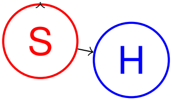
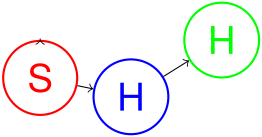
CS 170: Algorithms

CS 170: Algorithms

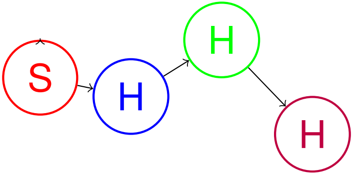
CS 170: Algorithms



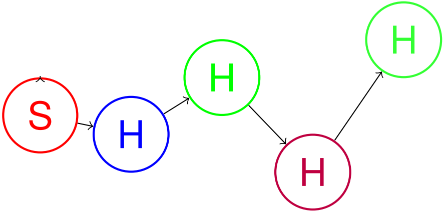
CS 170: Algorithms



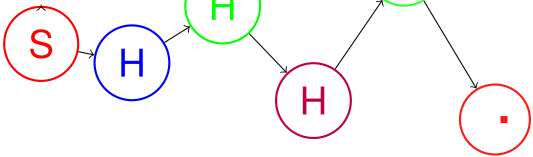
CS 170: Algorithms



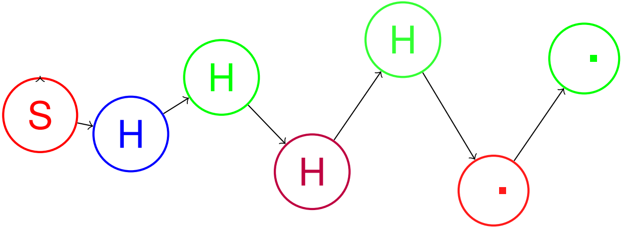
CS 170: Algorithms



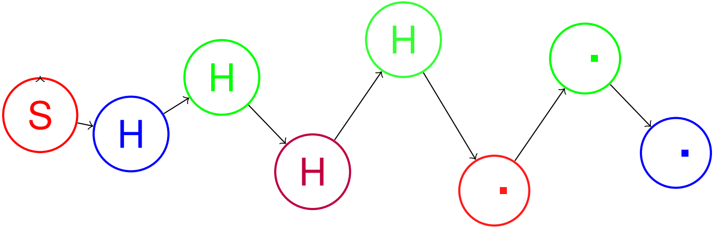
CS 170: Algorithms



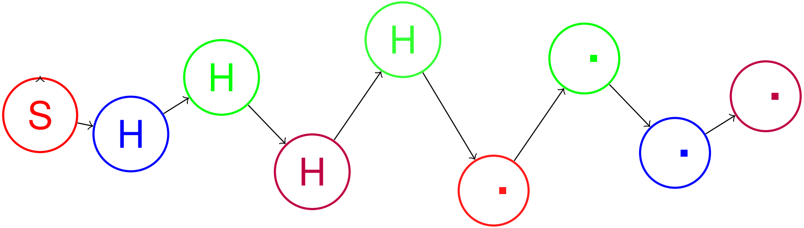
CS 170: Algorithms



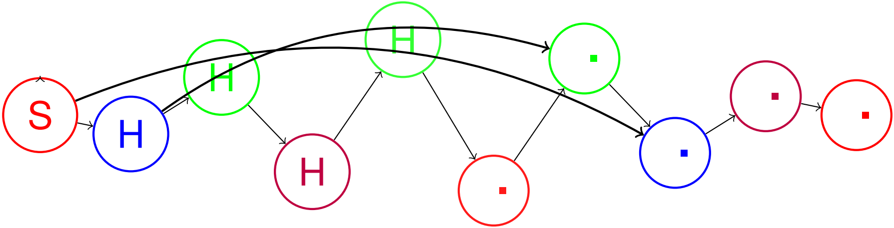
CS 170: Algorithms



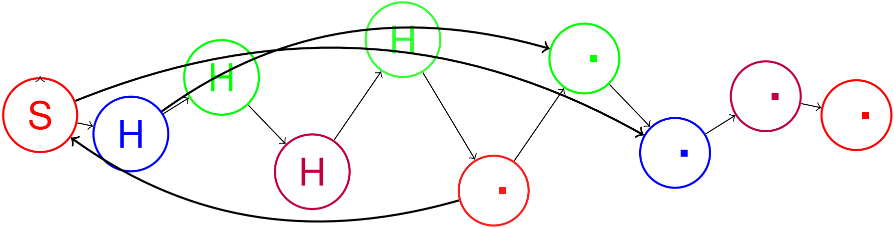
CS 170: Algorithms



CS 170: Algorithms



CS 170: Algorithms



Directed graphs...with cycles.

Lecture in a minute!

Quick Review:

DFS so far $\equiv$

Lecture in a minute!

Quick Review:
- DFS so far $\equiv$ how I learned to love the stack.
  - pre/post = time on stack.
- Topological Ordering:
  - Inverse post ordering $\equiv$ topological ordering.
  - Remove source, repeat.

Lecture in a minute!

Quick Review:
  DFS so far $\equiv$ how I learned to love the stack.
   pre/post = time on stack.
  Topological Ordering:
    Inverse post ordering $\equiv$ topological ordering.
    Remove source, repeat.

Strongly Connected Components: directed graphs.
 Strong Connectivity for u and v.
   On a cycle together.
 Easy: $O(|V||E|)$ algorithm.
 Linear time algorithm!

Lecture in a minute!

Quick Review:
DFS so far $\equiv$ how I learned to love the stack.
pre/post = time on stack.
Topological Ordering:
Inverse post ordering $\equiv$ topological ordering.
Remove source, repeat.

Strongly Connected Components: directed graphs.
Strong Connectivity for u and v.
On a cycle together.
Easy: $O(|V||E|)$ algorithm.
Linear time algorithm!
Observation: Highest post in "source component".
Find vertex in sink component.
Explore.
Repeat.

DFS: so far.

DFS: so far.

How I learned to stop worrying

DFS: so far.

How I learned to stop worrying
...and love the stack.

DFS: so far.

How I learned to stop worrying

...and love the stack. See "Dr. Strangelove".

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”

$\vdots$

u

$\vdots$

v

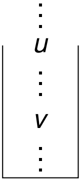
$\vdots$

Who is an ancestor of whom?

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”



Who is an ancestor of whom?

v is an ancestor of *u*

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”

$\vdots$
u
$\vdots$
v
$\vdots$

Who is an ancestor of whom?

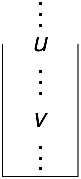
v is an ancestor of u

No cycles: no back edge

DFS: so far.

How I learned to stop worrying
...and love the stack. See "Dr. Strangelove".

$[pre(u), post(u)]$ is "clock interval on stack."



Who is an ancestor of whom?

v is an ancestor of u

No cycles: no back edge
Back edge – edge to ancestor in "tree" of explore calls."

DFS: so far.

How I learned to stop worrying
...and love the stack. See "Dr. Strangelove".

$[pre(u), post(u)]$ is "clock interval on stack."

$\vdots$
u
$\vdots$
v
$\vdots$

Who is an ancestor of whom?

v is an ancestor of u

No cycles: no back edge
Back edge – edge to ancestor in "tree" of explore calls."

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”

$$\begin{array}{|c|} \vdots \\ u \\ \vdots \\ v \\ \vdots \\ \hline \end{array}$$

Who is an ancestor of whom?

v is an ancestor of u

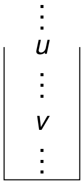
No cycles: no back edge
Back edge – edge to ancestor in “tree” of explore calls.”

Topological order - edge (u, v) means u before v

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”



Who is an ancestor of whom?

v is an ancestor of u

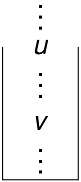
No cycles: no back edge
Back edge – edge to ancestor in “tree” of explore calls.”

Topological order - edge (u, v) means u before v
Inverse post ordering is topological order.

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”



Who is an ancestor of whom?

v is an ancestor of u

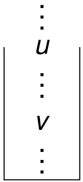
No cycles: no back edge
Back edge – edge to ancestor in “tree” of explore calls.”

Topological order - edge (u, v) means u before v
Inverse post ordering is topological order.
Bonus Topological Order Algorithm:

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”



Who is an ancestor of whom?

v is an ancestor of u

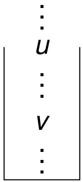
No cycles: no back edge
Back edge – edge to ancestor in “tree” of explore calls.”

Topological order - edge (u, v) means u before v
Inverse post ordering is topological order.
Bonus Topological Order Algorithm:
remove source,

DFS: so far.

How I learned to stop worrying
...and love the stack. See “Dr. Strangelove”.

$[pre(u), post(u)]$ is “clock interval on stack.”



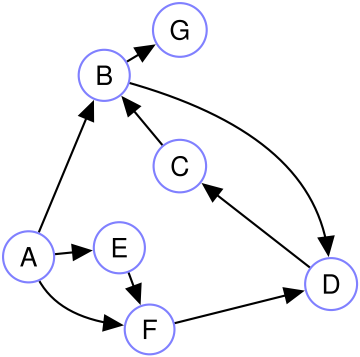
Who is an ancestor of whom?

v is an ancestor of u

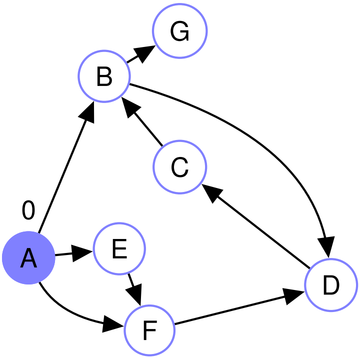
No cycles: no back edge
Back edge – edge to ancestor in “tree” of explore calls.”

Topological order - edge (u, v) means u before v
Inverse post ordering is topological order.
Bonus Topological Order Algorithm:
remove source, repeat.

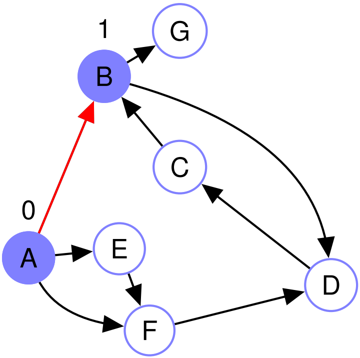
Depth first search: directed.



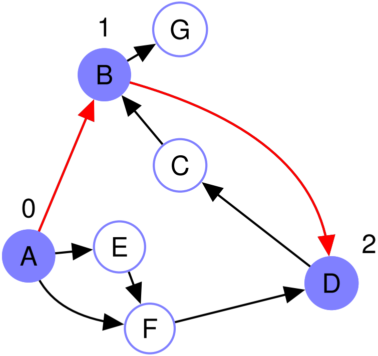
Depth first search: directed.



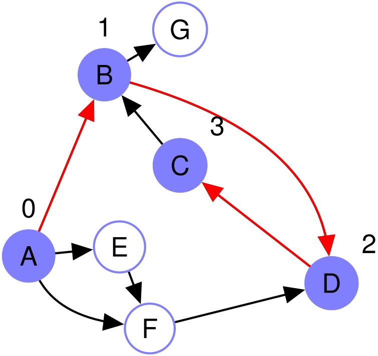
Depth first search: directed.



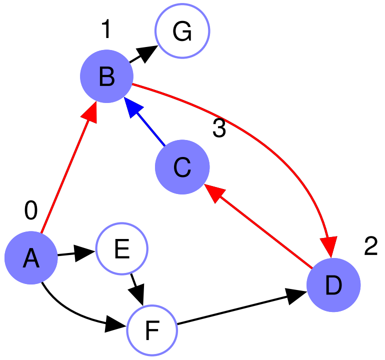
Depth first search: directed.



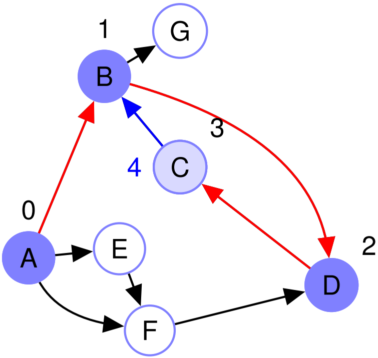
Depth first search: directed.



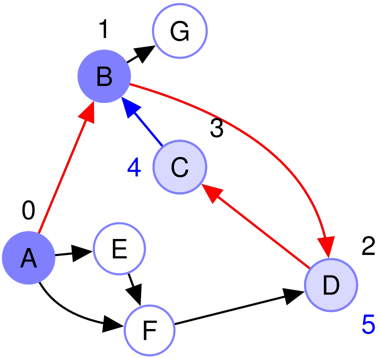
Depth first search: directed.



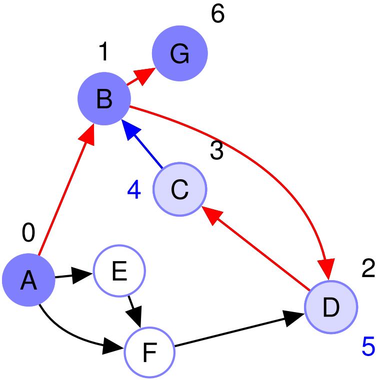
Depth first search: directed.



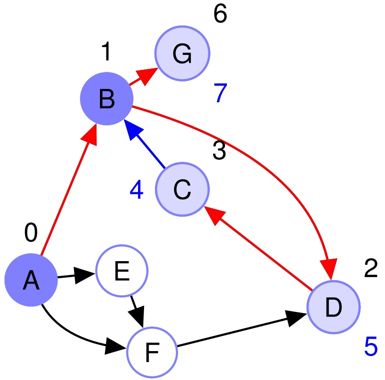
Depth first search: directed.



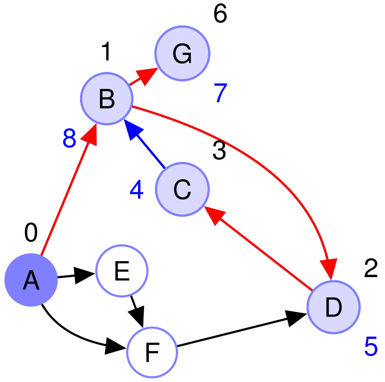
Depth first search: directed.



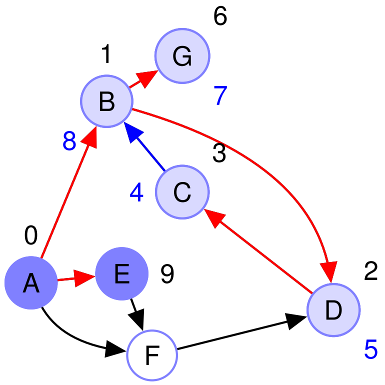
Depth first search: directed.



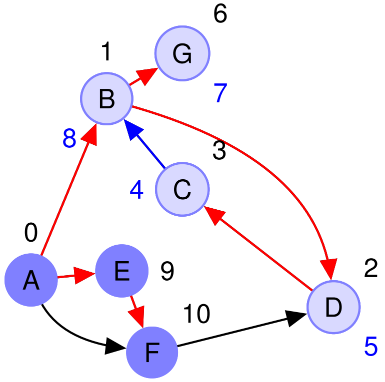
Depth first search: directed.



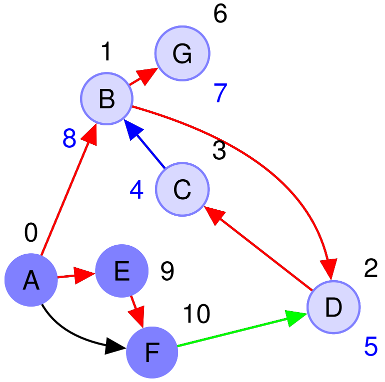
Depth first search: directed.



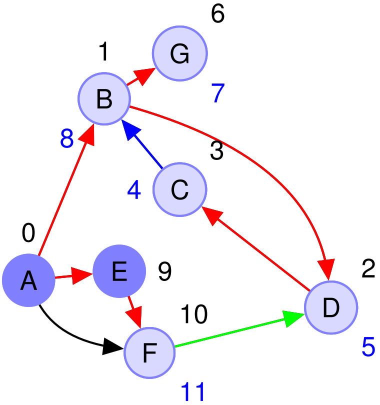
Depth first search: directed.



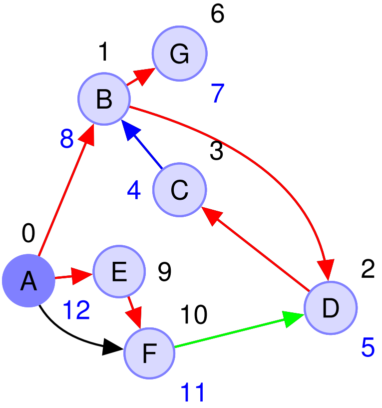
Depth first search: directed.



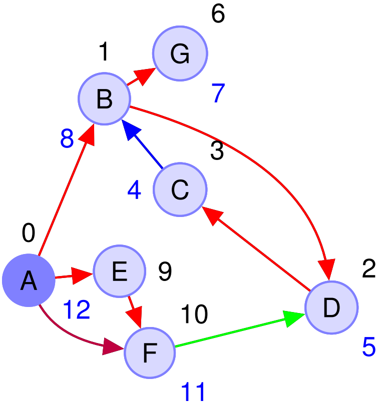
Depth first search: directed.



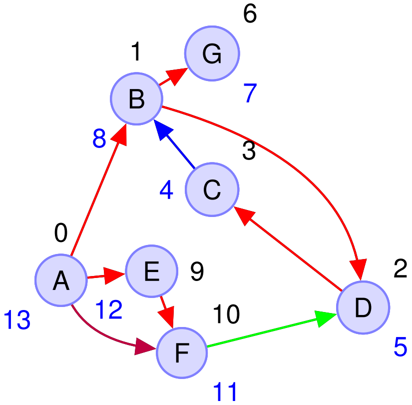
Depth first search: directed.



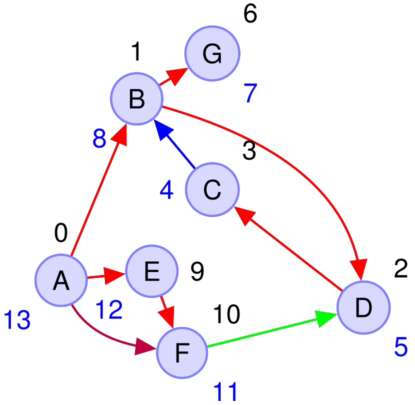
Depth first search: directed.



Depth first search: directed.

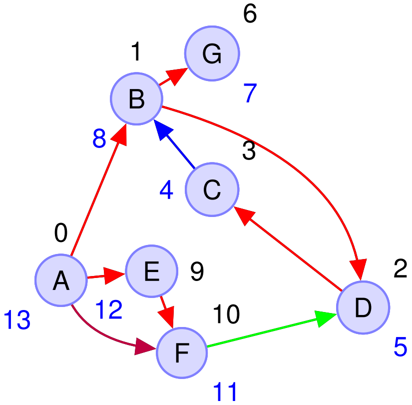


Depth first search: directed.



Tree/forward edge (u, v): $\text{int}(v)$ = [pre(v),post(v)] in $\text{int}(u)$ = [pre(u),post(u)].

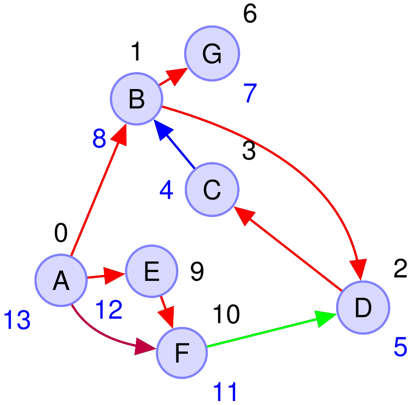
Depth first search: directed.



Tree/forward edge (u, v): int(v) = [pre(v),post(v)] in int(u) = [pre(u),post(u)].

Forward (A, F): [10,11] in [0,13] or [0,[10,11],13]

Depth first search: directed.

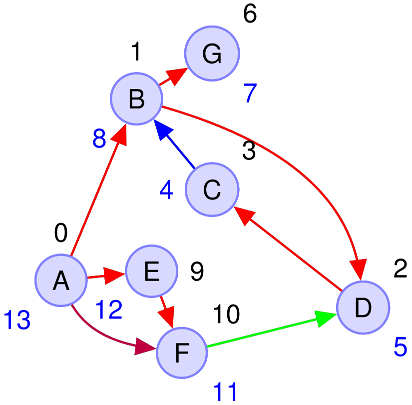


Tree/forward edge (u, v): int(v) = [pre(v),post(v)] in int(u) = [pre(u),post(u)].

Forward (A, F): [10,11] in [0,13] or [0,[10,11],13]

Back edge (u, v): int(v) contains int(u).

Depth first search: directed.



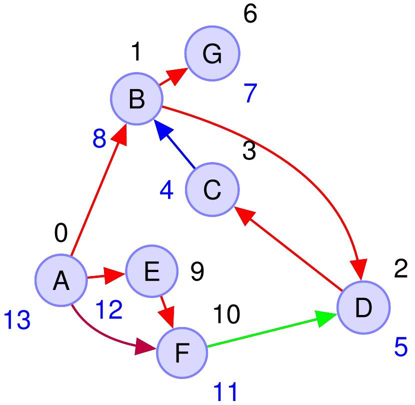
Tree/forward edge (u, v): int(v) = [pre(v),post(v)] in int(u) = [pre(u),post(u)].

Forward (A, F): [10,11] in [0,13] or [0,[10,11],13]

Back edge (u, v): int(v) contains int(u).

(C, B): [3,4] in [1,8] or [1, [3, 4], 8]

Depth first search: directed.



Tree/forward edge (u, v): int(v) = [pre(v),post(v)] in int(u) = [pre(u),post(u)].

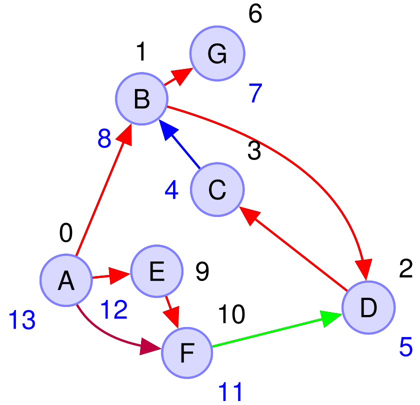
Forward (A, F): [10,11] in [0,13] or [0,[10,11],13]

Back edge (u, v): int(v) contains int(u).

(C, B): [3,4] in [1,8] or [1, [3, 4], 8]

Cross edge (u, v): int(v) before int(u).

Depth first search: directed.



Tree/forward edge (u, v): int(v) = [pre(v),post(v)] in int(u) = [pre(u),post(u)].

Forward (A, F): [10,11] in [0,13] or [0,[10,11],13]

Back edge (u, v): int(v) contains int(u).

(C, B): [3,4] in [1,8] or [1, [3, 4], 8]

Cross edge (u, v): int(v) before int(u).

(F, D): [2,5] before [10,11]

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea:

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.
Edge from "last" vertex, v_k, to v_0 is back edge.

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.
Edge from "last" vertex, v_k, to v_0 is back edge.

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.
Edge from "last" vertex, v_k, to v_0 is back edge.

Topological Ordering: π, where for all edges (u, v), $\pi(u) < \pi(v)$.

Testing for cycle/Topological sort.

Back Edge: (u,v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.
Edge from "last" vertex, v_k, to v_0 is back edge.

Topological Ordering: π, where for all edges (u,v), $\pi(u) < \pi(v)$.

Thm: Reverse order of post number is a topological ordering.

Testing for cycle/Topological sort.

Back Edge: (u, v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.
Edge from “last” vertex, v_k, to v_0 is back edge.

Topological Ordering: π, where for all edges (u, v), $\pi(u) < \pi(v)$.

Thm: Reverse order of post number is a topological ordering.

Proof: No back edges!

Testing for cycle/Topological sort.

Back Edge: (u,v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.
Edge from "last" vertex, v_k, to v_0 is back edge.

Topological Ordering: π, where for all edges (u,v), $\pi(u) < \pi(v)$.

Thm: Reverse order of post number is a topological ordering.

Proof: No back edges!
Tree/Forward edge: (u,v) : $[pre(u), post(u)] \in [pre(v), post(v)]$
$\implies post(u) > post(v)$.
Cross edge: (u,v) : $[pre(u), post(u)] > [pre(v), post(v)]$
$\implies post(u) > post(v)$.

Testing for cycle/Topological sort.

Back Edge: (u,v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.
Edge from "last" vertex, v_k, to v_0 is back edge.

Topological Ordering: π, where for all edges (u,v), $\pi(u) < \pi(v)$.

Thm: Reverse order of post number is a topological ordering.

Proof: No back edges!
Tree/Forward edge: (u,v) : $[pre(u), post(u)] \in [pre(v), post(v)]$
$\implies post(u) > post(v)$.
Cross edge: (u,v) : $[pre(u), post(u)] > [pre(v), post(v)]$
$\implies post(u) > post(v)$.

$\implies$ for every edge, (u,v), $post(u) > post(v)$

Testing for cycle/Topological sort.

Back Edge: (u,v) where $int(v)$ contains $int(u)$.

Thm: A graph has a cycle if and only if there is back edge.

Proof Idea: Back edge $\implies$ cycle!
Edge to ancestor, plus path from ancestor is cycle.

Interval of first explored vertex, v_0, contains all others.
Edge from "last" vertex, v_k, to v_0 is back edge.

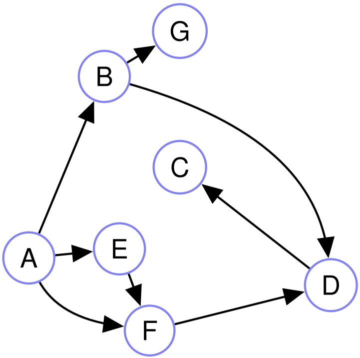
Topological Ordering: π, where for all edges (u,v), $\pi(u) < \pi(v)$.

Thm: Reverse order of post number is a topological ordering.

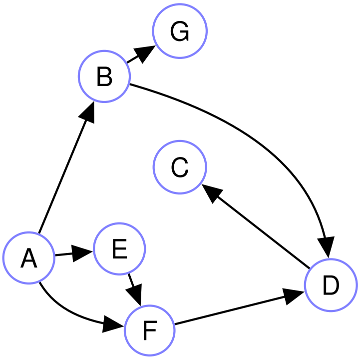
Proof: No back edges!
Tree/Forward edge: (u,v) : $[pre(u), post(u)] \in [pre(v), post(v)]$
$\implies post(u) > post(v)$.
Cross edge: (u,v) : $[pre(u), post(u)] > [pre(v), post(v)]$
$\implies post(u) > post(v)$.

$\implies$ for every edge, (u,v), $post(u) > post(v)$ $\square$

Topological Sort Example.

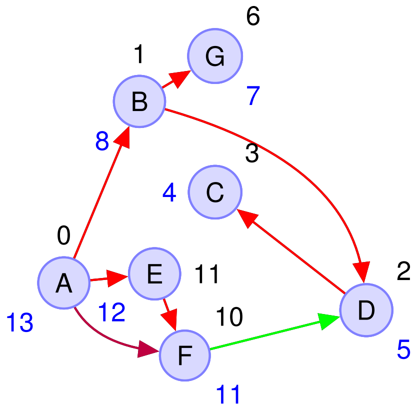


Topological Sort Example.



A linear order:

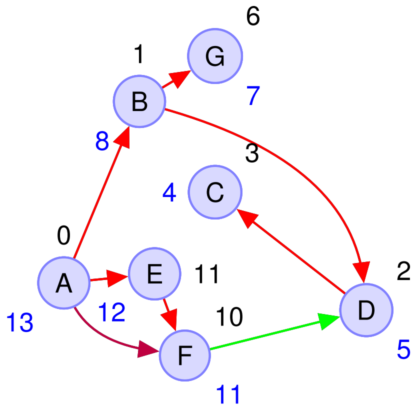
Topological Sort Example.



A linear order:

A, E, F, B, G, D, C

Topological Sort Example.

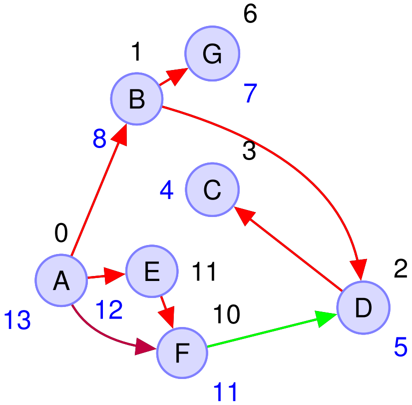


A linear order:

A, E, F, B, G, D, C

In DFS: When is A popped off stack?

Topological Sort Example.



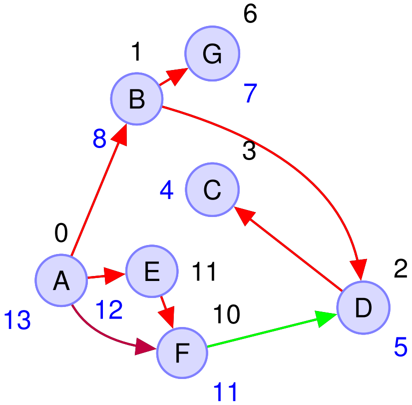
A linear order:

A, E, F, B, G, D, C

In DFS: When is A popped off stack?

plus Induction.

Topological Sort Example.



A linear order:

$$A, E, F, B, G, D, C$$

In DFS: When is A popped off stack?

plus Induction. $\implies$ Reverse order of post numbering is topological ordering.

Topological Sort: DFS

Last post order should..

(A) be first in linearization!

(B) be last in linearization!

Topological Sort: DFS

Last post order should..

(A) be first in linearization!

(B) be last in linearization!

(A). First!

Connectivity in Directed Graphs.

Two nodes are connected...

Connectivity in Directed Graphs.

Two nodes are connected...when?

Connectivity in Directed Graphs.

Two nodes are connected...when?

When there is a path from u to v?

Connectivity in Directed Graphs.

Two nodes are connected...when?

When there is a path from u to v?
When there is a path from v to u?

Connectivity in Directed Graphs.

Two nodes are connected...when?

When there is a path from u to v?
When there is a path from v to u?

Both!

Connectivity in Directed Graphs.

Two nodes are connected...when?

When there is a path from u to v?
When there is a path from v to u?

Both!

Nodes u and v are **strongly connected**

Connectivity in Directed Graphs.

Two nodes are connected...when?

When there is a path from u to v?
When there is a path from v to u?

Both!

Nodes u and v are **strongly connected**
 if there is a path from u to v

Connectivity in Directed Graphs.

Two nodes are connected...when?

When there is a path from u to v?
When there is a path from v to u?

Both!

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Connectivity in Directed Graphs.

Two nodes are connected...when?

When there is a path from u to v?
When there is a path from v to u?

Both!

Nodes u and v are **strongly connected**
  if there is a path from u to v
  and a path from v to u.

Note: Nodes are strongly connected to themselves.

Connectivity in Directed Graphs.

Two nodes are connected...when?

When there is a path from u to v?
When there is a path from v to u?

Both!

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Note: Nodes are strongly connected to themselves.

Path with zero edges in both directions!

Properties..

Nodes *u* and *v* are **strongly connected**
if there is a path from *u* to *v*
and a path from *v* to *u*.

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

True/False?

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

True/False?

If u is strongly connected to v and v is strongly connected to w
$\implies$ u connected to w.

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

True/False?

If u is strongly connected to v and v is strongly connected to w
$\implies$ u connected to w.

True!

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

True/False?

If u is strongly connected to v and v is strongly connected to w
$\implies$ u connected to w.

True!
path from u (through v) to w and path from w (through v) to u!

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

True/False?

If u is strongly connected to v and v is strongly connected to w
$\implies$ u connected to w.

True!
path from u (through v) to w and path from w (through v) to u!

Transitive: u strongly connected to v strongly connected to w

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

True/False?

If u is strongly connected to v and v is strongly connected to w
$\implies$ u connected to w.

True!
path from u (through v) to w and path from w (through v) to u!

Transitive: u strongly connected to v strongly connected to w
$\implies$ u connected to w.

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

True/False?

If u is strongly connected to v and v is strongly connected to w
$\implies$ u connected to w.

True!
path from u (through v) to w and path from w (through v) to u!

Transitive: u strongly connected to v strongly connected to w
$\implies$ u connected to w.

Relation $\implies$ a partition into equivalence classes.

Properties..

Nodes u and v are **strongly connected**
if there is a path from u to v
and a path from v to u.

Remember: Nodes are strongly connected to themselves.

True/False?

If u is strongly connected to v and v is strongly connected to w
$\implies$ u connected to w.

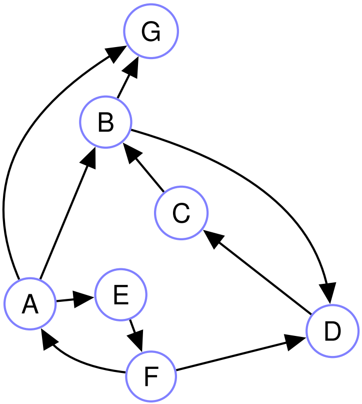
True!
path from u (through v) to w and path from w (through v) to u!

Transitive: u strongly connected to v strongly connected to w
$\implies$ u connected to w.

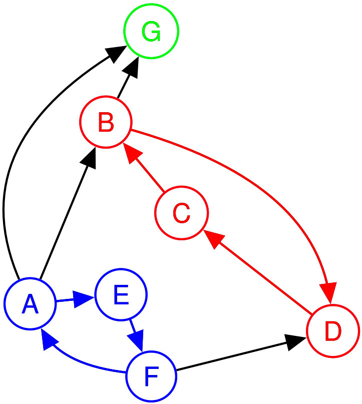
Relation $\implies$ a partition into equivalence classes.

Strongly connected components: sets of nodes which are strongly connected.

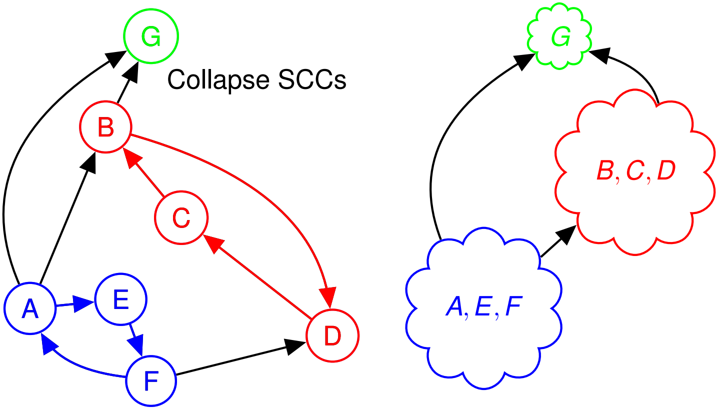
Example.



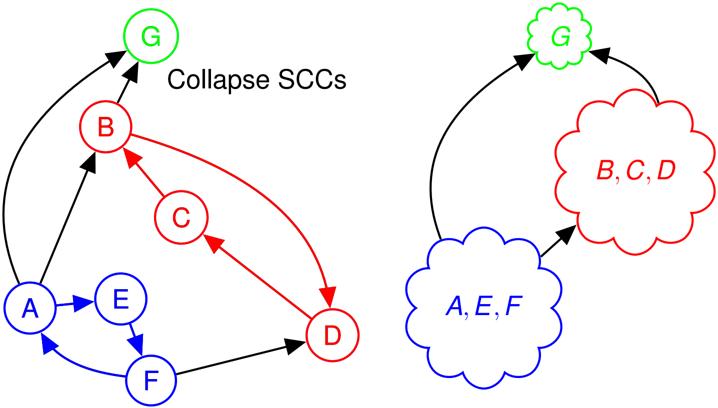
Example.



Example.

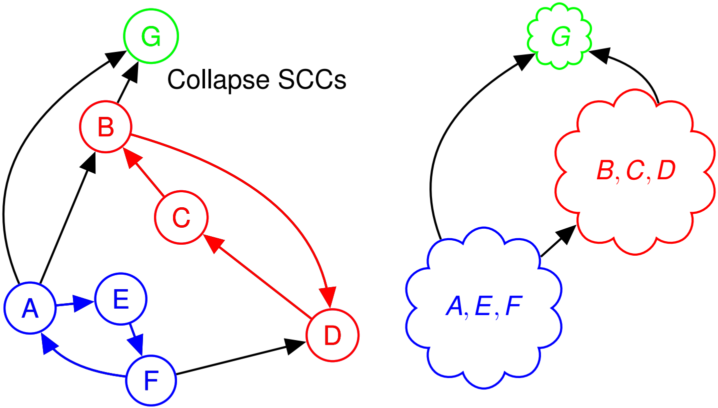


Example.



Collapsing strongly connected components (SCCs)..
..yields a DAG!

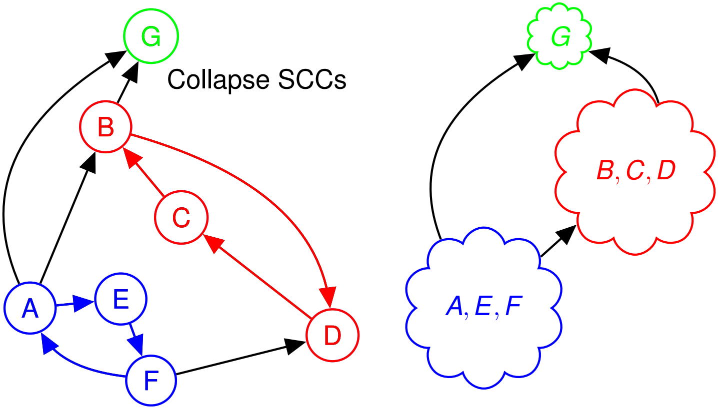
Example.



Collapsing strongly connected components (SCCs)..
..yields a DAG!

Why?

Example.

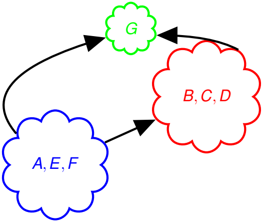


Collapsing strongly connected components (SCCs)..
..yields a DAG!

Why?
..any cycle collapses nodes into a single SCC.

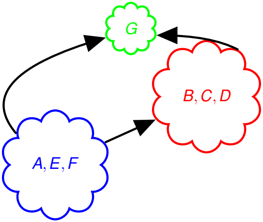
Dag of SCCs

Property: Every directed graph is a DAG of strongly connected components.



Dag of SCCs

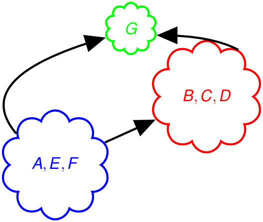
Property: Every directed graph is a DAG of strongly connected components.



Finding the strongly connected components?

Dag of SCCs

Property: Every directed graph is a DAG of strongly connected components.

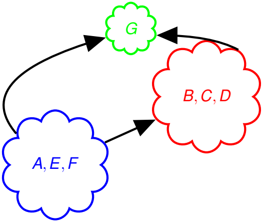


Finding the strongly connected components?

Property: **explore**(u) visits all nodes reachable from u.

Dag of SCCs

Property: Every directed graph is a DAG of strongly connected components.



Finding the strongly connected components?

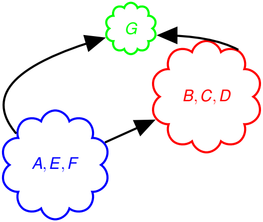
Property: explore(u) visits all nodes reachable from u.

Algorithm:

1. Run explore on node in sink component.

Dag of SCCs

Property: Every directed graph is a DAG of strongly connected components.



Finding the strongly connected components?

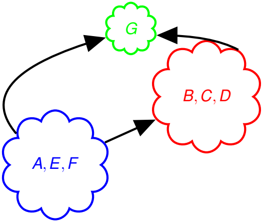
Property: explore(u) visits all nodes reachable from u.

Algorithm:

1. Run explore on node in sink component.
   get all nodes in sink.

Dag of SCCs

Property: Every directed graph is a DAG of strongly connected components.



Finding the strongly connected components?

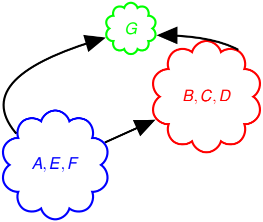
Property: **explore**(u) visits all nodes reachable from u.

Algorithm:

1. Run explore on node in sink component.
   get all nodes in sink.
2. Output visited nodes.

Dag of SCCs

Property: Every directed graph is a DAG of strongly connected components.



Finding the strongly connected components?

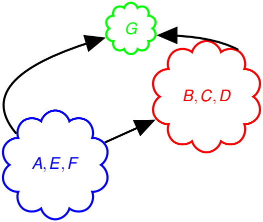
Property: explore(u) visits all nodes reachable from u.

Algorithm:

1. Run explore on node in sink component.
   get all nodes in sink.
2. Output visited nodes.
3. Repeat.

Dag of SCCs

Property: Every directed graph is a DAG of strongly connected components.



Finding the strongly connected components?

Property: **explore**(u) visits all nodes reachable from u.

Algorithm:

1. Run explore on node in sink component.
   get all nodes in sink.
2. Output visited nodes.
3. Repeat.

How do we find a node in the sink component?

Finding a source.

Property: The node with the highest post order number is in a source component.

Finding a source.

Property: The node with the highest post order number is in a source component.

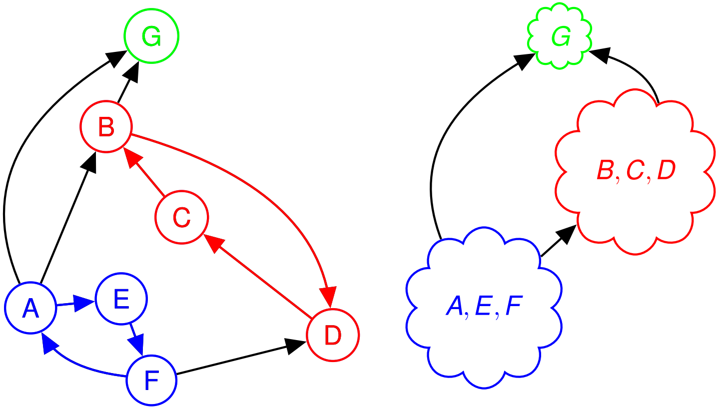
Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Post numbers of SCCs

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

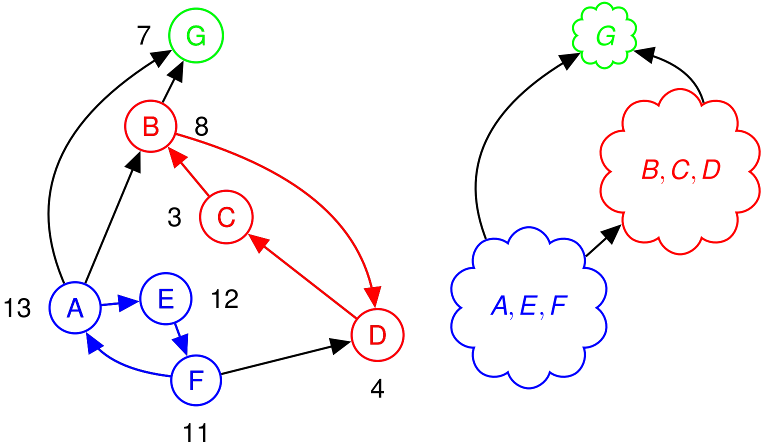
Post numbers of SCCs

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.



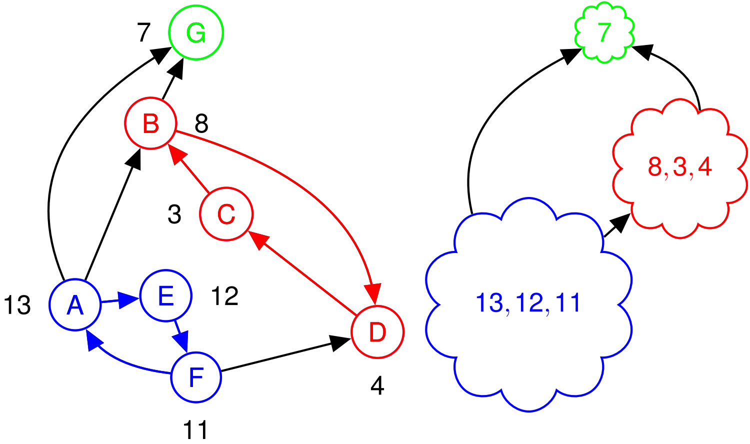
Post numbers of SCCs

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.



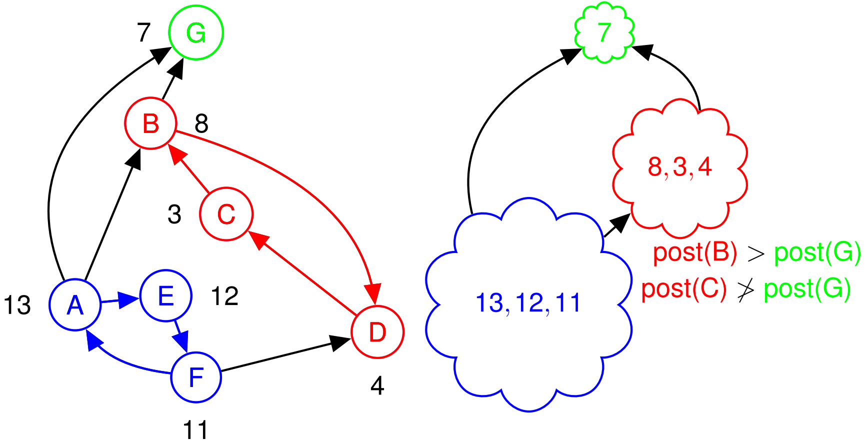
Post numbers of SCCs

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.



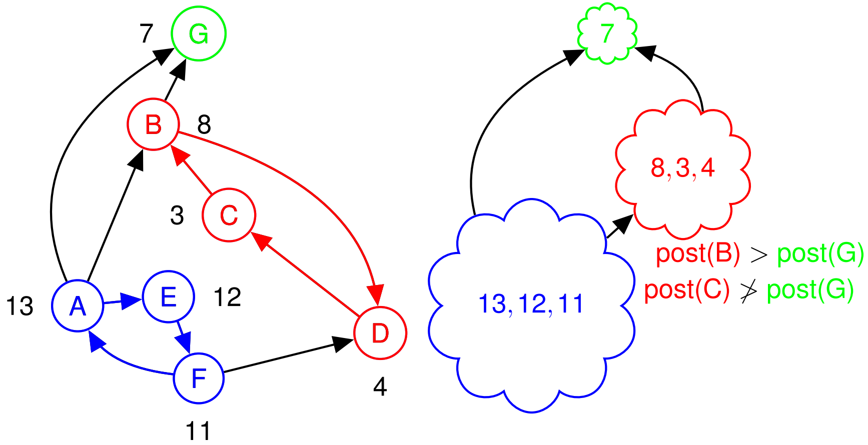
Post numbers of SCCs

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.



Post numbers of SCCs

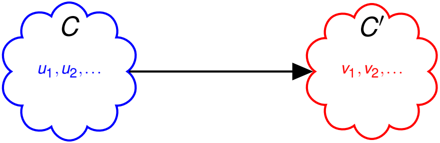
Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.



Highest post# in C bigger than any in C'

not true every post# in C greater than any in C'

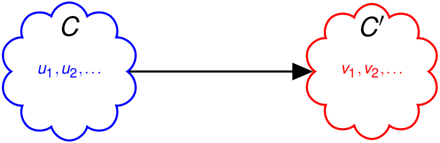
Proof:



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Proof:

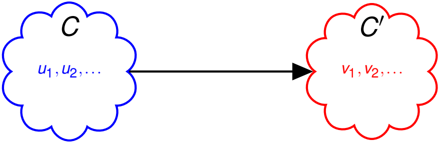
Proof:



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.

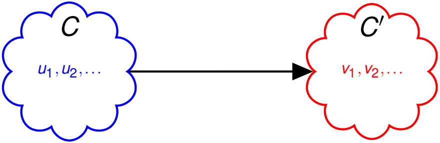
Proof:



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.
explore(v) gets to all of C'
and none of C!
So every node in C explored after every node in C'.

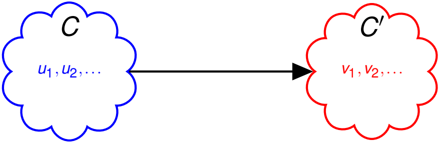
Proof:



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.
explore(v) gets to all of C'
and none of C!
So every node in C explored after every node in C'.
$\equiv$ every post # in C larger than every post # in C'.

Proof:

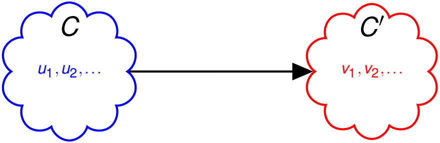


Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.
explore(v) gets to all of C'
and none of C!
So every node in C explored after every node in C'.
$\equiv$ every post # in C larger than every post # in C'.

If a node u in C is explored first

Proof:

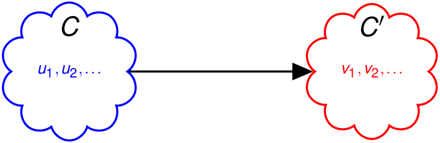


Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.
explore(v) gets to all of C'
and none of C!
So every node in C explored after every node in C'.
$\equiv$ every post # in C larger than every post # in C'.

If a node u in C is explored first
All of C and C' will be explored before returning

Proof:

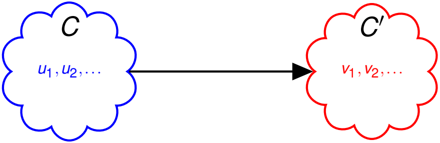


Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.
    explore(v) gets to all of C'
    and none of C!
So every node in C explored after every node in C'.
 $\equiv$ every post # in C larger than every post # in C'.

If a node u in C is explored first
    All of C and C' will be explored before returning
    from **explore**(u)

Proof:

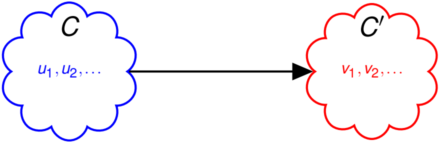


Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.
explore(v) gets to all of C'
and none of C!
So every node in C explored after every node in C'.
$\equiv$ every post # in C larger than every post # in C'.

If a node u in C is explored first
All of C and C' will be explored before returning
from **explore**(u)
So u has higher post number than any node in C'.

Proof:



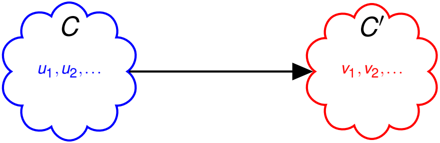
Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.
  explore(v) gets to all of C'
  and none of C!
So every node in C explored after every node in C'.
 $\equiv$ every post # in C larger than every post # in C'.

If a node u in C is explored first
  All of C and C' will be explored before returning
  from **explore**(u)
So u has higher post number than any node in C'.

$\square$

Proof:



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

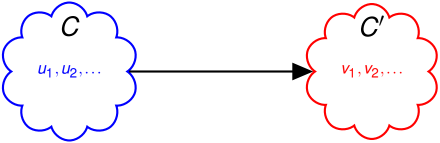
If a node v in C' is explored first.
  explore(v) gets to all of C'
  and none of C!
So every node in C explored after every node in C'.
$\equiv$ every post # in C larger than every post # in C'.

If a node u in C is explored first
  All of C and C' will be explored before returning
  from **explore**(u)
So u has higher post number than any node in C'.

$\square$

Implies highest post numbered node is in source component.

Proof:



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.
Proof:

If a node v in C' is explored first.
    explore(v) gets to all of C'
    and none of C!
So every node in C explored after every node in C'.
$\equiv$ every post # in C larger than every post # in C'.

If a node u in C is explored first
    All of C and C' will be explored before returning
    from **explore**(u)
So u has higher post number than any node in C'.

$\square$

Implies highest post numbered node is in source component.

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes!

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily.

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

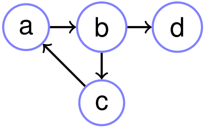
(A) Yes! (B) Not Necessarily. ... (B)

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)

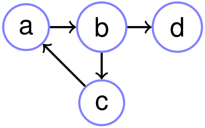


Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)



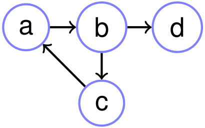
Explore(a)

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)



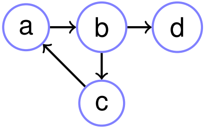
Explore(a) $\Longrightarrow$

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)



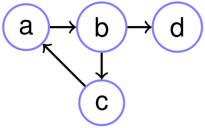
Explore(a) $\Longrightarrow$ Explore(b)

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)



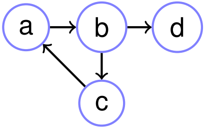
Explore(a) $\Longrightarrow$ Explore(b) $\Longrightarrow$ Explore (c)

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)



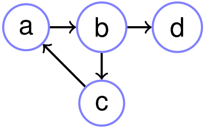
Explore(a) $\Longrightarrow$ Explore(b) $\Longrightarrow$ Explore (c)
$\Longrightarrow$ Return from (c)

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)



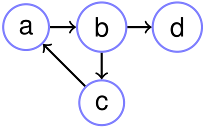
Explore(a) $\Longrightarrow$ Explore(b) $\Longrightarrow$ Explore (c)
$\Longrightarrow$ Return from (c) $\Longrightarrow$ Explore(d)

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)



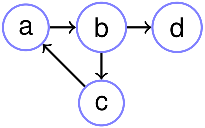
Explore(a) $\Longrightarrow$ Explore(b) $\Longrightarrow$ Explore (c)
$\Longrightarrow$ Return from (c) $\Longrightarrow$ Explore(d) $\Longrightarrow$ Return from (d) ...

Test your understanding..

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Does every node in C have a higher post order number than every node in C'?

(A) Yes! (B) Not Necessarily. ... (B)



Explore(a) $\Longrightarrow$ Explore(b) $\Longrightarrow$ Explore (c)
$\Longrightarrow$ Return from (c) $\Longrightarrow$ Explore(d) $\Longrightarrow$ Return from (d) ...

(c) has lower post order number than (d).

Back to SCC Algorithm..

Property: The highest post numbered node is in source component.

Back to SCC Algorithm..

Property: The highest post numbered node is in source component.

Algorithm:

1. Run explore on node in sink component.
2. Output visited nodes.
3. Repeat.

Back to SCC Algorithm..

Property: The highest post numbered node is in source component.

Algorithm:

1. Run explore on node in sink component.
2. Output visited nodes.
3. Repeat.

Uh...oh.

Back to SCC Algorithm..

Property: The highest post numbered node is in source component.

Algorithm:

1. Run explore on node in sink component.
2. Output visited nodes.
3. Repeat.

Uh...oh.

How should we fix this?

SCC Algorithm.

Property: The highest post numbered node is in source component.

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges!

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

1. DFS on G^R to compute post($\cdot$)

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

1. DFS on G^R to compute post$(\cdot)$
   Highest post # vertex, v, in G^R in sink comp. of G.

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

1. DFS on G^R to compute post$(\cdot)$
   Highest post # vertex, v, in G^R in sink comp. of G.
2. Output nodes visited in: explore(v)

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

1. DFS on G^R to compute post$(\cdot)$
   Highest post # vertex, v, in G^R in sink comp. of G.
2. Output nodes visited in: explore(v)
   Then what?

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

1. DFS on G^R to compute post($\cdot$)
   Highest post # vertex,v, in G^R in sink comp. of G.
2. Output nodes visited in: explore(v)
   Then what?

Find another node in sink of unvisited part of G!

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

1. DFS on G^R to compute post$(\cdot)$
   Highest post # vertex,v, in G^R in sink comp. of G.
2. Output nodes visited in: explore(v)

Then what?

Find another node in sink of unvisited part of G!

Recompute DFS in G^R...

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

1. DFS on G^R to compute post$(\cdot)$
   Highest post # vertex, v, in G^R in sink comp. of G.
2. Output nodes visited in: explore(v)
   Then what?

Find another node in sink of unvisited part of G!

Recompute DFS in G^R...or...

SCC Algorithm.

Property: The highest post numbered node is in source component.

Find node in sink component?

Reverse edges! G^R

Source component in G^R is sink component in G.

Algorithm:

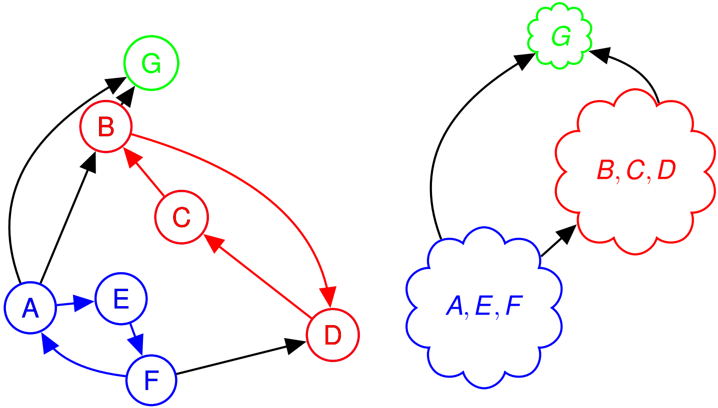
1. DFS on G^R to compute post($\cdot$)
   Highest post # vertex, v, in G^R in sink comp. of G.
2. Output nodes visited in: explore(v)
   Then what?

Find another node in sink of unvisited part of G!

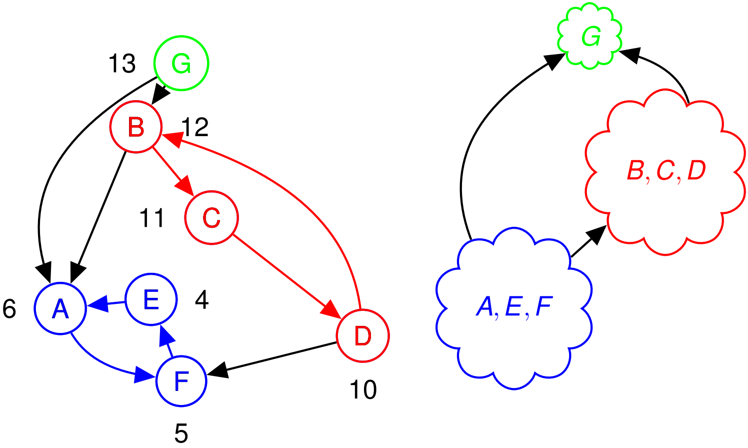
Recompute DFS in G^R...or...

.... use post($\cdot$) again!

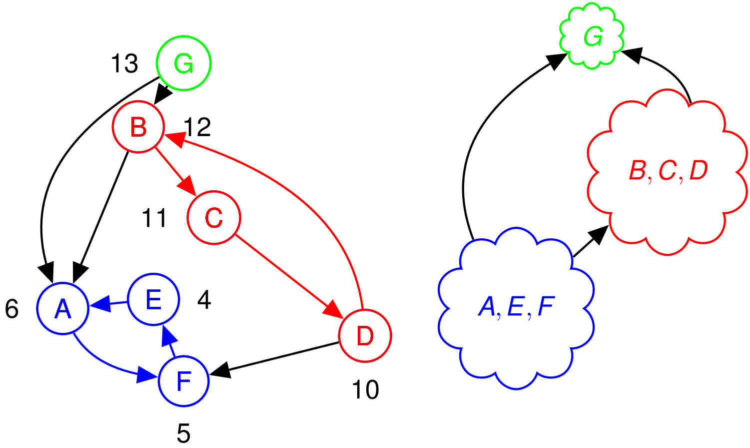
Post from Reverse graph



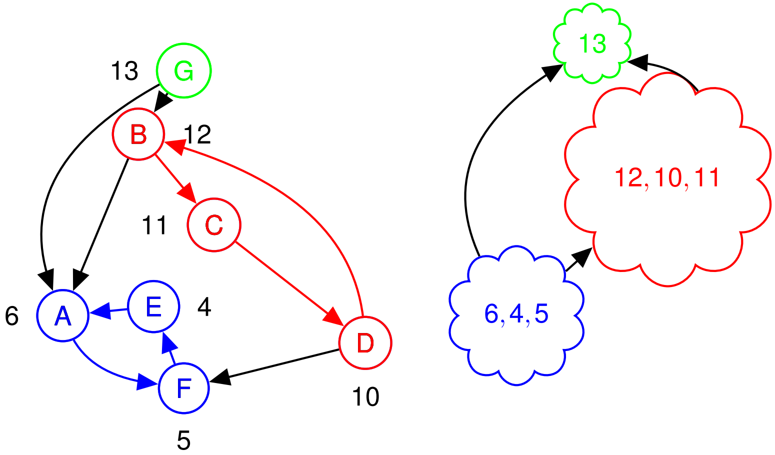
Post from Reverse graph



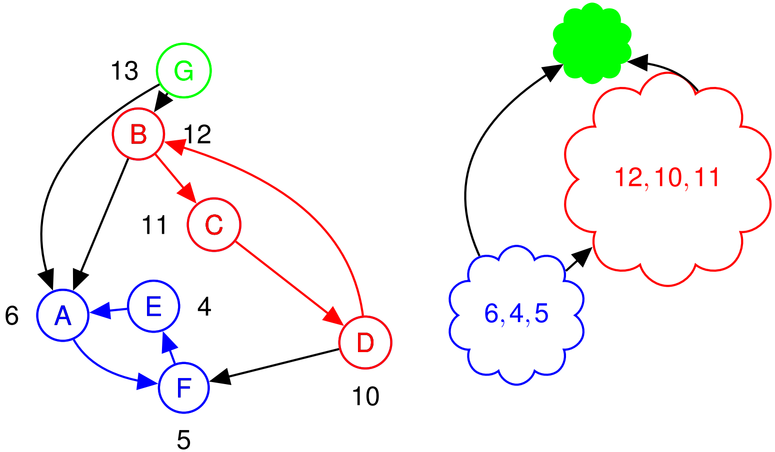
Post from Reverse graph



Post from Reverse graph

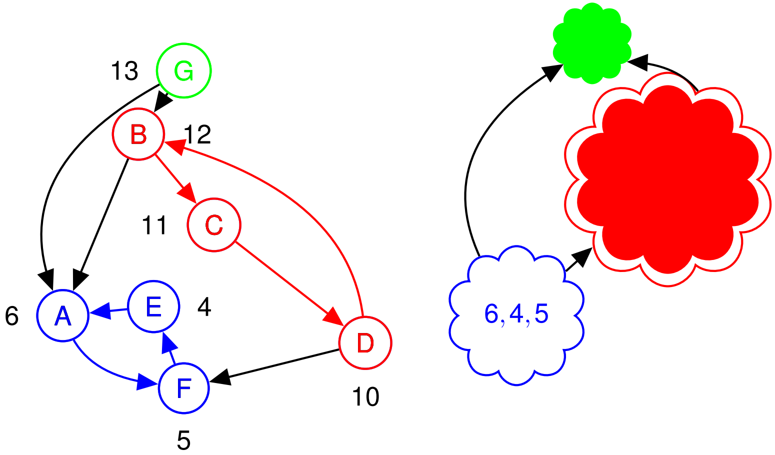


Post from Reverse graph



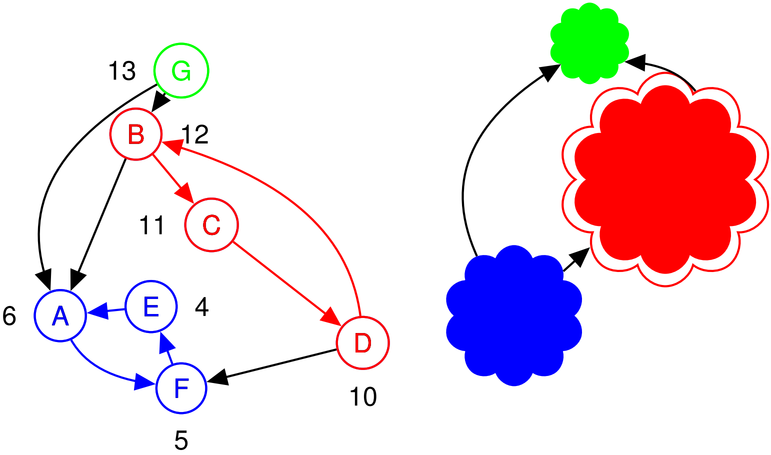
Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Post from Reverse graph



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Post from Reverse graph



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\implies$ in source component of G^R

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\implies$ in source component of G^R
$\implies$ v in sink component of G!

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\implies$ in source component of G^R
$\implies$ v in sink component of G!

SCC Algorithm:

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\implies$ in source component of G^R
$\implies$ v in sink component of G!

SCC Algorithm:
1. DFS of G^R.

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\implies$ in source component of G^R
$\implies$ v in sink component of G!

SCC Algorithm:
1. DFS of G^R.
2. Run undirected components algorithm on G

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\implies$ in source component of G^R
$\implies$ v in sink component of G!

SCC Algorithm:
1. DFS of G^R.
2. Run undirected components algorithm on G
— in reverse post order number from step 1.

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\implies$ in source component of G^R
$\implies$ v in sink component of G!

SCC Algorithm:
1. DFS of G^R.
2. Run undirected components algorithm on G
— in reverse post order number from step 1.

$O(|V| + |E|)$ time ...

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

First explore of G:
Removes sink component of G.
$\Longrightarrow$ removes source component of G^R.
$\Longrightarrow$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\Longrightarrow$ in source component of G^R
$\Longrightarrow$ v in sink component of G!

SCC Algorithm:
1. DFS of G^R.
2. Run undirected components algorithm on G
— in reverse post order number from step 1.

$O(|V| + |E|)$ time ...

Compute G^R in linear time?..

All sinks from one dfs.

Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

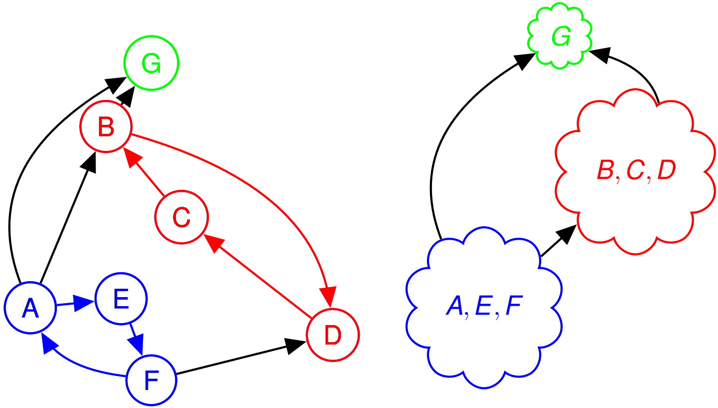
First explore of G:
Removes sink component of G.
$\implies$ removes source component of G^R.
$\implies$ highest rem. post # vertex, v,
in G^R in component with no in-edges
$\implies$ in source component of G^R
$\implies$ v in sink component of G!

SCC Algorithm:
1. DFS of G^R.
2. Run undirected components algorithm on G
— in reverse post order number from step 1.

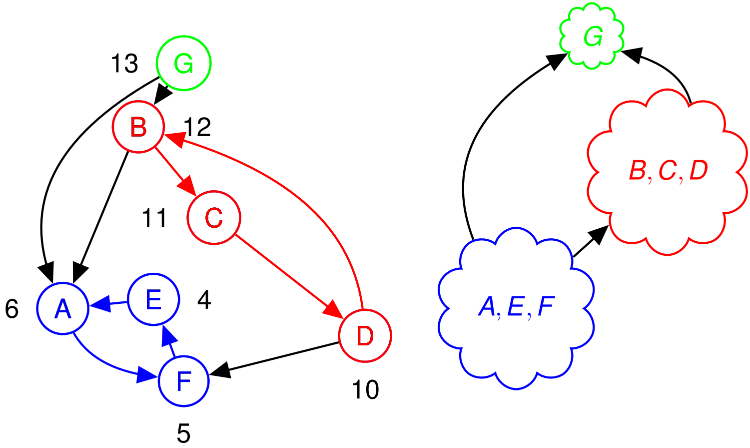
$O(|V| + |E|)$ time ...

Compute G^R in linear time?.. exercise.

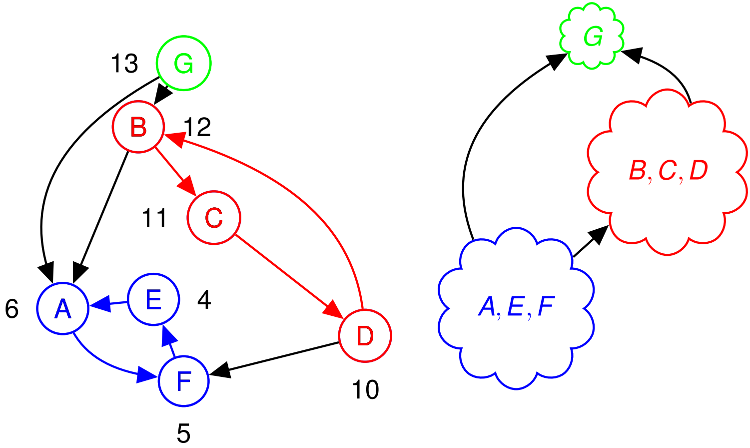
Example Again: think runtime.



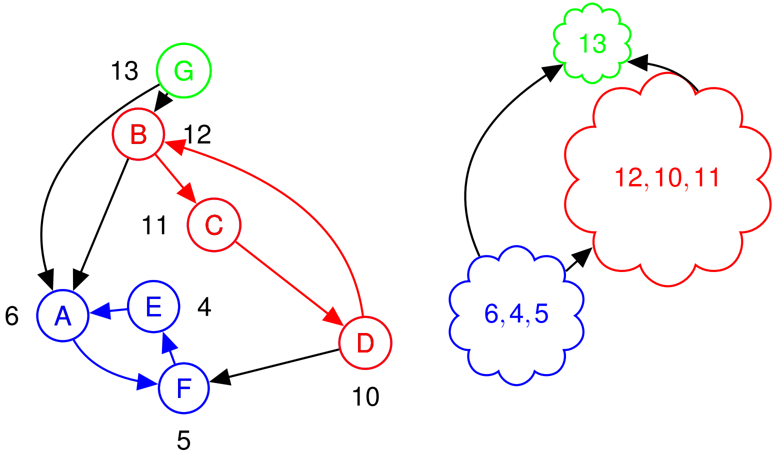
Example Again: think runtime.



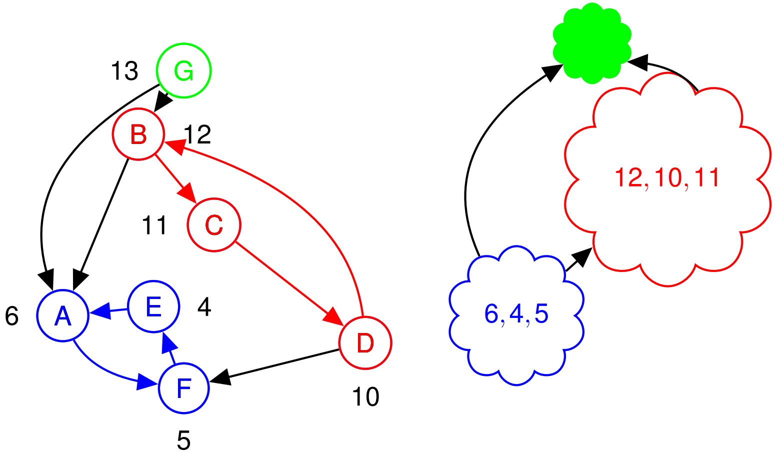
Example Again: think runtime.



Example Again: think runtime.

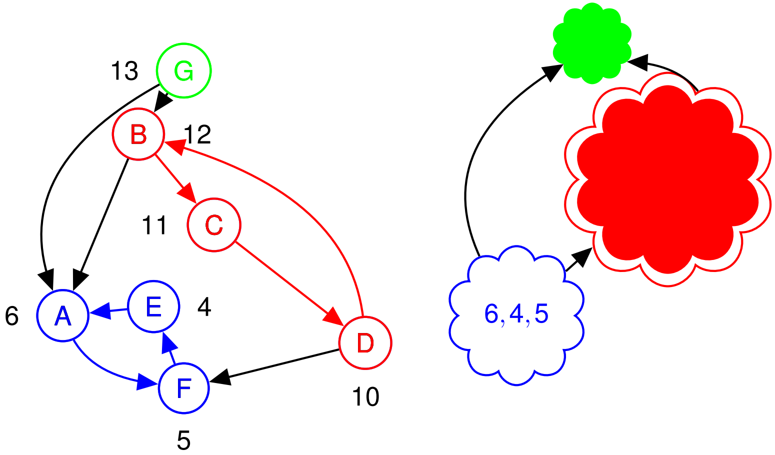


Example Again: think runtime.



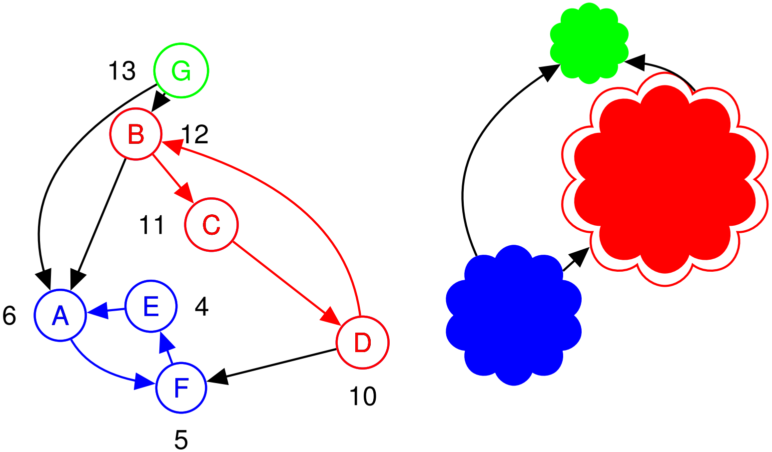
Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Example Again: think runtime.



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Example Again: think runtime.



Property++: If C and C' are SCCs with an edge from C to C', highest post# of a node in C larger than post# of any node in C'.

Lecture in a minute!

Quick Review:
DFS so far $\equiv$

Lecture in a minute!

Quick Review:

DFS so far $\equiv$ how I learned to love the stack.

pre/post = time on stack.

Topological Ordering:

Inverse post ordering $\equiv$ topological ordering.

Remove source, repeat.

Lecture in a minute!

Quick Review:
DFS so far $\equiv$ how I learned to love the stack.
pre/post = time on stack.
Topological Ordering:
Inverse post ordering $\equiv$ topological ordering.
Remove source, repeat.

Strongly Connected Components: directed graphs.
Strong Connectivity for u and v.
On a cycle together.
Easy: $O(|V||E|)$ algorithm.
Linear time algorithm!

Lecture in a minute!

Quick Review:
- DFS so far $\equiv$ how I learned to love the stack.
  - pre/post = time on stack.
- Topological Ordering:
  - Inverse post ordering $\equiv$ topological ordering.
  - Remove source, repeat.

Strongly Connected Components: directed graphs.
- Strong Connectivity for u and v.
  - On a cycle together.
- Easy: $O(|V||E|)$ algorithm.
- Linear time algorithm!
  - Observation: Highest post in "source component".
  - Find vertex in sink component.
    - Explore.
  - Repeat.