Pre-lecture brain teaser

Last time we looked at the BasicSearch algorithm:

```
Explore(G, u):
    Visited[1 .. n] ← FALSE
    Add u to S
    Visited[u] ← TRUE
    ExploreStep(G, u, Visited, S)
    Output S

ExploreStep(G, x, Visited, S):
    for each edge xy in Adj(x) do
        if (Visited[y] = FALSE)
            Visited[y] ← TRUE
            ExploreStep(G, y, Visited, S):
    return
```

We said that if ToExplore was a:

- Stack, the algorithm is equivalent to **DFS**
- Queue, the algorithm is equivalent to **BFS**

What if the algorithm was written recursively (instead of the while loop, you recursively call explore). What would the algorithm be equivalent to?

ECE-374-B: Lecture 15 - Directed Graphs (DFS, DAGs, Topological Sort)

Instructor: Abhishek Kumar Umrawal

March 19, 2024

University of Illinois at Urbana-Champaign

Pre-lecture brain teaser

Last time we looked at the BasicSearch algorithm:

```
Explore(G,u):
    Visited[1 .. n] ← FALSE
    Add u to S
    Visited[u] ← TRUE
    ExploreStep(G,u,Visited, S)
    Output S

ExploreStep(G,x,Visited, S):
    for each edge xy in Adj(x) do
        if (Visited[y] = FALSE)
            Visited[y] ← TRUE
            ExploreStep(G,y,Visited, S):
    return
```

We said that if ToExplore was a:

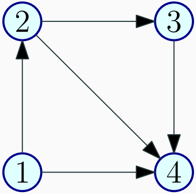
- Stack, the algorithm is equivalent to **DFS**
- Queue, the algorithm is equivalent to **BFS**

What if the algorithm was written recursively (instead of the while loop, you recursively call explore). What would the algorithm be equivalent to?

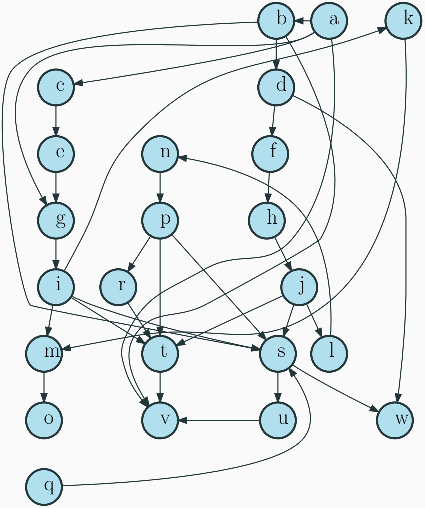
Directed Acyclic Graphs - definition and basic properties

Directed Acyclic Graphs

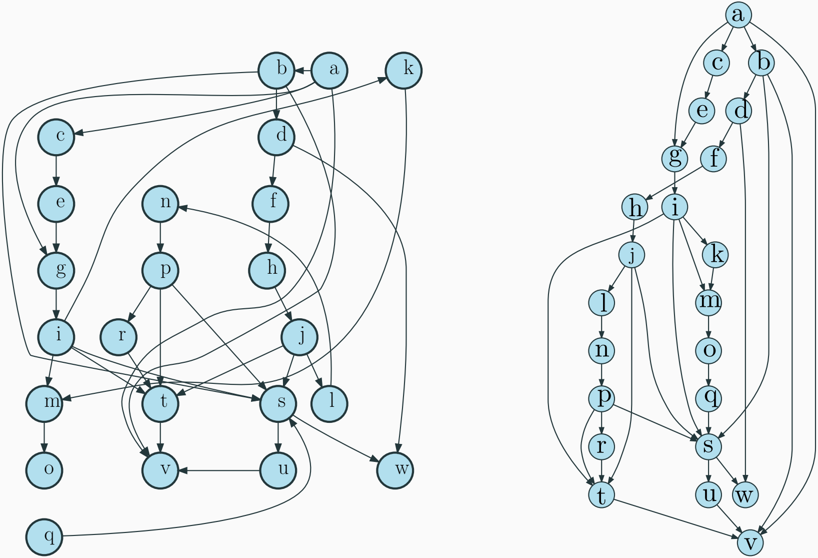
Definition
A directed graph G is a directed acyclic graph (DAG) if there is no directed cycle in G.



Is this a DAG?



Is this a DAG?



Sources and Sinks

Definition

- A vertex u is a source if it has no in-coming edges.
- A vertex u is a sink if it has no out-going edges.

Simple DAG Properties

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

Simple DAG Properties

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

Proof.
Let $P = v_1, v_2, \ldots, v_k$ be a longest path in G. Claim that v_1 is a source and v_k is a sink. Suppose not. Then v_1 has an incoming edge which either creates a cycle or a longer path both of which are contradictions. Similarly if v_k has an outgoing edge. □

Topological ordering

Total recall: Order on a set

Order or strict total order on a set X is a binary relation $\prec$ on X, such that

- Transitivity: $\forall x.y, z \in X \qquad x \prec y$ and $y \prec z \implies x \prec z$.
- For any $x, y \in X$, exactly one of the following holds: $x \prec y$, $y \prec x$ or $x = y$.

Convention about writing edges

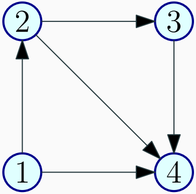
- Undirected graph edges:

$$uv = \{u, v\} = vu \in \mathsf{E}$$

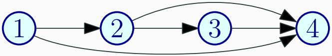
- Directed graph edges:

$$u \to v \qquad \equiv \qquad (u, v) \qquad \equiv \qquad (u \to v)$$

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 $(u \to v) \in E$ 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.

Topological ordering in linear time

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

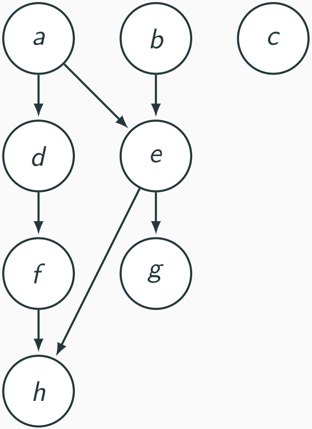
Topological ordering in linear time

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

Simple Algorithm:

1. Calculate the in-degree of each vertex
2. For each vertex that is source ($deg_{in}(v) = 0$):
   2.1 Add v to the topological sort
   2.2 Lower the in-degree of vertices v is connected to. [1]

Topological Sort: Example



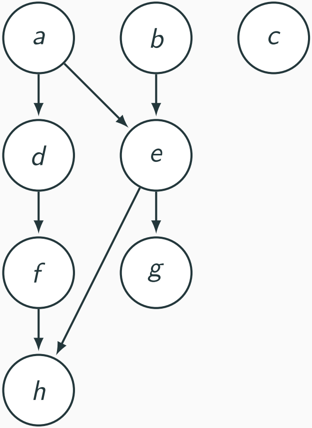
Adjacency List:

Node	Neighbors	
a	d	e
b	e	
c		
d	f	
e	h	g
f	h	
g		
h		

Generate $deg_{in}(v)$:

In-degree	Vertices
0	a, b, c
1	d, f, g
2	e, h

Topological Sort: Example



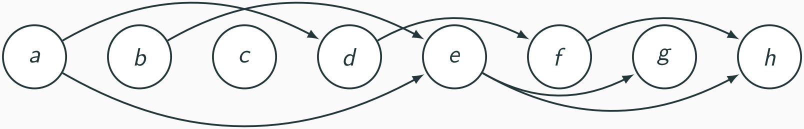
Adjacency List:

Node	Neighbors	
a	d	e
b	e	
c		
d	f	
e	h	g
f	h	
g		
h		

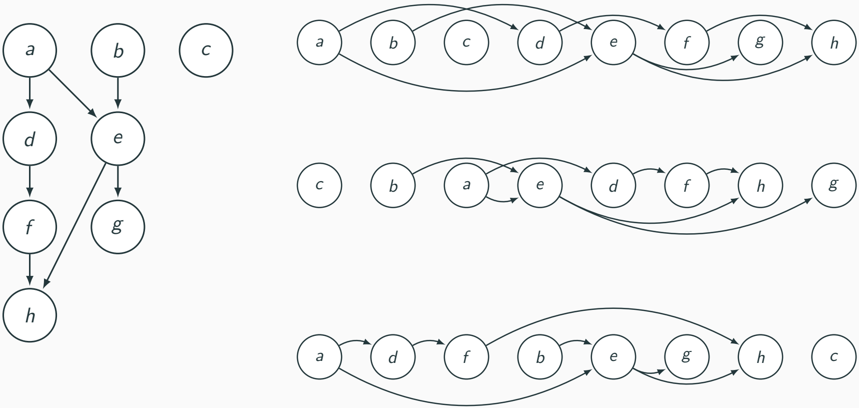
Generate $deg_{in}(v)$:

In-degree	Vertices
0	a, b, c
1	d, f, g
2	e, h

Topological Ordering:



Multiple possible topological orderings



DAGs and Topological Sort

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

- Exercise: What is a DAG with the most number of distinct topological sorts for a given number *n* of vertices?

- Exercise: What is a DAG with the least number of distinct topological sorts for a given number *n* of vertices?

Direct Topological ordering - code

```
TopSort(G):
    Sorted ← NULL
    deg_in[1 .. n] ← −1
    Tdeg_in[1 .. n] ← NULL
    Generate in-degree for each vertex
    for each edge xy in G do
        deg_in[y] + +
    for each vertex v in G do
        Tdeg_in[deg_in[v]].append(v)
    Next we recursively add vertices
     with in-degree = 0 to the sort list
    while (Tdeg_in[0] is non-empty) do
        Remove node x from Tdeg_in[0]
        Sorted.append(x)
        for each edge xy in Adj(x) do
            deg_in[y] − −
            move y to Tdeg_in[deg_in[y]]
    Output Sorted
```

DAGs and Topological Sort

Lemma
A directed graph G can be topologically ordered $\implies$ G is a DAG.

Proof.
Proof by contradiction. Suppose G is not a DAG and has a topological ordering $\prec$. G has a cycle

$$C = u_1 \to u_2 \to \cdots \to u_k \to u_1.$$

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

DAGs and Topological Sort

Lemma
A directed graph G can be topologically ordered $\implies$ G is a DAG.

Proof.
Proof by contradiction. Suppose G is not a DAG and has a topological ordering $\prec$. G has a cycle

$$C = u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \rightarrow u_1.$$

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

$\implies u_1 \prec u_1.$

A contradiction (to $\prec$ being an order). Not possible to topologically order the vertices. □

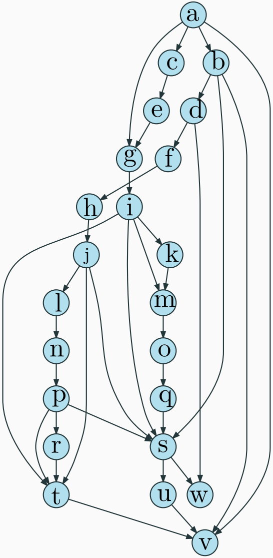
An explicit definition of what topological ordering of a graph is

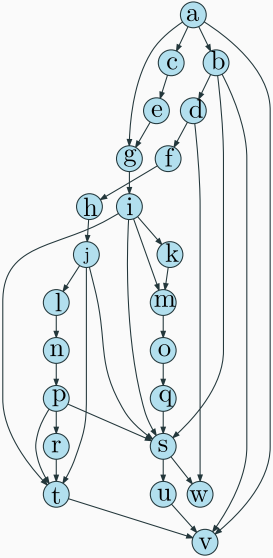
For a graph $G = (V, E)$ a topological ordering of a graph is a numbering $\pi : V \rightarrow \{1, 2, \ldots, n\}$, such that

$$\forall (u \rightarrow v) \in E(G) \implies \pi(u) < \pi(v).$$

(That is, π is one-to-one, and $n = |V|$)

Example...





Assuming:

$V = \{a, \ldots w\}$
$\pi = \{1, \ldots 23\}$

Depth First Search (DFS)

Depth First Search (DFS) in Undirected Graphs

Depth First Search

- **DFS** special case of Basic Search.
- **DFS** is useful in understanding graph structure.
- **DFS** used to obtain linear time ($O(m+n)$) algorithms for
  - Finding cut-edges and cut-vertices of undirected graphs
  - Finding strong connected components of directed graphs
- ...many other applications as well.

DFS in Undirected Graphs

Recursive version. Easier to understand some properties.

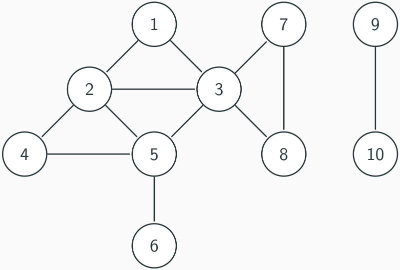
```
DFS(G)
    for all u ∈ V(G) do
        Mark u as unvisited
        Set pred(u) to null
    T is set to ∅
    while ∃ unvisited u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    for each uv in Out(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.

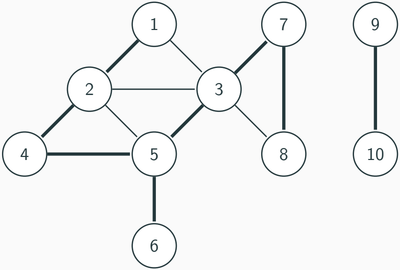
Example



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

- tree edge: belongs to T
- non-tree edge: does not belong to T

Example



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

- tree edge: belongs to T
- non-tree edge: does not belong to T

DFS with pre-post numbering

DFS with Visit Times

Keep track of when nodes are visited.

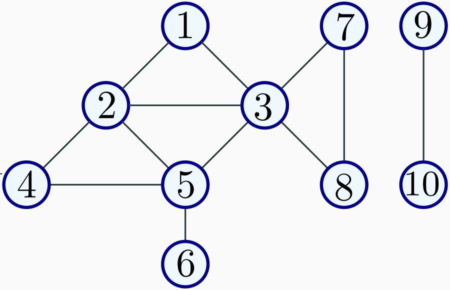
```
DFS(G)
    for all u ∈ V(G) do
        Mark u as unvisited
    T is set to ∅
    time = 0
    while ∃ unvisited u do
        DFS(u)
    Output T
```

```
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
```

Animation

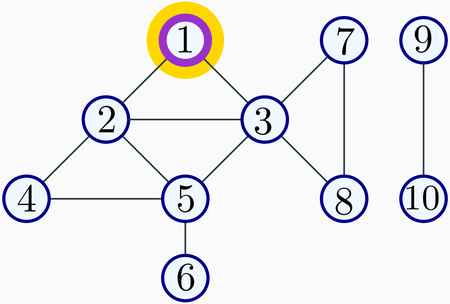
$time = 0$

vertex	[*pre*, *post*]



$time = 1$

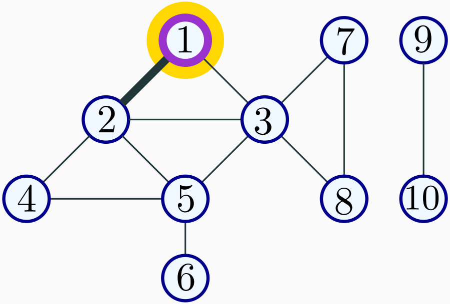
vertex	[*pre*, *post*]
1	[1,]



Animation

$time = 1$

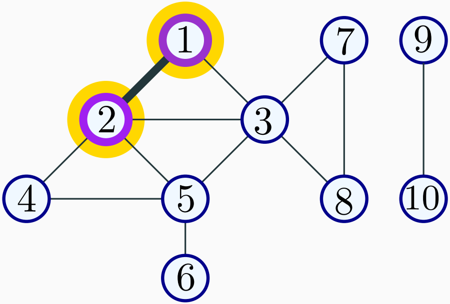
vertex	[pre, $post$]
1	[1,]



Animation

$time = 2$

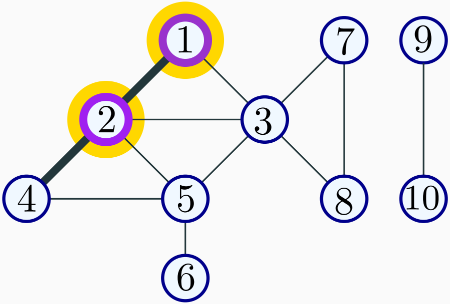
vertex	[*pre*, *post*]
1	[1,]
2	[2,]



Animation

$time = 2$

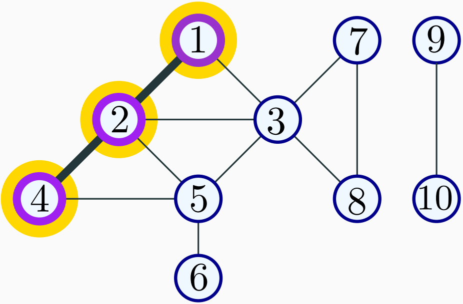
vertex	[pre, $post$]
1	[1,]
2	[2,]



Animation

$time = 3$

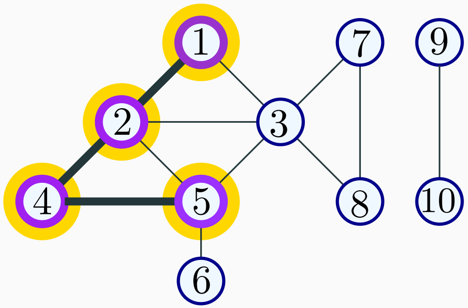
vertex	$[pre, post]$
1	$[1,]$
2	$[2,]$
4	$[3,]$



Animation

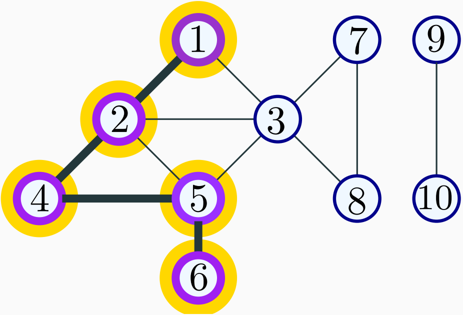
$time = 4$

vertex	[*pre*, *post*]
1	[1,]
2	[2,]
4	[3,]
5	[4,]



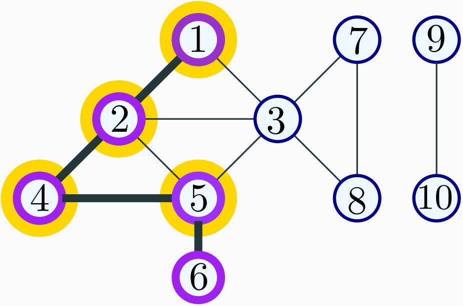
$time = 5$

vertex	[*pre*, *post*]
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5,]



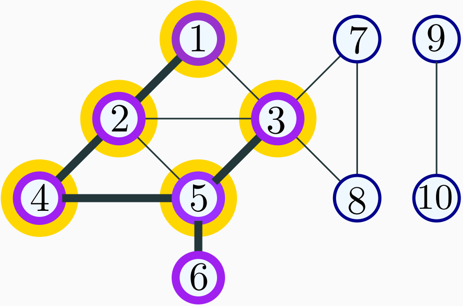
$time = 6$

vertex	$[pre, post]$
1	$[1,]$
2	$[2,]$
4	$[3,]$
5	$[4,]$
6	$[5, 6]$



$time = 7$

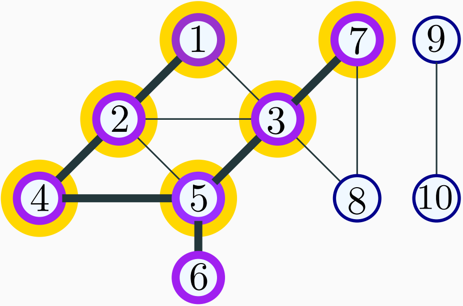
vertex	$[pre, post]$
1	$[1,]$
2	$[2,]$
4	$[3,]$
5	$[4,]$
6	$[5, 6]$
3	$[7,]$



Animation

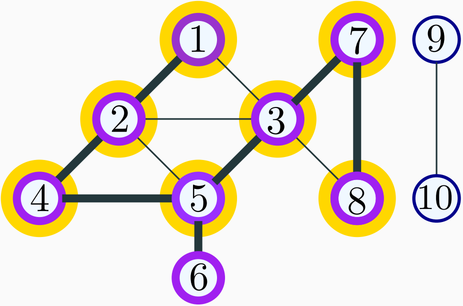
$time = 8$

vertex	$[pre, post]$
1	$[1,]$
2	$[2,]$
4	$[3,]$
5	$[4,]$
6	$[5, 6]$
3	$[7,]$
7	$[8,]$



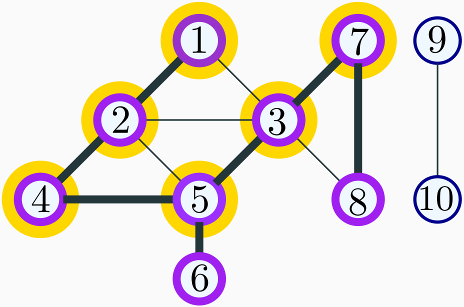
$time = 9$

vertex	$[pre, post]$
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]
8	[9,]



$time = 10$

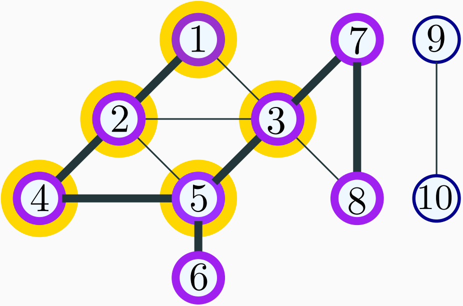
vertex	[*pre*, *post*]
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]
8	[9, 10]



Animation

$time = 11$

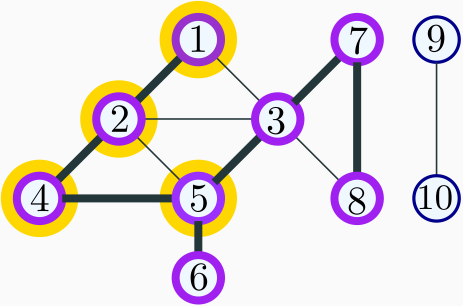
vertex	$[pre, post]$
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8, 11]
8	[9, 10]



Animation

$time = 12$

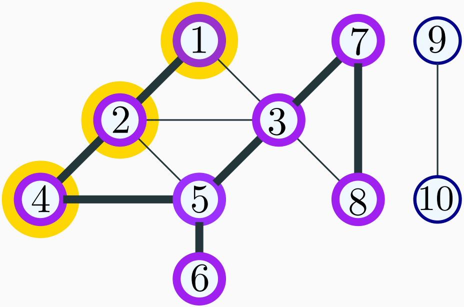
vertex	$[pre, post]$
1	$[1,]$
2	$[2,]$
4	$[3,]$
5	$[4,]$
6	$[5, 6]$
3	$[7, 12]$
7	$[8, 11]$
8	$[9, 10]$



Animation

$time = 13$

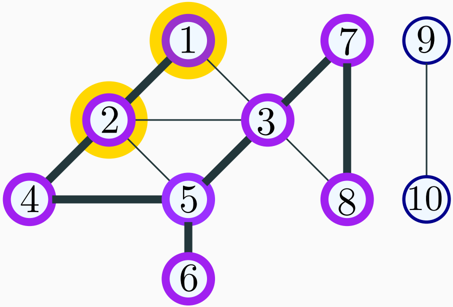
vertex	$[pre, post]$
1	$[1,]$
2	$[2,]$
4	$[3,]$
5	$[4, 13]$
6	$[5, 6]$
3	$[7, 12]$
7	$[8, 11]$
8	$[9, 10]$



Animation

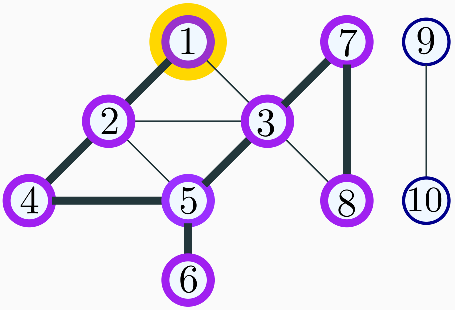
$time = 14$

vertex	$[pre, post]$
1	$[1,]$
2	$[2,]$
4	$[3, 14]$
5	$[4, 13]$
6	$[5, 6]$
3	$[7, 12]$
7	$[8, 11]$
8	$[9, 10]$



$time = 15$

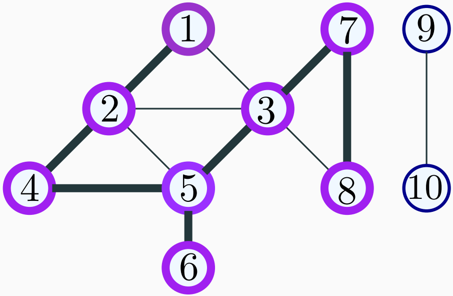
vertex	$[pre, post]$
1	$[1,]$
2	$[2, 15]$
4	$[3, 14]$
5	$[4, 13]$
6	$[5, 6]$
3	$[7, 12]$
7	$[8, 11]$
8	$[9, 10]$



Animation

$time = 16$

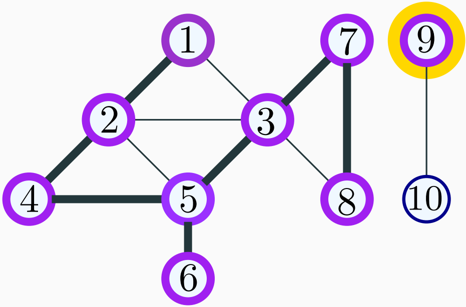
vertex	[*pre*, *post*]
1	[1, 16]
2	[2, 15]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



Animation

$time = 17$

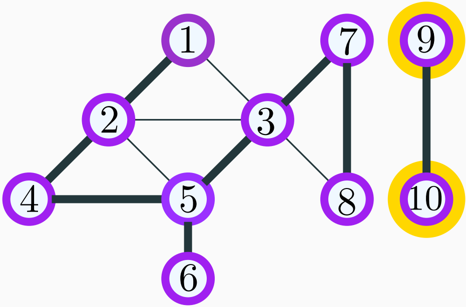
vertex	[*pre*, *post*]
1	[1, 16]
2	[2, 15]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]
9	[17,]



Animation

$$time = 18$$

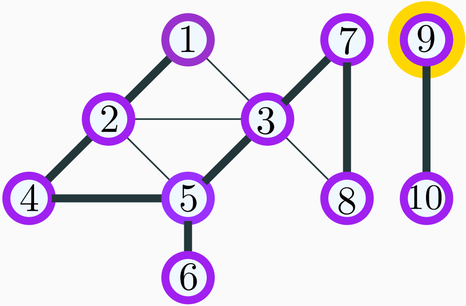
vertex	$[pre, post]$
1	$[1, 16]$
2	$[2, 15]$
4	$[3, 14]$
5	$[4, 13]$
6	$[5, 6]$
3	$[7, 12]$
7	$[8, 11]$
8	$[9, 10]$
9	$[17,]$
10	$[18,]$



Animation

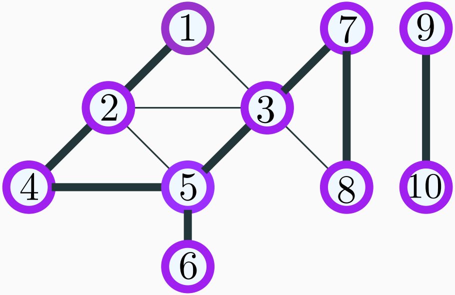
$time = 19$

vertex	$[pre, post]$
1	$[1, 16]$
2	$[2, 15]$
4	$[3, 14]$
5	$[4, 13]$
6	$[5, 6]$
3	$[7, 12]$
7	$[8, 11]$
8	$[9, 10]$
9	$[17,]$
10	$[18, 19]$



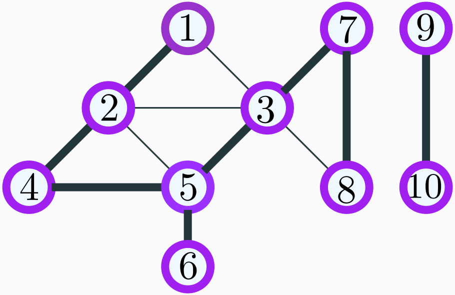
$time = 20$

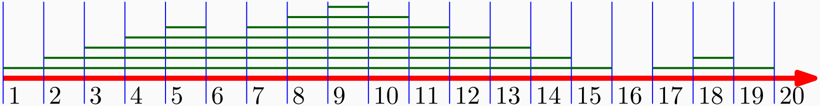
vertex	$[pre, post]$
1	$[1, 16]$
2	$[2, 15]$
4	$[3, 14]$
5	$[4, 13]$
6	$[5, 6]$
3	$[7, 12]$
7	$[8, 11]$
8	$[9, 10]$
9	$[17, 20]$
10	$[18, 19]$



Animation

vertex	[*pre*, *post*]
1	[1, 16]
2	[2, 15]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]
9	[17, 20]
10	[18, 19]





pre **and** post **numbers**

Node u is <u>active</u> in time interval $[\text{pre}(u), \text{post}(u)]$

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

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

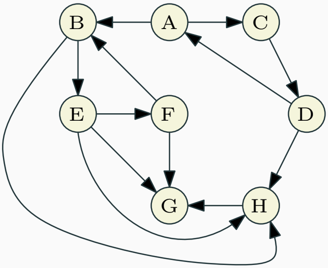
DFS in Directed Graphs

DFS in Directed Graphs

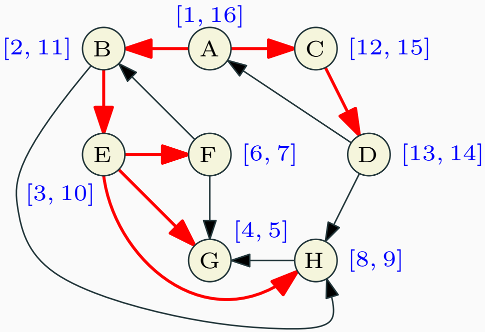
```
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    time = 0
    while there is an unvisited node u do
        DFS(u)
    Output T
```

```
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 of DFS in directed graph



Example of DFS in directed graph



DFS Properties

Generalizing ideas from undirected graphs:

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

DFS Properties

Generalizing ideas from undirected graphs:

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

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

- **DFS**(G) takes $O(m+n)$ time.
- Edges added form a branching: a forest of out-trees. **Output of** $DFS(G)$ **depends on the order in which vertices are considered.**
- 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 $v \in \text{rch}(u)$
- 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

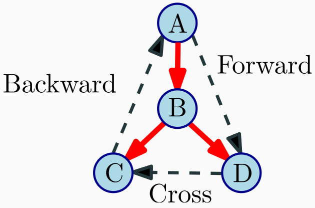
Generalizing ideas from undirected graphs:

- **DFS**(G) takes $O(m+n)$ time.
- Edges added form a branching: a forest of out-trees. **Output of** $DFS(G)$ **depends on the order in which vertices are considered.**
- 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 $v \in \text{rch}(u)$
- 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 tree and related edges

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

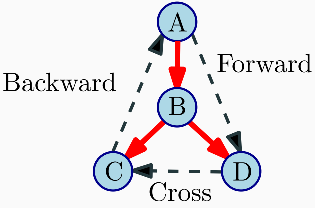
- Tree edges that belong to T
- A forward edge is a non-tree edges (x, y) such that y is a descendant of x .
- A backward edge is a non-tree edge (x, y) such that y is an ancestor of x.
- A cross edge is a non-tree edges (x, y) such that they don't have a ancestor/descendant relationship between them.



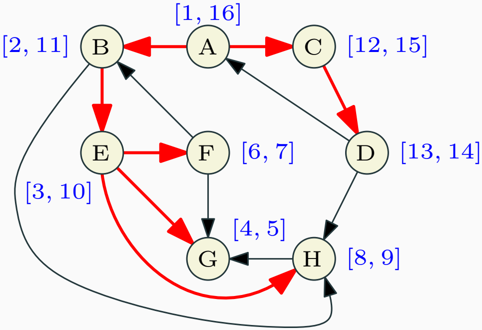
DFS tree and related edges

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

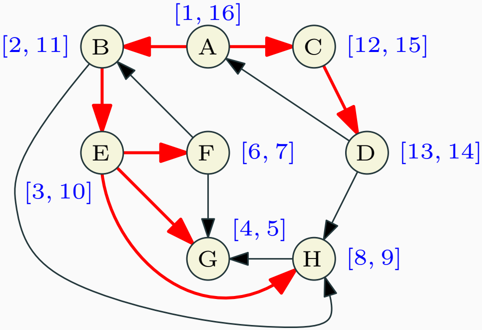
- Tree edges that belong to T
- A forward edge is a non-tree edges (x, y) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.
- A backward edge is a non-tree edge (x, y) such that $\text{pre}(y) < \text{pre}(x) < \text{post}(x) < \text{post}(y)$.
- A cross edge is a non-tree edges (x, y) such that the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are disjoint.



Types of Edges



Types of Edges



- **Back edges:** (F,B), (D,A)
- **Forward edges:** (B,H)
- **Cross edges:** (F,G), (H,G), (D,H)

DFS and cycle detection: Topological sorting using DFS

Cycles in graphs

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

Cycles in graphs

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?

Cycle detection in directed graph using topological sorting

Question
Given G, is it a DAG?

If it is, compute a topological sort.

If it fails, then output the cycle C.

Topological sort a graph using **DFS**

DFS based algorithm:

- Compute **DFS**(G)
- 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).
- Otherwise output nodes in decreasing post-visit order. Note: no need to sort, $DFS(G)$ can output nodes in this order.

Topological sort a graph using DFS

DFS based algorithm:

- Compute **DFS**(G)
- 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).
- Otherwise output nodes in decreasing post-visit order. Note: no need to sort, $DFS(G)$ can output nodes in this order.

Computes topological ordering of the vertices.

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

Topological sort a graph using DFS

DFS based algorithm:

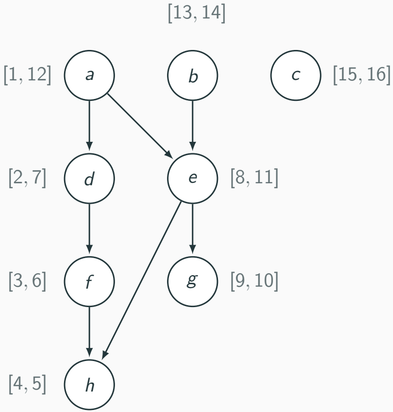
- Compute **DFS**(G)
- If there is a back edge $e = (v, u)$ then G is not a DAG. Output cycle C formed by path from u to v in T plus edge (v, u).
- Otherwise output nodes in decreasing post-visit order. Note: no need to sort, $DFS(G)$ can output nodes in this order.

Computes topological ordering of the vertices.

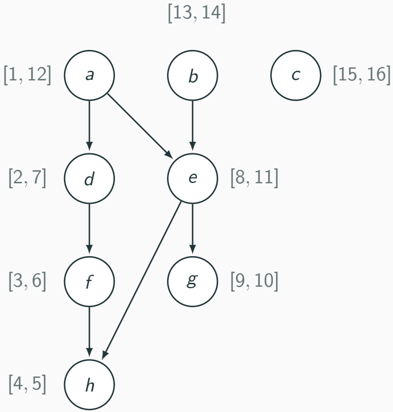
Algorithm runs in $O(n + m)$ time. Correctness is not so obvious.

See next two propositions.

Example



Example

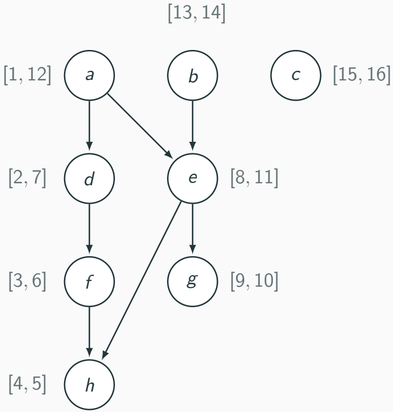


Listing out the vertices in post-number decreasing gives:

c,b,a,e,g,d,f,h

Remind you of anything?

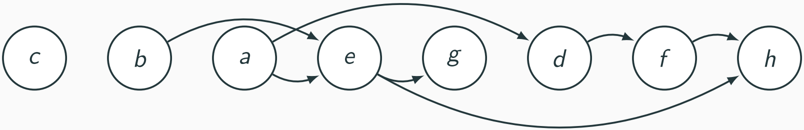
Example



Listing out the vertices in post-number decreasing gives:

c,b,a,e,g,d,f,h

Remind you of anything?



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 $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.

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. □

Decreasing post numbering is valid

Proposition
If G is a DAG and $\mathrm{post}(v) > \mathrm{post}(u)$, *then* $(u \to v)$ *is not in* G.

Proof.
Assume $\mathrm{post}(u) < \mathrm{post}(v)$ and $(u \to v)$ is an edge in G.

Decreasing post numbering is valid

Proposition
If G is a DAG and $\mathrm{post}(v) > \mathrm{post}(u)$, *then* $(u \to v)$ *is not in* G.

Proof.
Assume $\mathrm{post}(u) < \mathrm{post}(v)$ and $(u \to v)$ is an edge in G. One of two holds:

- Case 1: $[pre(u), post(u)]$ is contained in $[pre(v), post(v)]$.
- Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

□

Decreasing post numbering is valid

Proposition
If G is a DAG and $\mathrm{post}(v) > \mathrm{post}(u)$*, then* $(u \to v)$ *is not in G.*

Proof.
Assume $\mathrm{post}(u) < \mathrm{post}(v)$ and $(u \to v)$ is an edge in G. One of two holds:

- 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.

□

Translation

We just proved:

Proposition
If G is a DAG and $\mathrm{post}(v) > \mathrm{post}(u)$, *then* $(u \to v)$ *is not in G.*

$\Longrightarrow$ sort the vertices of a DAG by decreasing post nubmering in decreasing order, then this numbering is valid.

Topological sorting

Theorem
$G = (V, E)$: Graph with n vertices and m edges.

Comptue a topological sorting of G using **DFS** *in $O(n + m)$ time.*

That is, compute a numbering $\pi : V \to \{1, 2, \ldots, n\}$, such that

$$(u \to v) \in E(G) \implies \pi(u) < \pi(v).$$

The meta graph of strong connected components

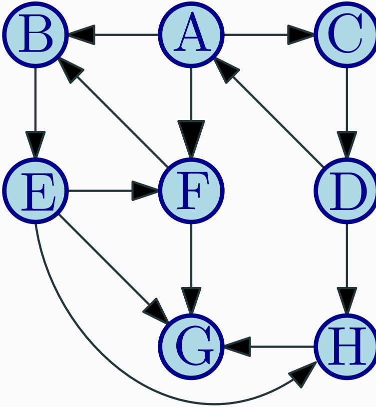
Strong Connected Components (SCCs)

Algorithmic Problem
Find all SCCs of a given directed graph.

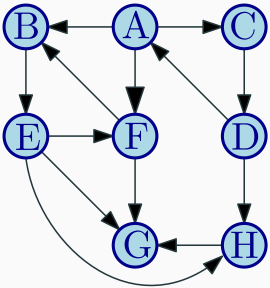
Previous lecture:

Saw an $O(n \cdot (n + m))$ time algorithm.

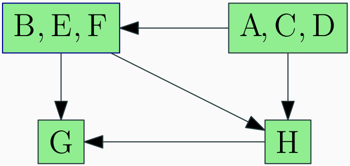
This lecture: sketch of a $O(n + m)$ time algorithm.



Graph of SCCs



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}

- Vertices are $S_1, S_2, \ldots S_k$
- 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.

The meta graph of SCCs is a DAG...

Proposition

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

Proof.

If $\mathsf{G}^{\mathrm{SCC}}$ 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. $\square$

To Remember: Structure of Graphs

Undirected graph: connected components of $G = (V, E)$ partition V and can be computed in $O(m + n)$ time.

Directed graph: the meta-graph $\mathrm{G}^{\mathrm{SCC}}$ of G can be computed in $O(m + n)$ time. $\mathrm{G}^{\mathrm{SCC}}$ gives information on the partition of V into strong connected components and how they form a DAG structure.

Above structural decomposition will be useful in several algorithms

Linear time algorithm for finding all SCCs

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 ∈ 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: $O(n(n + m))$

Finding all SCCs of a Directed Graph

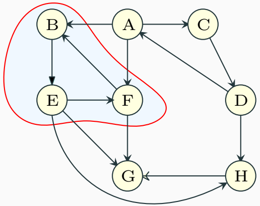
Problem
Given a directed graph $G = (V, E)$, output all its strong connected components.

Straightforward algorithm:

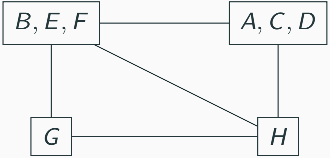
```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u): Mark u as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

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

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.

Linear-time Algorithm for SCCs: Ideas

Wishful Thinking Algorithm

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

Linear-time Algorithm for SCCs: Ideas

Wishful Thinking Algorithm

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

Justification

- **DFS**(u) only visits vertices (and edges) in $SCC(u)$

Linear-time Algorithm for SCCs: Ideas

Wishful Thinking Algorithm

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

Justification

- **DFS**(u) only visits vertices (and edges) in $SCC(u)$
- ... since there are no edges coming out a sink!
-
-

Linear-time Algorithm for SCCs: Ideas

Wishful Thinking Algorithm

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

Justification

- **DFS**(u) only visits vertices (and edges) in $SCC(u)$
- ... since there are no edges coming out a sink!
- **DFS**(u) takes time proportional to size of $SCC(u)$
-

Linear-time Algorithm for SCCs: Ideas

Wishful Thinking Algorithm

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

Justification

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

Big Challenge(s)

How do we find a vertex in a sink *SCC* of $\mathsf{G}^{\mathrm{SCC}}$?

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 $\mathrm{G}^{\mathrm{SCC}}$?

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

Answer: **DFS**(G) gives some information!

Maximum post numbering and the source of the meta-graph

Post numbering and the meta graph

Claim
Let v be the vertex with maximum post numbering in **DFS**(G).
Then v is in a SCC S, such that S is a source of G^{SCC}.

Reverse post numbering and the meta graph

Claim
Let v be the vertex with maximum post numbering in **DFS**(G^{rev})*.*
Then v is in a SCC S, such that S is a sink of G^{SCC}.

Reverse post numbering and the meta graph

Claim

Let v be the vertex with maximum post numbering in **DFS**(G^{rev})*. Then v is in a SCC S, such that S is a sink of G^{SCC}.*

Holds even after we delete the vertices of S (i.e., the vertex with the maximum post numbering, is in a sink of the meta graph).

The linear-time SCC algorithm itself

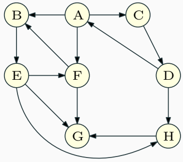
Linear Time Algorithm

```
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
```

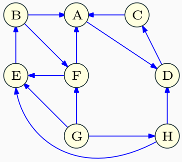
Theorem
Algorithm runs in time $O(m+n)$ and correctly outputs all the SCCs of G.

Linear Time Algorithm: An Example - Initial steps 1

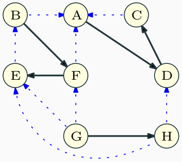
Graph G:



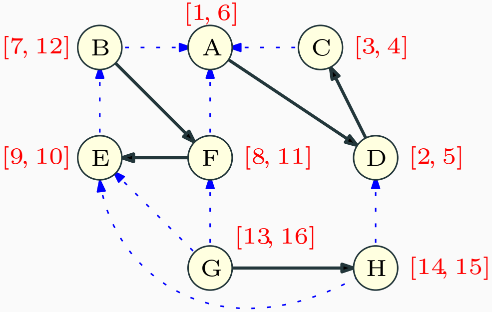
Reverse graph G^{rev}:



DFS of reverse graph:

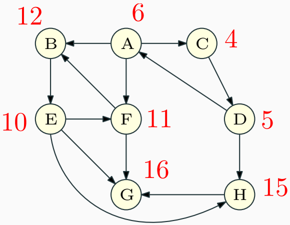


Pre/Post **DFS** numbering of reverse graph:



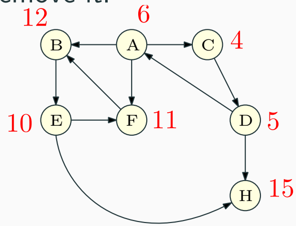
Linear Time Algorithm: An Example

Original graph G with rev post numbers:



$\Longrightarrow$

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

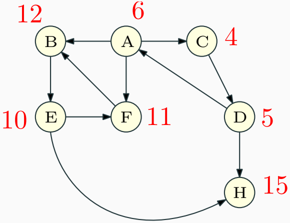


SCC computed:

$\{G\}$

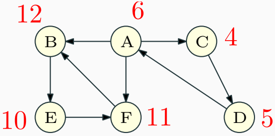
Linear Time Algorithm: An Example

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



$\Longrightarrow$

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



SCC computed:

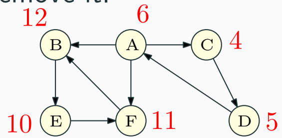
$\{G\}$

SCC computed:

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

Linear Time Algorithm: An Example

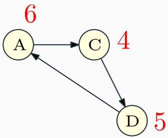
Do **DFS** from vertex H, remove it.



$\Longrightarrow$

Do **DFS** from vertex B

Remove visited vertices: $\{F, B, E\}$.



SCC computed:

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

SCC computed:

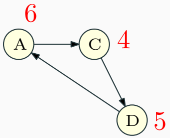
$\{G\}, \{H\}, \{F, B, E\}$

Linear Time Algorithm: An Example

Do **DFS** from vertex F

Remove visited vertices:

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



SCC computed:

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

$\Longrightarrow$

Do **DFS** from vertex A

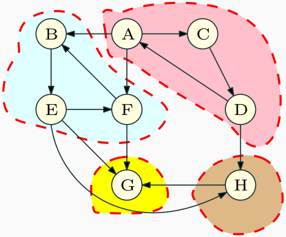
Remove visited vertices:

$\{A, C, D\}$.

SCC computed:

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

Linear Time Algorithm: An Example



SCC computed:

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

Which is the correct answer!

Obtaining the meta-graph...

Exercise:
Given all the strong connected components of a directed graph $G = (V, E)$ show that the meta-graph $\mathrm{G}^{\mathrm{SCC}}$ can be obtained in $O(m + n)$ time.

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}?

Summary

Take away Points

- DAGs
- Topological orderings.
- **DFS**: pre/post numbering.
- 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.
- There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.
- 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).

Scratch Figures