DAGs, DFS and SCC

Lecture 17

Part I

Directed Acyclic Graphs

DAG Properties

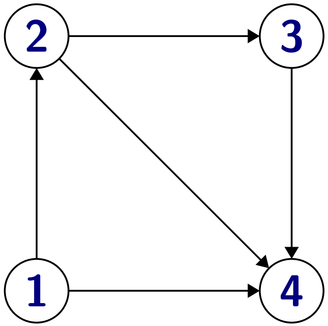
Proposition

Every DAG G *has at least one source and at least one sink.*

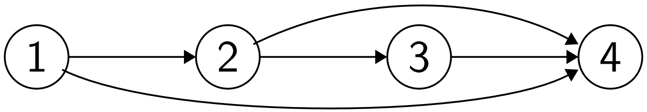
Proposition

A directed graph G *can be topologically ordered iff it is a* DAG.

Topological Ordering/Sorting



Graph G



Topological Ordering of G

Definition

A **topological ordering**/**topological sorting** of $G = (V, E)$ is an ordering $\prec$ on V such that if $(u, v) \in E$ then $u \prec v$.

Informal equivalent definition:

One can order the vertices of the graph along a line (say the x-axis) such that all edges are from left to right.

DAGs and Topological Sort

What does it mean?

DAGs and Topological Sort

What does it mean?

Consider a dependency graph.

Topological ordering

Find an order of events in which all dependencies are satisfied.

DAGs and Topological Sort

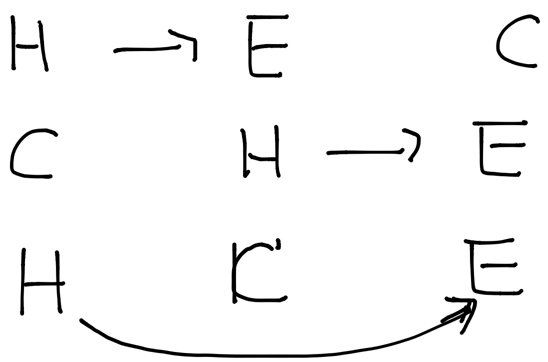
What does it mean?

Consider a dependency graph.

Topological ordering

Find an order of events in which all dependencies are satisfied.

Case 1: DAG. Heat a pizza → eat the pizza, have a Coke.



DAGs and Topological Sort

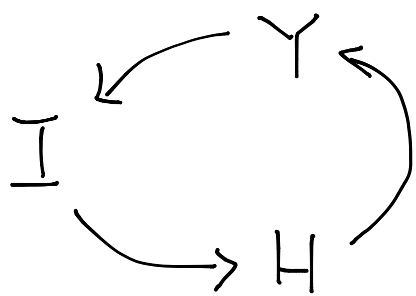
What does it mean?

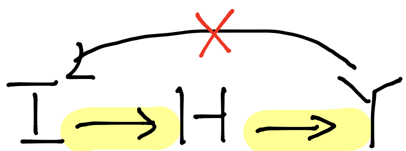
Consider a dependency graph.

Topological ordering

Find an order of events in which all dependencies are satisfied.

Case 1: DAG. Heat a pizza $\longrightarrow$ eat the pizza, have a Coke.
Case 2: Circular dependence.





DAGs and Topological Sort

Lemma

A directed graph G can be topologically ordered only if it is a DAG.

Proof.

Suppose G is not a DAG and has a topological ordering $\prec$. G has a cycle $C = u_1, u_2, \ldots, u_k, u_1$.

Then $u_1 \prec u_2 \prec \ldots \prec u_k \prec u_1$!

That is... $u_1 \prec u_1$.

A contradiction (to $\prec$ being an order).

Not possible to topologically order the vertices. □

DAGs and Topological Sort

Lemma

A directed graph G can be topologically ordered if it is a DAG.

Proof.

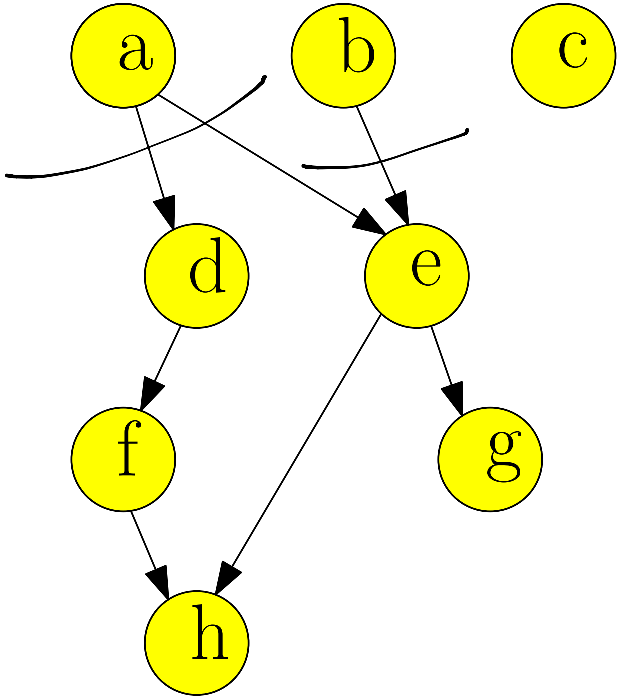
Consider the following algorithm:

1. Pick a source u, output it.
2. Remove u and all edges out of u.
3. Repeat until graph is empty.

Exercise: prove this gives toplogical sort. □

Exercise: show algorithm can be implemented in $O(m + n)$ time.

Topological Sort: Example



c a b d f e h g

DAGs and Topological Sort

Note: A DAG G may have many different topological sorts.

Question: What is a DAG with the largest number of distinct topological sorts for a given number ***n*** of vertices?

$n!$

Question: What is a DAG with the smallest number of distinct topological sorts for a given number ***n*** of vertices?

1

$1 \to 2 \to 3 \to \dots \to n$

Part II

DFS in Undirected Graphs

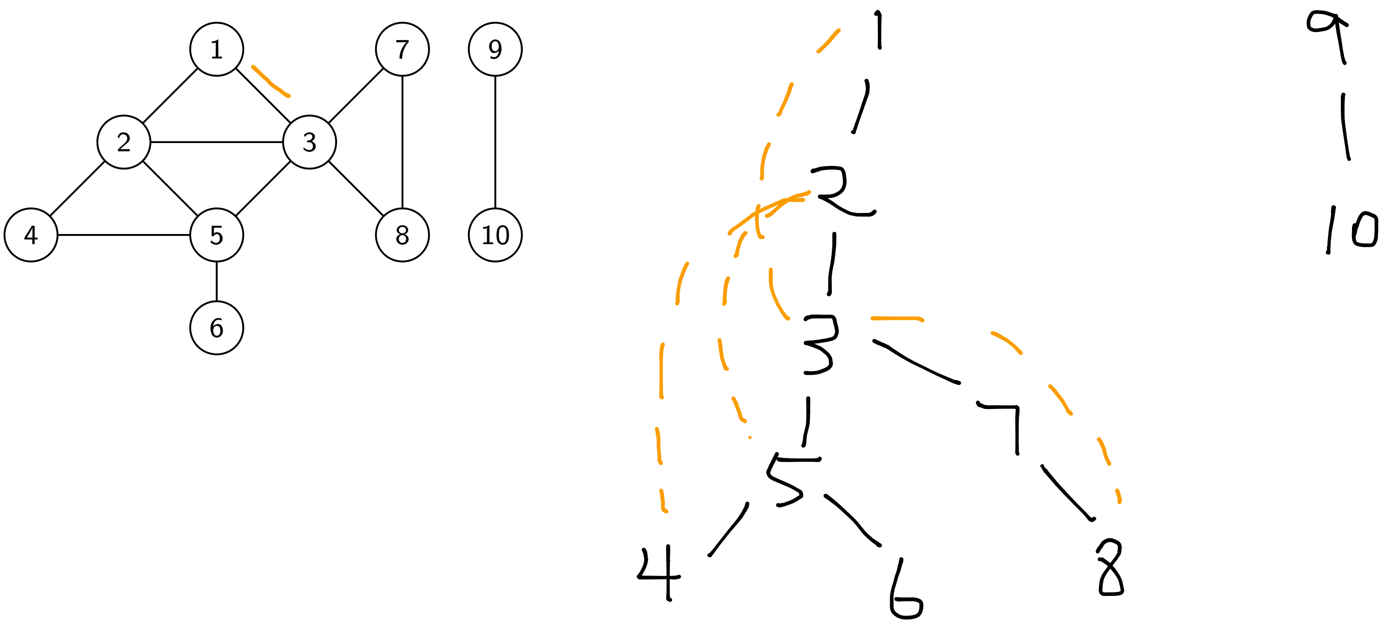
DFS in Undirected Graphs

Recursive version. Easier to understand some properties.

DFS(G)
    for all $u \in V(G)$ **do**
        Mark u as unvisited
        Set pred(u) to null
    T is set to $\emptyset$
    while $\exists$ unvisited u **do**
        DFS(u)
    Output T
	DFS(u)
	    Mark u as visited
	    for each uv in $Adj(u)$ **do**
	        if v is not visited **then**
	            add edge uv to T
	            set pred(v) to u
	            DFS(v)

Implemented using a global array *Visited* for all recursive calls.
T is the search tree/forest.

Example



Edges classified into two types: $uv \in E$ is a

1. tree edge: belongs to T
2. non-tree edge: does not belong to T

Properties of DFS tree

Proposition

1. *T is a forest*
2. *connected components of T are same as those of G.*
3. *If $uv \in E$ is a non-tree edge then, in T, either:*
   1. *u is an ancestor of v, or*
   2. *v is an ancestor of u.*

Question: Why are there no *cross-edges*?

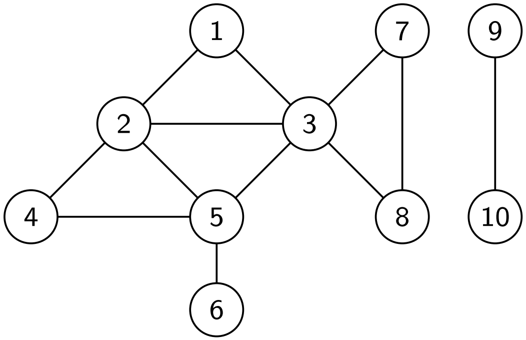
DFS with Visit Times

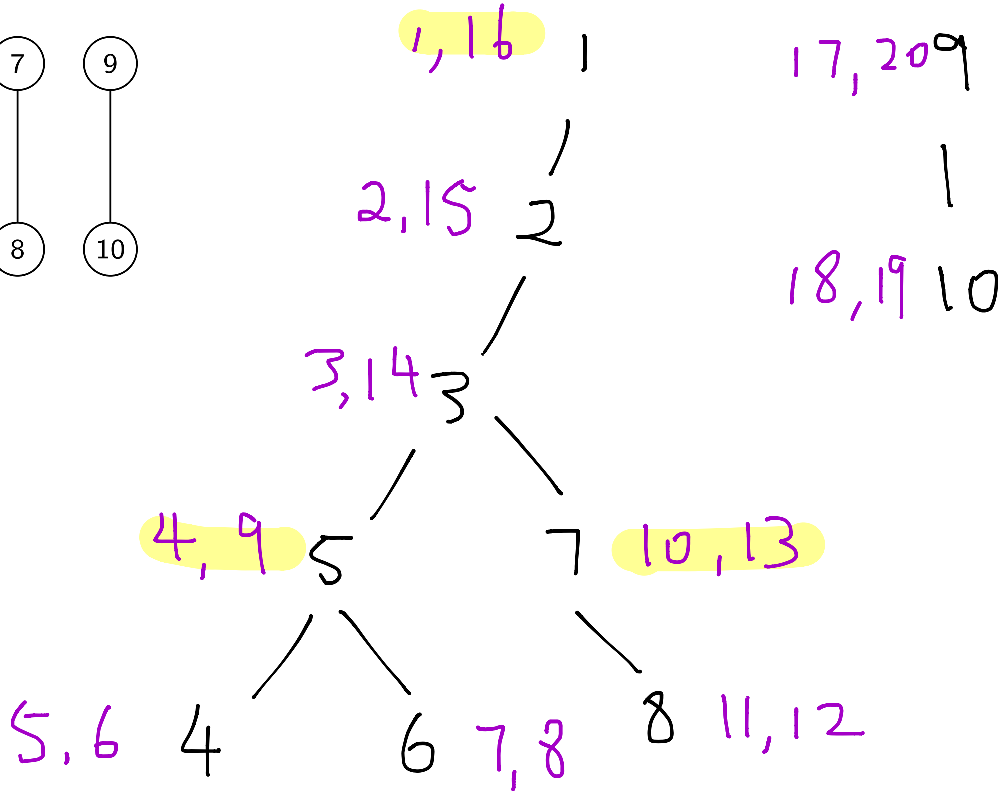
Keep track of when nodes are visited.

```
DFS(G)
    for all u ∈ V(G) do
        Mark u as unvisited
    T is set to ∅
    time = 0
    while ∃unvisited u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each uv in Out(u) do
        if v is not marked then
            add edge uv to T
            DFS(v)
    post(u) = ++time
```

Example


1
7
9
2
3
4
5
8
10
6


1,16 1
17,20 9
2,15 2
18,19 10
3,14 3
4,9 5
7 10,13
5,6 4
6 7,8
8 11,12

pre and post numbers

Node u is **active** in time interval $[\mathrm{pre}(u), \mathrm{post}(u)]$

Proposition

For any two nodes u *and* v*, the two intervals* $[\mathrm{pre}(u), \mathrm{post}(u)]$ *and* $[\mathrm{pre}(v), \mathrm{post}(v)]$ *are disjoint or one is contained in the other.*

pre and post numbers

Node u is **active** in time interval $[\mathrm{pre}(u), \mathrm{post}(u)]$

Proposition

For any two nodes u and v, the two intervals $[\mathrm{pre}(u), \mathrm{post}(u)]$ and $[\mathrm{pre}(v), \mathrm{post}(v)]$ are disjoint or one is contained in the other.

Proof.

pre and post numbers

Node u is **active** in time interval $[\mathrm{pre}(u), \mathrm{post}(u)]$

Proposition

For any two nodes u and v, the two intervals $[\mathrm{pre}(u), \mathrm{post}(u)]$ and $[\mathrm{pre}(v), \mathrm{post}(v)]$ are disjoint or one is contained in the other.

Proof.

- Assume without loss of generality that $\mathrm{pre}(u) < \mathrm{pre}(v)$. Then v visited after u.

pre and post numbers

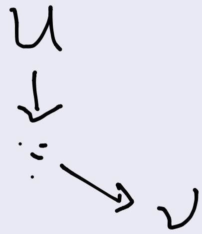
Node u is **active** in time interval $[\mathrm{pre}(u), \mathrm{post}(u)]$

Proposition

For any two nodes u and v, the two intervals $[\mathrm{pre}(u), \mathrm{post}(u)]$ and $[\mathrm{pre}(v), \mathrm{post}(v)]$ are disjoint or one is contained in the other.

Proof.

- Assume without loss of generality that $\mathrm{pre}(u) < \mathrm{pre}(v)$. Then v visited after u.
- If **DFS**(v) invoked before **DFS**(u) finished, $\mathrm{post}(v) < \mathrm{post}(u)$.



pre and post numbers

Node u is **active** in time interval $[\mathrm{pre}(u), \mathrm{post}(u)]$

Proposition

For any two nodes u and v, the two intervals $[\mathrm{pre}(u), \mathrm{post}(u)]$ and $[\mathrm{pre}(v), \mathrm{post}(v)]$ are disjoint or one is contained in the other.

Proof.

- Assume without loss of generality that $\mathrm{pre}(u) < \mathrm{pre}(v)$. Then v visited after u.
- If **DFS**(v) invoked before **DFS**(u) finished, $\mathrm{post}(v) < \mathrm{post}(u)$.
- If **DFS**(v) invoked after **DFS**(u) finished, $\mathrm{pre}(v) > \mathrm{post}(u)$. □

[u, v, v, u]

[u, u] [v, v]

pre and post numbers

Node u is **active** in time interval $[\mathrm{pre}(u), \mathrm{post}(u)]$

Proposition

For any two nodes u and v, the two intervals $[\mathrm{pre}(u), \mathrm{post}(u)]$ and $[\mathrm{pre}(v), \mathrm{post}(v)]$ are disjoint or one is contained in the other.

Proof.

- Assume without loss of generality that $\mathrm{pre}(u) < \mathrm{pre}(v)$. Then v visited after u.
- If **DFS**(v) invoked before **DFS**(u) finished, $\mathrm{post}(v) < \mathrm{post}(u)$.
- If **DFS**(v) invoked after **DFS**(u) finished, $\mathrm{pre}(v) > \mathrm{post}(u)$. □

pre and post numbers useful in several applications of **DFS**

Part III

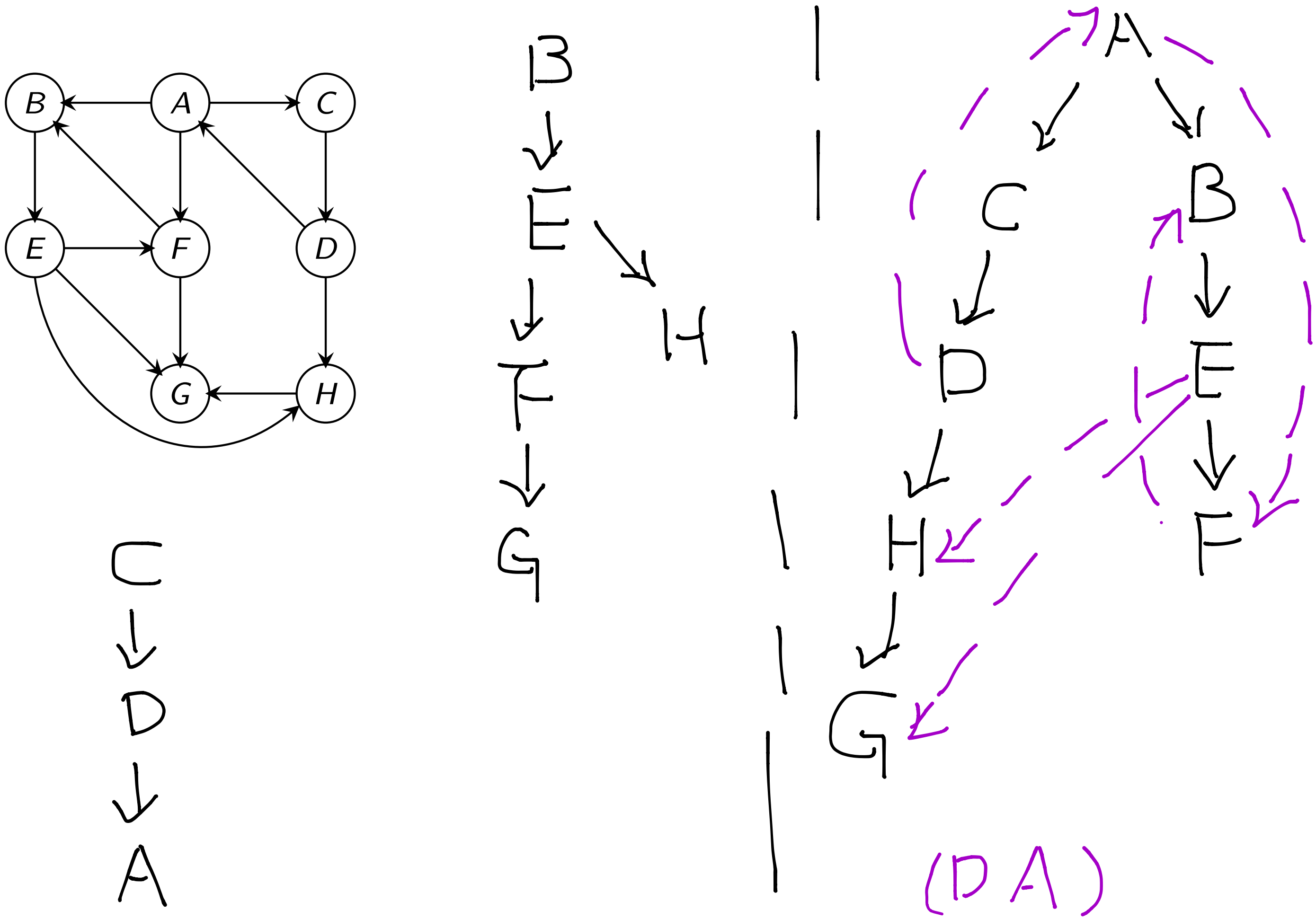
DFS in Directed Graphs

DFS in Directed Graphs

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    time = 0
    while there is an unvisited node u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each edge (u, v) in Out(u) do
        if v is not visited
            add edge (u, v) to T
            DFS(v)
    post(u) = ++time
```

Example



DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS**(G) takes $O(m+n)$ time.

DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS**(G) takes $O(m+n)$ time.
2. Edges added form a *branching*: a forest of out-trees. Output of $DFS(G)$ depends on the order in which vertices are considered.

DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS**(G) takes $O(m+n)$ time.
2. Edges added form a *branching*: a forest of out-trees. Output of $DFS(G)$ depends on the order in which vertices are considered.
3. If u is the first vertex considered by $DFS(G)$ then $DFS(u)$ outputs a directed out-tree T rooted at u and a vertex v is in T if and only if $v \in \text{rch}(u)$

DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS**(G) takes $O(m+n)$ time.
2. Edges added form a *branching*: a forest of out-trees. Output of $DFS(G)$ depends on the order in which vertices are considered.
3. If u is the first vertex considered by $DFS(G)$ then $DFS(u)$ outputs a directed out-tree T rooted at u and a vertex v is in T if and only if $v \in \mathrm{rch}(u)$
4. For any two vertices x, y the intervals $[\mathrm{pre}(x), \mathrm{post}(x)]$ and $[\mathrm{pre}(y), \mathrm{post}(y)]$ are either disjoint or one is contained in the other.

DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS**(G) takes $O(m + n)$ time.
2. Edges added form a *branching*: a forest of out-trees. Output of $DFS(G)$ depends on the order in which vertices are considered.
3. If u is the first vertex considered by $DFS(G)$ then $DFS(u)$ outputs a directed out-tree T rooted at u and a vertex v is in T if and only if $v \in \mathrm{rch}(u)$
4. For any two vertices x, y the intervals $[\mathrm{pre}(x), \mathrm{post}(x)]$ and $[\mathrm{pre}(y), \mathrm{post}(y)]$ are either disjoint or one is contained in the other.

Note: Not obvious whether **DFS**(G) is useful in dir graphs but it is.

DFS Tree

Edges of G can be classified with respect to the **DFS** tree T as:

1. **Tree edges** (x, y) that belong to T:
   $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
2. A **forward edge** is a non-tree edges (x, y) such that
   $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
3. A **backward edge** is a non-tree edge (x, y) such that
   $\mathrm{pre}(y) < \mathrm{pre}(x) < \mathrm{post}(x) < \mathrm{post}(y)$.
4. A **cross edge** is a non-tree edges (x, y) such that
   $\mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{pre}(x) < \mathrm{post}(x)$.

DFS Tree

Edges of G can be classified with respect to the **DFS** tree T as:

1. **Tree edges** (x, y) that belong to T:
   $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
2. A **forward edge** is a non-tree edges (x, y) such that
   $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
3. A **backward edge** is a non-tree edge (x, y) such that
   $\mathrm{pre}(y) < \mathrm{pre}(x) < \mathrm{post}(x) < \mathrm{post}(y)$.
4. A **cross edge** is a non-tree edges (x, y) such that
   $\mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{pre}(x) < \mathrm{post}(x)$.

Note what makes a backward edge special is $\mathrm{post}(x) < \mathrm{post}(y)$.

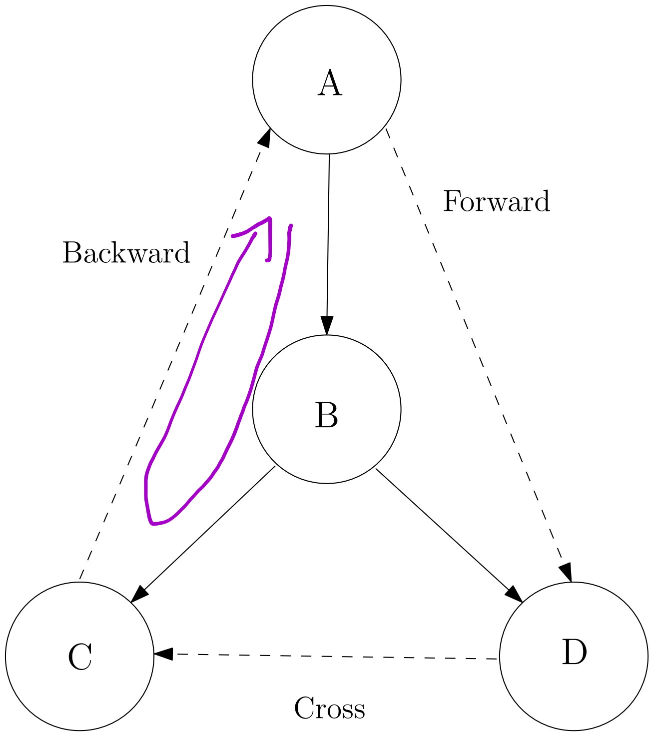
DFS Tree

Edges of G can be classified with respect to the **DFS** tree T as:

1. **Tree edges** (x, y) that belong to T:
   $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
2. A **forward edge** is a non-tree edges (x, y) such that
   $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
3. A **backward edge** is a non-tree edge (x, y) such that
   $\mathrm{pre}(y) < \mathrm{pre}(x) < \mathrm{post}(x) < \mathrm{post}(y)$.
4. A **cross edge** is a non-tree edges (x, y) such that
   $\mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{pre}(x) < \mathrm{post}(x)$.

Note what makes a backward edge special is $\mathrm{post}(x) < \mathrm{post}(y)$.
Also note both backward and cross edge have $\mathrm{pre}(y) < \mathrm{pre}(x)$.

Types of Edges



Cycles in graphs

Question: Given an *undirected* graph how do we check whether it has a cycle and output one if it has one?

Question: Given an *directed* graph how do we check whether it has a cycle and output one if it has one?

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS**(G).

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS**(G).

Proof.

If: (u, v) is a back edge implies there is a cycle C consisting of the path from v to u in **DFS** search tree and the edge (u, v).

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS**(G).

Proof.

If: (u, v) is a back edge implies there is a cycle C consisting of the path from v to u in **DFS** search tree and the edge (u, v).

Only if: Suppose there is a cycle $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$.
Let v_i be first node in C visited in **DFS**.
All other nodes in C are descendants of v_i since they are reachable from v_i.
Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge. □

An Edge in DAG

Proposition

If G is a DAG *and* $\mathrm{post}(u) < \mathrm{post}(v)$, *then* (u, v) *is not in* G.
i.e., for all edges (u, v) *in a* DAG, $\mathrm{post}(u) > \mathrm{post}(v)$.

$u \prec v$

An Edge in DAG

Proposition

If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.
i.e., for all edges (u, v) in a DAG, $\mathrm{post}(u) > \mathrm{post}(v)$.

Proof.

Assume $\mathrm{post}(u) < \mathrm{post}(v)$ *and* (u, v) is an edge in G. We derive a contradiction. One of two cases holds from DFS property.

- Case 1: $[pre(u), post(u)]$ is contained in $[pre(v), post(v)]$. Implies that u is explored during **DFS**(v) and hence is a descendent of v. Edge (u, v) implies a cycle in G but G is assumed to be DAG!
- Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$. This cannot happen since v would be explored from u.

□

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given G, is it a DAG? If it is, generate a topological sort. Else output a cycle C.

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given G, is it a DAG? If it is, generate a topological sort. Else output a cycle C.

DFS based algorithm:

1. Compute **DFS**(G)
2. If there is a back edge $e = (v, u)$ then G is not a DAG. Output cycle C formed by path from u to v in T plus edge (v, u).

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given G, is it a DAG? If it is, generate a topological sort. Else output a cycle C.

DFS based algorithm:

1. Compute **DFS**(G)
2. If there is a back edge $e = (v, u)$ then G is not a DAG. Output cycle C formed by path from u to v in T plus edge (v, u).
3. Otherwise output nodes in decreasing post-visit order.

Using DFS...

... to check for Acylicity and compute Topological Ordering

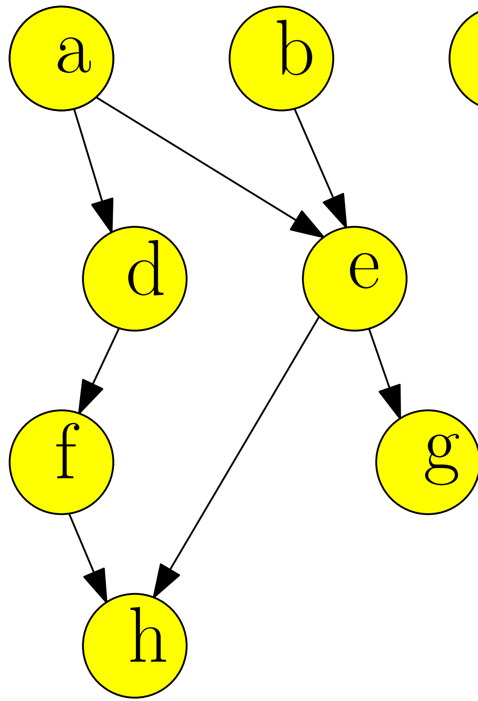
Question

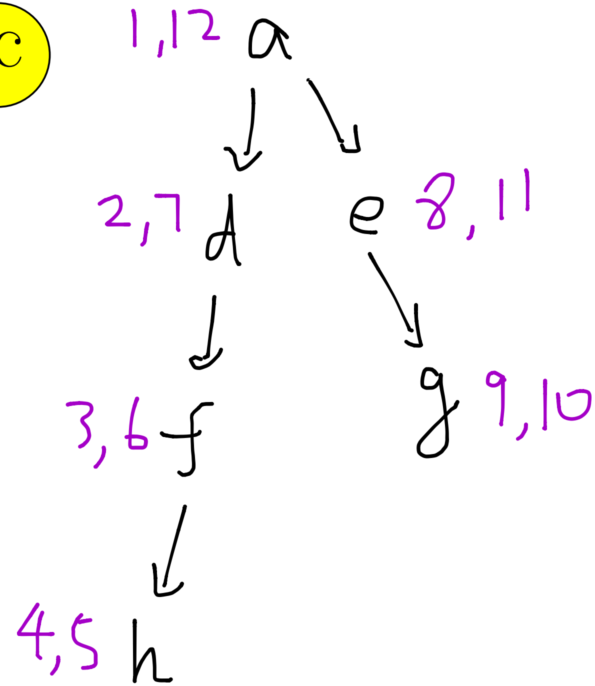
Given G, is it a DAG? If it is, generate a topological sort. Else output a cycle C.

DFS based algorithm:

1. Compute **DFS**(G)
2. If there is a back edge $e = (v, u)$ then G is not a DAG. Output cycle C formed by path from u to v in T plus edge (v, u).
3. Otherwise output nodes in decreasing post-visit order.
   Note: no need to sort, $DFS(G)$ can output nodes in this order.

Algorithm runs in $O(n + m)$ time.





c b a e g d f h

b 13,14 c 15,16

Part IV

DAGs, DFS and SCC in Linear Time

Finding all SCCs of a Directed Graph

Problem

Given a directed graph $G = (V, E)$, output *all* its strong connected components.

Finding all SCCs of a Directed Graph

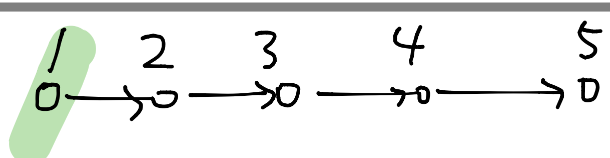
Problem

Given a directed graph $G = (V, E)$, output *all* its strong connected components.

Straightforward algorithm:

Mark all vertices in V as not visited.
for each vertex $u \in V$ not visited yet **do**
    find SCC(G, u) the strong component of u:
        Compute $\mathsf{rch}(G, u)$ using $DFS(G, u)$
        Compute $\mathsf{rch}(G^{\mathrm{rev}}, u)$ using $DFS(G^{\mathrm{rev}}, u)$
        SCC$(G, u) \Leftarrow \mathsf{rch}(G, u) \cap \mathsf{rch}(G^{\mathrm{rev}}, u)$
        $\forall u \in$ SCC(G, u): Mark u as visited.

Running time: $O(n(n + m))$



Finding all SCCs of a Directed Graph

Problem

Given a directed graph $G = (V, E)$, output *all* its strong connected components.

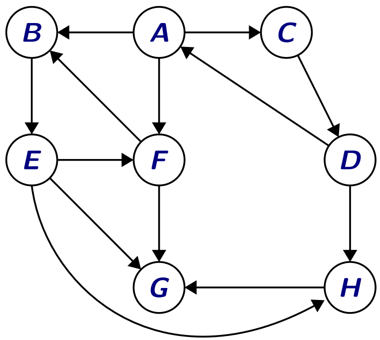
Straightforward algorithm:

```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u): Mark u as visited.
```

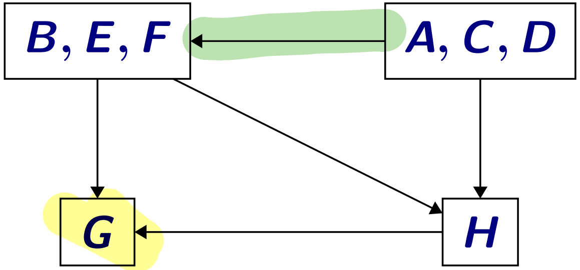
Running time: $O(n(n+m))$

Is there an $O(n+m)$ time algorithm?

Graph of SCCs



Graph G



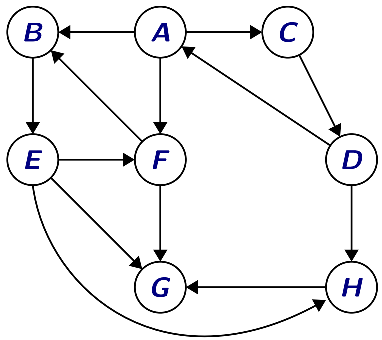
Graph of SCCs G^{SCC}

Meta-graph of SCCs

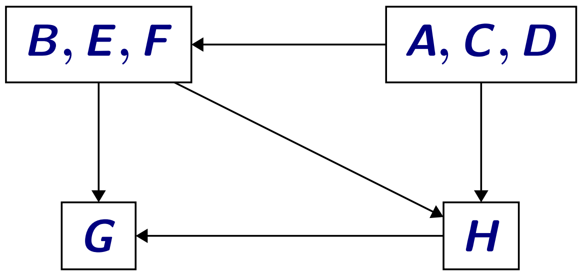
Let $S_1, S_2, \ldots S_k$ be the strong connected components (i.e., SCCs) of G. The graph of SCCs is G^{SCC}

1. Vertices are $S_1, S_2, \ldots S_k$
2. There is an edge (S_i, S_j) if there is some $u \in S_i$ and $v \in S_j$ such that (u, v) is an edge in G.

Structure of a Directed Graph



Graph G



Graph of SCCs G^{SCC}

Reminder

G^{SCC} is created by collapsing every strong connected component to a single vertex.

Proposition

For a directed graph G, its meta-graph G^{SCC} is a DAG.

SCCs and DAGs

Proposition

For any graph G, the graph G^{SCC} has no directed cycle.

Proof.

If G^{SCC} has a cycle $S_1, S_2, \ldots, S_k$ then $S_1 \cup S_2 \cup \cdots \cup S_k$ should be in the same SCC in G. Formal details: exercise. □

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $\mathrm{SCC}(u)$
3. Remove $\mathrm{SCC}(u)$ and repeat

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $\mathrm{SCC}(u)$
3. Remove $\mathrm{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $\mathrm{SCC}(u)$
3. Remove $\mathrm{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$
2. ... since there are no edges coming out a sink!
3.
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $\mathrm{SCC}(u)$
3. Remove $\mathrm{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$
2. ... since there are no edges coming out a sink!
3. **DFS**(u) takes time proportional to size of $\mathrm{SCC}(u)$
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $\mathrm{SCC}(u)$
3. Remove $\mathrm{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$
2. ... since there are no edges coming out a sink!
3. **DFS**(u) takes time proportional to size of $\mathrm{SCC}(u)$
4. Therefore, total time $O(n + m)$!

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

There is no easy way to find a node in a sink SCC, but there is a way to find a node in a source SCC.

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

There is no easy way to find a node in a sink SCC, but there is a way to find a node in a source SCC.
Then we can find a node in the source SCC of the the reversal of G^{SCC}!

Reversal and SCCs

Proposition

For any graph G, the graph of SCCs of $\boldsymbol{G}^{\mathbf{rev}}$ is the same as the reversal of G^{SCC}.

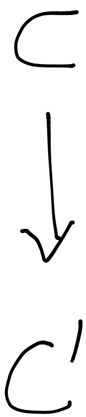
Proof.

The SCCs of $\boldsymbol{G}^{\mathbf{rev}}$ are the same as those of G. Formal proof as exercise. □

How to linearize SCCs

Proposition

If C and C' are SCC, and there is an edge from a node in C to a node in C', then the highest post number in C is bigger than the highest post number in C'.



How to linearize SCCs

Proposition

If C and C' are SCC, and there is an edge from a node in C to a node in C', then the highest post number in C is bigger than the highest post number in C'.

Proof

Consider two cases.

1. Case 1: **DFS** visits C first.

How to linearize SCCs

Proposition

If C and C' are SCC, and there is an edge from a node in C to a node in C', then the highest post number in C is bigger than the highest post number in C'.

Proof

Consider two cases.

1. Case 1: **DFS** visits C first.
   then all the vertices will be traversed. The first node visited in C will have the highest post number.

How to linearize SCCs

Proposition

If C *and* C' *are* SCC*, and there is an edge from a node in* C *to a node in* C'*, then the highest post number in* C *is bigger than the highest post number in* C'*.*

Proof

Consider two cases.

1. Case 1: **DFS** visits C first.
   then all the vertices will be traversed. The first node visited in C will have the highest post number.
2. Case 2: **DFS** visits C' first.

$C' \longleftarrow C$

How to linearize SCCs

Proposition

If C *and* C' *are* SCC, *and there is an edge from a node in* C *to a node in* C', *then the highest post number in* C *is bigger than the highest post number in* C'.

Proof

Consider two cases.

$C \longrightarrow C'$

1. Case 1: **DFS** visits C first.
   then all the vertices will be traversed. The first node visited in C will have the highest post number.
2. Case 2: **DFS** visits C' first.
   then **DFS** will stop after visiting all nodes in C' but before seeing any of C.

How to linearize SCCs

Proposition

The node that receives the highest post number in DFS must lie in a source SCC*.*

How to linearize SCCs

Proposition

The node that receives the highest post number in DFS must lie in a source SCC.

In other words, the SCCs are topologically sorted by arranging them in decreasing order of their highest post number.

How to linearize SCCs

Proposition

The node that receives the highest post number in DFS must lie in a source SCC.

In other words, the SCCs are topologically sorted by arranging them in decreasing order of their highest post number.

A generalization of topological sort for DAGs.

Linear Time Algorithm

...for computing the strong connected components in **G**

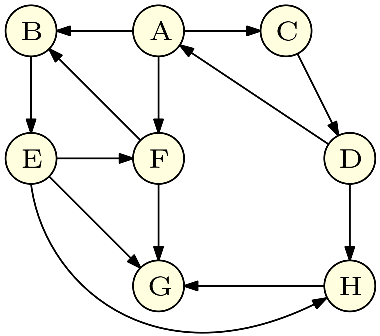
```
do DFS(G^rev) and output vertices in decreasing post order.
Mark all nodes as unvisited
for each u in the computed order do
    if u is not visited then
        DFS(u)
        Let S_u be the nodes reached by u
        Output S_u as a strong connected component
        Remove S_u from G
```

Theorem

Algorithm runs in time $O(m+n)$ and correctly outputs all the SCCs of G.

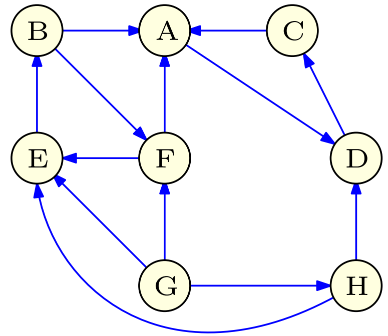
Linear Time Algorithm: An Example - Initial steps

Graph G:



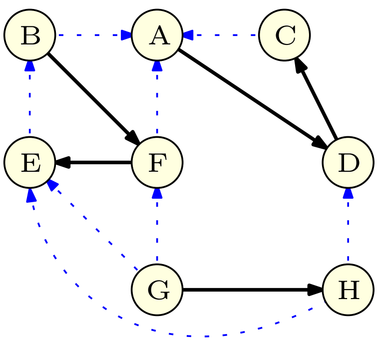
⟹

Reverse graph G^{rev}:



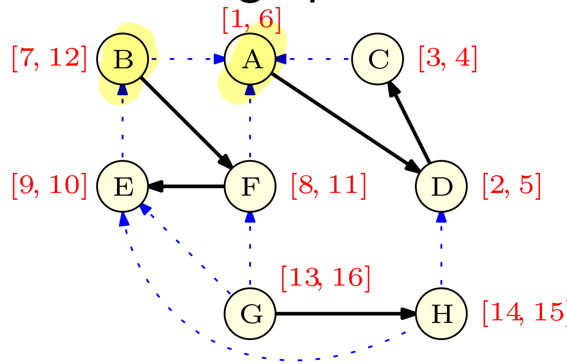
⟹

DFS of reverse graph:



⟹

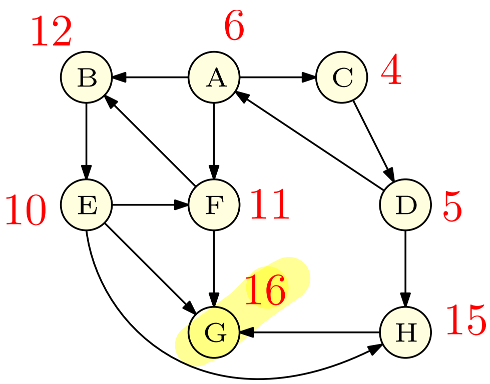
Pre/Post **DFS** numbering of reverse graph:



Linear Time Algorithm: An Example

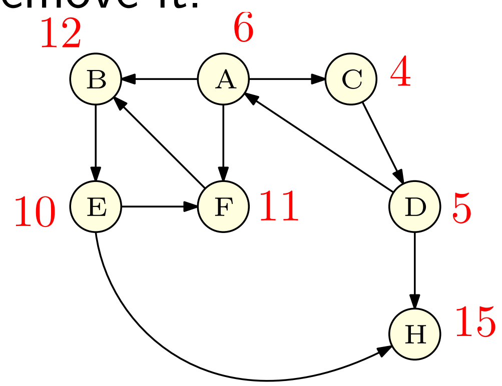
Removing connected components: 1

Original graph G with rev post numbers:



$\Longrightarrow$

Do **DFS** from vertex G remove it.



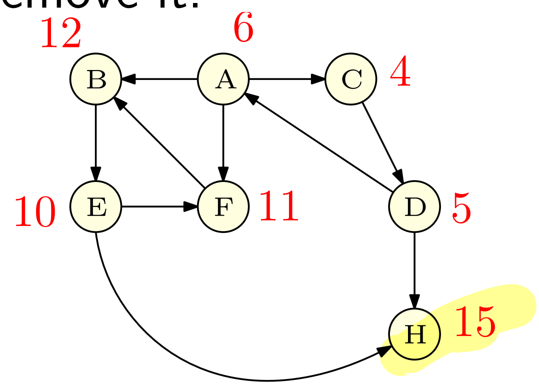
SCC computed:

$\{G\}$

Linear Time Algorithm: An Example

Removing connected components: 2

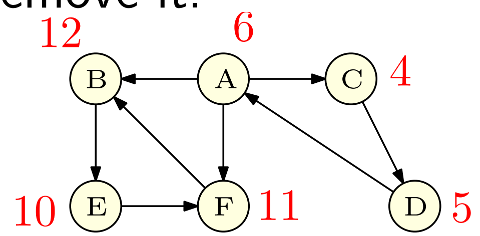
Do **DFS** from vertex G remove it.



SCC computed:
$\{G\}$

$\implies$

Do **DFS** from vertex H, remove it.

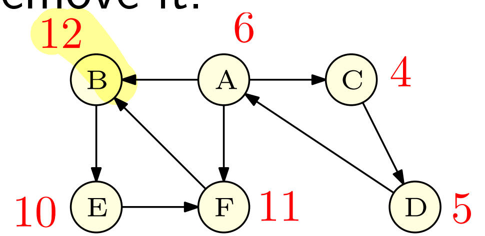


SCC computed:
$\{G\}, \{H\}$

Linear Time Algorithm: An Example

Removing connected components: 3

Do **DFS** from vertex H, remove it.

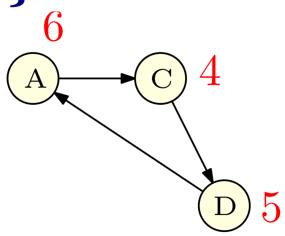


$\Longrightarrow$

Do **DFS** from vertex B

Remove visited vertices:

$\{F, B, E\}$.



SCC computed:

$\{G\}, \{H\}$

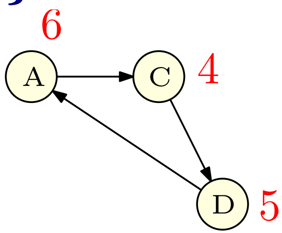
SCC computed:

$\{G\}, \{H\}, \{F, B, E\}$

Linear Time Algorithm: An Example

Removing connected components: 4

Do **DFS** from vertex F
Remove visited vertices:
$\{F, B, E\}$.



SCC computed:
$\{G\}, \{H\}, \{F, B, E\}$

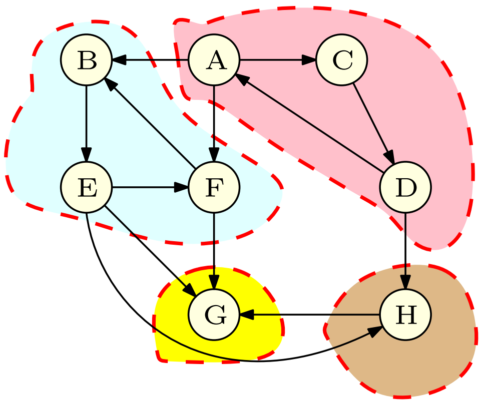
$\Longrightarrow$

Do **DFS** from vertex A
Remove visited vertices:
$\{A, C, D\}$.

SCC computed:
$\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}$

Linear Time Algorithm: An Example

Final result



SCC computed:

$\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}$

Which is the correct answer!

Solving Problems on Directed Graphs

A template for a class of problems on directed graphs:

- Is the problem solvable when G is strongly connected?
- Is the problem solvable when G is a DAG?
- If the above two are feasible then is the problem solvable in a general directed graph G by considering the meta graph G^{SCC}?

Take away Points

1. Given a directed graph G, its SCCs and the associated acyclic meta-graph G^{SCC} give a structural decomposition of G that should be kept in mind.
2. There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.
3. DAGs arise in many application and topological sort is a key property in algorithm design. Linear time algorithms to compute a topological sort (there can be many possible orderings so not unique).