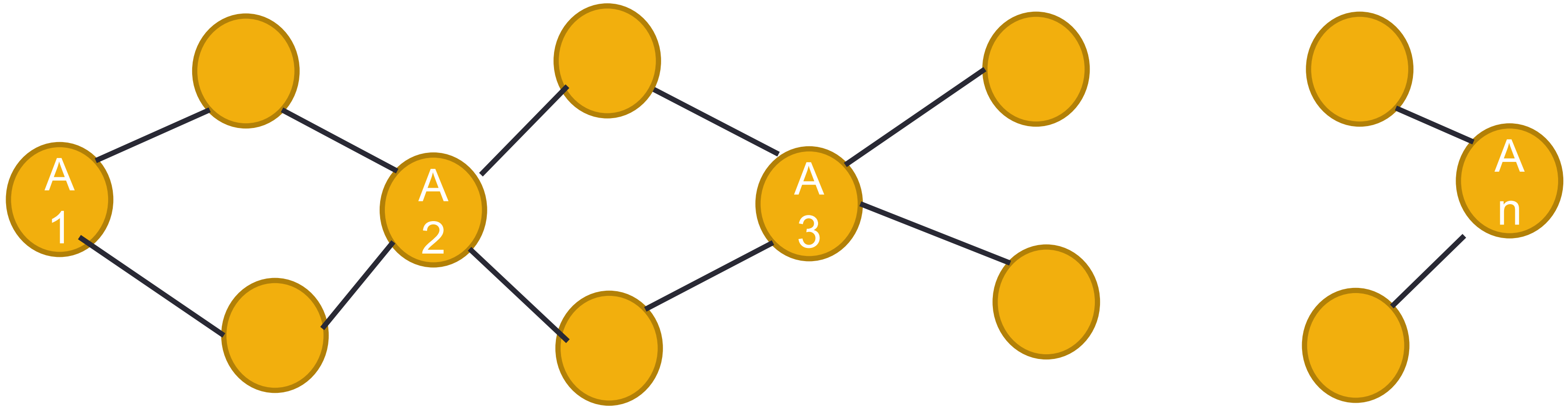
- While DFS finds all the reachable vertices, it doesn't consider all paths between them. No feasible algorithm could.



How many paths from A1 to An?

All reachable vertices, not all paths

- While DFS finds all the reachable vertices, it doesn't consider all paths between them. No feasible algorithm could.



2^{n-1} paths from A1 to An

Finding paths: the DFS tree

- After the DFS, we know which vertices are reachable, but not how to get there

How long could a path in a graph be?

How about a simple path?

How many paths do we have to find?

Finding paths: the DFS tree

- After the DFS, we know which vertices are reachable, but not how to get there
- We have up to |V|-1 paths to find, and each path can be up to length |V|.

Synergy

- After the DFS, we know which vertices are reachable, but not how to get there

- We have up to |V|-1 paths to find, and each path can be up to length |V|.

- Sometimes, doing something similar many times costs less than doing it from scratch each time.

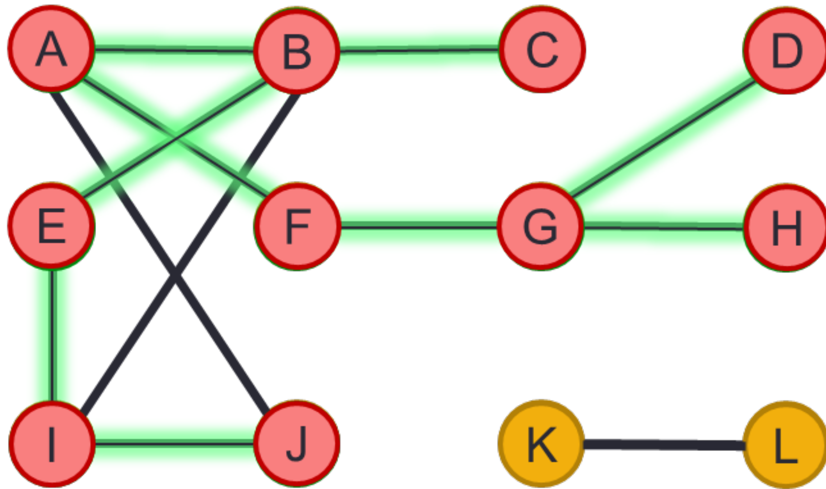
For DFS, the paths overlap, and form a |V|-1 edge tree

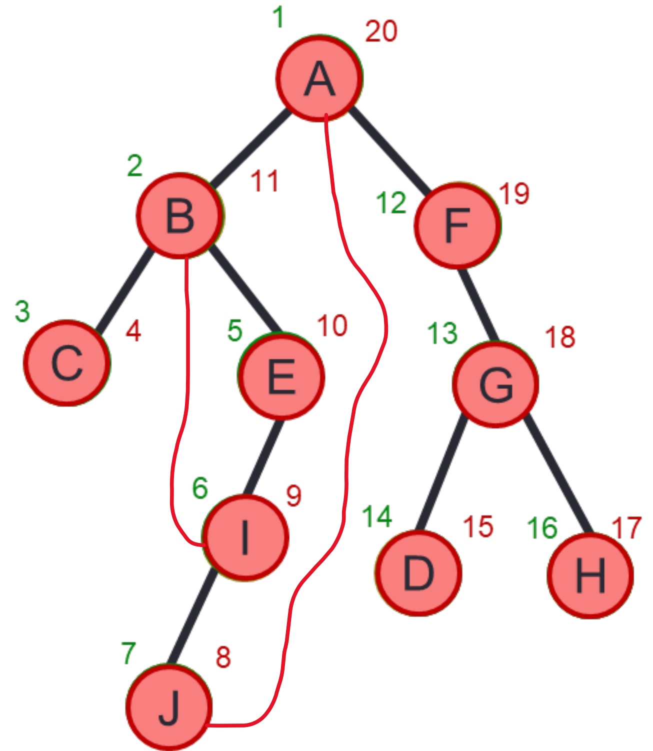
DFS augmented to create DFS tree

- procedure explore(G, v)
- Input: graph $G = (V, E)$; node v in V output:
- Output: array visited[u]; parent[u]
- 1. visited[v] = true
- 2. for each edge $(v, u) \in E$ do:
- if not visited[u]: parent[u]=v; explore(G, u);

keeping track of paths

Example:





DFS augmtd. with pre, post numbers

- procedure explore(G, v)
- Input: graph $G = (V, E)$; node $v \in V$ output: count starts at 1
- Output: array visited[u]; parent[u]; pre[u]; post[u]
- 1. visited[v] = true ;
- 2. for each edge $(v, u) \in E$ do:
- if not visited[u]: parent[u]=v; pre[u]=count; count++; explore(G, u);
- 3. post[v] = count, count++

Depth first search

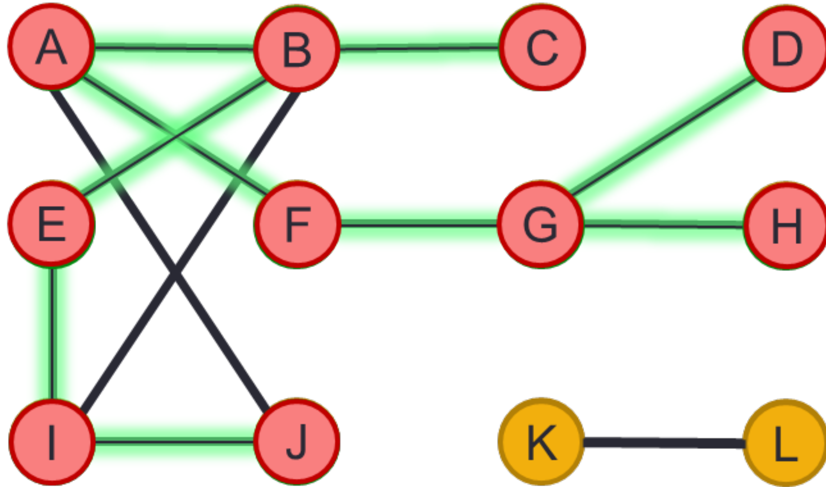
```
procedure DFS(G)
cc = 0
clock = 1
for each vertex v:
        visited(v) = false
for each vertex v:
        if not visited(v):
                cc++
                explore(G,v)
```

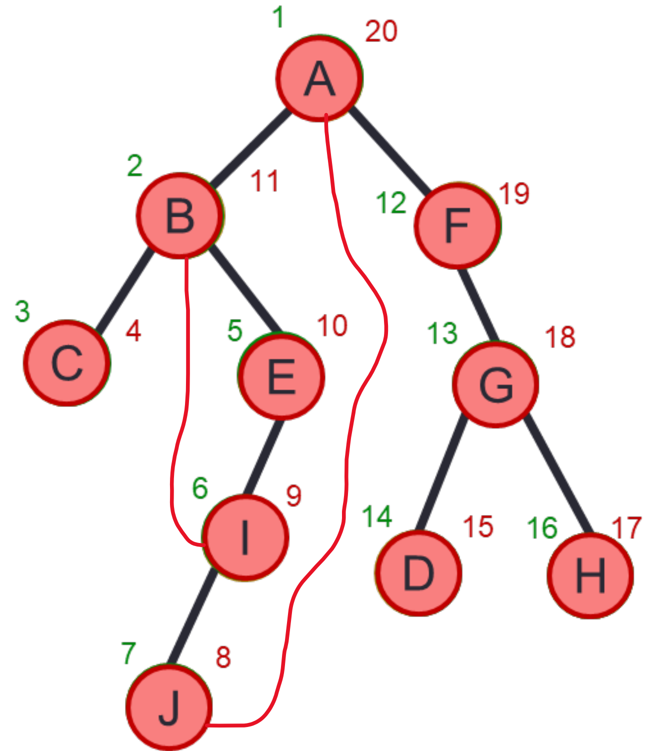
```
procedure previsit(v)
        pre(v)=clock
        clock++
```

```
procedure post visit(v)
        post(v)=clock
        clock++
```

keeping track of paths

Example:





Inferring relative position in tree

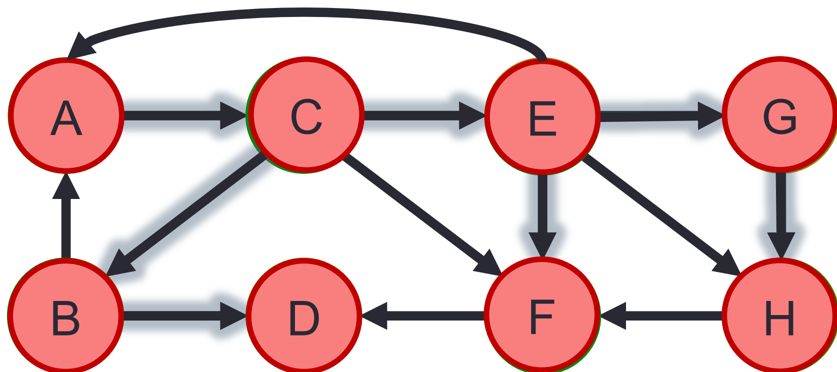
- u is below v in the DFS tree iff pre(v) < pre(u) and post(u) < post(v).
  - **In this case, an edge from u to v creates a cycle**

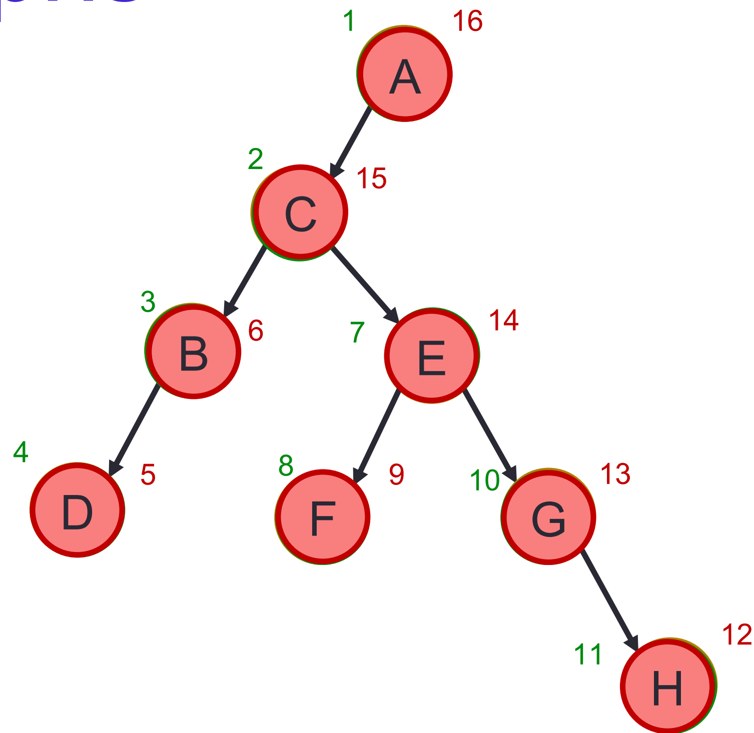
- u is to the right of v iff pre(v) < pre(u) and post(v) < post(u)

Edge types (directed graph)

- Tree edge: solid edge included in the DFS output tree
- Back edge: leads to an ancestor
- Forward edge: leads to a descendent
- Cross edge: leads to neither anc. or des.: always from right to left

DFS on Directed Graphs



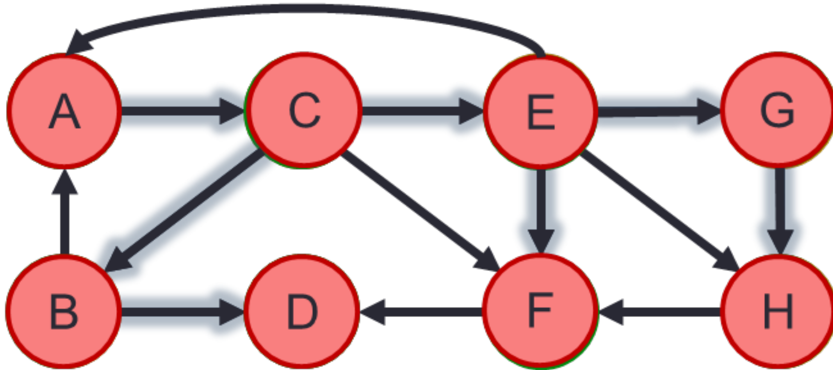


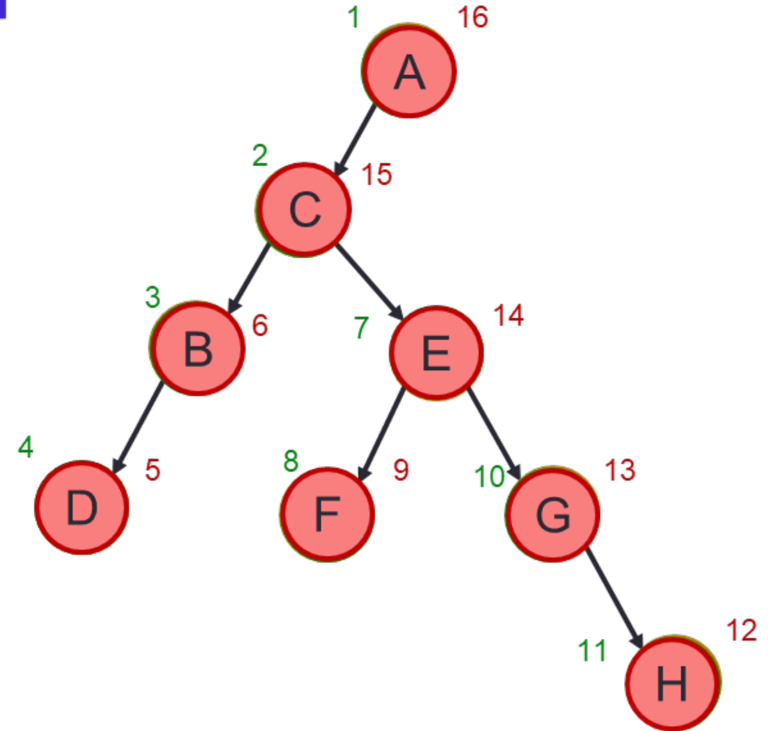
Edge types and pre/post numbers

The different types of edges can be determined from the pre/post numbers for the edge (u, v)

- (u, v) is a tree/forward edge then $pre(u) < pre(v) < post(v) < post(u)$
- (u, v) is a back edge then $pre(v) < pre(u) < post(u) < post(v)$
- (u, v) is a cross edge then $pre(v) < post(v) < pre(u) < post(u)$

DFS on Directed Graphs





Cycles in Directed Graphs

- A cycle in a directed graph is a path that starts and ends with the same vertex

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0$$

$$A \to C \to E \to A$$

A directed graph has a directed cycle iff its dfs output tree has a back edge

Proof: $\rightarrow$

Suppose G has a cycle:

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0$$

A directed graph has a directed cycle iff its dfs output tree has a back edge

Proof: $\rightarrow$

Suppose G has a cycle:

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0$$

Suppose v_0 is the first vertex to be discovered.

(What does that mean about v_0?)

A directed graph has a directed cycle iff its dfs output tree has a back edge

Proof: →

Suppose G has a cycle:

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0$$

Suppose v_0 is the first vertex to be discovered. (the vertex with the lowest pre-number.)

All other v_j are reachable from it and therefore, they are all descendants in the DFS tree.

A directed graph has a directed cycle iff its dfs output tree has a back edge

Proof: $\rightarrow$

Suppose G has a cycle:

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0$$

Suppose v_0 is the first vertex to be discovered. (the vertex with the lowest pre-number.)

All other v_j are reachable from it and therefore, they are all descendants in the dfs tree.

Therefore the edge (v_k, v_0) is a back edge.

A directed graph has a directed cycle iff its dfs output tree has a back edge

Proof: $\leftarrow$

Suppose (b, a) is a back edge.

A directed graph has a directed cycle iff its dfs output tree has a back edge

Proof: $\leftarrow$

Suppose (b, a) is a back edge.

Then by definition a is a ancestor of b so there is a path from a to b in the DFS output tree.

A directed graph has a directed cycle iff its dfs output tree has a back edge

Proof: $\leftarrow$

Suppose (b, a) is a back edge.

Then by definition a is a ancestor of b so there is a path from a to b in the DFS output tree.

Directed Acyclic Graphs (DAG)

- A directed graph without a cycle is called <u>acyclic</u>. (DAG)

Corollary:

A directed graph G is a DAG if and only if it's DFS output tree does not have any back edges.

How to spot a DAG?

Step 1: perform dfs on the graph

Step 2: loop through each edge to see if it is a back edge.

i.e.:

for each edge (u, v),

if $\mathrm{pre}(v) < \mathrm{pre}(u) < \mathrm{post}(u) < \mathrm{post}(v)$:

return “not DAG”

return “DAG”

Linearization aka Topological Sort

- Is it possible to order the vertices such that all edges go in only one direction?
- For what types of DAGs is this possible?
- How do we find such an ordering?

Property of DAGS

Theorem: every edge in a DAG goes from a higher post number to lower post number.

Property of DAGS

Theorem: every edge in a DAG goes from a higher post number to lower post number.

proof:

suppose (u, v) is an edge in a DAG then it can’t be a back edge, therefore it can only be a forward edge/tree edge or a cross edge.

All of which have the property that post(v) < post(u).

Corollary: Sorting by post numbers is a topological sort

Property of DAGS

Linearization of a DAG:

Since we know that edges go in the direction of decreasing post numbers, if we order the vertices by decreasing post numbers then we will have a linearization

procedure **linearize**(DAG $G = (V, E)$)

run DFS(G)

return list of vertices in decreasing order of post numbers (by putting at start of list when post number assigned)

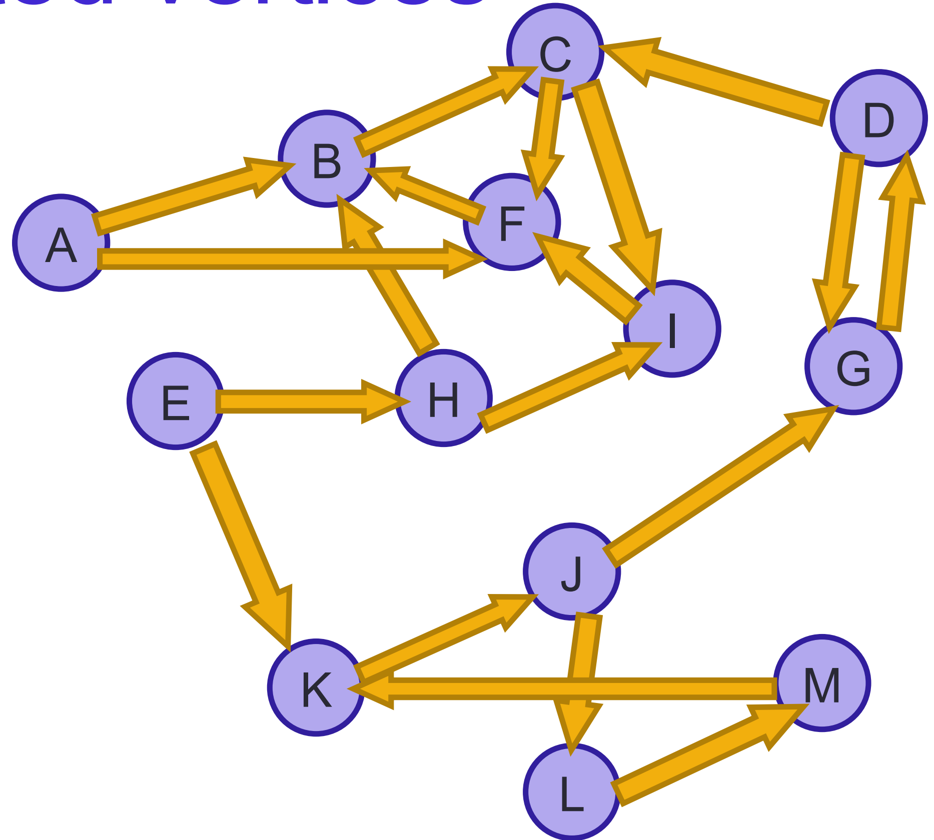
Sources and sinks

- Since all DAGs can be linearized, that means the first vertex in the ordering does not have any edges coming in and the last vertex does not have any edges going out.
- **Definitions:**
  - A vertex with no incoming edges is called a **source**
  - A vertex with no outgoing edges is called a **sink**
- **Theorem:** All DAGs have at least one source and one sink.

Strongly connected vertices

Two vertices u and v in a directed graph are strongly connected if there exists a path from u to v and a path from v to u.

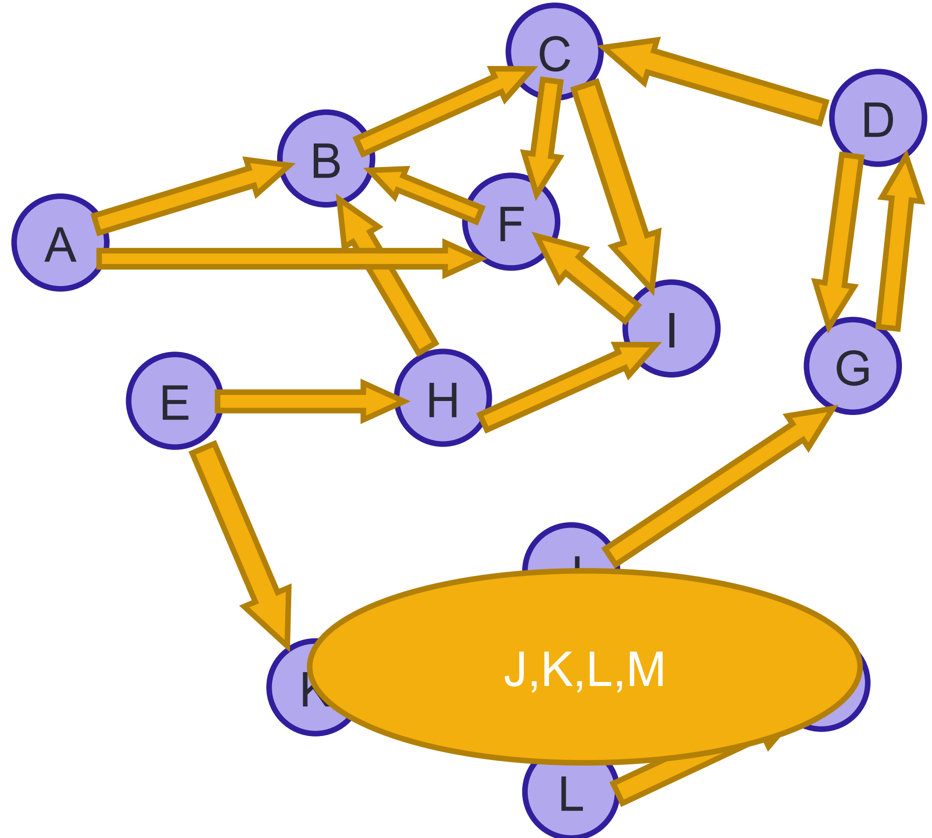
Which vertices are strongly connected to J?



Strongly connected vertices

Two vertices u and v in a directed graph are strongly connected if there exists a path from u to v and a path from v to u.

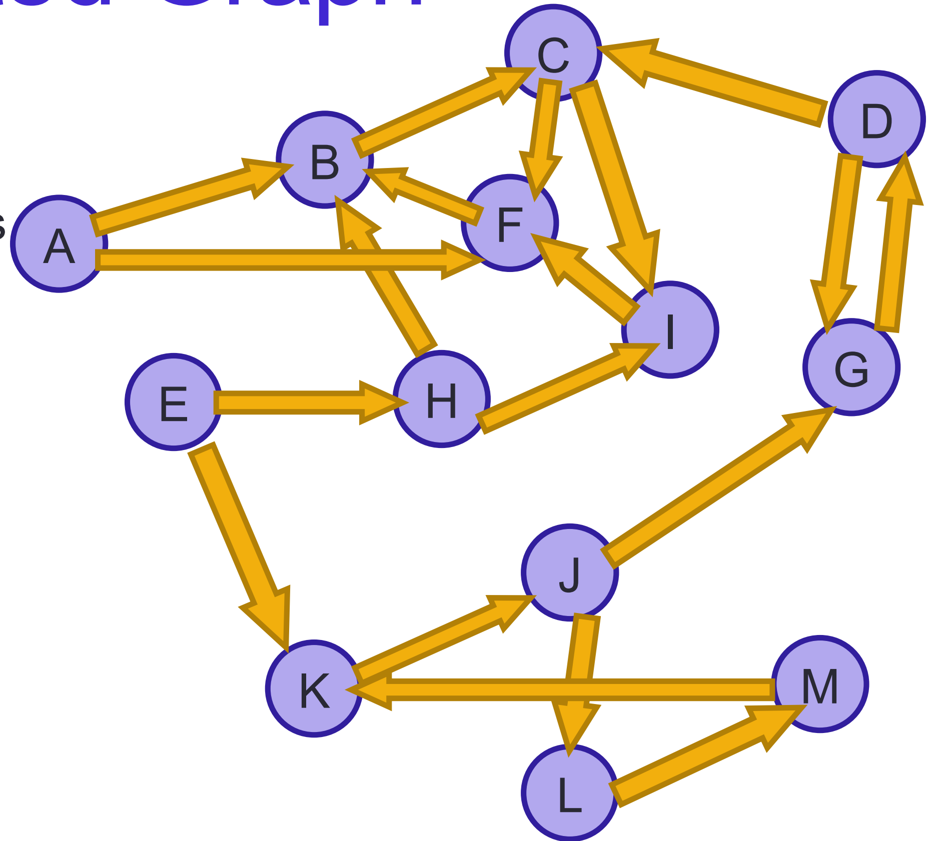
Which vertices are strongly connected to J? {J, K, L, M}



Strongly connected Graph

A graph is called strongly connected if for each pair of vertices v, u there is a path from v to u and a path from u to v.

Is this a strongly connected graph?



Strongly connected components

Consider the relation uRv if u is strongly connected to v.

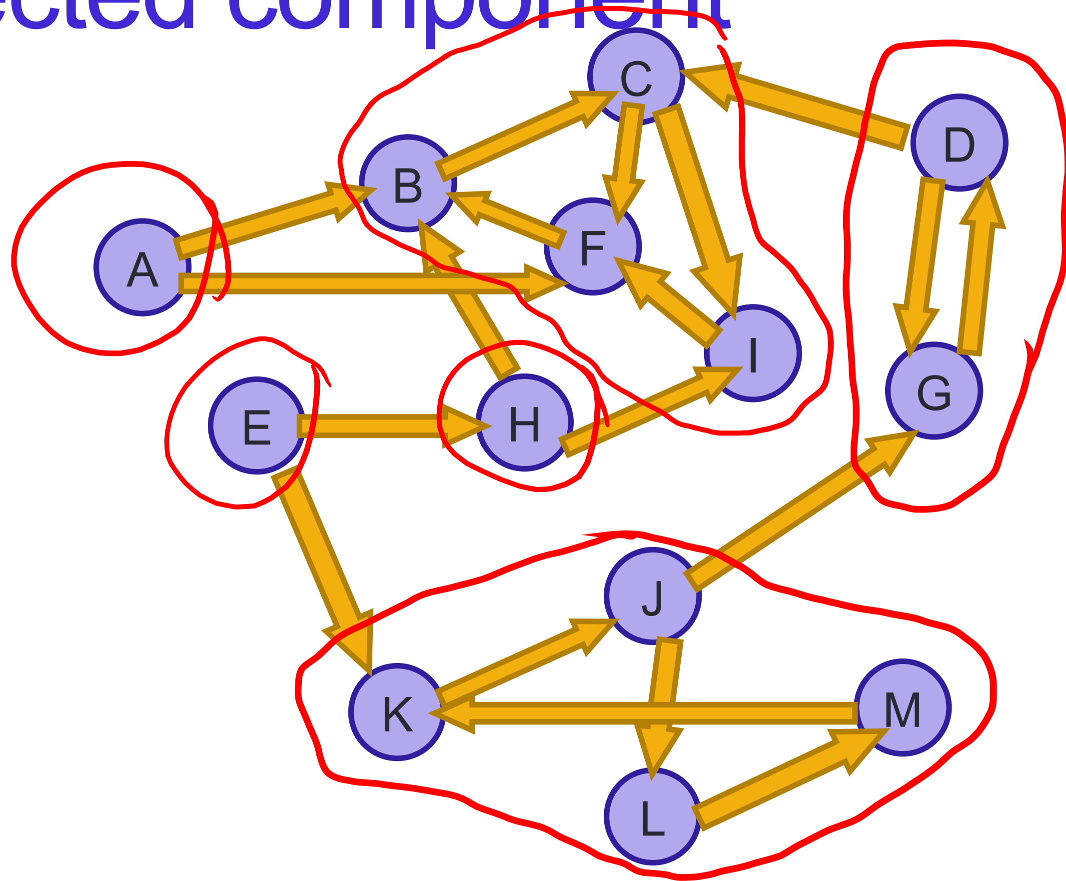
Then R is an equivalence relation. It is reflexive, symmetric and transitive.

So R partitions V, the set of vertices into equivalence classes.

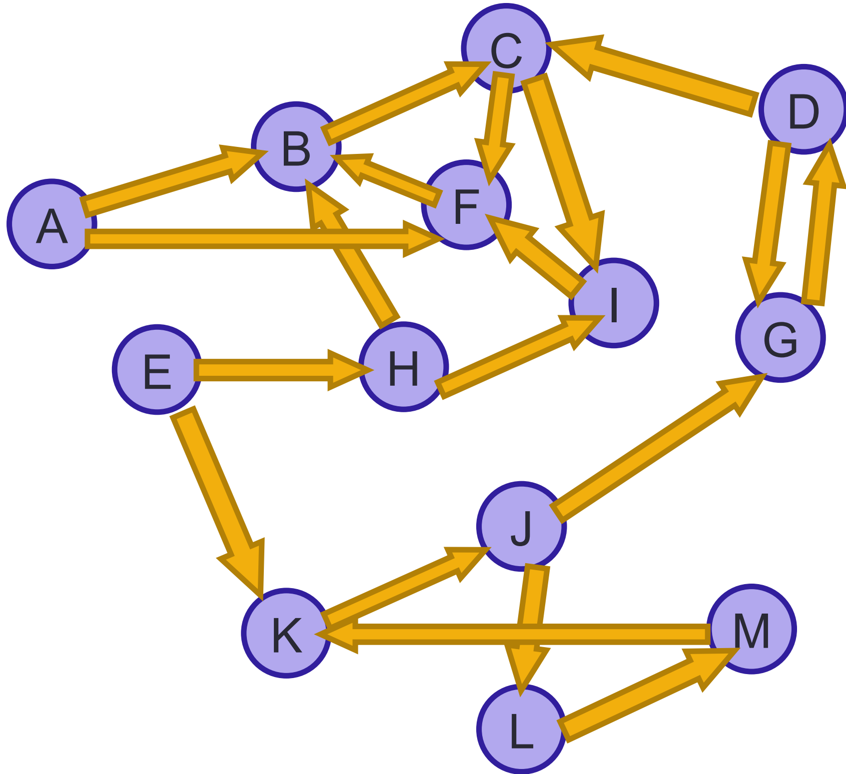
These equivalence classes are called strongly connected components.

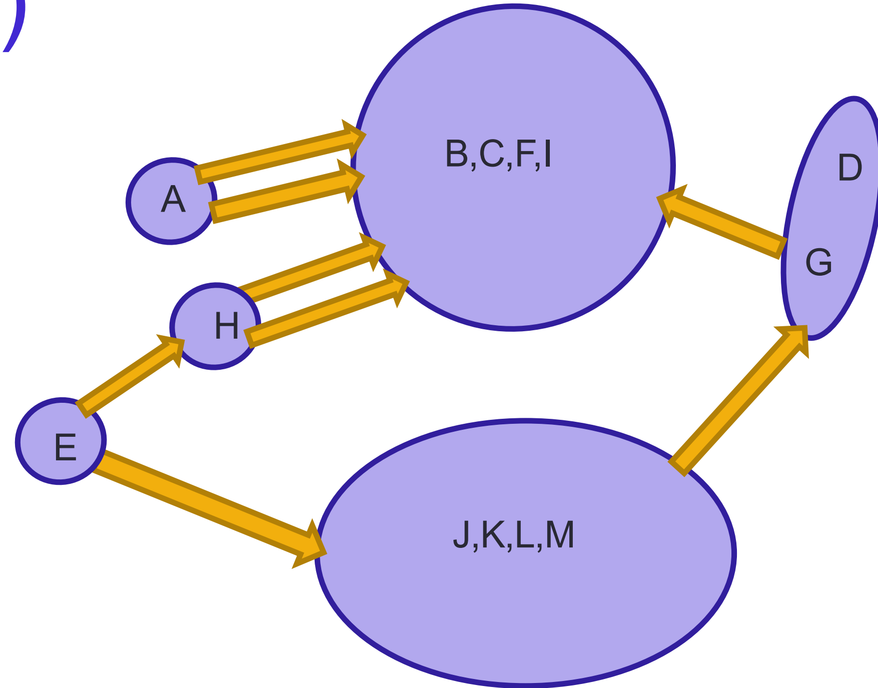
Strongly connected component

What are the strongly connected components of this graph?



Strongly connected components as vertices. (Meta-graph)





Directed Graphs as DAGs of SCCs

Every Directed graph is a DAG of its strongly connected components.

Some SCCs are sink SCCs and some are source SCCs.

Decomposition

There is a linear time algorithm that decomposes a directed graph into its strongly connected components.

If explore is performed on a vertex u, then it will visit only the vertices that are reachable by u.

What vertices will be visited when explore is performed on u if u is in a sink SCC?

Sink SCCs

If explore is performed on a vertex that is in a sink SCC, then only the vertices from that SCC will be visited.

This suggests a way to look for SCCs.

- Start explore on a vertex in a sink SCC and visit its SCC.
- Remove the sink SCC from the graph and repeat.

Source SCCs

Ideally we would like to find a vertex in a sink SCC.

Unfortunately, there is not a direct way to do this.

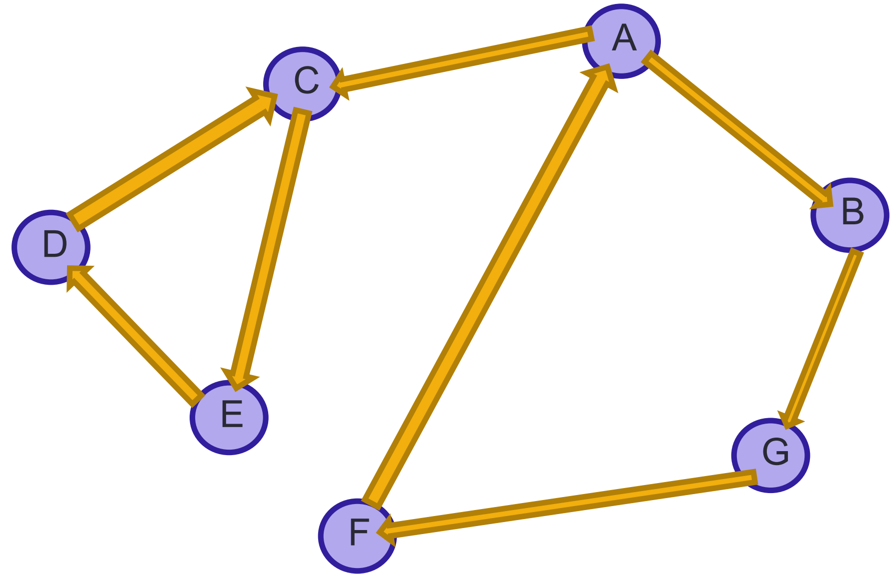
Source SCCs

However, there is a way to find a vertex in a source SCC.

The vertex with the greatest post number in any DFS output tree belongs to a source SCC.

The vertex with the least post number in a dfs output does not necessarily belong to a sink SCC.

Example of low post number not in a sink.



Vertices in Source SCCs

The vertex with the greatest post number in any DFS output tree belongs to a source SCC.

To prove this, we will state a more general property:

If C and C' are strongly connected components and there is an edge from a vertex in C to a vertex in C' then the highest post number in C is greater than the highest post number in C'

Vertices in S SCCs

The vertex with the greatest post number in any DFS output tree belongs to a source SCC.

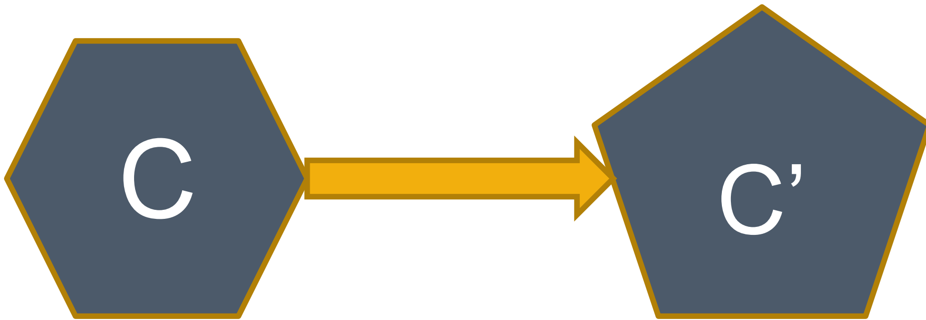
To prove this, we will state a more general property:

If C and C' are strongly connected components and there is an edge from a vertex in C to a vertex in C' then the highest post number in C is greater than the highest post number in C'

Proof

Case 1: DFS searches C before C':

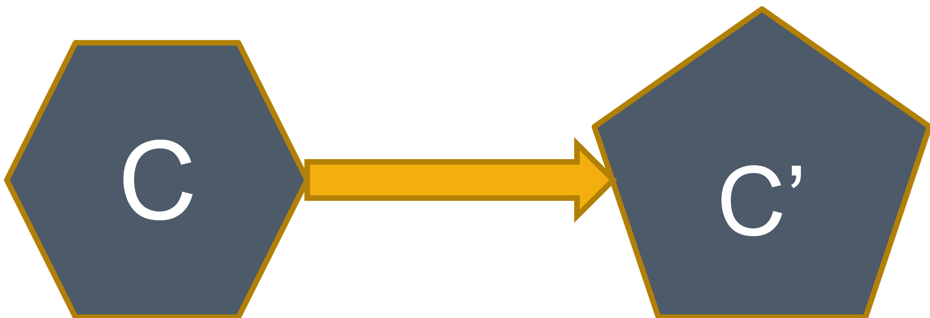
Then at some point dfs will cross into C' and visit every edge in C' then it will retrace its steps until it gets back to the first node in C it started with and assign it the highest post number



Proof

Case 2: DFS searches C' before C:

Then DFS will visit all vertices of C' before getting stuck and assign a post number to all vertices of C'.
Then it will visit some vertex of C later and assign post numbers to those vertices.



Corollary

The strongly connected components can be linearized by arranging them in decreasing order of their highest post numbers.

How to find sink SCCs

Given a graph G, let G^R be the reverse graph of G.
Then the sources of G^R are the sinks of G,

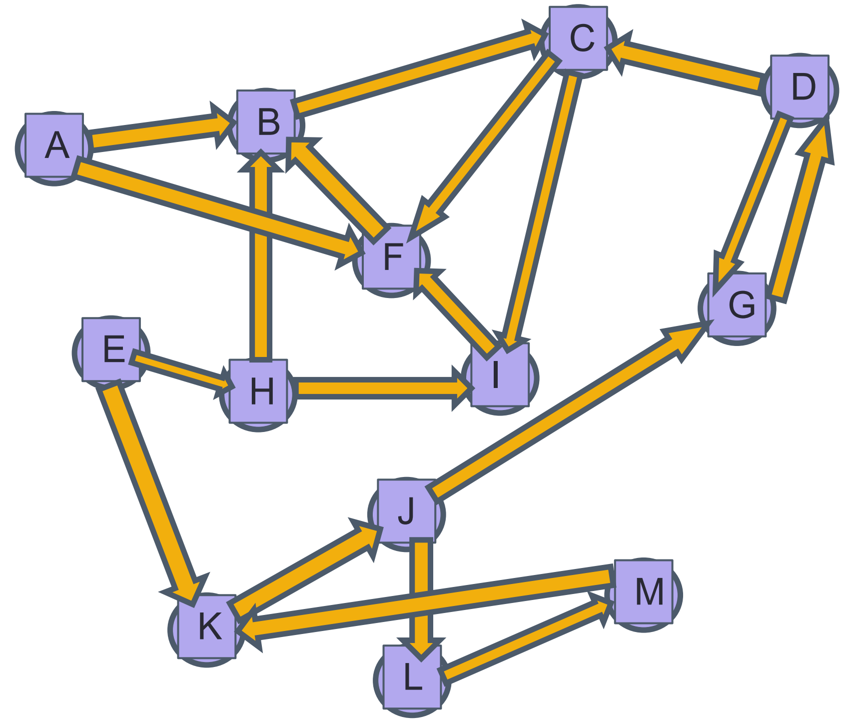
So if we perform DFS on G^R then the vertex with the highest post number is in a source. This means that this vertex will be in a sink of G.

So start with this vertex and explore the SCC.

Then the vertex with the next greatest post number in G^R is in the next SCC in linear order so start with that one next.

How to decompose a graph into its SCCs:

- Construct G^R.
- Run DFS on G^R and keep track of the post numbers.
- Run DFS on G and order the vertices in decreasing order of the post numbers from the previous step. Every time DFS increments **cc**, you have found a new SCC!!


A
B
C
D
E
F
G
H
I
J
K
L
M

1 A 2

3 B 26

4 F 25

5 C 20

21 I 24

6 D 19

22 H 23

7 G 18

8 J 17

10 E 11

9 K 16

12 M 15

13 L 14

C

B

D

A

F

I

E

H

G

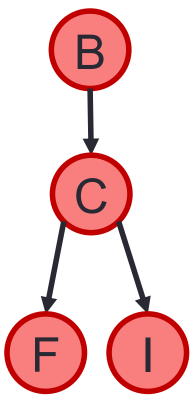
J

K

M

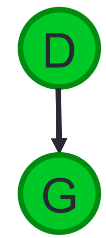
L

cc = 1

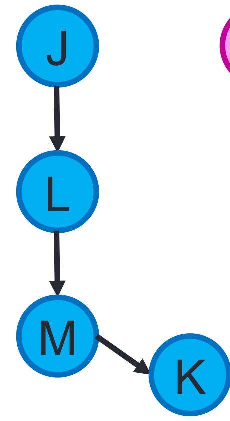


cc = 2

cc = 3



cc = 4



cc = 5

cc = 6

B, F, I, H, C, D, G, J, K, M, L, E, A

How to decompose a graph into its SCCs:

- Run DFS on G^R and keep track of the post numbers.
- Run DFS on G and order the vertices in decreasing order of the postnumbers from the previous step. Every time DFS increments **cc**, you have found a new SCC!!

How long does this take?

I claim it is linear time for each step and so it is linear time in general

DFS is good for

DFS is good for

- Find what vertices can be reached by a given vertex
- Divide an undirected graph into connected components
- Find cycles in graphs (directed or undirected.)
- Find sinks and sources in DAGs
- Topologically sort a DAG
- Make a directed graph into a DAG of its SCCs

DFS is not good for

DFS is not good for

- Finding shortest distances between vertices.