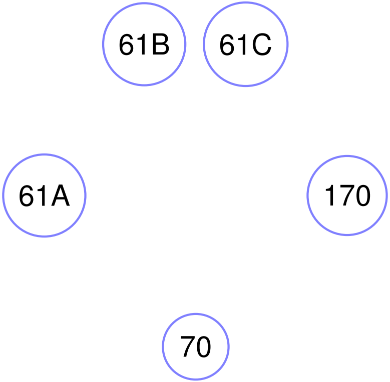
CS170 - Lecture 6

Sanjam Garg
UC Berkeley

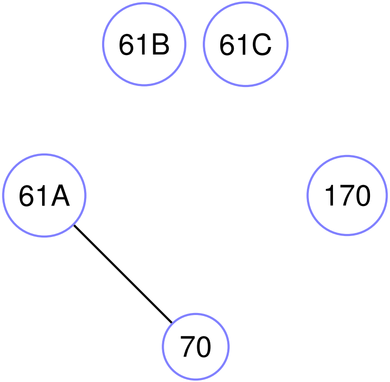
Today

1. Graphs
2. Depth First Search
3. Reachability

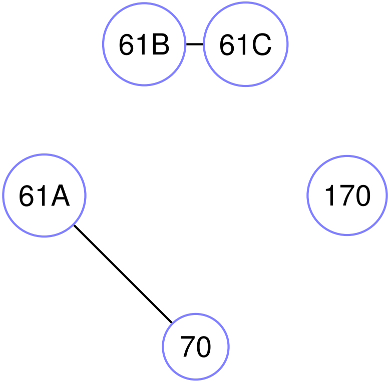
Scheduling: coloring.



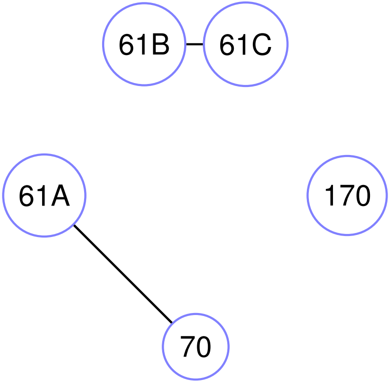
Scheduling: coloring.



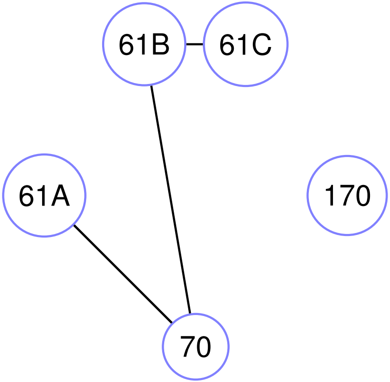
Scheduling: coloring.



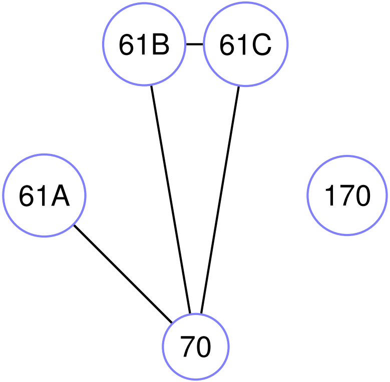
Scheduling: coloring.



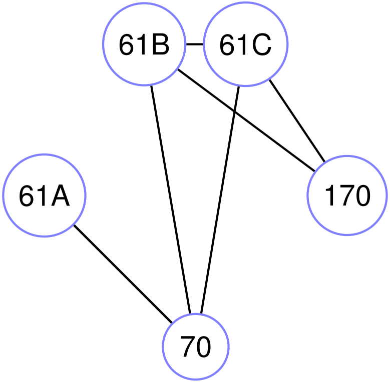
Scheduling: coloring.



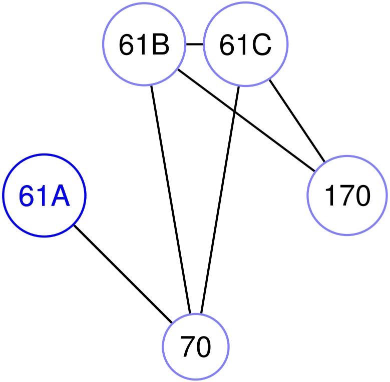
Scheduling: coloring.



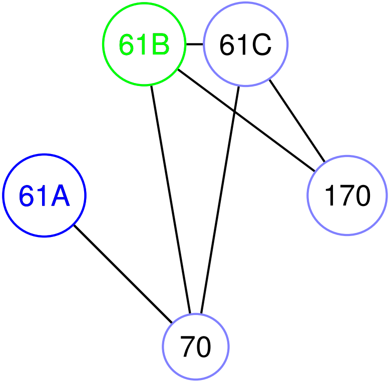
Scheduling: coloring.



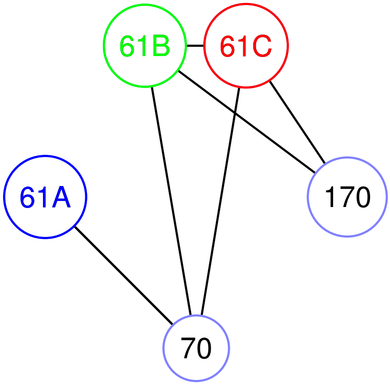
Scheduling: coloring.



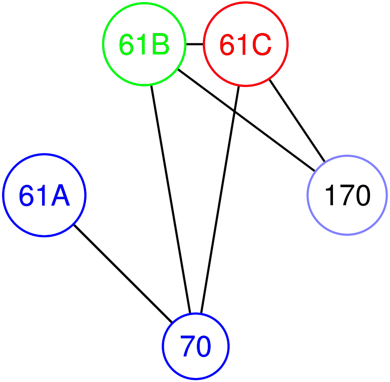
Scheduling: coloring.



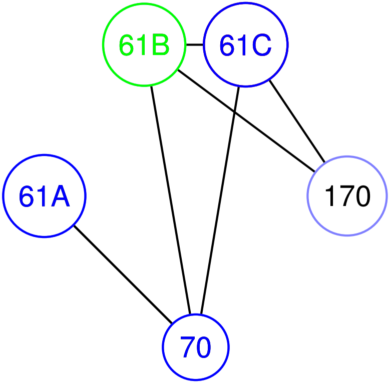
Scheduling: coloring.



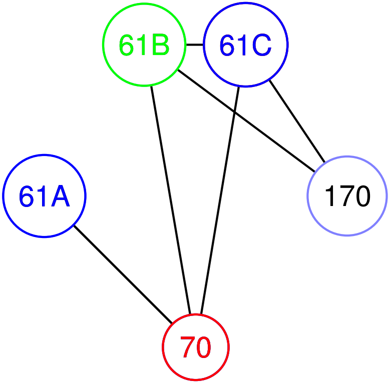
Scheduling: coloring.



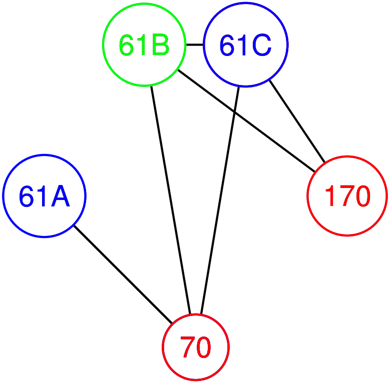
Scheduling: coloring.



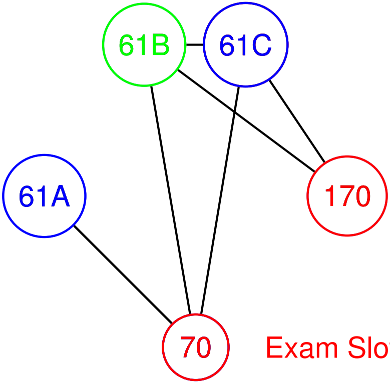
Scheduling: coloring.



Scheduling: coloring.



Scheduling: coloring.



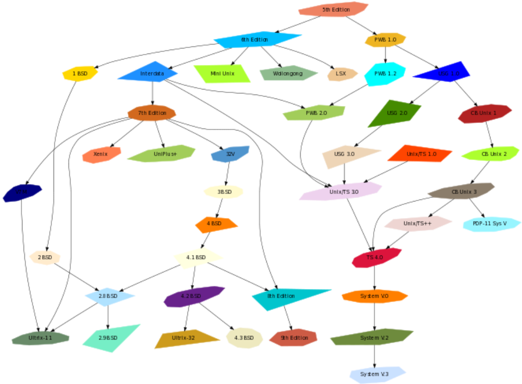
Exam Slot 1.

Exam Slot 2.

Exam Slot 3.

Directed acyclic graphs.

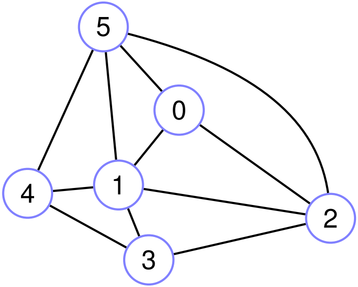
Heritage of Unix.



Object Oriented Graphs
Stephen North, 3/19/93

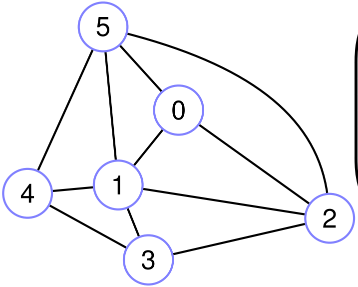
From http://www.graphviz.org/content/crazy.

Graph $G = (V, E)$.



$$V = \{0, 1, 2, 3, 4, 5\}$$
$$E = \{(0,1), (0,2), (0,5), (1,3) \ldots\}$$

Graph $G = (V, E)$.



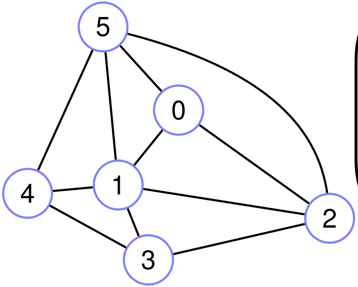
Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

$$E = \{(0,1), (0,2), (0,5), (1,3) \ldots\}$$

Graph $G = (V, E)$.



Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

$$E = \{(0,1), (0,2), (0,5), (1,3) \ldots\}$$

Adjacency List

$0 :\quad 1, 2, 5$

$1 :\quad 0, 2, 3, 4, 5$

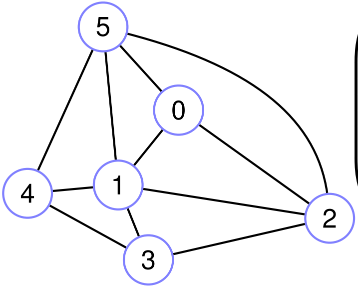
$2 :\quad 0, 1, 3, 5$

$3 :\quad 1, 2, 4$

$4 :\quad 1, 3, 5$

$5 :\quad 0, 1, 2, 4$

Graph $G = (V, E)$.



Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

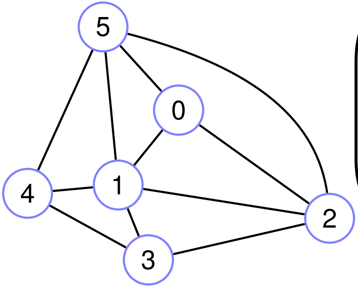
$$E = \{(0,1), (0,2), (0,5), (1,3) \ldots\}$$

Adjacency List

0 : 1, 2, 5
1 : 0, 2, 3, 4, 5
2 : 0, 1, 3, 5
3 : 1, 2, 4
4 : 1, 3, 5
5 : 0, 1, 2, 4

	Matrix	Adj. List
Edge (u, v)?		

Graph $G = (V, E)$.



Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

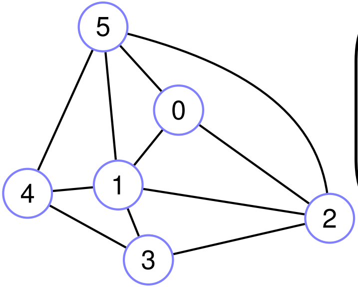
$$E = \{(0, 1), (0, 2), (0, 5), (1, 3) \ldots\}$$

Adjacency List

0 : $1, 2, 5$
1 : $0, 2, 3, 4, 5$
2 : $0, 1, 3, 5$
3 : $1, 2, 4$
4 : $1, 3, 5$
5 : $0, 1, 2, 4$

	Matrix	Adj. List
Edge (u, v)?	$O(1)$	

Graph $G = (V, E)$.



Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

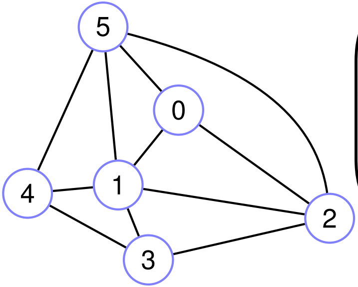
$$E = \{(0,1), (0,2), (0,5), (1,3) \ldots\}$$

Adjacency List

$0 :$ $1, 2, 5$
$1 :$ $0, 2, 3, 4, 5$
$2 :$ $0, 1, 3, 5$
$3 :$ $1, 2, 4$
$4 :$ $1, 3, 5$
$5 :$ $0, 1, 2, 4$

	Matrix	Adj. List
Edge (u, v)?	$O(1)$	$O(d)$
Neighbors of u		

Graph $G = (V, E)$.



Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

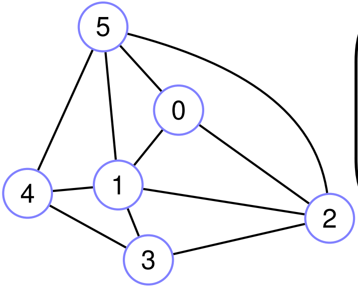
$$E = \{(0, 1), (0, 2), (0, 5), (1, 3) \ldots\}$$

Adjacency List

$0 : \quad 1, 2, 5$
$1 : \quad 0, 2, 3, 4, 5$
$2 : \quad 0, 1, 3, 5$
$3 : \quad 1, 2, 4$
$4 : \quad 1, 3, 5$
$5 : \quad 0, 1, 2, 4$

	Matrix	Adj. List
Edge (u, v)?	$O(1)$	$O(d)$
Neighbors of u	$O(\vert V \vert)$	

Graph $G = (V, E)$.



Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

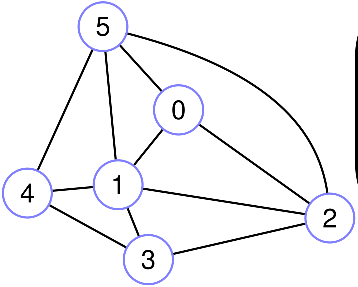
$$E = \{(0,1), (0,2), (0,5), (1,3) \ldots\}$$

Adjacency List

0 :	1, 2, 5
1 :	0, 2, 3, 4, 5
2 :	0, 1, 3, 5
3 :	1, 2, 4
4 :	1, 3, 5
5 :	0, 1, 2, 4

	Matrix	Adj. List
Edge (u, v)?	$O(1)$	$O(d)$
Neighbors of u	$O(\lvert V \rvert)$	$O(d)$
Space		

Graph $G = (V, E)$.



Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

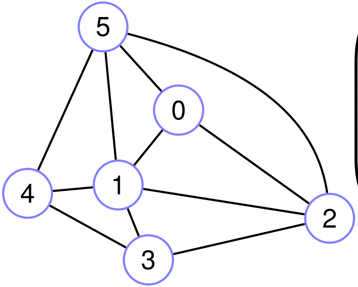
$$E = \{(0,1), (0,2), (0,5), (1,3) \ldots\}$$

Adjacency List

$0:$ $1, 2, 5$
$1:$ $0, 2, 3, 4, 5$
$2:$ $0, 1, 3, 5$
$3:$ $1, 2, 4$
$4:$ $1, 3, 5$
$5:$ $0, 1, 2, 4$

	Matrix	Adj. List
Edge (u, v)?	$O(1)$	$O(d)$
Neighbors of u	$O(\lvert V \rvert)$	$O(d)$
Space	$O(\lvert V \rvert^2)$	

Graph $G = (V, E)$.



Matrix Representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$V = \{0, 1, 2, 3, 4, 5\}$$

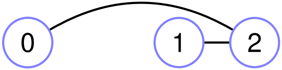
$$E = \{(0,1), (0,2), (0,5), (1,3) \ldots\}$$

Adjacency List

0 : 1, 2, 5
1 : 0, 2, 3, 4, 5
2 : 0, 1, 3, 5
3 : 1, 2, 4
4 : 1, 3, 5
5 : 0, 1, 2, 4

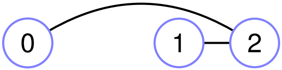
	Matrix	Adj. List
Edge (u, v)?	$O(1)$	$O(d)$
Neighbors of u	$O(\|V\|)$	$O(d)$
Space	$O(\|V\|^2)$	$O(\|E\|)$

Test your understanding..



Adjacency list of node 0?

Test your understanding..



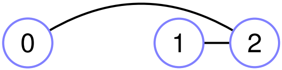
Adjacency list of node 0?

(A) $0 : 1$

(B) $0 : 1, 2$

(C) $0 : 2$

Test your understanding..



Adjacency list of node 0?

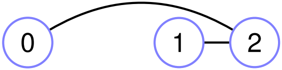
(A) $0 : 1$

(B) $0 : 1, 2$

(C) $0 : 2$

(C)

Test your understanding..



Adjacency list of node 0?

(A) $0:1$

(B) $0:1,2$

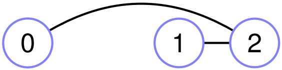
(C) $0:2$

(C)

How many edges?

(A) 2

Test your understanding..



Adjacency list of node 0?

(A) 0 : 1

(B) 0 : 1, 2

(C) 0 : 2

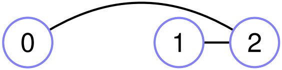
(C)

How many edges?

(A) 2

Total length of adacency lists?

Test your understanding..



Adjacency list of node 0?

(A) $0 : 1$

(B) $0 : 1, 2$

(C) $0 : 2$

(C)

How many edges?

(A) 2

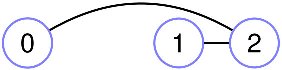
Total length of adacency lists?

(A) 2

(B) 3

(C) 4

Test your understanding..



Adjacency list of node 0?

(A) $0:1$

(B) $0:1,2$

(C) $0:2$

(C)

How many edges?

(A) 2

Total length of adacency lists?

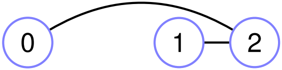
(A) 2

(B) 3

(C) 4

(C)

Test your understanding..



Adjacency list of node 0?

(A) $0 : 1$

(B) $0 : 1, 2$

(C) $0 : 2$

(C)

How many edges?

(A) 2

Total length of adacency lists?

(A) 2

(B) 3

(C) 4

(C) 2 entries for each edge!

Exploring a maze.

Theseus: Wants to find the minatour in the maze.

Exploring a maze.

Theseus: Wants to find the minatour in the maze.

Exploring a maze.

Theseus: Wants to find the minatour in the maze.

Theseus has access to a Ball of Thread and a Chalk!

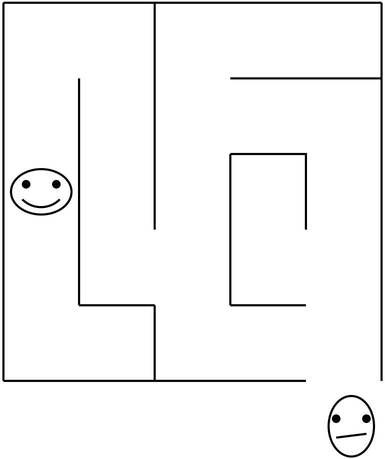
Exploring a maze.

Theseus: Wants to find the minatour in the maze.

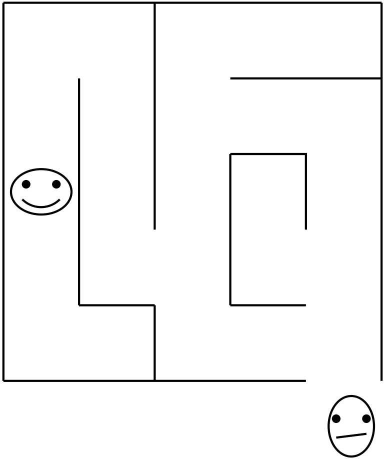
Theseus has access to a Ball of Thread and a Chalk!

Explore a room: Mark room with chalk.
For each exit.
  Look through exit. If marked, next exit.
  Otherwise go in room unwind thread.
    Explore that room.
Wind thread to go back to “previous” room.

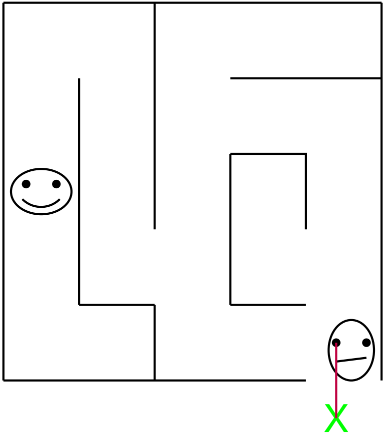
Where is the minatour?



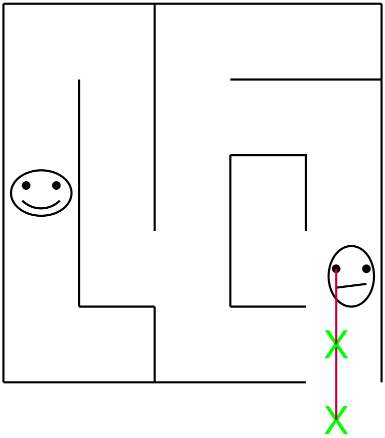
Where is the minatour?



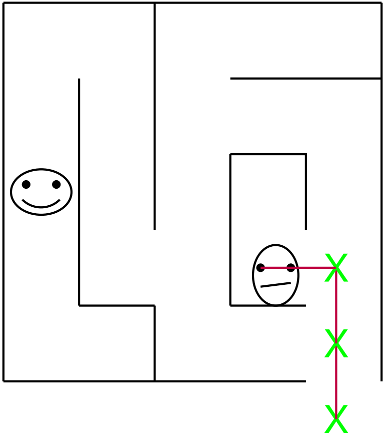
Where is the minatour?



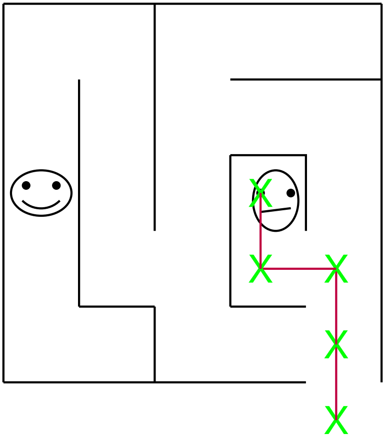
Where is the minatour?



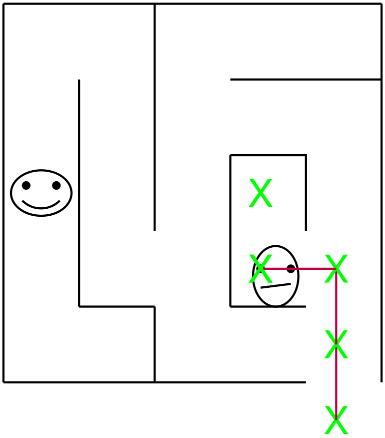
Where is the minatour?



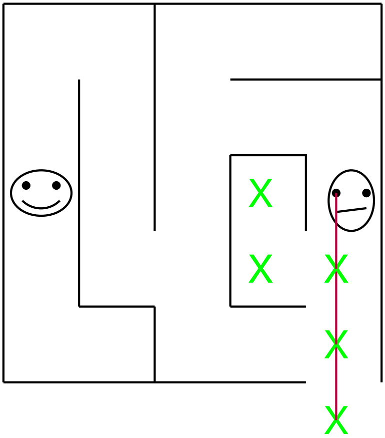
Where is the minatour?



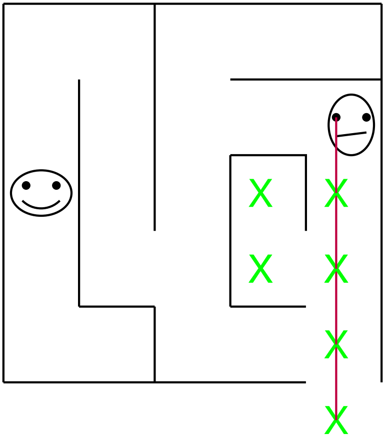
Where is the minatour?



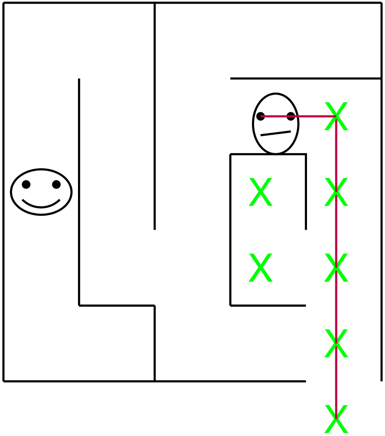
Where is the minatour?



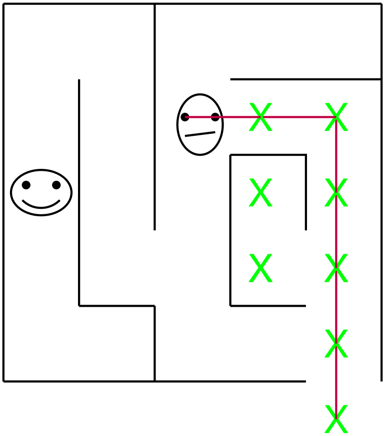
Where is the minatour?



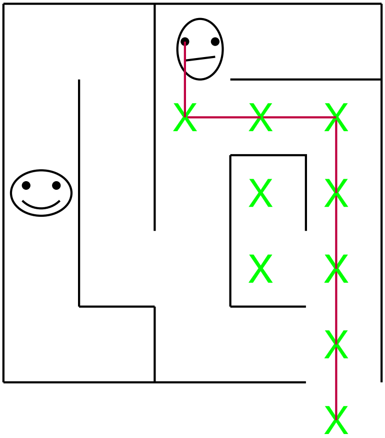
Where is the minatour?



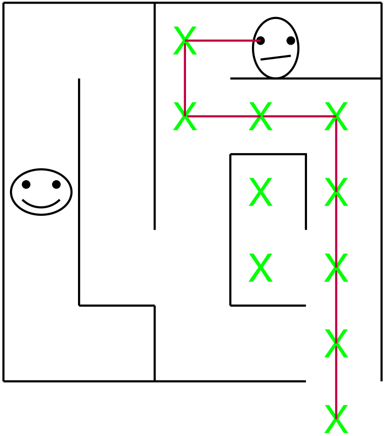
Where is the minatour?



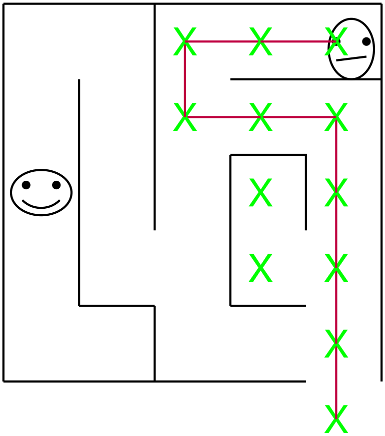
Where is the minatour?



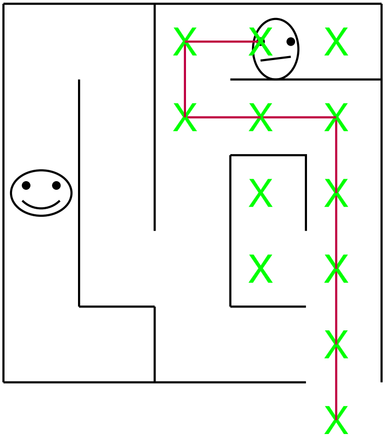
Where is the minatour?



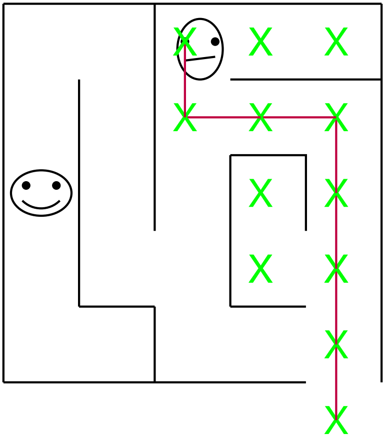
Where is the minatour?



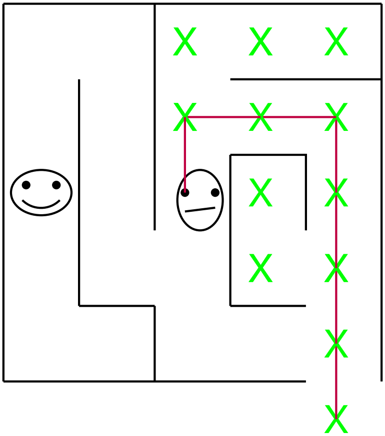
Where is the minatour?



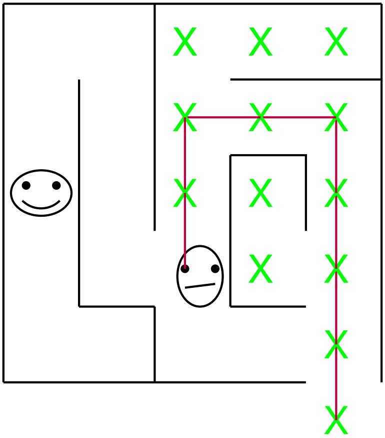
Where is the minatour?



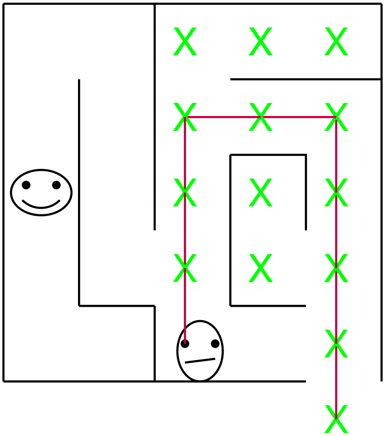
Where is the minatour?



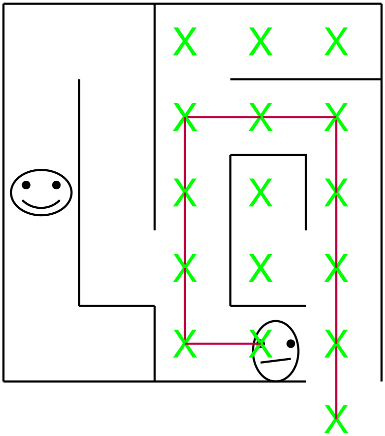
Where is the minatour?



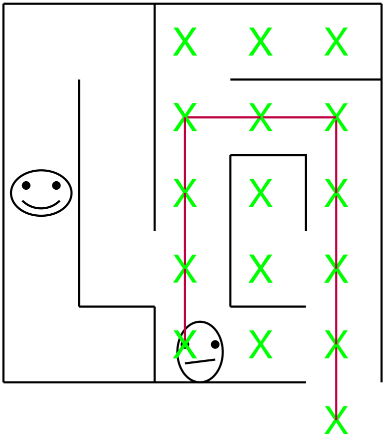
Where is the minatour?



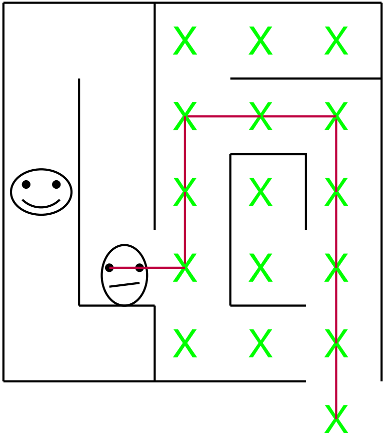
Where is the minatour?



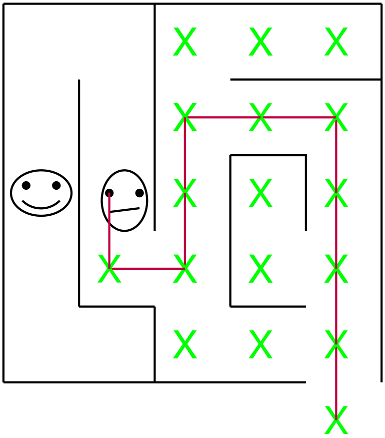
Where is the minatour?



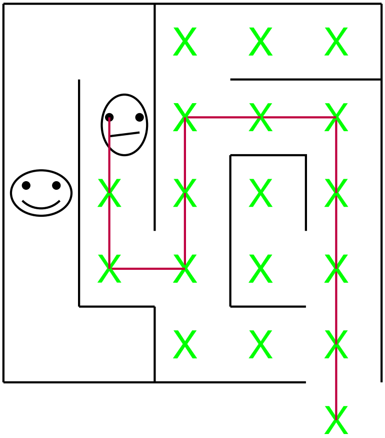
Where is the minatour?



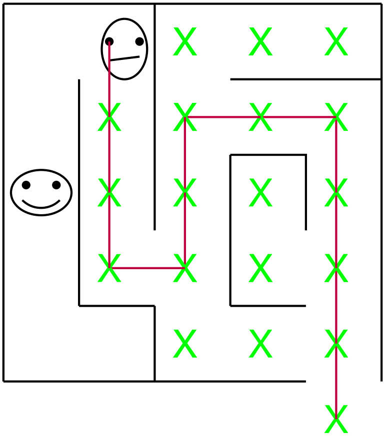
Where is the minatour?



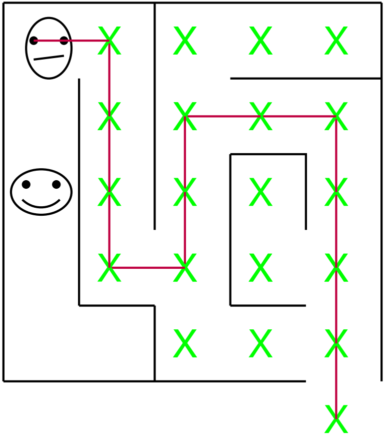
Where is the minatour?



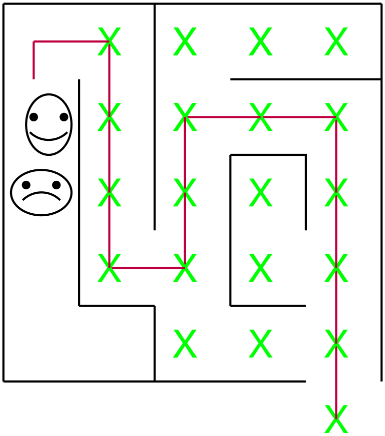
Where is the minatour?



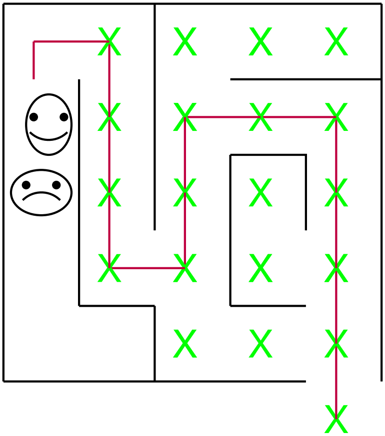
Where is the minatour?



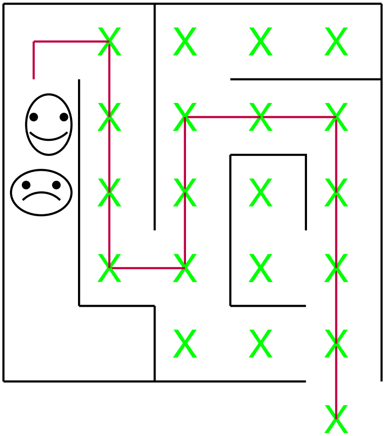
Where is the minatour?



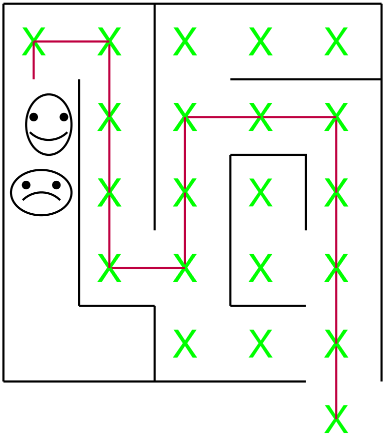
Where is the minatour?



Where is the minatour?



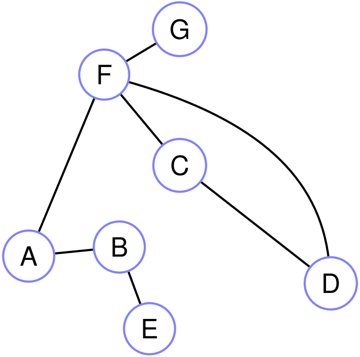
Where is the minatour?



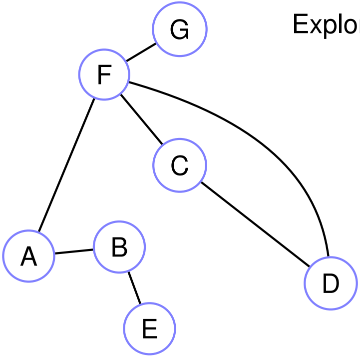
Reachability problem in a Graph.

Problem: Find out which nodes are reachable from A.
Need digital analogues of the chalk and ball of thread.
We will use array (visited) for chalk and stack for thread.

Explore.



Explore.



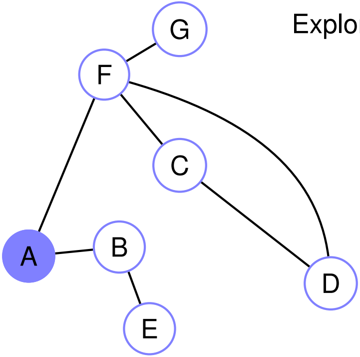
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



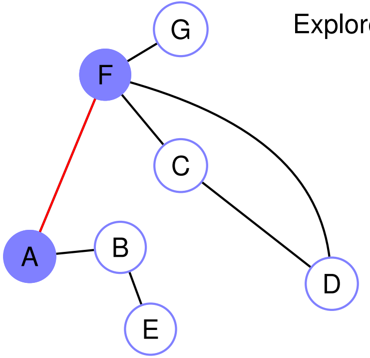
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



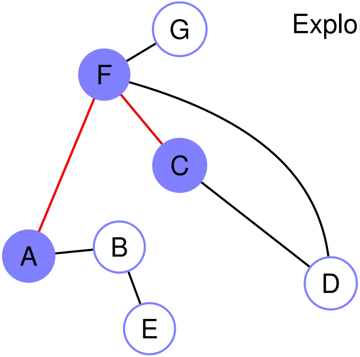
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



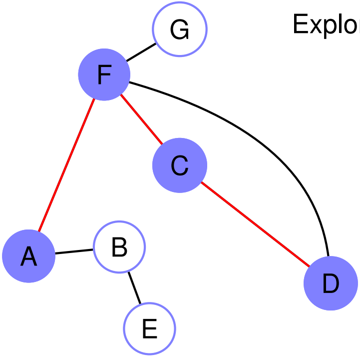
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



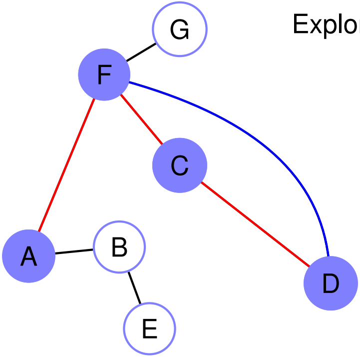
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



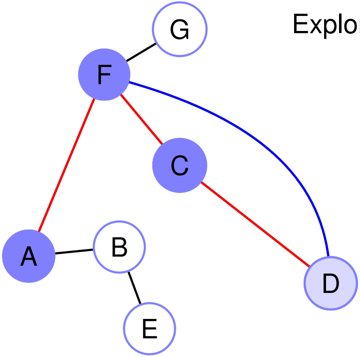
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



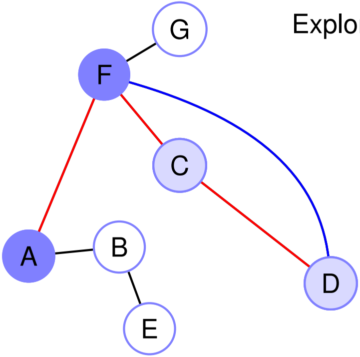
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



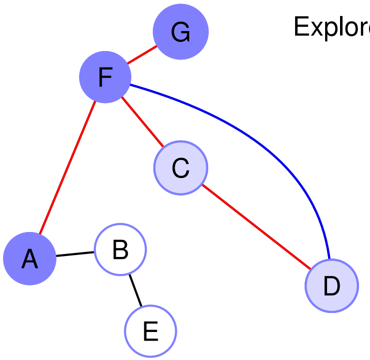
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



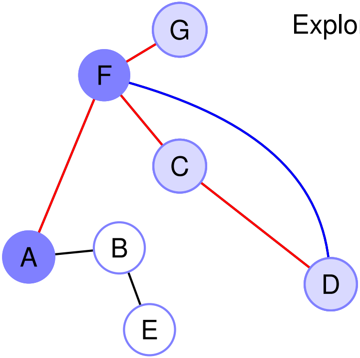
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.

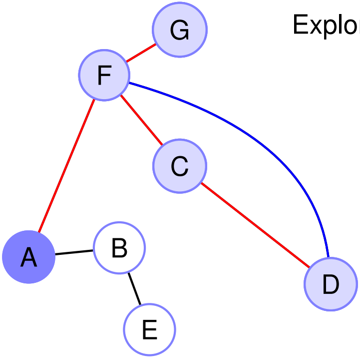


Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.
Stack is Thread.

Explore.



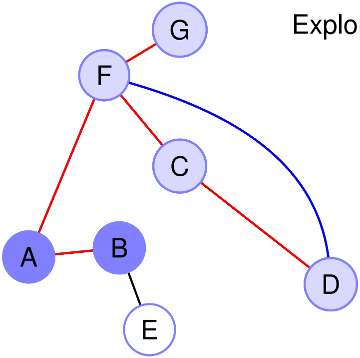
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

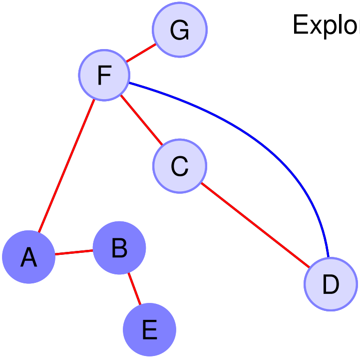
Explore.



Explore(v):
1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.
Stack is Thread.

Explore.



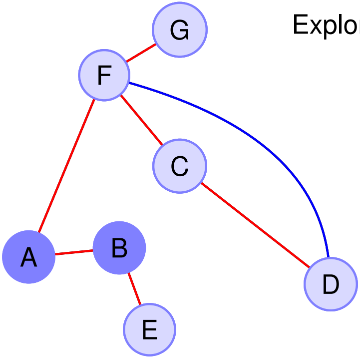
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



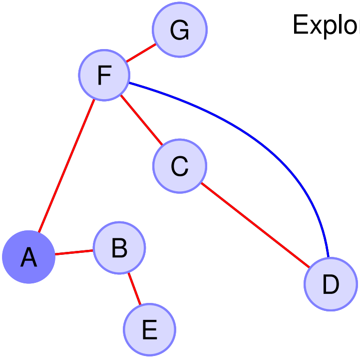
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



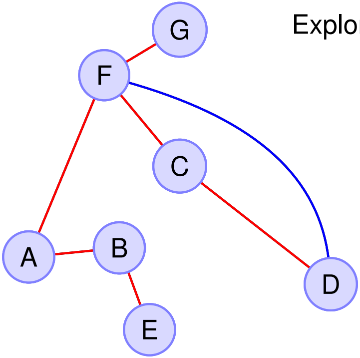
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



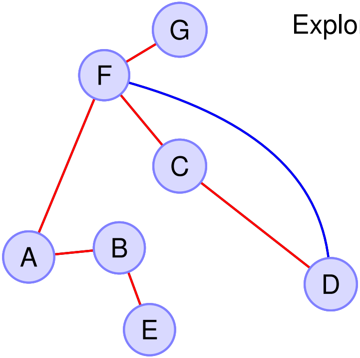
Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.

Stack is Thread.

Explore.



Explore(v):

1. Set visited[v] := true
2. for each edge (v,w) in E
3. if not visited[w]: Explore(w).

Chalk.
Stack is Thread.

Explore builds tree.

Tree and *back* edges.

Correctness.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w)

Property:

All and only nodes reachable from *A* are reached by explore.

Correctness.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w)

Property:

All and only nodes reachable from *A* are reached by explore.

Only: when *u* visited.

Correctness.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w)

Property:

All and only nodes reachable from *A* are reached by explore.

Only: when *u* visited.

stack contains nodes in a path from *a* to *u*.

Correctness.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w)

Property:

All and only nodes reachable from *A* are reached by explore.

Only: when *u* visited.
stack contains nodes in a path from *a* to *u*.

All: if a node *u* is reachable.
there is a path to it. Assume: *u* not found.

Correctness.

Explore(v):

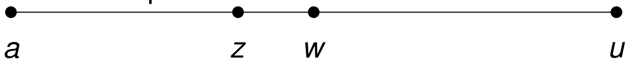
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w)

Property:

All and only nodes reachable from *A* are reached by explore.

Only: when *u* visited.
stack contains nodes in a path from *a* to *u*.

All: if a node *u* is reachable.
there is a path to it. Assume: *u* not found.



Correctness.

Explore(v):

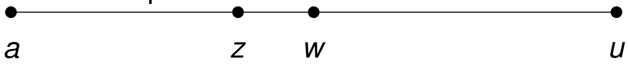
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w)

Property:

All and only nodes reachable from *A* are reached by explore.

Only: when *u* visited.
stack contains nodes in a path from *a* to *u*.

All: if a node *u* is reachable.
there is a path to it. Assume: *u* not found.



z is explored.

Correctness.

Explore(v):

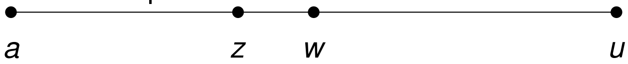
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w)

Property:

All and only nodes reachable from *A* are reached by explore.

Only: when *u* visited.
  stack contains nodes in a path from *a* to *u*.

All: if a node *u* is reachable.
  there is a path to it. Assume: *u* not found.



z is explored. *w* is not!

Correctness.

Explore(v):

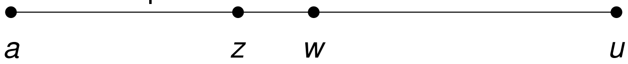
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w)

Property:

All and only nodes reachable from A are reached by explore.

Only: when u visited.
stack contains nodes in a path from a to u.

All: if a node u is reachable.
there is a path to it. Assume: u not found.



z is explored. w is not!
Explore (z) would explore(w), or it was already explored!

Correctness.

Explore(v):

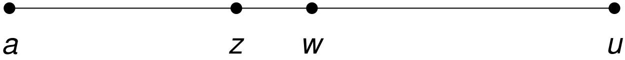
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w)

Property:

All and only nodes reachable from *A* are reached by explore.

Only: when *u* visited.
  stack contains nodes in a path from *a* to *u*.

All: if a node *u* is reachable.
  there is a path to it. Assume: *u* not found.



z is explored. *w* is not!
Explore (*z*) would explore(*w*), or it was already explored!
Contradiction.

□

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

$T(n,m) \leq (d)T(n-1,m) + O(d)$

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

$T(n,m) \leq (d)T(n-1,m) + O(d)$ Exponential

?!?!?!

Running Time.

Explore(v):

1. Set visited[v] := **true.**
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

$T(n,m) \leq (d)T(n-1,m) + O(d)$ Exponential
?!?!?!

Don't use recurrence!

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w).

How to analyse?

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

"Charge work to something."

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

"Charge work to something."

For node x:

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

"Charge work to something."

For node x:
  Explore once!

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

"Charge work to something."

For node x:

Explore once!
Process each incident edge.

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

"Charge work to something."

For node x:

Explore once!
Process each incident edge.

Each edge processed twice.

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

"Charge work to something."

For node x:
  Explore once!
  Process each incident edge.

Each edge processed twice.

$O(n)$ - call explore on n nodes.

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

"Charge work to something."

For node x:

Explore once!
Process each incident edge.

Each edge processed twice.

$O(n)$ - call explore on n nodes.
$O(m)$ - process each edge twice.

Running Time.

Explore(v):

1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.   if not visited[w]: Explore(w).

How to analyse?

Let $n = |V|$, and $m = |E|$.

"Charge work to something."

For node x:
  Explore once!
  Process each incident edge.

Each edge processed twice.

$O(n)$ - call explore on n nodes.
$O(m)$ - process each edge twice.
Total: $O(n+m)$.

Depth first search.

Process whole graph.

Depth first search.

Process whole graph.

DFS(G)

1: For each node u,

Depth first search.

Process whole graph.

DFS(G)

1: For each node u,
2: visited[u] = **false**.

Depth first search.

Process whole graph.

DFS(G)

For each node *u*,
    visited[*u*] = **false**.
For each node *u*,

Depth first search.

Process whole graph.

DFS(G)

```
For each node u,
    visited[u] = false.
For each node u,
    if not visited[u] explore(u)
```

Depth first search.

Process whole graph.

DFS(G)

1: For each node u,
2:    visited[u] = **false**.
3: For each node u,
4:    if not visited[u] **explore(**u**)**

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

Depth first search.

Process whole graph.

DFS(G)

For each node u,
    visited[u] = **false**.
For each node u,
    if not visited[u] **explore(**u**)**

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

Intuitively: tree for each “connected component”.

Depth first search.

Process whole graph.

DFS(G)

1: For each node u,
2: visited[u] = **false**.
3: For each node u,
4: if not visited[u] **explore(**u**)**

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

Intuitively: tree for each "connected component".
Several trees

Depth first search.

Process whole graph.

DFS(G)

```
For each node u,
    visited[u] = false.
For each node u,
    if not visited[u] explore(u)
```

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

Intuitively: tree for each "connected component".
Several trees or Forest!

Depth first search.

Process whole graph.

DFS(G)

1: For each node u,
2:    visited[u] = **false**.
3: For each node u,
4:    if not visited[u] **explore(**u**)**

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

Intuitively: tree for each "connected component".
Several trees or Forest! Output connected components?

DFS and connected components.

DFS and connected components.

Change explore a bit:

DFS and connected components.

Change explore a bit:

explore(v):

1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

DFS and connected components.

Change explore a bit:

explore(v):

1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

previsit(v):

1. Set cc[v] := ccnum.

DFS and connected components.

Change explore a bit:

explore(v):

1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

previsit(v):

1. Set cc[v] := ccnum.

DFS(G):

0. Set ccnum := 0.
1. for each v in V:
2.   if not visited[v]:
3.   explore(v)
4.   ccnum = ccnum+1

DFS and connected components.

Change explore a bit:

explore(v):

1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

previsit(v):

1. Set cc[v] := ccnum.

DFS(G):

0. Set ccnum := 0.
1. for each v in V:
2.   if not visited[v]:
3.   explore(v)
4.   ccnum = ccnum+1

Each node will be labelled with connected component number.

DFS and connected components.

Change explore a bit:

explore(v):

1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

previsit(v):

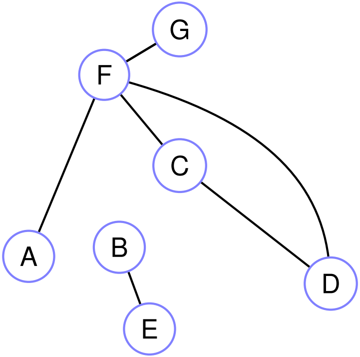
1. Set cc[v] := ccnum.

DFS(G):

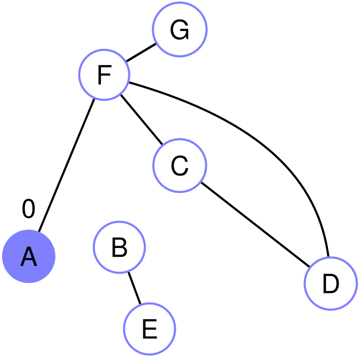
0. Set ccnum := 0.
1. for each v in V:
2.   if not visited[v]:
3.   explore(v)
4.   ccnum = ccnum+1

Each node will be labelled with connected component number.
Runtime: $O(|V| + |E|)$.

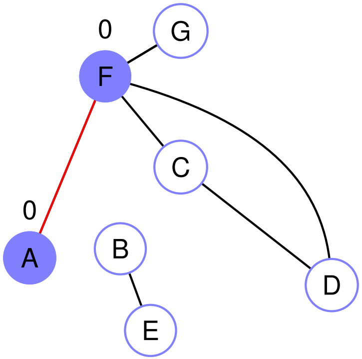
Connected Components.



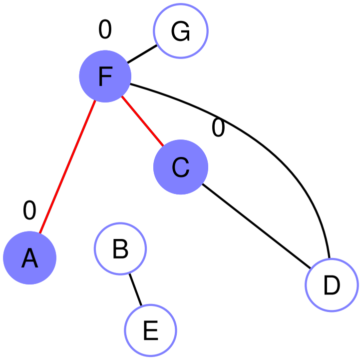
Connected Components.



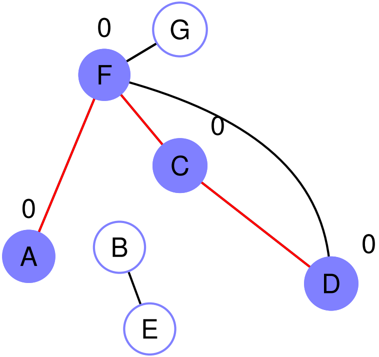
Connected Components.



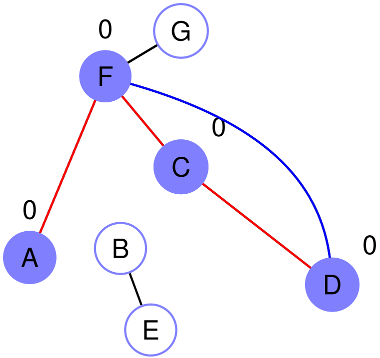
Connected Components.



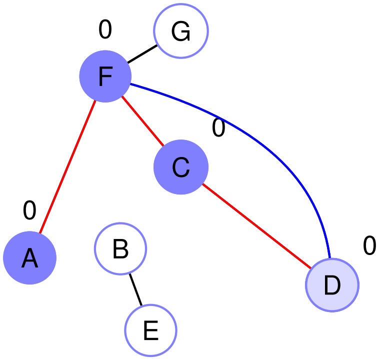
Connected Components.



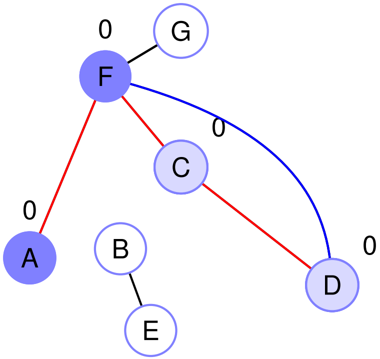
Connected Components.



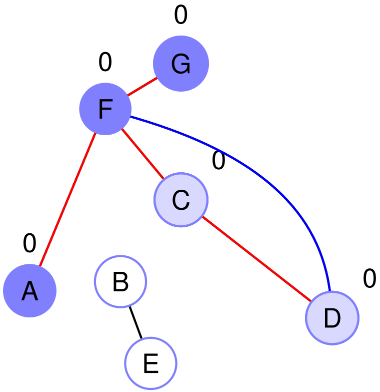
Connected Components.



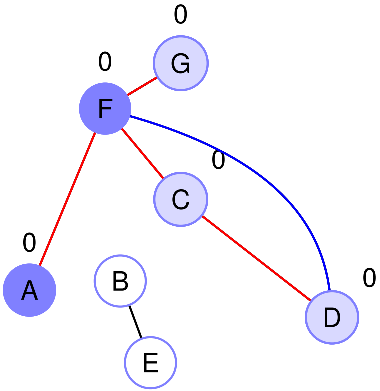
Connected Components.



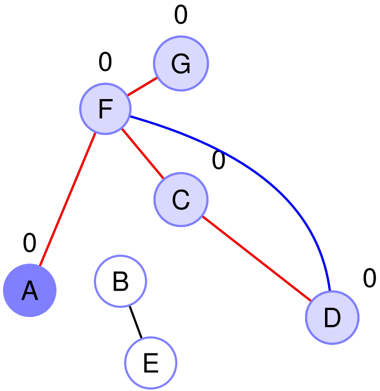
Connected Components.



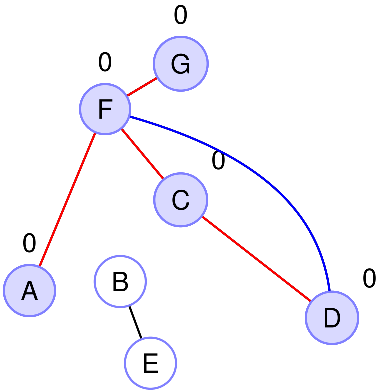
Connected Components.



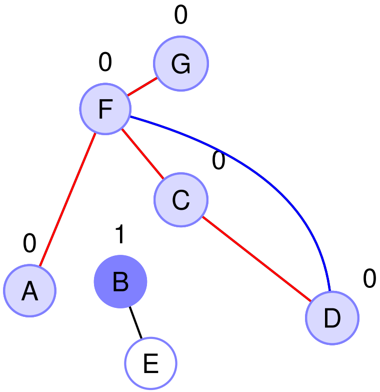
Connected Components.



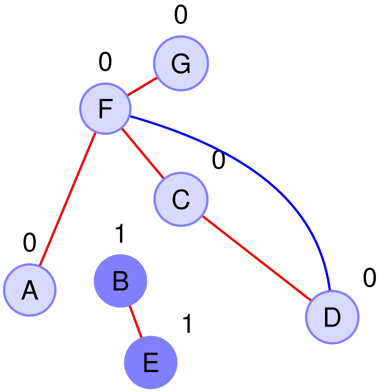
Connected Components.



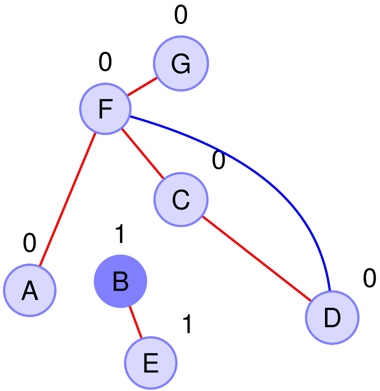
Connected Components.



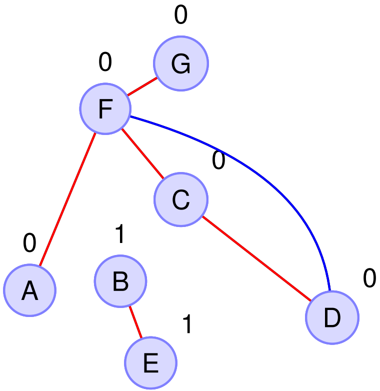
Connected Components.



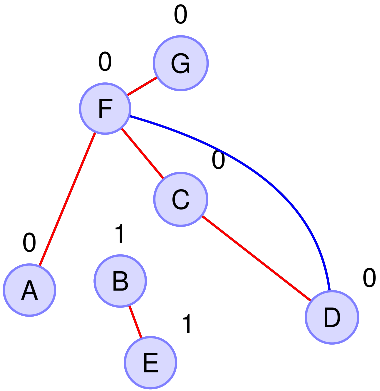
Connected Components.



Connected Components.



Connected Components.



Introspection: pre/post.

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

⋮

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

   ⋮

Clock: goes up to

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

   ⋮

Clock: goes up to 2 times number of vertices.

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

⋮

Clock: goes up to 2 times number of vertices.
First pre:

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

⋮

Clock: goes up to 2 times number of vertices.
First pre: 0

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

$\vdots$

Clock: goes up to 2 times number of vertices.

First pre: 0

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

$\vdots$

Clock: goes up to 2 times number of vertices.
First pre: 0

Property:
For any two nodes, u and v, $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either **disjoint** or **one is contained in other.**

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

$\vdots$

Clock: goes up to 2 times number of vertices.
First pre: 0

Property:
For any two nodes, u and v, $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either **disjoint** or **one is contained in other.**

Interval is "clock interval on stack."

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

   ⋮

Clock: goes up to 2 times number of vertices.
First pre: 0

Property:
For any two nodes, u and v, $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either **disjoint** or **one is contained in other.**

Interval is "clock interval on stack."

Either both on stack at some point (contained) or not (disjoint.)

Introspection: pre/post.

previsit(v):

1. Set pre[v] := clock.
2. clock := clock+1

postvisit(v):

1. Set post[v] := clock.
2. clock := clock+1

DFS(G):

0. Set clock := 0.

   ⋮

Clock: goes up to 2 times number of vertices.
First pre: 0

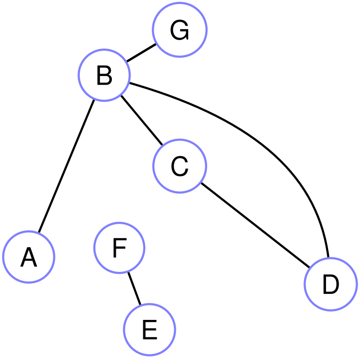
Property:
For any two nodes, u and v, $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either **disjoint** or **one is contained in other.**

Interval is “clock interval on stack.”

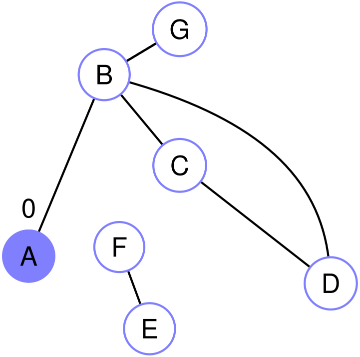
Either both on stack at some point (contained) or not (disjoint.)

Let’s just watch it work!

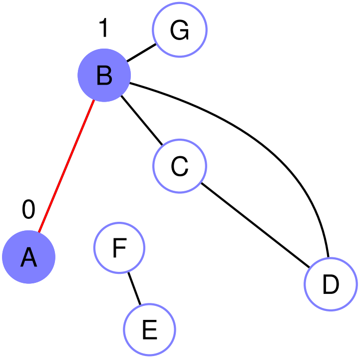
Example: Pre/Post numbering.



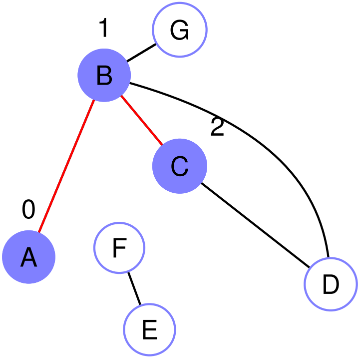
Example: Pre/Post numbering.



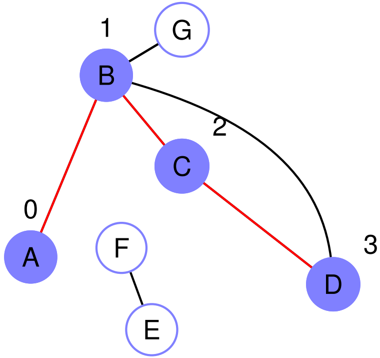
Example: Pre/Post numbering.



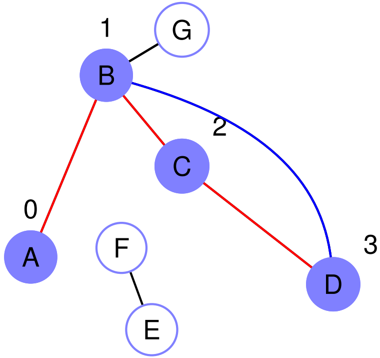
Example: Pre/Post numbering.



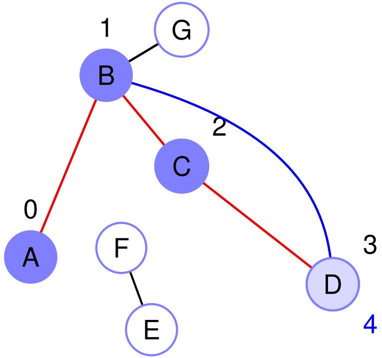
Example: Pre/Post numbering.



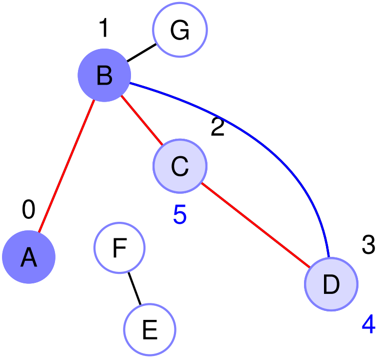
Example: Pre/Post numbering.



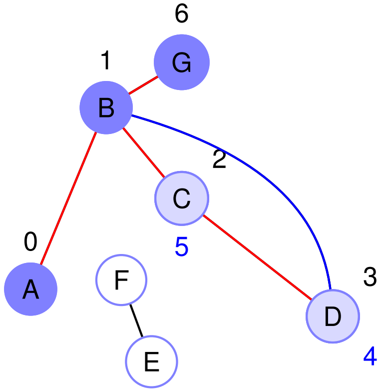
Example: Pre/Post numbering.



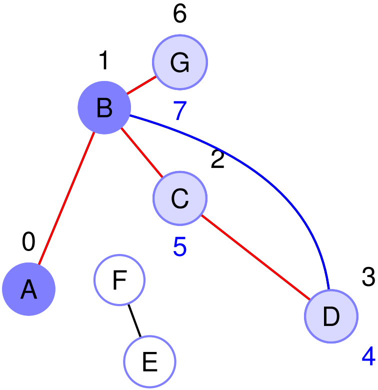
Example: Pre/Post numbering.



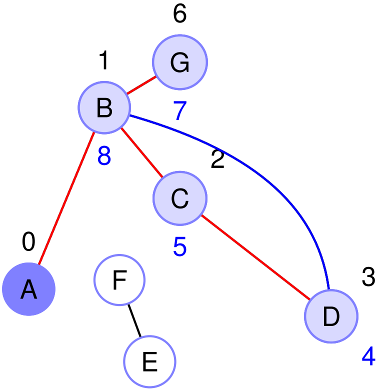
Example: Pre/Post numbering.



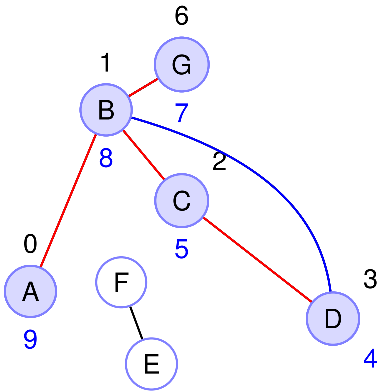
Example: Pre/Post numbering.



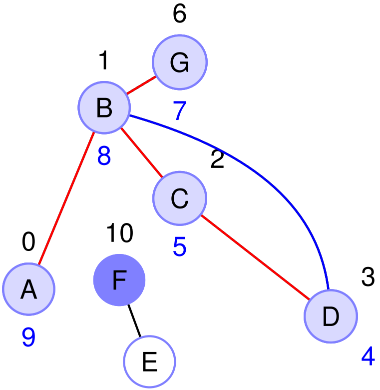
Example: Pre/Post numbering.



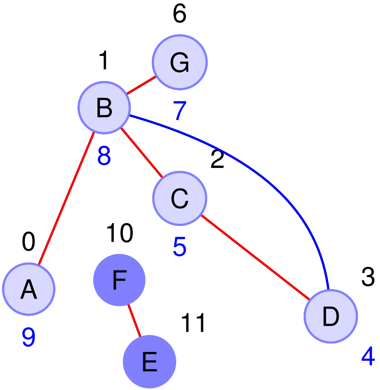
Example: Pre/Post numbering.



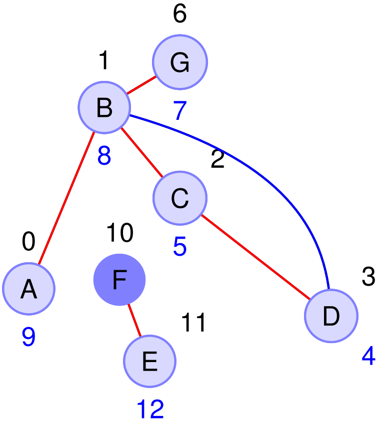
Example: Pre/Post numbering.



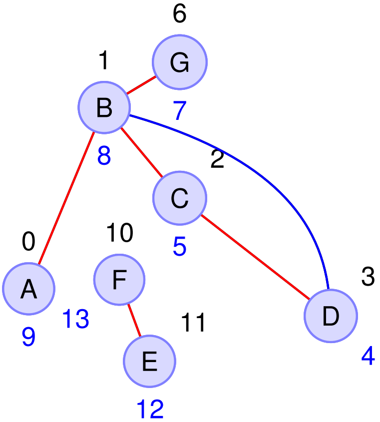
Example: Pre/Post numbering.



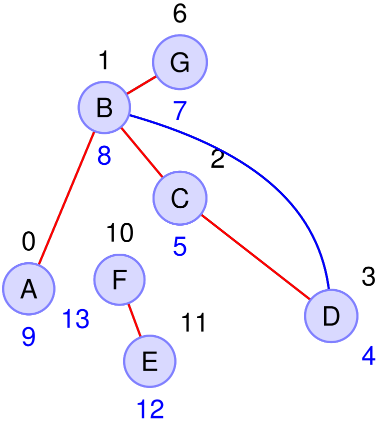
Example: Pre/Post numbering.



Example: Pre/Post numbering.



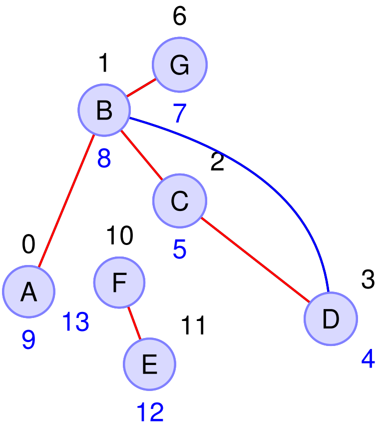
Example: Pre/Post numbering.



Edge (u, v) is tree edge iff $[pre[v], post[v]] \subset [pre[u], post[u]]$.

u on stack before v.

Example: Pre/Post numbering.



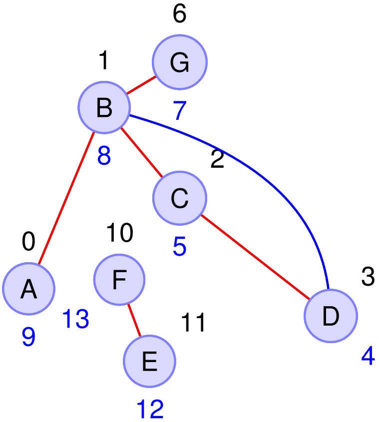
Edge (u, v) is tree edge iff $[pre[v], post[v]] \subset [pre[u], post[u]]$.

u on stack before v.

Edge (u, v) is back edge iff $[pre[u], post[u]] \subset [pre[v], post[v]]$.

v on stack before u on stack. Path from v to u! Cycle!

Example: Pre/Post numbering.



Edge (u, v) is tree edge iff $[pre[v], post[v]] \subset [pre[u], post[u]]$.

u on stack before v.

Edge (u, v) is back edge iff $[pre[u], post[u]] \subset [pre[v], post[v]]$.

v on stack before u on stack. Path from v to u! Cycle!

No edge between u and v if disjoint intervals.

Directed graphs.

$G = (V, E)$

Directed graphs.

$G = (V, E)$

vertices V.

Directed graphs.

$G = (V, E)$

vertices V.

edges $E \subseteq V \times V$.

Directed graphs.

$G = (V, E)$
vertices V.
edges $E \subseteq V \times V$.

Edge: (u, v)

Directed graphs.

$G = (V, E)$
vertices V.
edges $E \subseteq V \times V$.

Edge: (u, v)
From u to v.

Directed graphs.

$G = (V, E)$
vertices V.
edges $E \subseteq V \times V$.

Edge: (u, v)
From u to v.
Tail – u

Directed graphs.

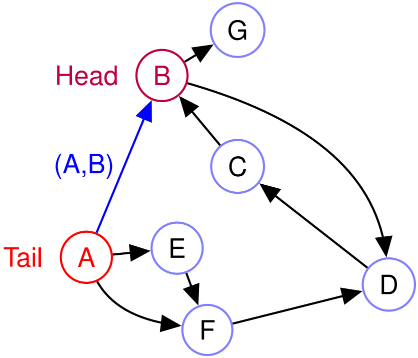
$G = (V, E)$
- vertices V.
- edges $E \subseteq V \times V$.

Edge: (u, v)
- From u to v.
  - Tail – u
  - Head – v

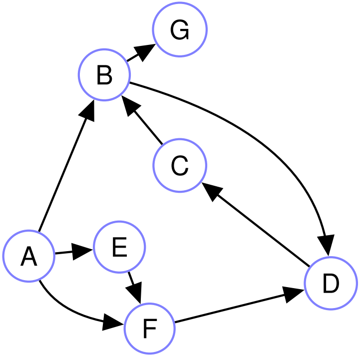
Directed graphs.

$G = (V, E)$
vertices V.
edges $E \subseteq V \times V$.

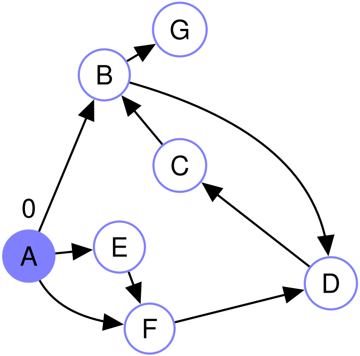
Edge: (u, v)
From u to v.
Tail – u
Head – v



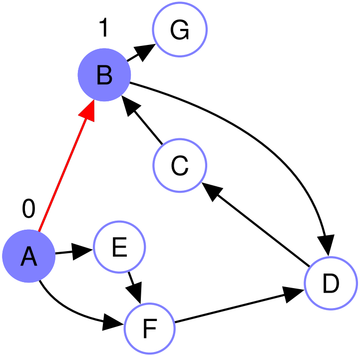
Depth first search: directed.



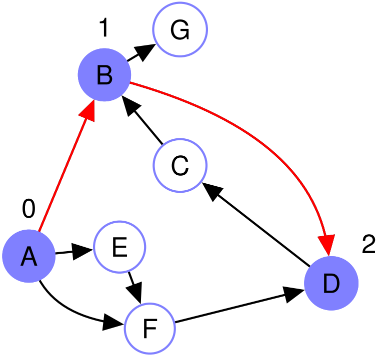
Depth first search: directed.



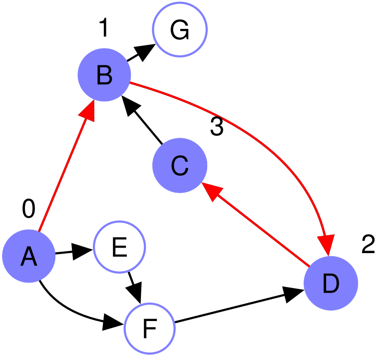
Depth first search: directed.



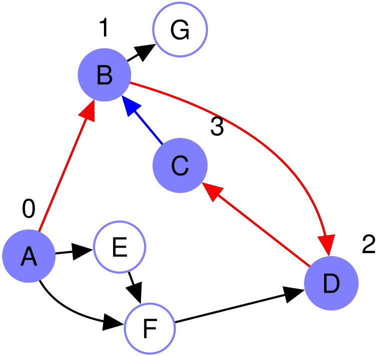
Depth first search: directed.



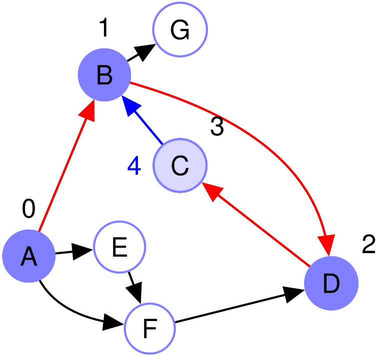
Depth first search: directed.



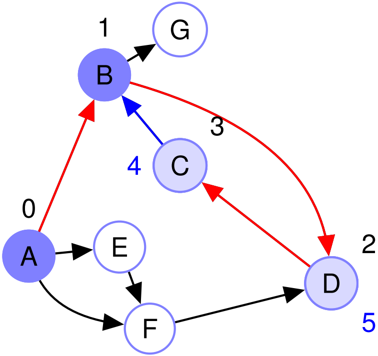
Depth first search: directed.



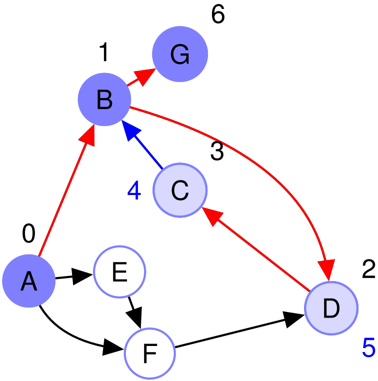
Depth first search: directed.



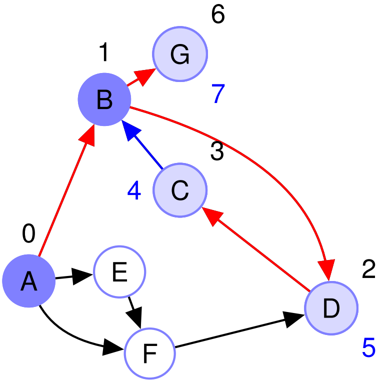
Depth first search: directed.



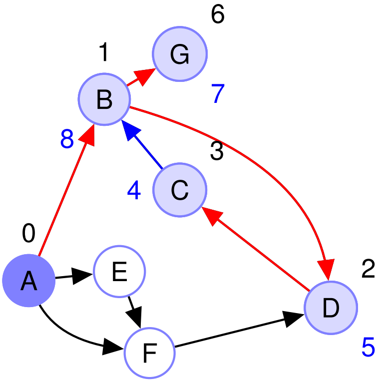
Depth first search: directed.



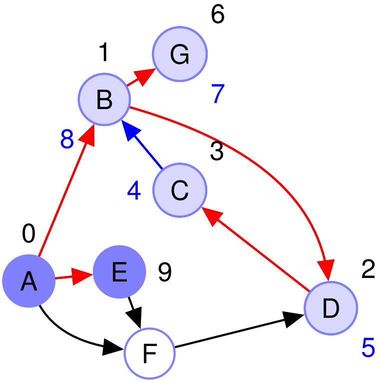
Depth first search: directed.



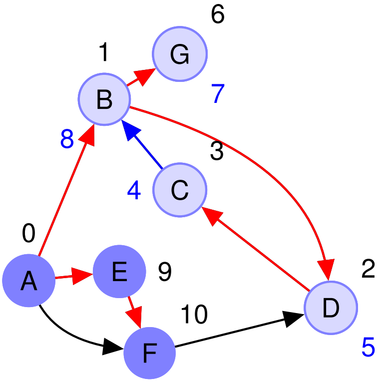
Depth first search: directed.



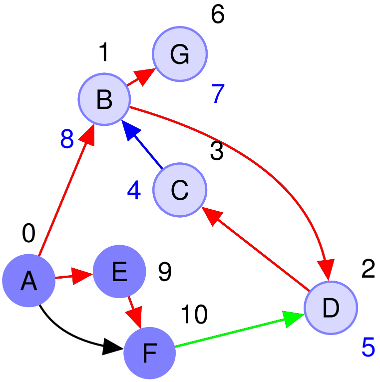
Depth first search: directed.



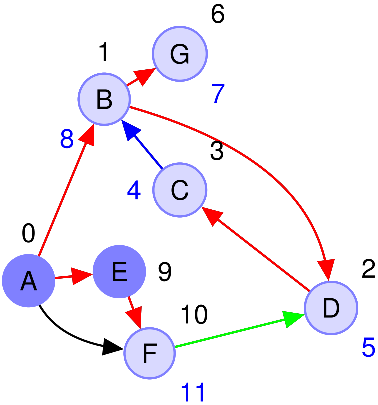
Depth first search: directed.



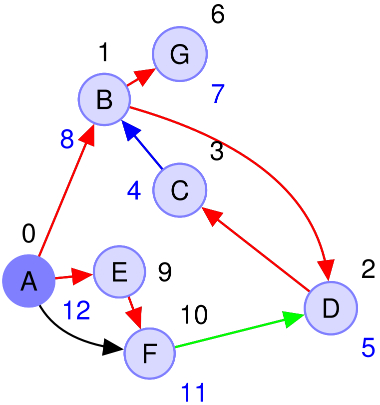
Depth first search: directed.



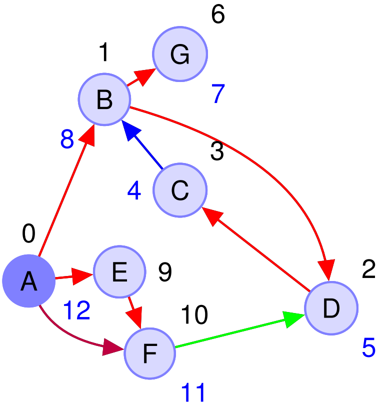
Depth first search: directed.



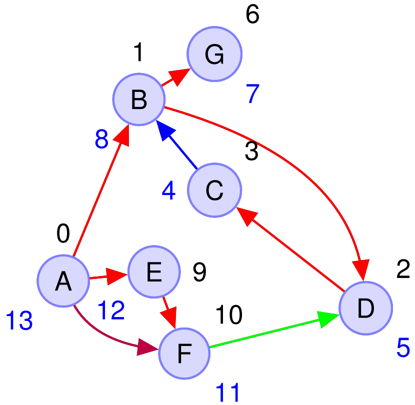
Depth first search: directed.



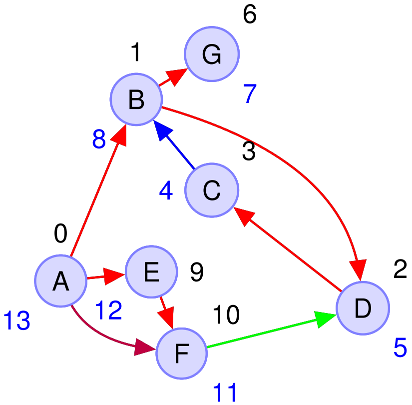
Depth first search: directed.



Depth first search: directed.

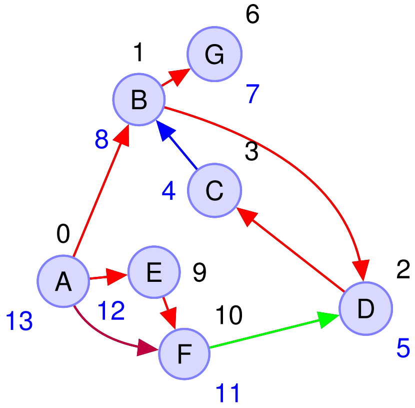


Depth first search: directed.



Tree/forward edge (u,v): $int(v) \subset int(u)$. $inv(v) = [pre(v), post(v)]$

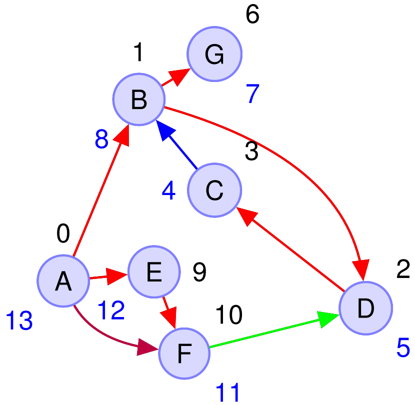
Depth first search: directed.



Tree/forward edge (u, v): $int(v) \subset int(u)$. $inv(v) = [pre(v), post(v)]$

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

Depth first search: directed.

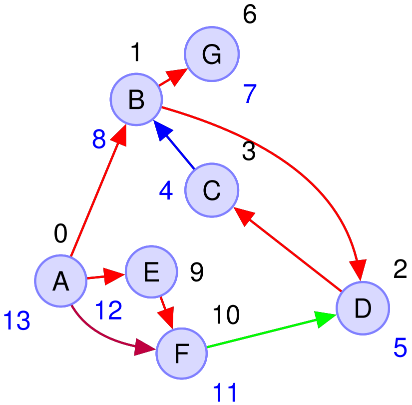


Tree/forward edge (u, v): $int(v) \subset int(u)$. $inv(v) = [pre(v), post(v)]$

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

Back edge (u, v): $int(u) \subset int(v)$.

Depth first search: directed.



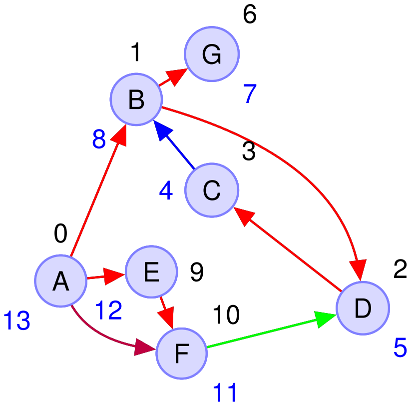
Tree/forward edge (u, v): $int(v) \subset int(u)$. $inv(v) = [pre(v), post(v)]$

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

Back edge (u, v): $int(u) \subset int(v)$.

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

Depth first search: directed.



Tree/forward edge (u, v): $int(v) \subset int(u)$. $inv(v) = [pre(v), post(v)]$

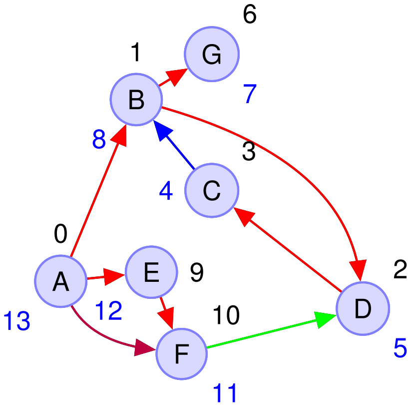
Forward (A, F): [10,11] in [0,13] or [0,[10,11],13]

Back edge (u, v): $int(u) \subset int(v)$.

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

Cross edge (u, v): $int(v) < int(u)$.

Depth first search: directed.



Tree/forward edge (u, v): $int(v) \subset int(u)$. $inv(v) = [pre(v), post(v)]$

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

Back edge (u, v): $int(u) \subset int(v)$.

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

Cross edge (u, v): $int(v) < int(u)$.

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

Cycle in a directed graph?

Fast algorithm for finding out whether directed graph has cycle?

Cycle in a directed graph?

Fast algorithm for finding out whether directed graph has cycle?

For each edge (u, v) remove, check if v is connected to u

Cycle in a directed graph?

Fast algorithm for finding out whether directed graph has cycle?

For each edge (u, v) remove, check if v is connected to u
$O(|E|(|E| + |V|))$.

Cycle in a directed graph?

Fast algorithm for finding out whether directed graph has cycle?

For each edge (u, v) remove, check if v is connected to u
$O(|E|(|E| + |V|))$.

Linear Time (i.e. $O(|V| + |E|)$)?

Testing for cycle.

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

Testing for cycle.

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

Proof:

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

Testing for cycle.

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

Proof:

We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

$v_0 \to v_1$

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

$v_0 \rightarrow v_1 \rightarrow v_2$

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

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

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

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

Assume that v_0 is the first node explored in the cycle
(without loss of generality since can renumber vertices.)

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

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

Assume that v_0 is the first node explored in the cycle
(without loss of generality since can renumber vertices.)

When **explore**(v_0) returns all nodes on cycle explored.

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

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

Assume that v_0 is the first node explored in the cycle
(without loss of generality since can renumber vertices.)

When **explore**(v_0) returns all nodes on cycle explored.

All int[v_i] in int[v_0]!

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

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

Assume that v_0 is the first node explored in the cycle
(without loss of generality since can renumber vertices.)

When **explore**(v_0) returns all nodes on cycle explored.

All int[v_i] in int[v_0]!

$\implies (v_k, v_0)$ is a back edge.

Testing for cycle.

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

Proof:
We just saw: Back edge $\implies$ cycle!

In the other direction: Assume there is a cycle

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

Assume that v_0 is the first node explored in the cycle
(without loss of generality since can renumber vertices.)

When **explore**(v_0) returns all nodes on cycle explored.

All int[v_i] in int[v_0]!

$\implies (v_k, v_0)$ is a back edge.

Cycle $\implies$ back edge! $\square$

Fast checking algorithm.

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

Fast checking algorithm.

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

Algorithm ??

Fast checking algorithm.

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

Algorithm ??

Fast checking algorithm.

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

Algorithm ??

Run DFS.

Fast checking algorithm.

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

Algorithm ??

Run DFS.

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

Fast checking algorithm.

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

Algorithm ??

Run DFS.

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

For each edge (u, v): is int(u) in int(v)?

Fast checking algorithm.

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

Algorithm ??

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

For each edge (u, v): is int(u) in int(v)?
$O(|E|)$ time.

Fast checking algorithm.

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

Algorithm ??

Run DFS.

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

For each edge (u, v): is int(u) in int(v)?

$O(|E|)$ time.

$O(|V| + |E|)$ time algorithm for checking if graph is acyclic!

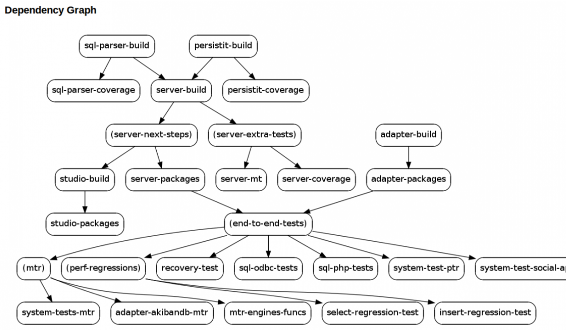
Directed Acyclic Graph

Hello

▼

Goodbye

“Hello” before “Goodbye



Directed Acyclic Graph

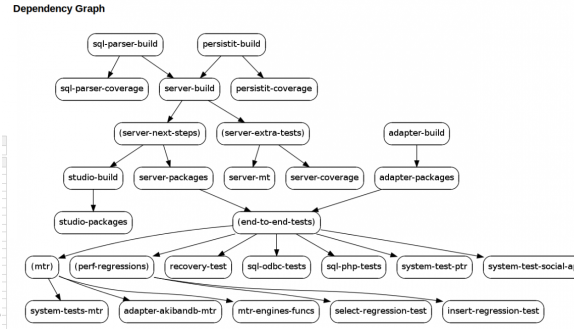
Hello

▼

Goodbye

“Hello” before “Goodbye

No cycles!



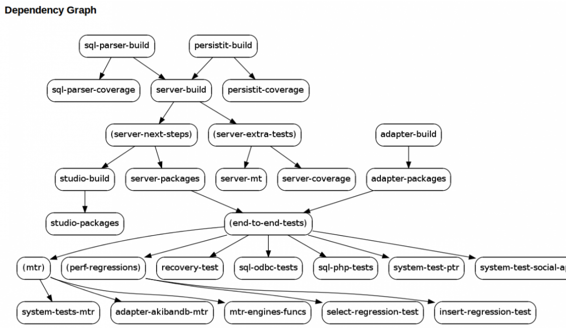
Directed Acyclic Graph

Hello

▼

Goodbye

“Hello” before “Goodbye



No cycles! Can tell in linear time!

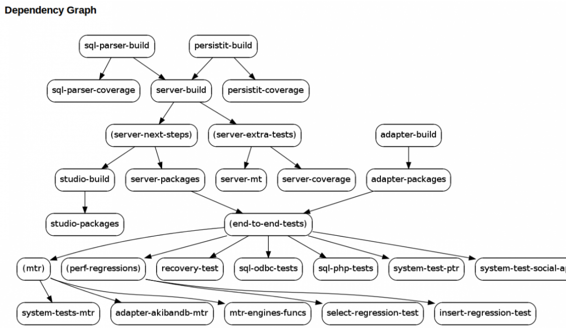
Directed Acyclic Graph

Hello

▼

Goodbye

“Hello” before “Goodbye”



No cycles! Can tell in linear time!

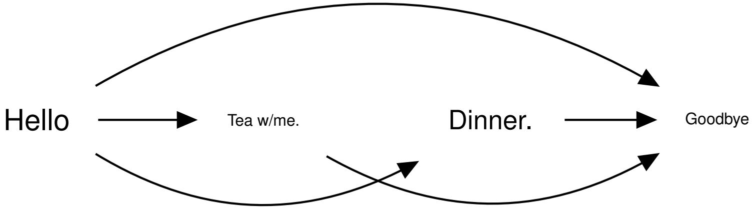
Really want to find ordering for build!

Linearize.

Topological Sort: For $G = (V, E)$, find ordering of all vertices where each edge goes from earlier vertex to later in acyclic graph.

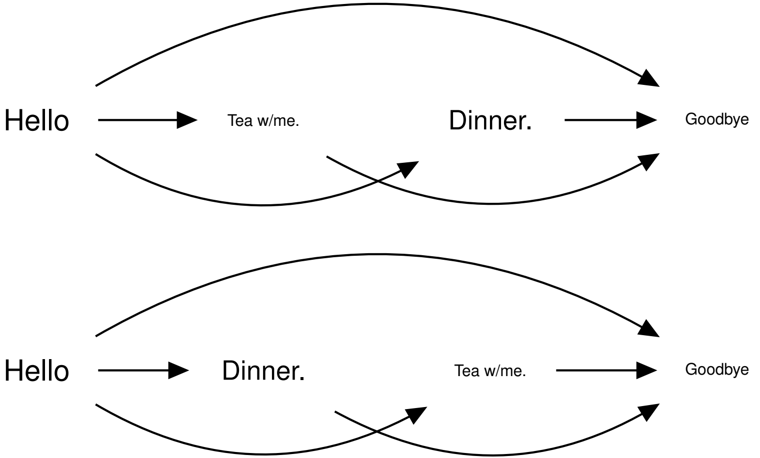
Linearize.

Topological Sort: For $G = (V, E)$, find ordering of all vertices where each edge goes from earlier vertex to later in acyclic graph.

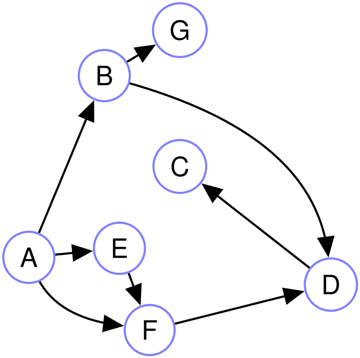


Linearize.

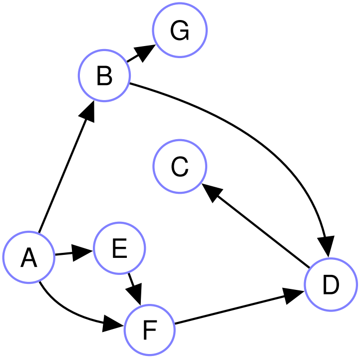
Topological Sort: For $G = (V, E)$, find ordering of all vertices where each edge goes from earlier vertex to later in acyclic graph.



Topological Sort Example.

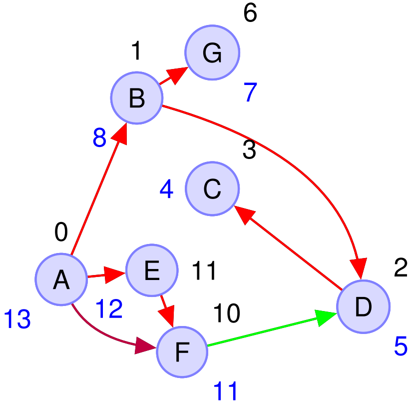


Topological Sort Example.



A linear order:

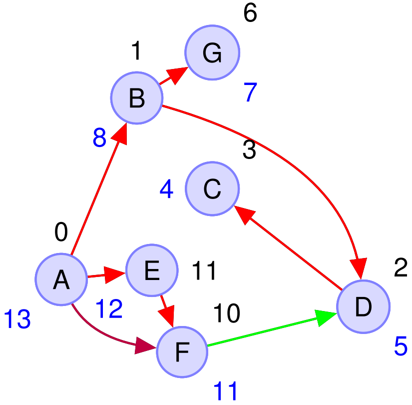
Topological Sort Example.



A linear order:

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

Topological Sort Example.

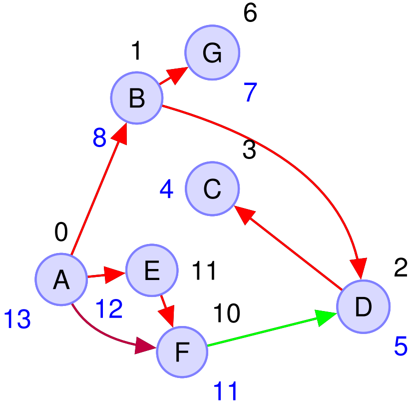


A linear order:

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

In DFS: When is A popped off stack?

Topological Sort Example.



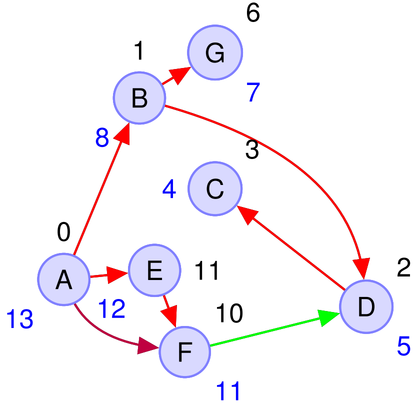
A linear order:

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

In DFS: When is A popped off stack?

Last!

Topological Sort Example.



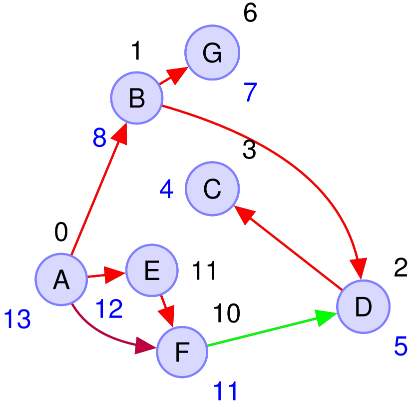
A linear order:

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

In DFS: When is A popped off stack?

Last! When is E popped off?

Topological Sort Example.



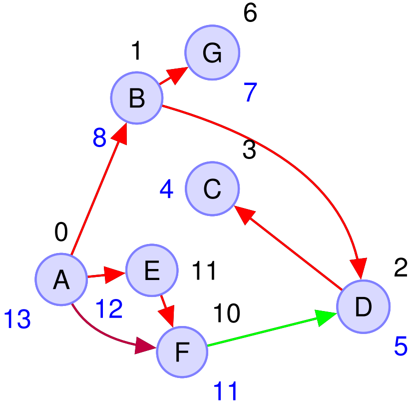
A linear order:

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

In DFS: When is *A* popped off stack?

Last! When is *E* popped off? second to last.

Topological Sort Example.



A linear order:

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

In DFS: When is *A* popped off stack?

Last! When is *E* popped off? second to last. ...

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!

Topological Sort: DFS

Last post order should..

(A) be first in linearization!

(B) be last in linearization!

(A). First!

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Topological Sort: DFS

Last post order should..

(A) be first in linearization!

(B) be last in linearization!

(A). First!

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Proof: No back edges in DAG.

Topological Sort: DFS

Last post order should..

(A) be first in linearization!

(B) be last in linearization!

(A). First!

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Proof: No back edges in DAG.
Tree and Forward edge (u, v):
int(u) contains int(v): $pre(u), pre[v], post[v], post[u]$

Topological Sort: DFS

Last post order should..

(A) be first in linearization!

(B) be last in linearization!

(A). First!

Property: Every edge in a DAG (u, v) has *post*$(u) >$ *post*(v).

Proof: No back edges in DAG.
Tree and Forward edge (u, v):
int(u) contains int(v): *pre*(u), *pre*$[v]$, *post*$[v]$, *post*$[u]$

Cross edge (u, v): int$(u) >$ int(v)

Topological Sort: DFS

Last post order should..

(A) be first in linearization!

(B) be last in linearization!

(A). First!

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Proof: No back edges in DAG.
Tree and Forward edge (u, v):
int(u) contains int(v): $pre(u), pre[v], post[v], post[u]$

Cross edge (u, v): int$(u) >$ int$(v) \implies post[u] > post[v]$

$\square$

Topological Sort: linearize.

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Topological Sort: linearize.

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Top Sort: output in reverse post order number.

Topological Sort: linearize.

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Top Sort: output in reverse post order number.

Runtime: $O(|V| + |E|)$.

Topological Sort: linearize.

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Top Sort: output in reverse post order number.

Runtime: $O(|V| + |E|)$.

Topological Sort: linearize.

Property: Every edge in a DAG (u,v) has $post(u) > post(v)$.

Top Sort: output in reverse post order number.

Runtime: $O(|V|+|E|)$.

..procedure **postvisit** outputs during DFS

Topological Sort: linearize.

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Top Sort: output in reverse post order number.

Runtime: $O(|V| + |E|)$.

..procedure **postvisit** outputs during DFS

def postvisit(u): result.append(u).

Topological Sort: linearize.

Property: Every edge in a DAG (u,v) has $post(u) > post(v)$.

Top Sort: output in reverse post order number.

Runtime: $O(|V|+|E|)$.

..procedure **postvisit** outputs during DFS

def postvisit(u): result.append(u).

..reverse

Topological Sort: linearize.

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.

Top Sort: output in reverse post order number.

Runtime: $O(|V| + |E|)$.

..procedure **postvisit** outputs during DFS

def postvisit(u): result.append(u).

..reverse **result**.