Intro. Algorithms & Models of Computation

CS/ECE 374A, Fall 2024

DAGs, DFS, topological sorting, linear time algorithm for SCC

Lecture 17

Thursday, October 24, 2024

LaTeXed: August 25, 2024 14:23

17.1

Overview: Depth First Search and SCC

Overview

Topics:

- Structure of directed graphs
- DAGs: Directed acyclic graphs.
- Topological ordering.
- **DFS** pre/post number, and its properties.
- Linear time algorithm for SCCs.

17.2

Directed Acyclic Graphs

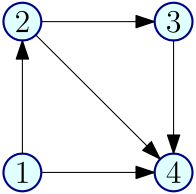
17.2.1

DAGs definition and basic properties

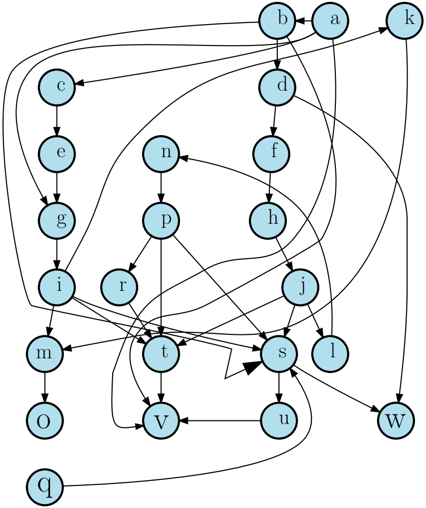
Directed Acyclic Graphs

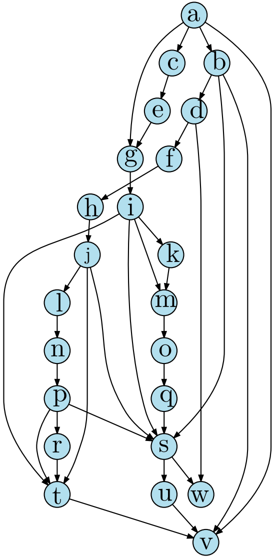
Definition 17.1.

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

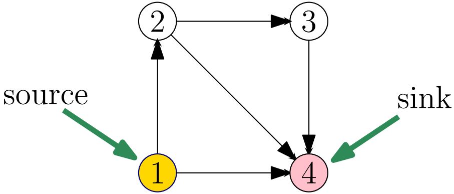


Is this a DAG?





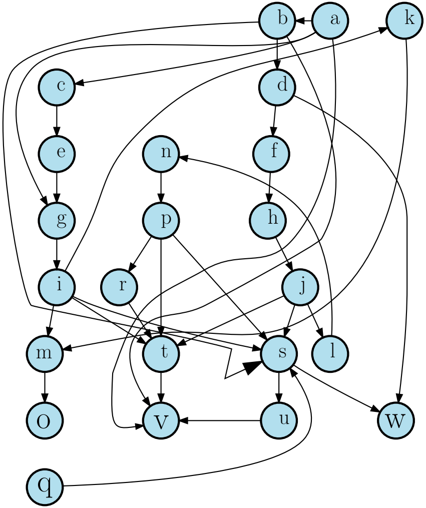
Sources and Sinks



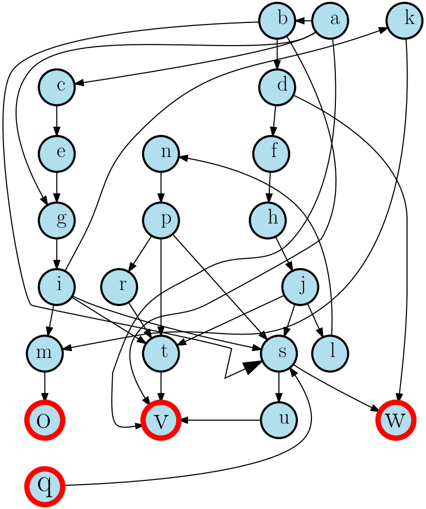
Definition 17.2.

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

Is this a DAG?

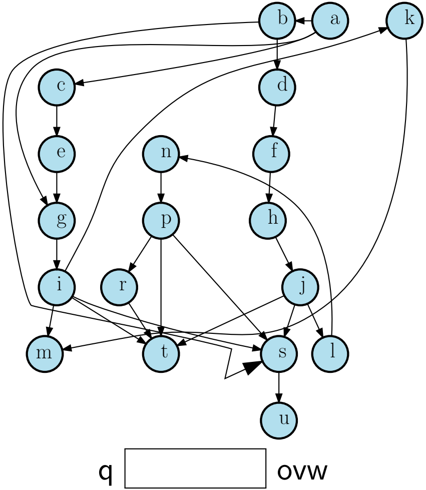


Is this a DAG?

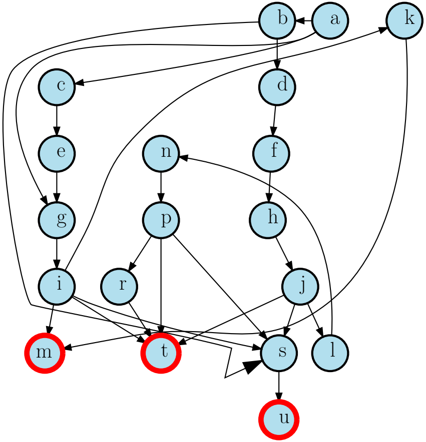


q ☐ ovw

Is this a DAG?

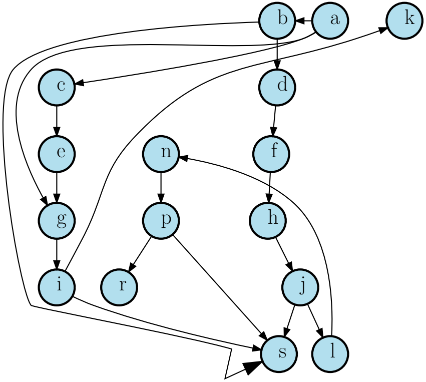


Is this a DAG?



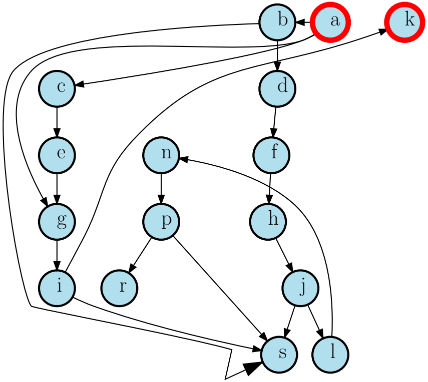
q mtu | ovw

Is this a DAG?



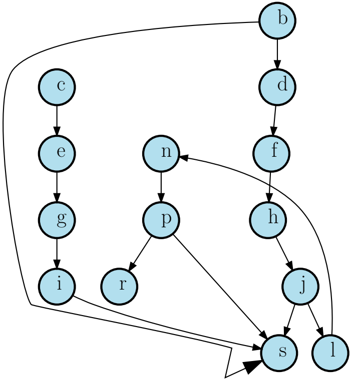
q mtu | ovw

Is this a DAG?



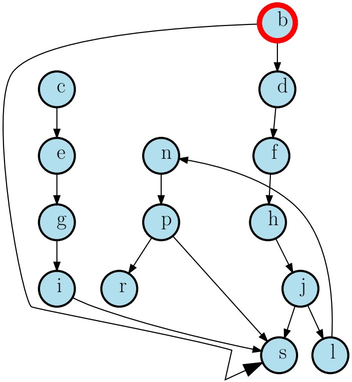
a|q mtu | ovw |k

Is this a DAG?



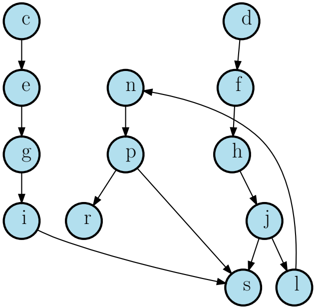
a|q ☐ mtu | ovw |k

Is this a DAG?



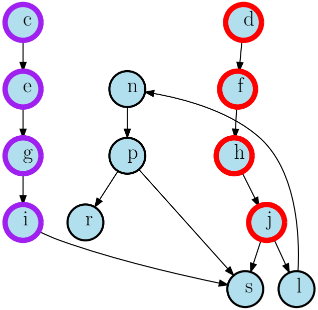
a|q|b mtu|ovw |k

Is this a DAG?



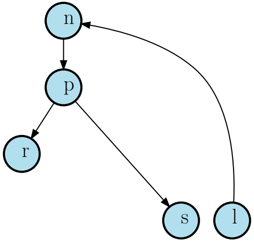
a|q|b mtu|ovw |k

Is this a DAG?



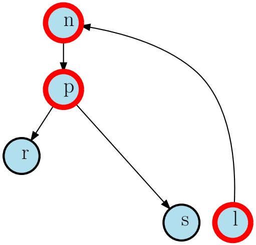
a|q|b|cegidfhj mtu | ovw |k

Is this a DAG?



a|q|b|cegidfhj mtu | ovw |k

Is this a DAG?

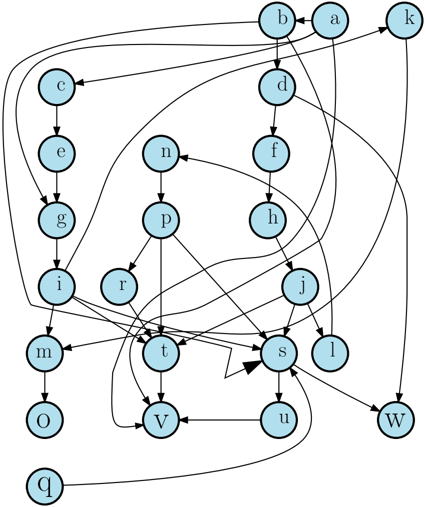


a|q|b|cegidfhj|lnp mtu | ovw |k

Is this a DAG?

a|q|b|cegidfhj|lnp|rs|mtu|ovw|k

Is this a DAG?



a|q|b|cegidfhj|lnp|rs|mtu|ovw|k

abcdefghijklmnopqrstuvw

Simple DAG Properties

Proposition 17.3.

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. ☐

Simple DAG Properties

Proposition 17.3.

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. □

DAG properties

1. G is a DAG if and only if $G^{\mathbf{rev}}$ is a DAG.
2. G is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

17.2.2

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

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

Cannot have $x_1, \ldots, x_m \in X$, such that $x_1 \prec X_2, \ldots, x_{m-1} \prec x_m, x_m \prec x_1$, because...

Order on a (finite) set X: listing the elements of X from smallest to largest.

Total recall: Order on a set

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

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

Cannot have $x_1, \ldots, x_m \in X$, such that $x_1 \prec X_2, \ldots, x_{m-1} \prec x_m, x_m \prec x_1$, because...

Order on a (finite) set X: listing the elements of X from smallest to largest.

Total recall: Order on a set

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

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

Cannot have $x_1, \ldots, x_m \in X$, such that $x_1 \prec X_2, \ldots, x_{m-1} \prec x_m, x_m \prec x_1$, because...

Order on a (finite) set X: listing the elements of X from smallest to largest.

Convention about writing edges

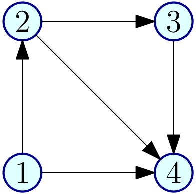
1. Undirected graph edges:

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

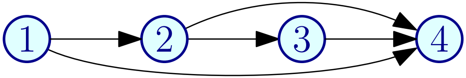
2. Directed graph edges:

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

Topological Ordering/Sorting



Graph G



Topological Ordering of G

Definition 17.4.

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.

DAGs and Topological Sort

Lemma 17.5.

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

Need to show both directions.

DAGs and Topological Sort

Lemma 17.6.

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

Proof.

Consider the following algorithm:

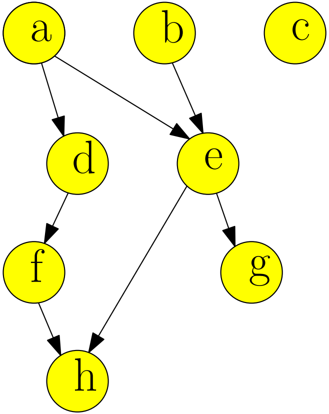
1. Pick a source $\boldsymbol{u}$, output it.
2. Remove $\boldsymbol{u}$ and all edges out of $\boldsymbol{u}$.
3. Repeat until graph is empty.

Exercise: prove this gives topological sort. □

Topological ordering in linear time

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

Topological Sort: Example



DAGs and Topological Sort

Lemma 17.7.

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. ☐

DAGs and Topological Sort

Lemma 17.7.

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. □

Regular sorting and DAGs

DAGs and Topological Sort

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

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

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

17.2.2.1

Explicit definition of what topological ordering

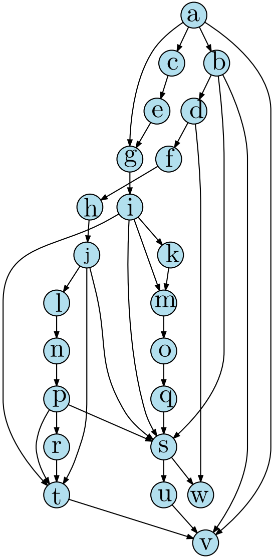
An explicit definition of what topological ordering of a graph is

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

$$\forall\, (u \to v) \in \mathsf{E}(\mathsf{G}) \implies \pi(u) < \pi(v).$$

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

Example...



17.3

Depth First Search (DFS)

17.3.1

Depth First Search (DFS) in Undirected Graphs

Depth First Search

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

DFS in Undirected Graphs

Recursive version. Easier to understand some properties.

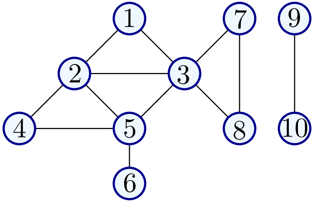
```
DFS($G$)
    for all $u \in V(G)$ do
        Mark $u$ as unvisited
        Set pred($u$) to null
    $T$ is set to $\emptyset$
    while $\exists$ unvisited $u$ do
        DFS($u$)
    Output $T$
```

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

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

Properties of DFS tree

Proposition 17.1.

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:*
    - 3.1 *u is an ancestor of v, or*
    - 3.2 *v is an ancestor of u.*

Question: Why are there no cross-edges?

Exercise

Prove that **DFS** of a graph G with n vertices and m edges takes $O(n + m)$ time.

17.3.2

DFS with pre-post numbering

DFS with Visit Times

Keep track of when nodes are visited.

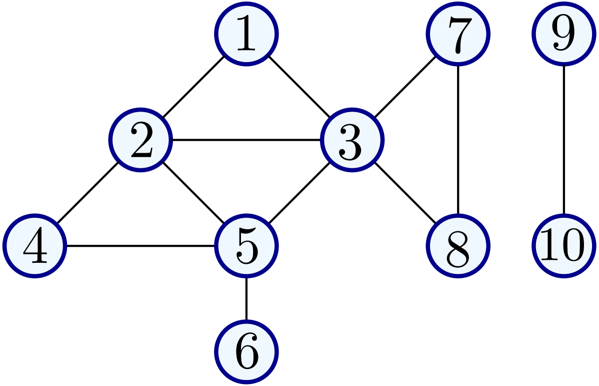
```
DFS($G$)
    for all $u \in V(G)$ do
        Mark $u$ as unvisited
    $T$ is set to $\emptyset$
    $time = 0$
    while $\exists$ unvisited $u$ do
        DFS($u$)
    Output $T$
```

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

Animation

$$\boldsymbol{time = 0}$$

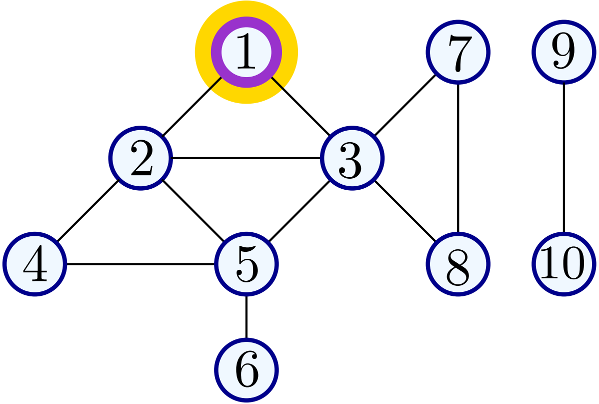
vertex	$[pre, post]$



Animation

$time = 1$

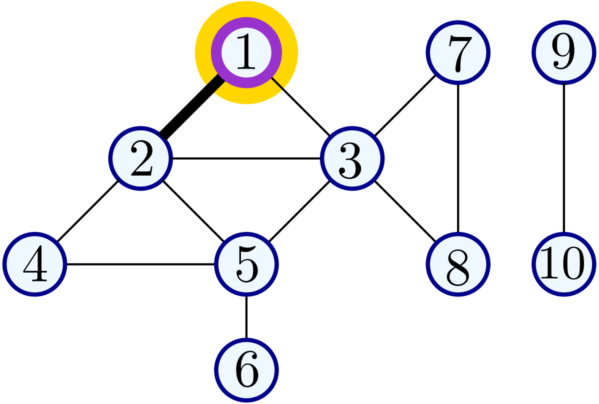
vertex	$[pre, post]$
1	$[1,]$



Animation

$$\boldsymbol{time = 1}$$

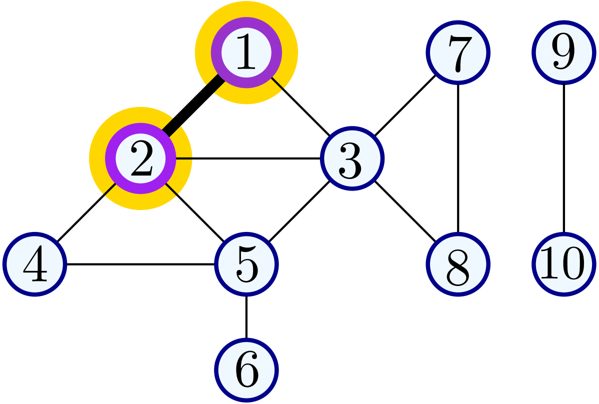
vertex	[*pre*, *post*]
1	**[1,]**



Animation

***time* = 2**

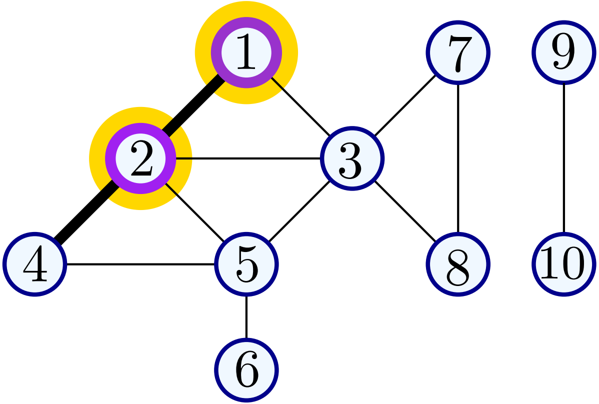
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**



Animation

time **= 2**

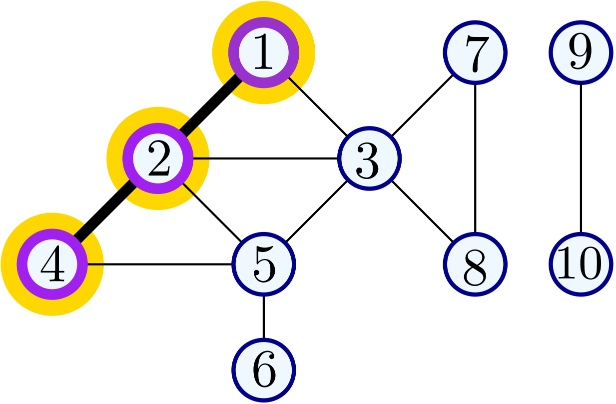
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**



Animation

***time* = 3**

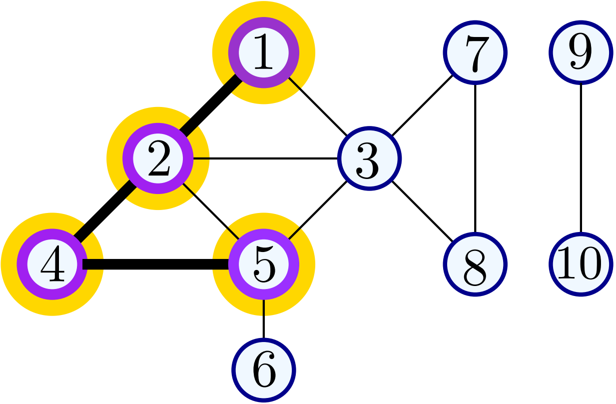
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**



Animation

***time* = 4**

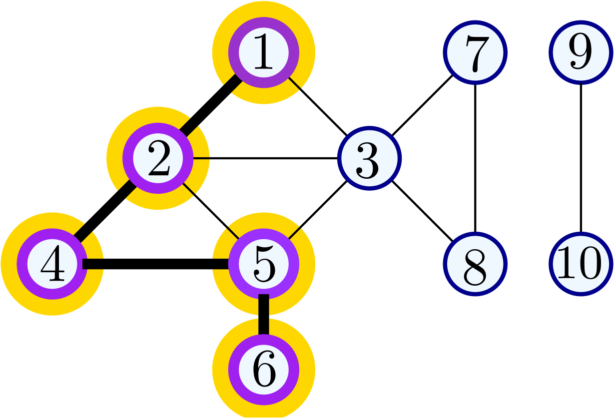
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**



Animation

$time = 5$

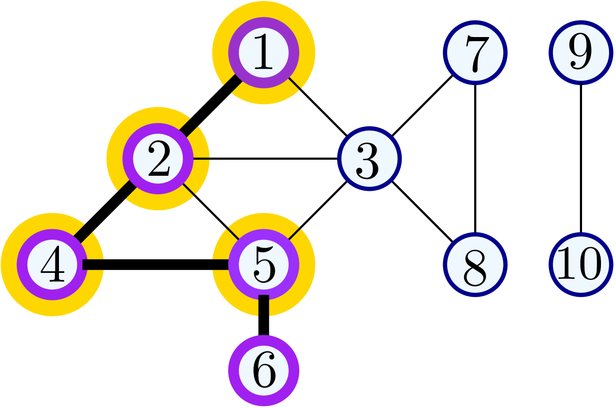
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**
6	**[5,]**



Animation

$time = 6$

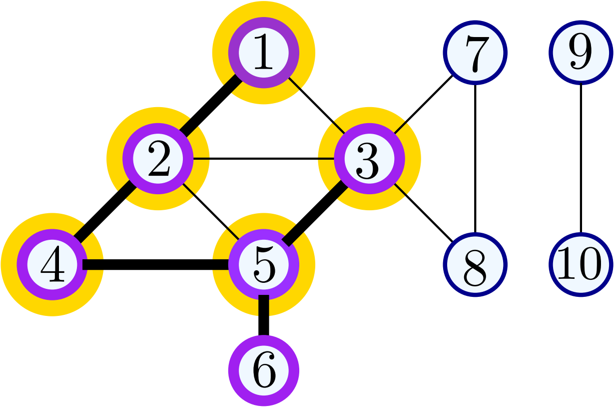
vertex	$[pre, post]$
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**
6	**[5, 6]**



Animation

***time* = 7**

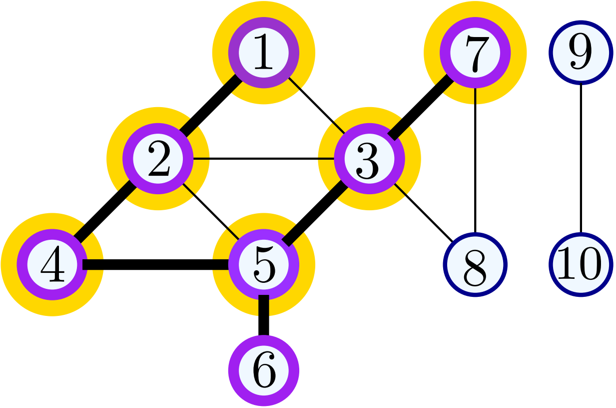
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**
6	**[5, 6]**
3	**[7,]**



Animation

***time* = 8**

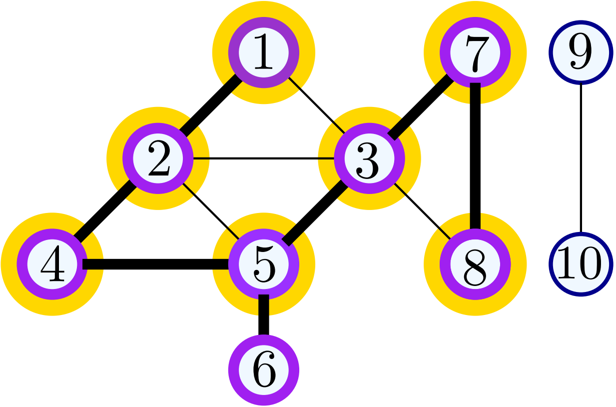
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**
6	**[5, 6]**
3	**[7,]**
7	**[8,]**



Animation

$\textit{time} = 9$

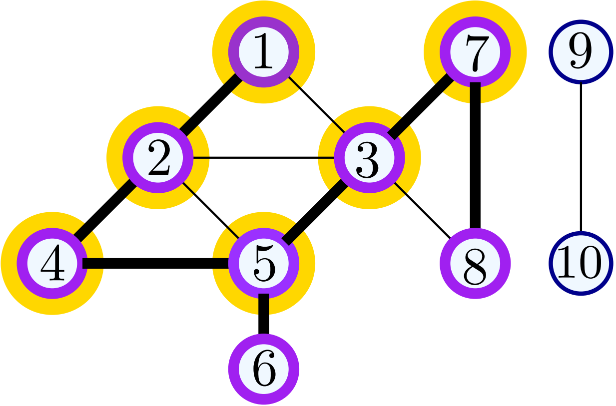
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**
6	**[5, 6]**
3	**[7,]**
7	**[8,]**
8	**[9,]**



Animation

time = 10

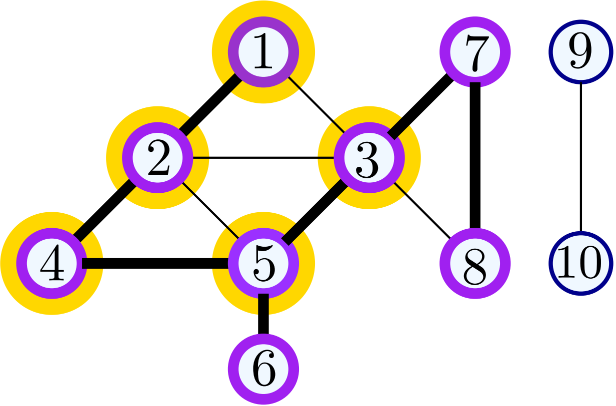
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**
6	**[5, 6]**
3	**[7,]**
7	**[8,]**
8	**[9, 10]**



Animation

time = 11

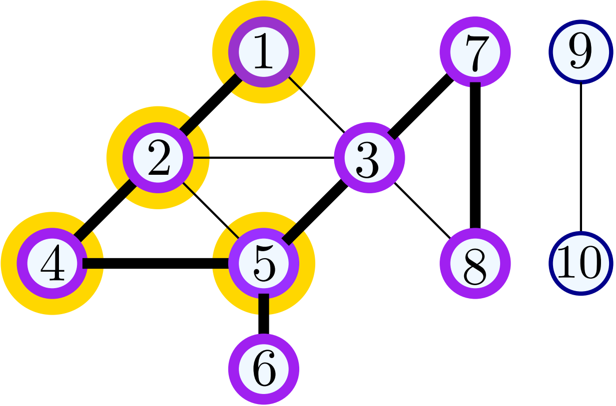
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**
6	**[5, 6]**
3	**[7,]**
7	**[8, 11]**
8	**[9, 10]**



Animation

***time* = 12**

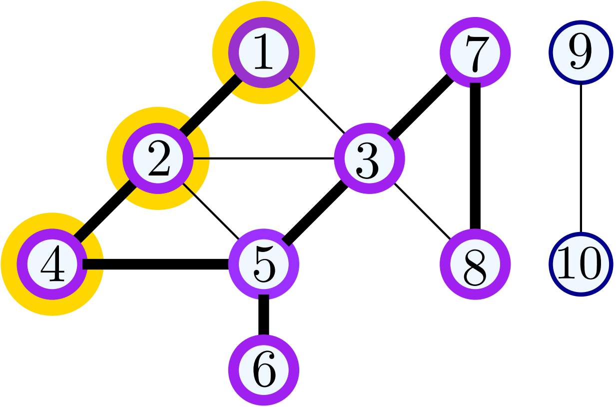
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4,]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**



Animation

time = 13

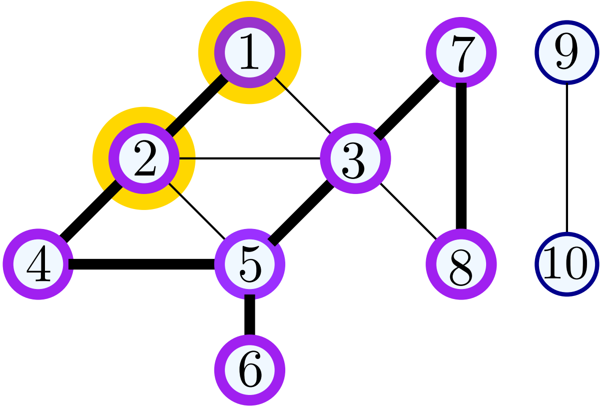
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3,]**
5	**[4, 13]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**



Animation

***time* = 14**

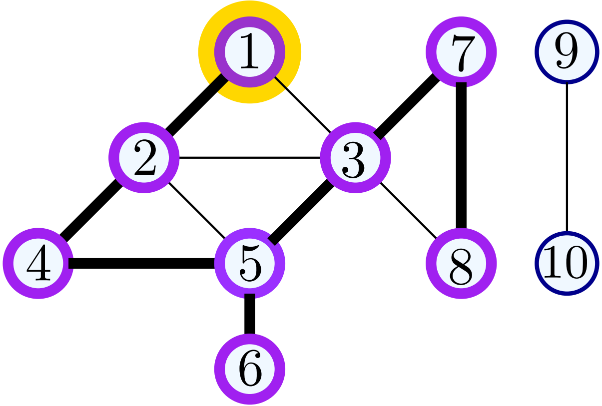
vertex	[*pre*, *post*]
1	**[1,]**
2	**[2,]**
4	**[3, 14]**
5	**[4, 13]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**



Animation

***time* = 15**

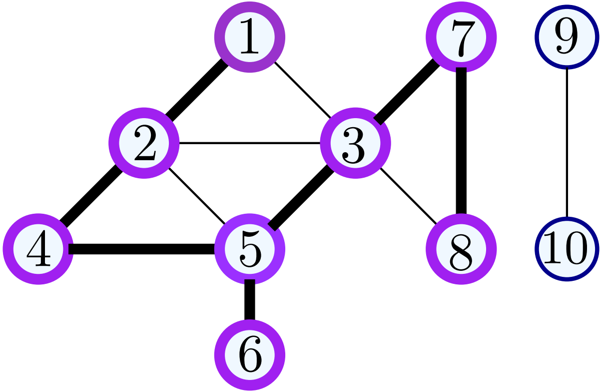
vertex	**[*pre*, *post*]**
1	**[1,]**
2	**[2, 15]**
4	**[3, 14]**
5	**[4, 13]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**



Animation

***time* = 16**

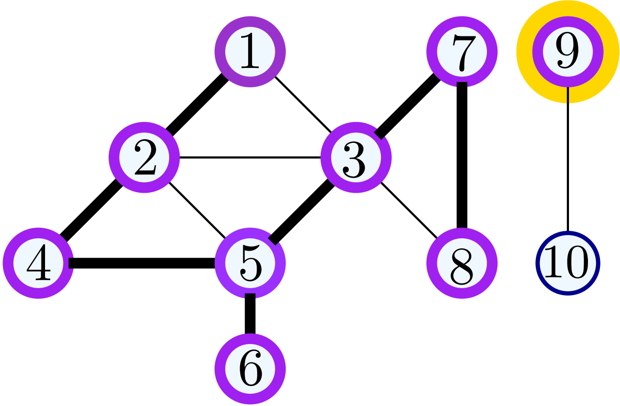
vertex	[*pre*, *post*]
1	**[1, 16]**
2	**[2, 15]**
4	**[3, 14]**
5	**[4, 13]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**



Animation

$\textit{time} = 17$

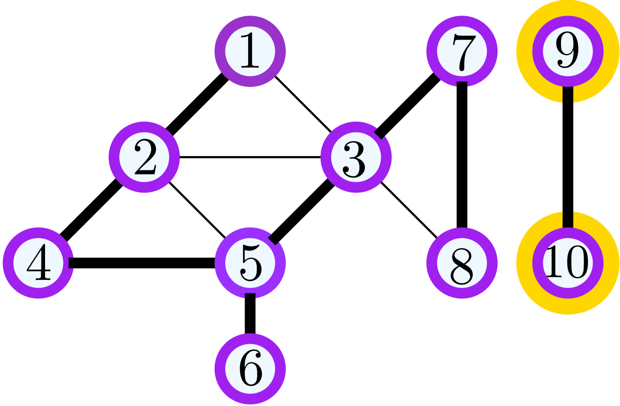
vertex	[*pre*, *post*]
1	**[1, 16]**
2	**[2, 15]**
4	**[3, 14]**
5	**[4, 13]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**
9	**[17,]**



Animation

$\boldsymbol{time = 18}$

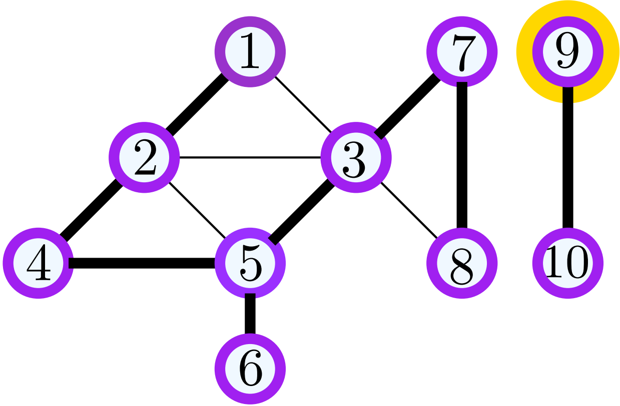
vertex	$[pre, post]$
1	**[1, 16]**
2	**[2, 15]**
4	**[3, 14]**
5	**[4, 13]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**
9	**[17,]**
10	**[18,]**



Animation

$\textit{time} = 19$

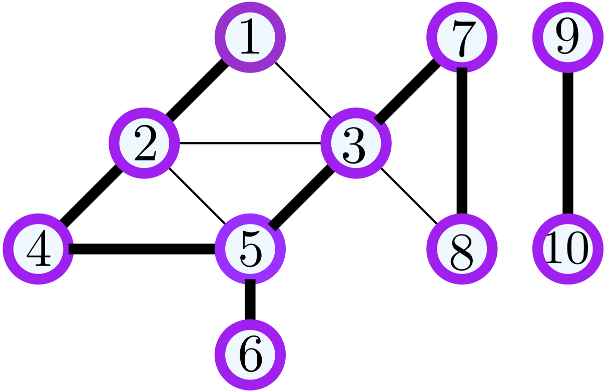
vertex	[*pre*, *post*]
1	**[1, 16]**
2	**[2, 15]**
4	**[3, 14]**
5	**[4, 13]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**
9	**[17,]**
10	**[18, 19]**



Animation

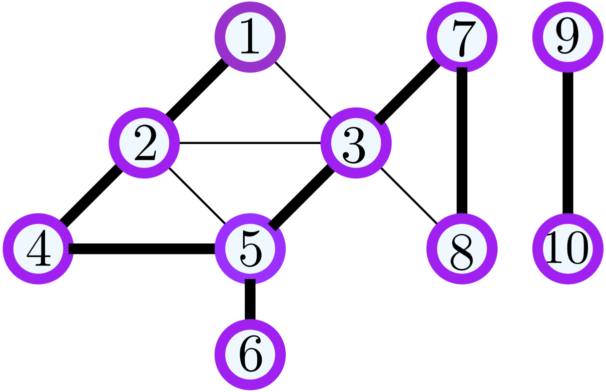
$time = 20$

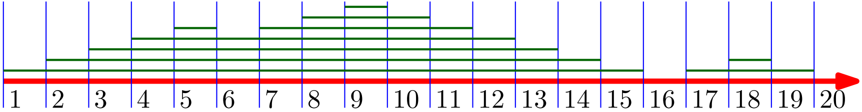
vertex	$[pre, post]$
1	**[1, 16]**
2	**[2, 15]**
4	**[3, 14]**
5	**[4, 13]**
6	**[5, 6]**
3	**[7, 12]**
7	**[8, 11]**
8	**[9, 10]**
9	**[17, 20]**
10	**[18, 19]**



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]**





$\mathbf{pre}$ and $\mathbf{post}$ numbers

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

Proposition 17.2.

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**

$\mathbf{pre}$ and $\mathbf{post}$ numbers

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

Proposition 17.2.

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, $\mathrm{pre}(v) > \mathrm{post}(u)$.

☐

$\mathbf{pre}$ and $\mathbf{post}$ numbers useful in several applications of **DFS**

pre and post numbers

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

Proposition 17.2.

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**

17.4

DFS in Directed Graphs

17.4.1

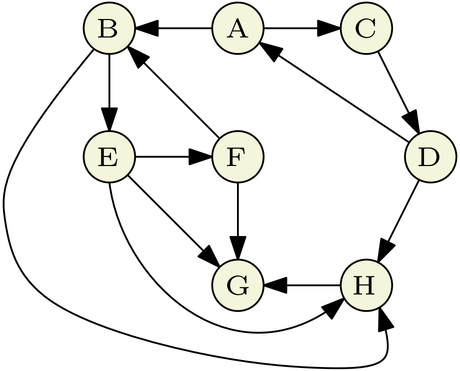
DFS in Directed Graphs: Pre/Post numbering

DFS in Directed Graphs

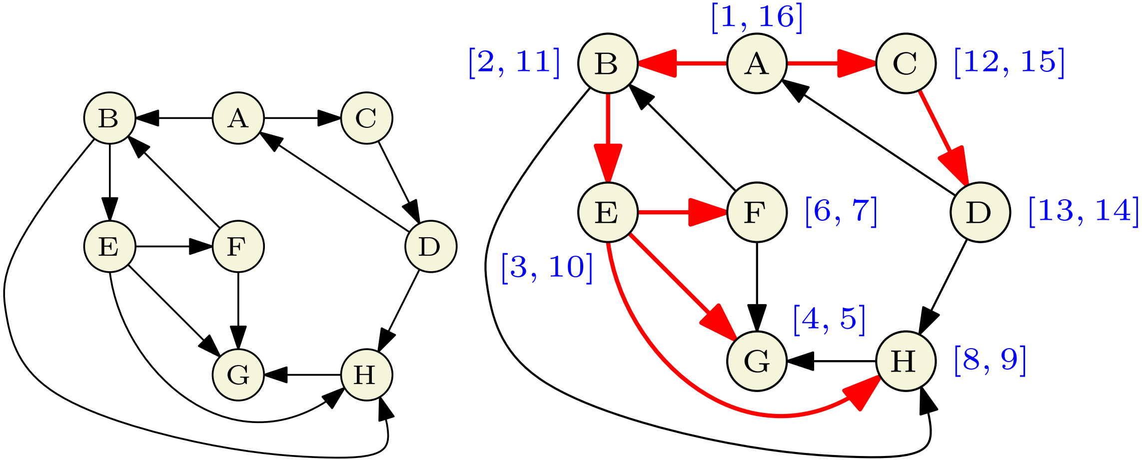
```
DFS($G$)
    Mark all nodes $u$ as unvisited
    $T$ is set to $\emptyset$
    $time = 0$
    while there is an unvisited node $u$ do
        DFS($u$)
    Output $T$
```

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

Example of DFS in directed graph



Example of DFS in directed graph



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 directed graphs but it is.

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 directed graphs but it is.

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 directed graphs but it is.

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 directed graphs but it is.

DFS Properties

Generalizing ideas from undirected graphs:

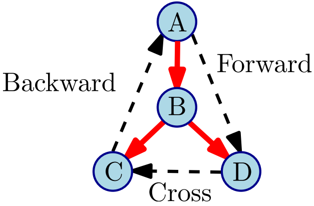
1. **DFS**(G) takes $O(m+n)$ time.
2. Edges added form a branching: a forest of out-trees. Output of $DFS(G)$ depends on the order in which vertices are considered.
3. If u is the first vertex considered by $DFS(G)$ then $DFS(u)$ outputs a directed out-tree T rooted at u and a vertex v is in T if and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

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

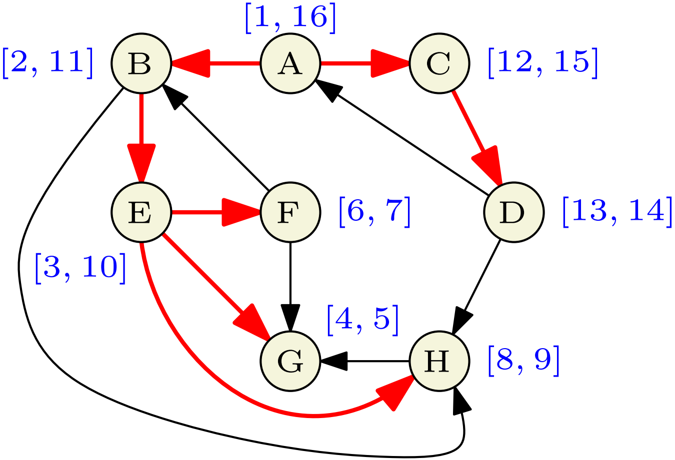
DFS tree and related edges

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

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



Types of Edges



17.4.2

DFS and cycle detection: Topological sorting using **DFS**

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?

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?

Cycle detection in directed graph using topological sorting

Question

Given G, is it a DAG?

If it is, compute a topological sort.
If it is not, then output the cycle C.

Topological sort a graph using DFS...

And detect a cycle in the process

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.

Computes topological ordering of the vertices.

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

Topological sort a graph using DFS...

And detect a cycle in the process

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.

Computes topological ordering of the vertices.

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

Topological sort a graph using DFS...

And detect a cycle in the process

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.

Computes topological ordering of the vertices.

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

Back edge and Cycles

Proposition 17.1.

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. ☐

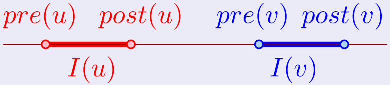
Decreasing post numbering is valid

Proposition 17.2.

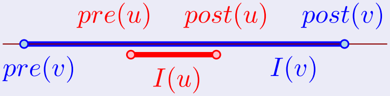
Let G be a DAG. *If* $\mathbf{post}(v) > \mathbf{post}(u)$, *then* $(u \to v)$ *is not in* G.

Proof.

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



: But if $(u \to v) \in E(G) \implies I(v) \subseteq I(v)$.



: u is decedent of v in **DFS** tree $\implies$ $(u \to v)$ is a back edge $\implies$ there is a cycle in G. Contradiction. □

Decreasing post numbering is valid (alt proof)

Proposition 17.3.

Let G be a DAG*. If* $\mathbf{post}(v) > \mathbf{post}(u)$*, then* $(u \rightarrow v)$ *is not in* G*.*

Proof.

Assume $\mathbf{post}(u) < \mathbf{post}(v)$ and $(u \rightarrow v)$ 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 (alt proof)

Proposition 17.3.

Let G be a DAG*. If* $\mathbf{post}(v) > \mathbf{post}(u)$*, then* $(u \to v)$ *is not in* G*.*

Proof.

Assume $\mathbf{post}(u) < \mathbf{post}(v)$ 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 (alt proof)

Proposition 17.3.

Let G be a DAG*. If* $\mathbf{post}(v) > \mathbf{post}(u)$*, then* $(u \rightarrow v)$ *is not in* G*.*

Proof.

Assume $\mathbf{post}(u) < \mathbf{post}(v)$ $\underline{\text{and}}$ $(u \rightarrow v)$ 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 descendant 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 17.4.

If G is a DAG and $\mathbf{post}(v) > \mathbf{post}(u)$, then $(u \rightarrow v)$ is not in G.

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

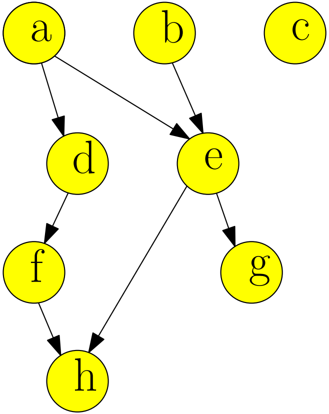
Topological sorting

Theorem 17.5.

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

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

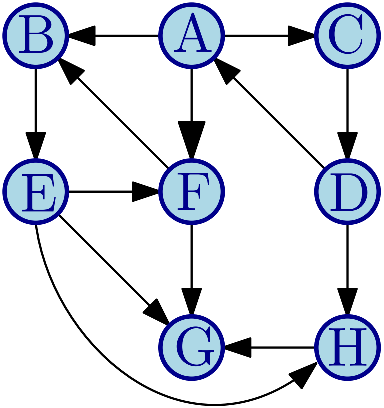
Example



17.5

The meta graph of strong connected components

Strong Connected Components (SCCs)

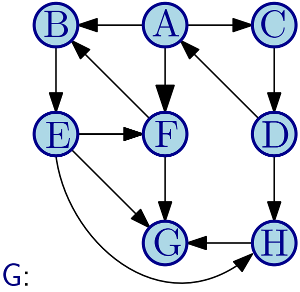


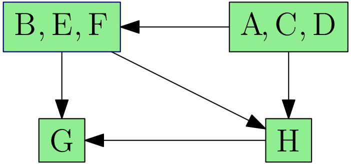
Algorithmic Problem

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





Graph of SCCs $\mathrm{G}^{\mathrm{SCC}}$

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

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.

Reversal and SCCs

Proposition 17.1.

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

Proof.

Exercise. $\square$

MUTTS by Patrick McDonnell | 08/04/11

The meta graph of SCCs is a DAG...

Proposition 17.2.

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. ☐

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 G^{SCC} of G can be computed in $O(m + n)$ time. G^{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

17.6

Linear time algorithm for finding all strong connected components of a directed graph

17.6.1

Wishful thinking linear-time SCC algorithm

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 $\mathrm{SCC}(G, u)$ the strong component of $u$:
        Compute $\mathrm{rch}(G, u)$ using $DFS(G, u)$
        Compute $\mathrm{rch}(G^{\mathrm{rev}}, u)$ using $DFS(G^{\mathrm{rev}}, u)$
        $\mathrm{SCC}(G, u) \Leftarrow \mathrm{rch}(G, u) \cap \mathrm{rch}(G^{\mathrm{rev}}, u)$
        $\forall u \in \mathrm{SCC}(G, u)$:  Mark $u$ as visited.
```

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

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

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 $\mathrm{SCC}(G, u)$ the strong component of $u$:
        Compute $\mathrm{rch}(G, u)$ using $DFS(G, u)$
        Compute $\mathrm{rch}(G^{\mathrm{rev}}, u)$ using $DFS(G^{\mathrm{rev}}, u)$
        $\mathrm{SCC}(G, u) \Leftarrow \mathrm{rch}(G, u) \cap \mathrm{rch}(G^{\mathrm{rev}}, u)$
        $\forall u \in \mathrm{SCC}(G, u)$:  Mark $u$ as visited.
```

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

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

Finding all SCCs of a Directed Graph

Problem

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

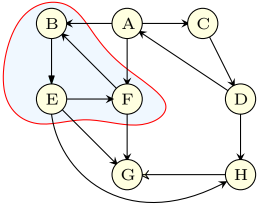
Straightforward algorithm:

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

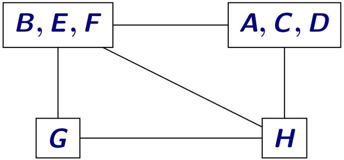
Running time: $O(n(n + m))$

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 17.1.

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

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 $\mathbf{SCC}(u)$
3. Remove $\mathbf{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$
2.
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 $\mathbf{SCC}(u)$
3. Remove $\mathbf{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$
2.
3.
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

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

Justification

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

Justification

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

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!

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}?

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

Big Challenge(s)

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

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

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

17.6.2

Maximum post numbering and the source of the meta-graph

Post numbering and the meta graph

Claim 17.2.

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 17.3.

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).

Reverse post numbering and the meta graph

Claim 17.3.

Let v be the vertex with maximum post numbering in **DFS**$(G^{\mathbf{rev}})$*. 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).

17.6.3

The linear-time SCC algorithm itself

Linear Time Algorithm

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

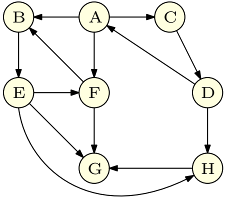
```
do DFS($G^{\text{rev}}$) and output vertices in decreasing post order.
Mark all nodes as unvisited
for each $u$ in the computed order do
    if $u$ is not visited then
        DFS($u$)
        Let $S_u$ be the nodes reached by $u$
        Output $S_u$ as a strong connected component
        Remove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

Theorem 17.4.

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

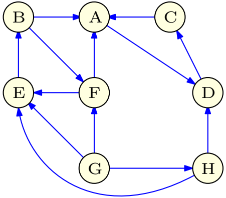
Linear Time Algorithm: An Example - Initial steps 1

Graph G:



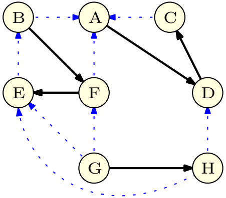
$\Longrightarrow$

Reverse graph G^{rev}:



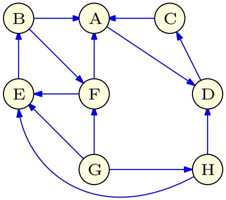
$\Longrightarrow$

DFS of reverse graph:



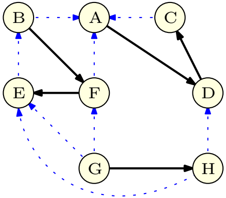
Linear Time Algorithm: An Example - Initial steps 2

Reverse graph G^{rev}:



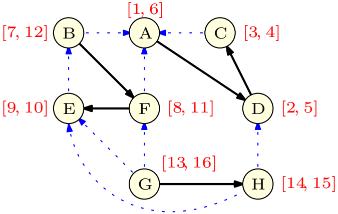
$\Longrightarrow$

DFS of reverse graph:



$\Longrightarrow$

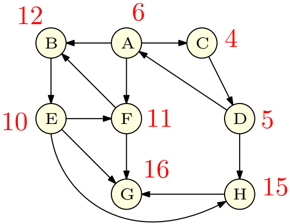
Pre/Post **DFS** numbering of reverse graph:



Linear Time Algorithm: An Example

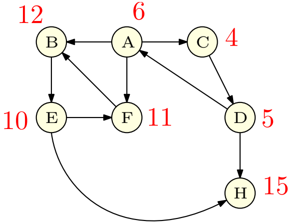
Removing connected components: 1

Original graph G with rev post numbers:



$\Longrightarrow$

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

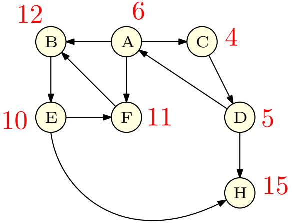


SCC computed:
$\{G\}$

Linear Time Algorithm: An Example

Removing connected components: 2

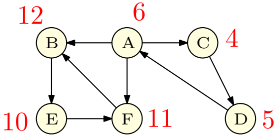
Do **DFS** from vertex G remove it.



SCC computed:
$\{G\}$

$\Longrightarrow$

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

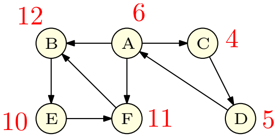


SCC computed:
$\{G\}, \{H\}$

Linear Time Algorithm: An Example

Removing connected components: 3

Do **DFS** from vertex $\boldsymbol{H}$, remove it.

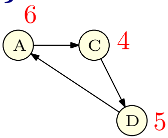


$\Longrightarrow$

Do **DFS** from vertex $\boldsymbol{B}$

Remove visited vertices:

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



SCC computed:

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

SCC computed:

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

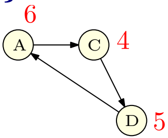
Linear Time Algorithm: An Example

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:

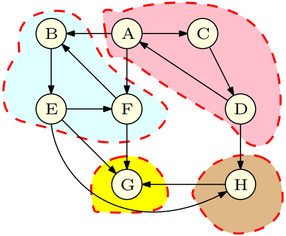
$\{A, C, D\}$.

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!

Obtaining the meta-graph...

Once the strong connected components are computed.

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}?

17.7

An Application of directed graphs to `make`

Make/Makefile

(A) I know what make/makefile is.

(B) I do NOT know what make/makefile is.

`make` Utility [Feldman]

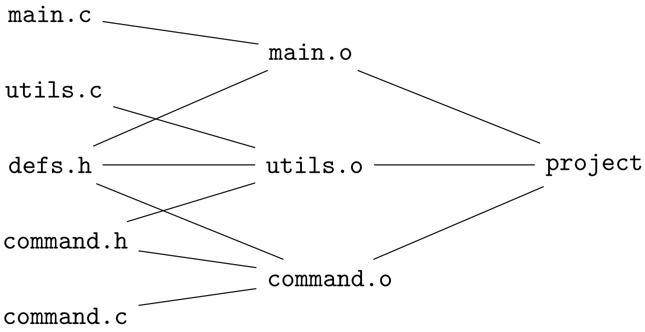
1. Unix utility for automatically building large software applications
2. A makefile specifies
    2.1 Object files to be created,
    2.2 Source/object files to be used in creation, and
    2.3 How to create them

An Example `makefile`

```
project:  main.o utils.o command.o
    cc -o project main.o utils.o command.o

main.o:  main.c defs.h
    cc -c main.c
utils.o:  utils.c defs.h command.h
    cc -c utils.c
command.o:  command.c defs.h command.h
    cc -c command.c
```

`makefile` as a Digraph



Computational Problems for `make`

1. Is the `makefile` reasonable?
2. If it is reasonable, in what order should the object files be created?
3. If it is not reasonable, provide helpful debugging information.
4. If some file is modified, find the fewest compilations needed to make application consistent.

Algorithms for `make`

1. Is the `makefile` reasonable? Is G a DAG?
2. If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
3. If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
4. If some file is modified, find the fewest compilations needed to make application consistent.
    4.1 Find all vertices reachable (using **DFS**/**BFS**) from modified files in directed graph, and recompile them in proper order. Verify that one can find the files to recompile and the ordering in linear time.

17.8

Summary

Take away Points

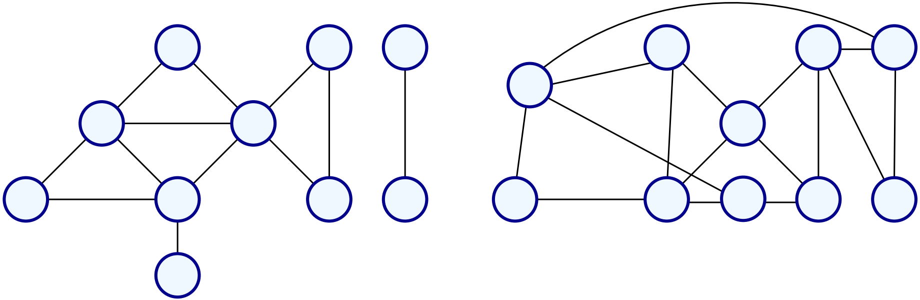
1. DAGs
2. Topological orderings.
3. **DFS**: pre/post numbering.
4. Given a directed graph G, its SCCs and the associated acyclic meta-graph G^{SCC} give a structural decomposition of G that should be kept in mind.
5. There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.
6. 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).

17.9

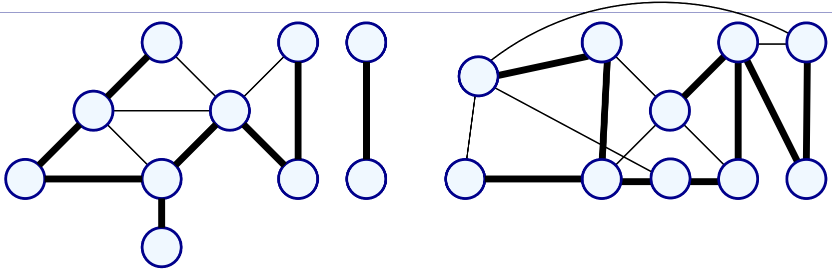
An example of DFS forests

Example: Undirected **DFS** forest

The input graph (disconnected in this case):

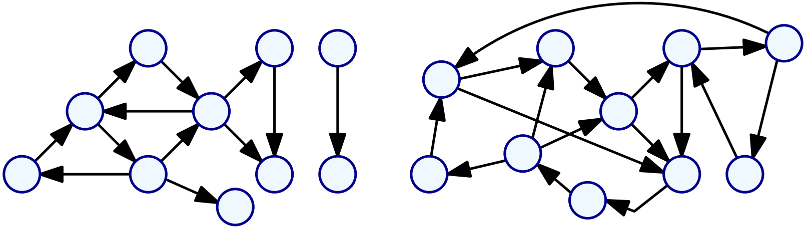


The resulting **DFS** forest:



Example: Directed **DFS** forest

The input graph:



The resulting **DFS** forest (numbers indicate the order of **DFS**):