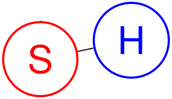
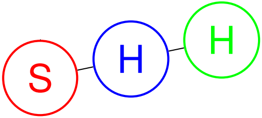
# CS 170: Algorithms

# CS 170: Algorithms

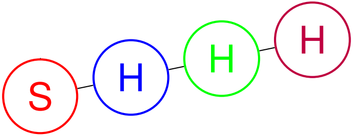
# CS 170: Algorithms



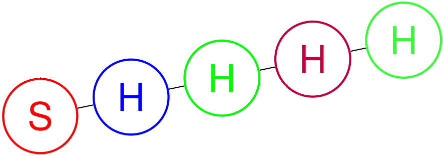
# CS 170: Algorithms



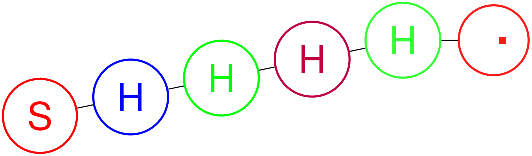
# CS 170: Algorithms



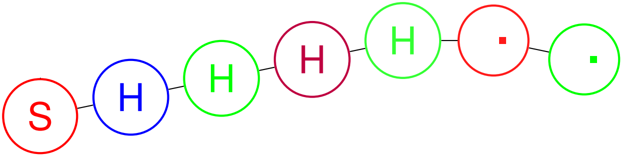
# CS 170: Algorithms



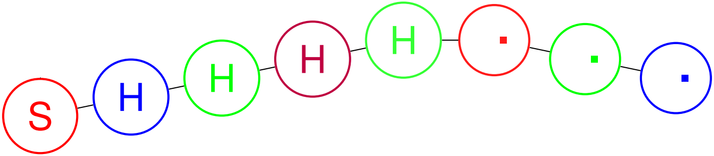
# CS 170: Algorithms



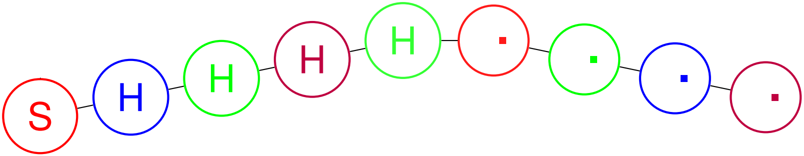
# CS 170: Algorithms



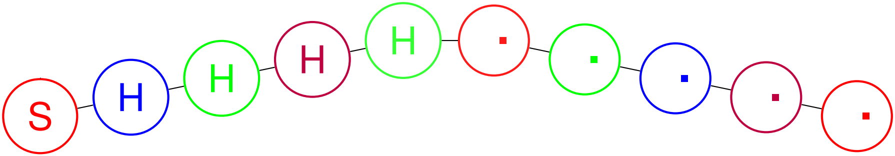
# CS 170: Algorithms



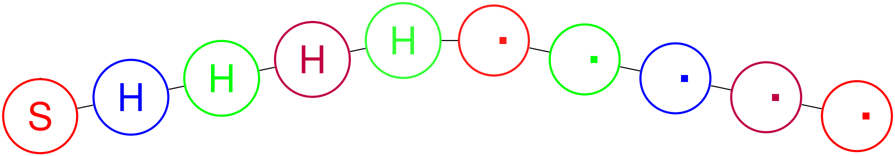
# CS 170: Algorithms



# CS 170: Algorithms

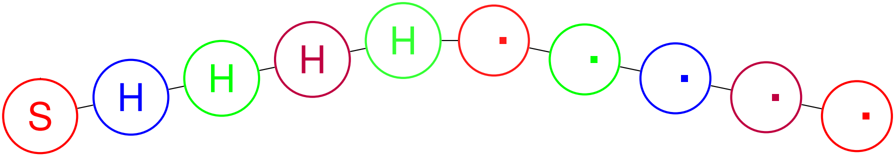


# CS 170: Algorithms




No laptops please.

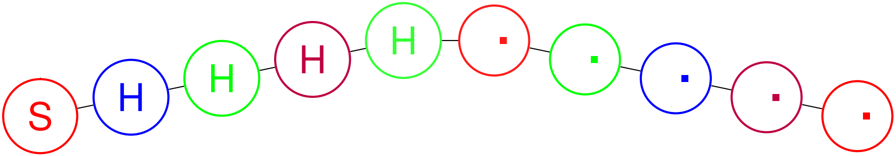
# CS 170: Algorithms




No laptops please.

Thank you

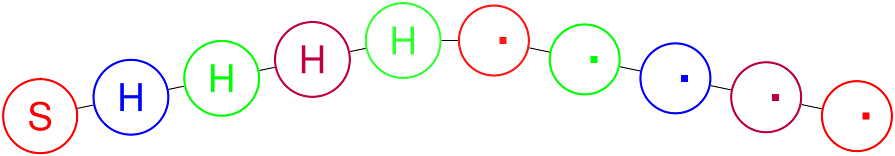
# CS 170: Algorithms




No laptops please.

Thank you !

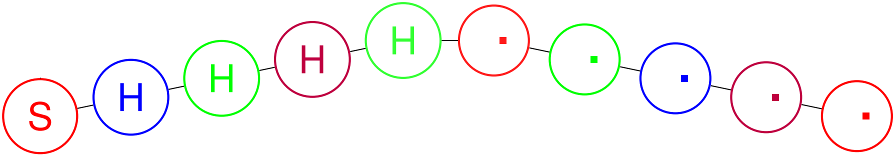
# CS 170: Algorithms




No laptops please.

Thank you ! !

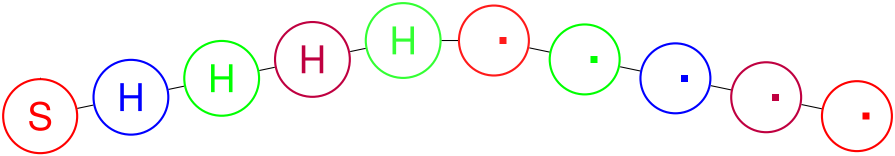
# CS 170: Algorithms




No laptops please.

Thank you ! ! !

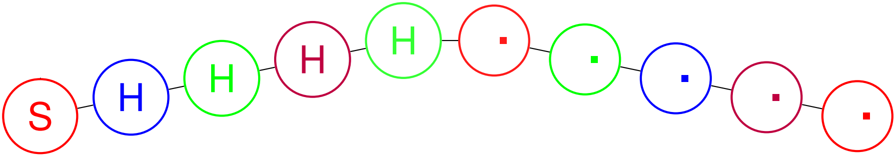
# CS 170: Algorithms




No laptops please.

Thank you ! ! ! !

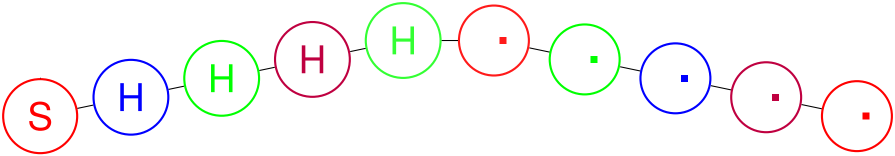
# CS 170: Algorithms




No laptops please.

Thank you ! ! ! ! !

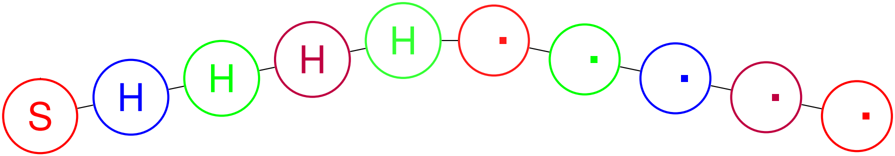
# CS 170: Algorithms




No laptops please.

Thank you ! ! ! ! ! !

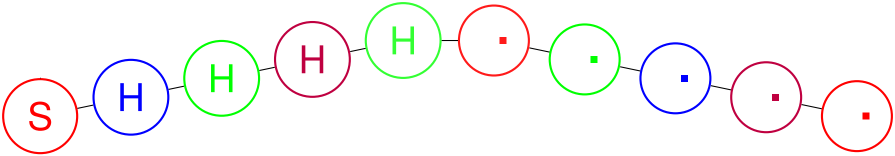
# CS 170: Algorithms




No laptops please.

Thank you ! ! ! ! ! ! !

# CS 170: Algorithms

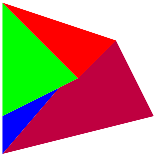


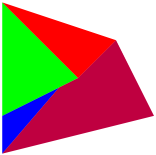

No laptops please.

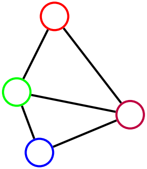
Thank you ! ! ! ! ! ! ! !

# Today

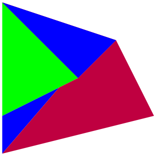
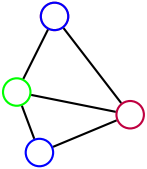
1. Graphs
2. Reachability.
3. Depth First Search



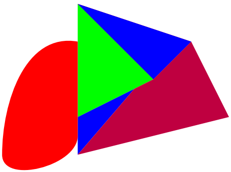


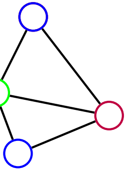


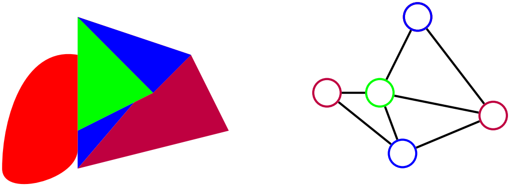
Fewer Colors?

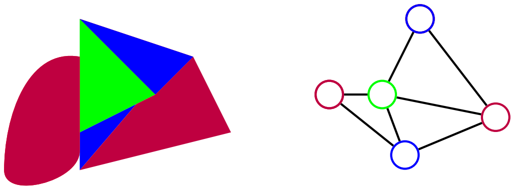
 

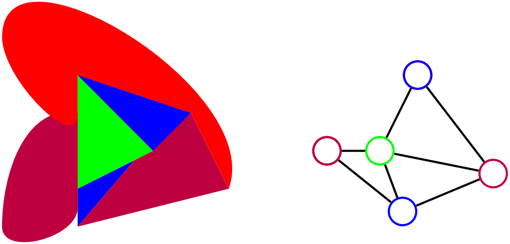
Yes! Three colors.

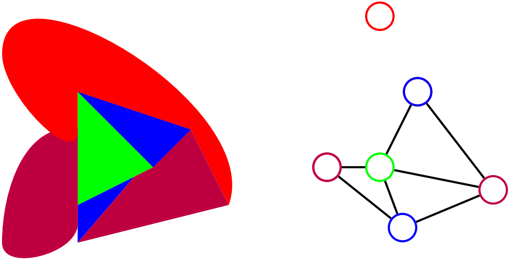


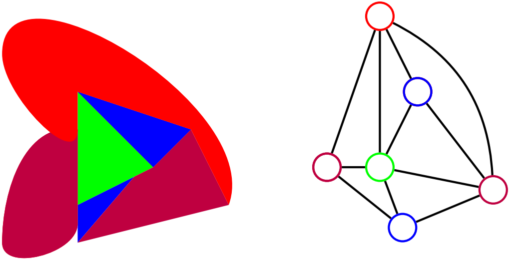


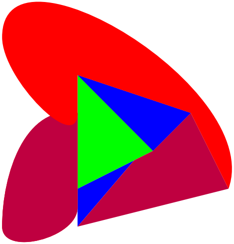


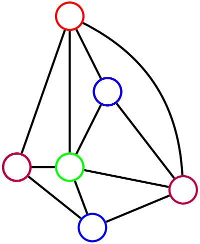




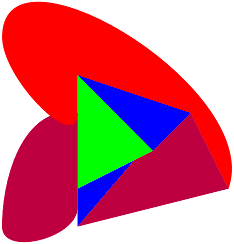


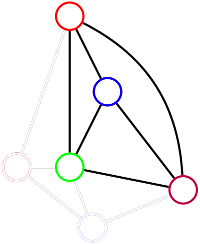


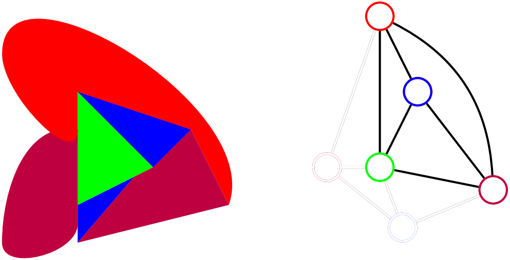




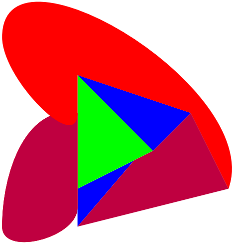
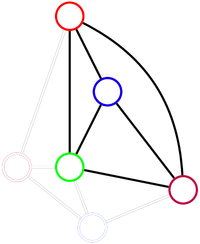
Fewer Colors?







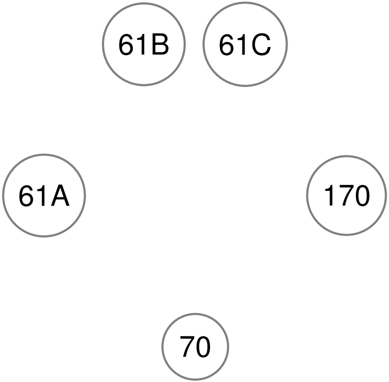
Four colors required!

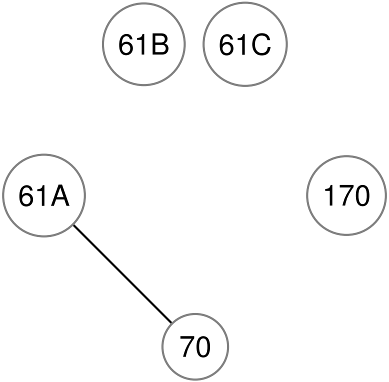
Four colors required!

Theorem: Four colors enough.

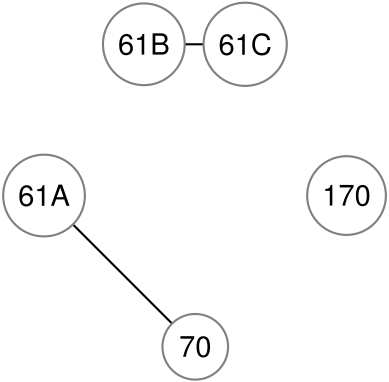
# Scheduling: coloring.



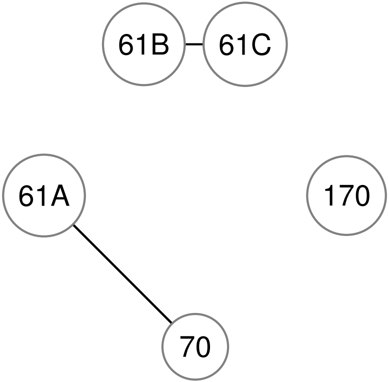
# Scheduling: coloring.



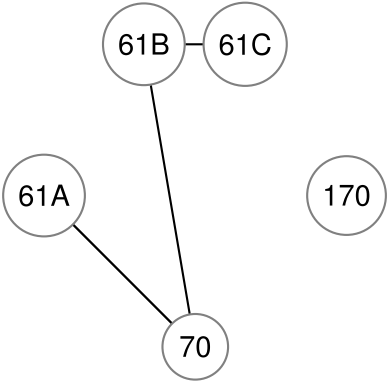
# Scheduling: coloring.



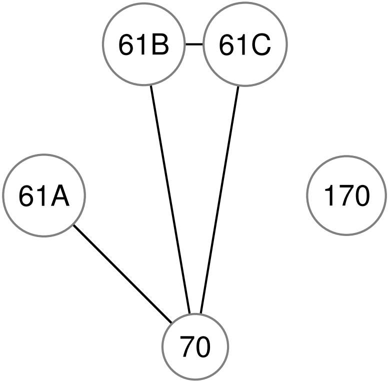
# Scheduling: coloring.



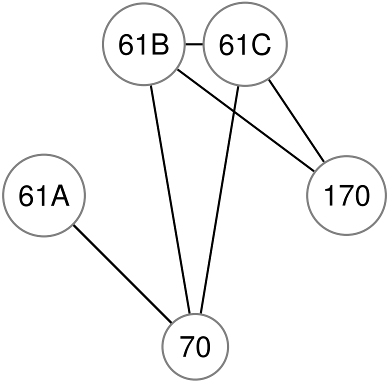
# Scheduling: coloring.



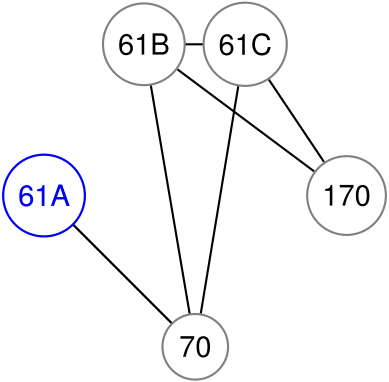
# Scheduling: coloring.



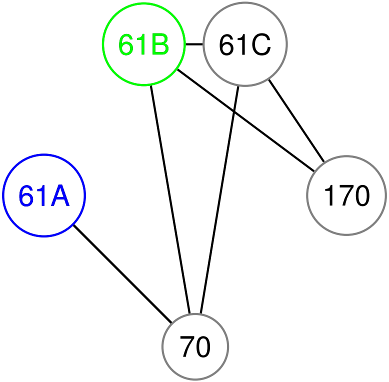
# Scheduling: coloring.



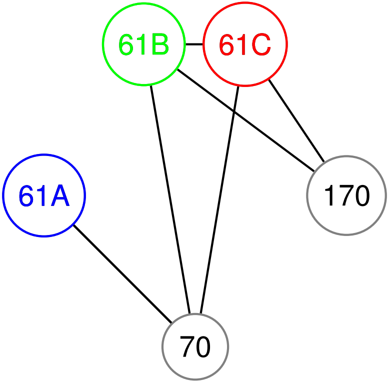
# Scheduling: coloring.



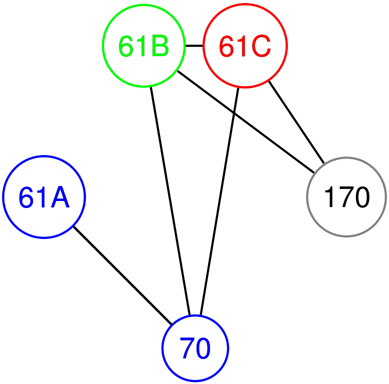
# Scheduling: coloring.



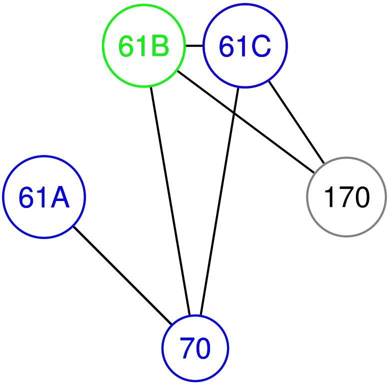
# Scheduling: coloring.



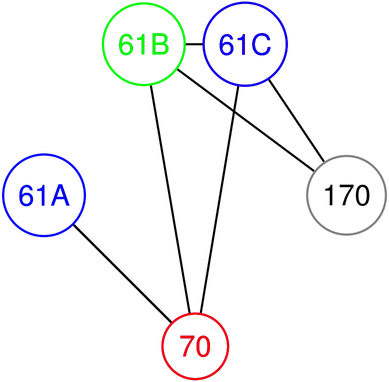
# Scheduling: coloring.



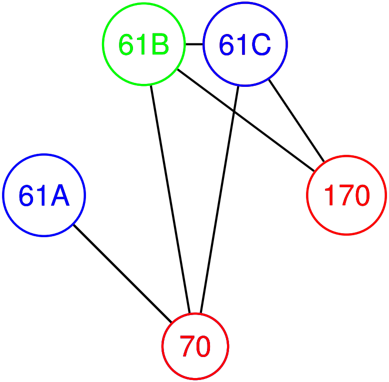
# Scheduling: coloring.



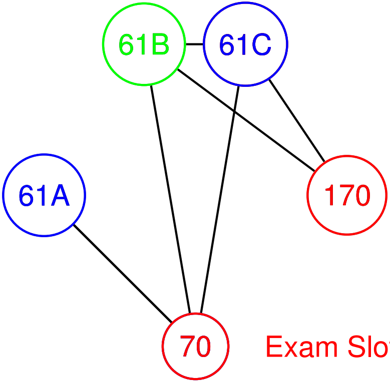
# Scheduling: coloring.



# Scheduling: coloring.



# Scheduling: coloring.



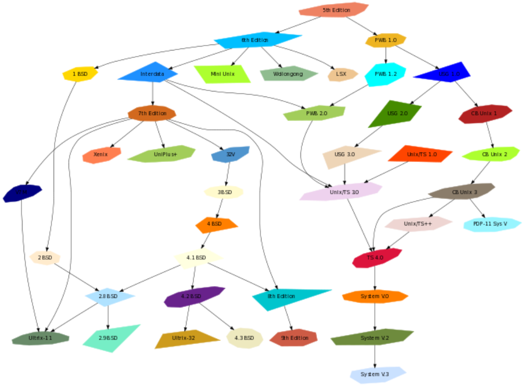

Exam Slot 1.

Exam Slot 2.

Exam Slot 3.

# Directed acyclic graphs.

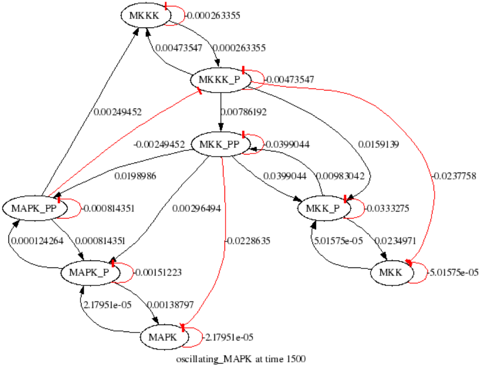
Heritage of Unix.




*Object Oriented Graphs*
*Stephen North, 3/19/93*

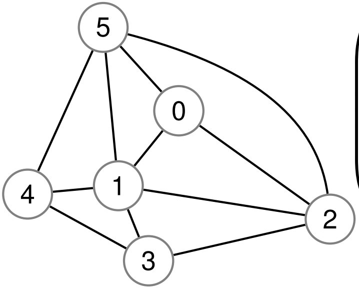
From http://www.graphviz.org/content/crazy.

# Chemical networks.




From http://www.tbi.univie.ac.at/ raim/odeSolver/doc/app.html.

Graph Implementations.



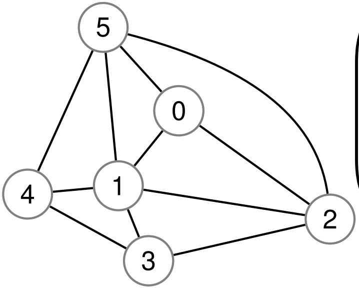

Matrix Representation.

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

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

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

Graph Implementations.




Matrix Representation.

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

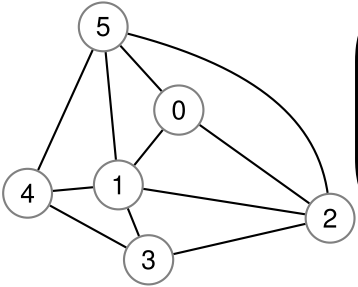
$$V = \{0,1,2,3,4,5\}$$

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

Adjacency List

$$\begin{array}{ll} 0: & 1,2,5 \\ 1: & 0,2,3,4,5 \\ 2: & 0,1,3 \\ 3: & 1,2,4 \\ 4: & 1,3,5 \\ 5: & 0,1,2,4 \end{array}$$

## Graph Implementations.




### Matrix Representation.

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

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

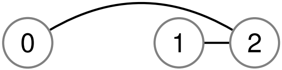
$$E = \{(0,1),(0,2),(0,5),(1,3)\ldots\}$$

### Adjacency List

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

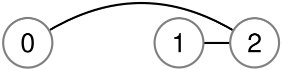
| | Matrix | Adj. List |
|---|---|---|
| Edge $(u,v)$? | $O(1)$ | $O(\|V\|)$ |
| Neighbors of $u$ | $O(\|V\|)$ | $O(d)$ |
| Space | $O(\|V\|^2)$ | $O(\|E\|)$ |

# Test your understanding..




Adjacency list of node 0?

# Test your understanding..



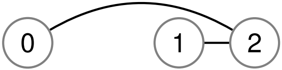

Adjacency list of node 0?

(A) $0:1$

(B) $0:1,2$

(C) $0:2$

# Test your understanding..




Adjacency list of node 0?

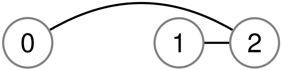
(A) $0 : 1$

(B) $0 : 1, 2$

(C) $0 : 2$

(C)

# Test your understanding..




Adjacency list of node 0?

(A) $0 : 1$

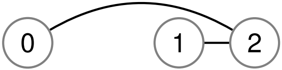
(B) $0 : 1, 2$

(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) $0:2$

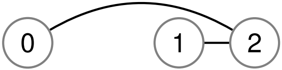
(C)

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

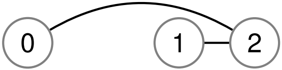
Total length of adacency lists?

(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?

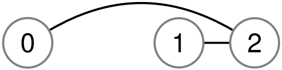
(A) 2

(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: ...

# Exploring a maze.

Theseus: ...gotta kill the minatour

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

Gives Theseus Ball of Thread and Chalk!

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

Gives Theseus Ball of Thread and Chalk!

Explore a room:

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

Gives Theseus Ball of Thread and Chalk!

Explore a room:
Mark room with chalk.

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

Gives Theseus Ball of Thread and Chalk!

Explore a room:
Mark room with chalk.
For each exit.

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

Gives Theseus Ball of Thread and Chalk!

Explore a room:
Mark room with chalk.
For each exit.
  Look through exit. If marked, next exit.

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

Gives Theseus Ball of Thread and Chalk!

Explore a room:
Mark room with chalk.
For each exit.
  Look through exit. If marked, next exit.
  Otherwise go in room unwind thread.

# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

Gives Theseus Ball of Thread and 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.

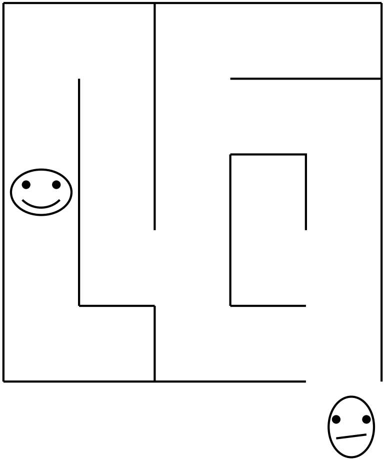
# Exploring a maze.

Theseus: ...gotta kill the minatour ..in the maze
Ariadne: he's cute..fortunately ..she's smart.

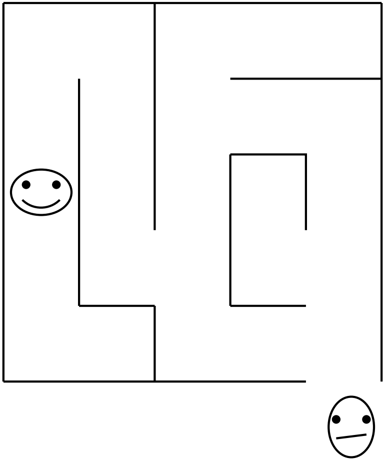
Gives Theseus Ball of Thread and Chalk!

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.

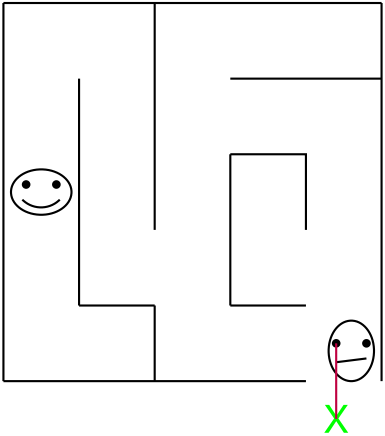
Where is the minatour?



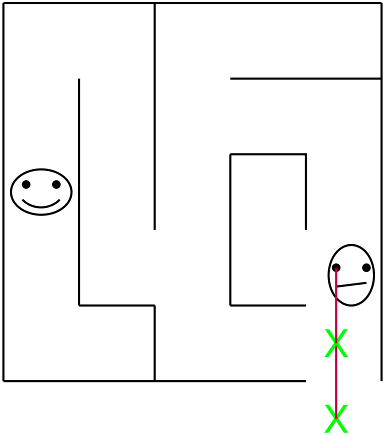
Where is the minatour?



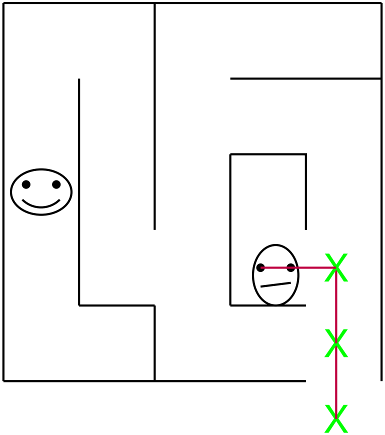
Where is the minatour?



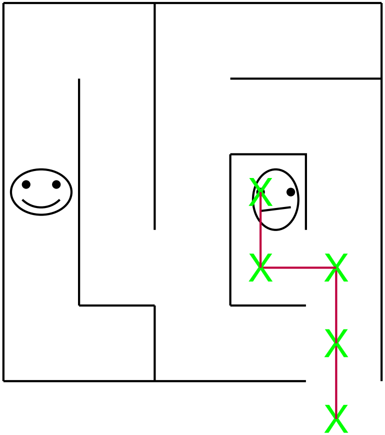
Where is the minatour?



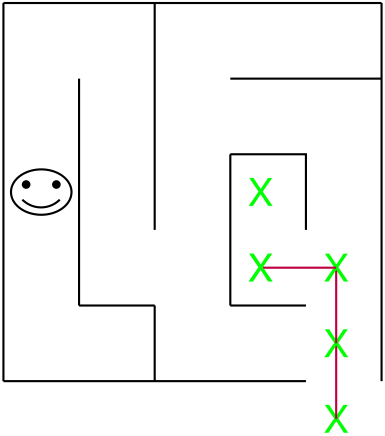
Where is the minatour?



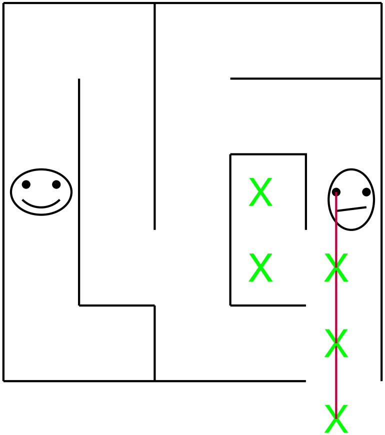
Where is the minatour?



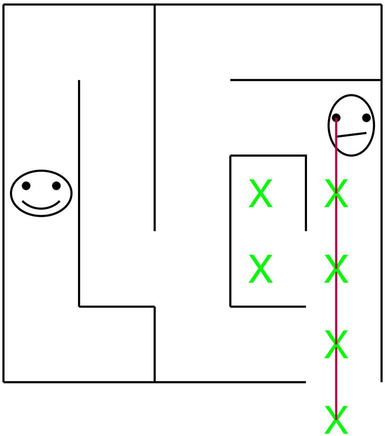
Where is the minatour?



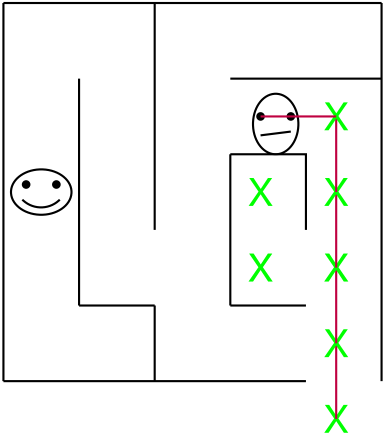
Where is the minatour?



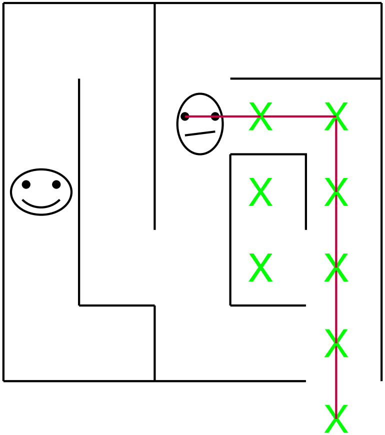
Where is the minatour?



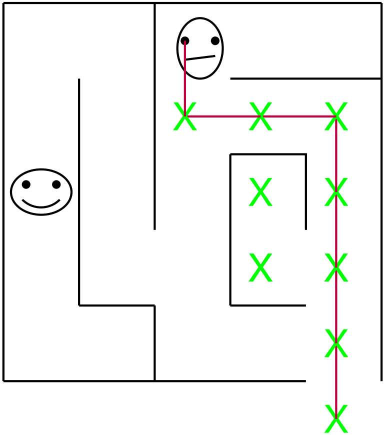
Where is the minatour?



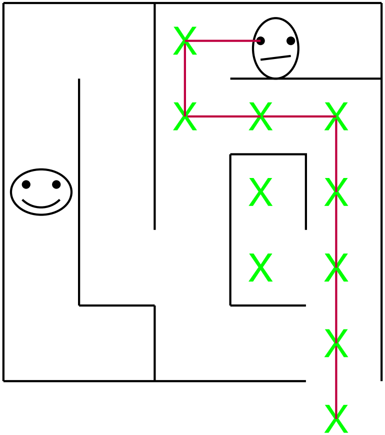
Where is the minatour?



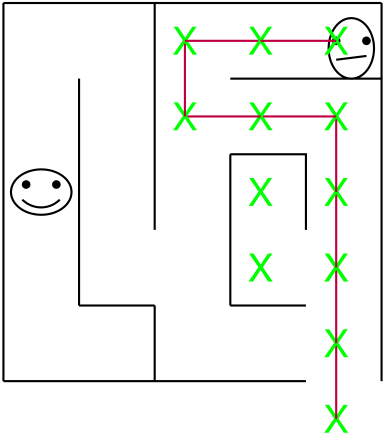
# Where is the minatour?



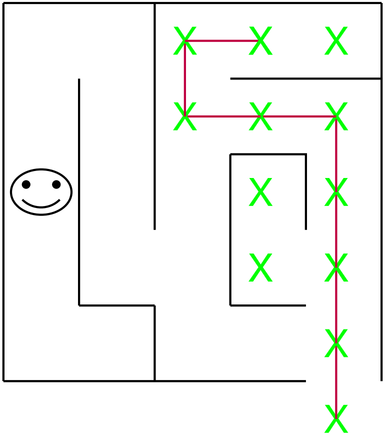
Where is the minatour?



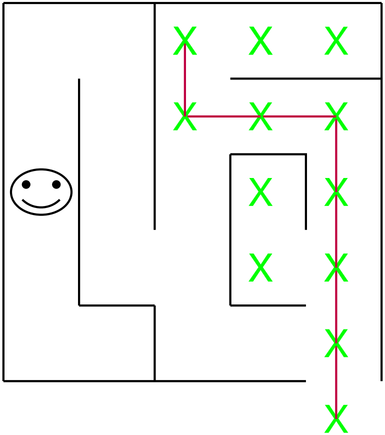
Where is the minatour?



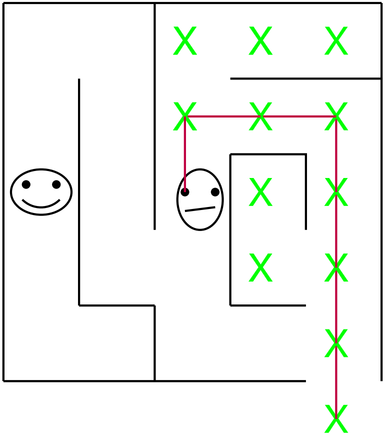
Where is the minatour?



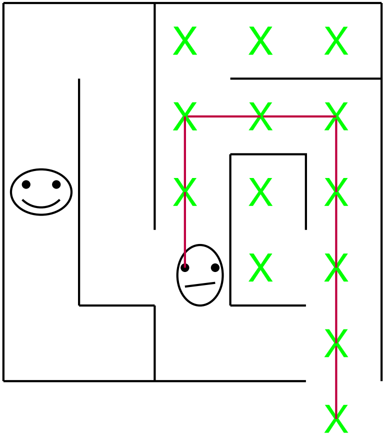
## Where is the minatour?



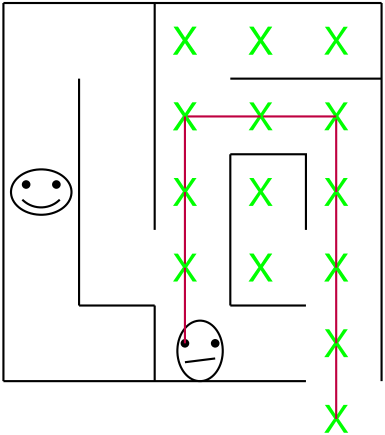
Where is the minatour?



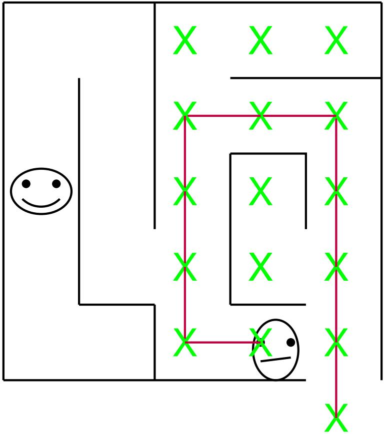
Where is the minatour?



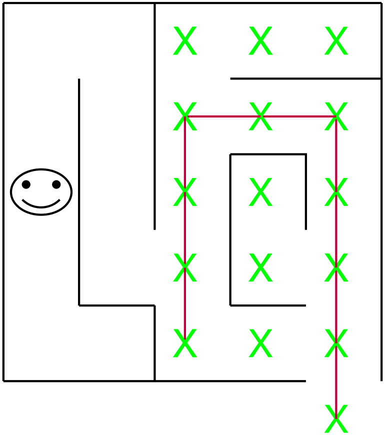
Where is the minatour?



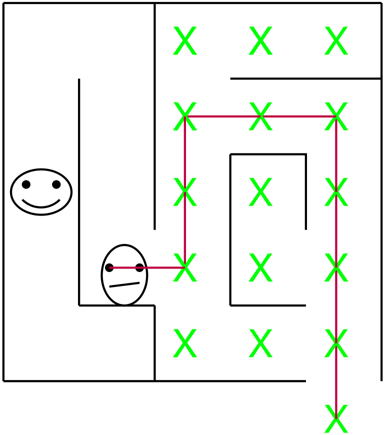
# Where is the minatour?



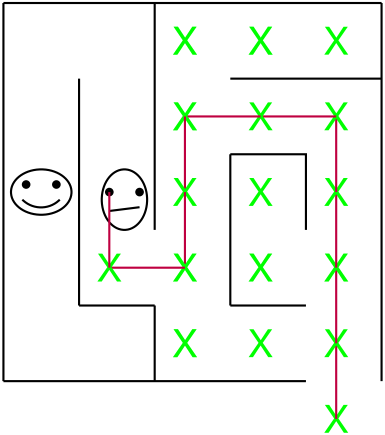
Where is the minatour?



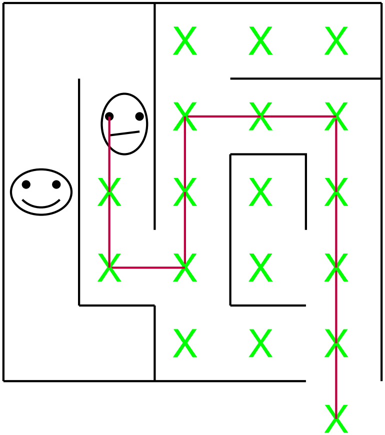
Where is the minatour?



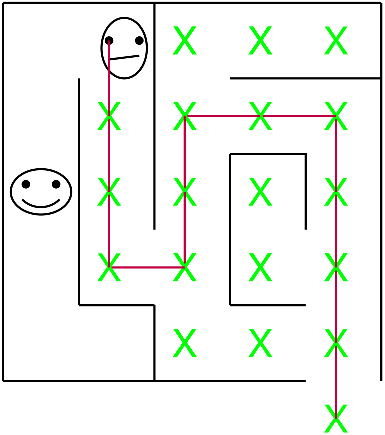
Where is the minatour?



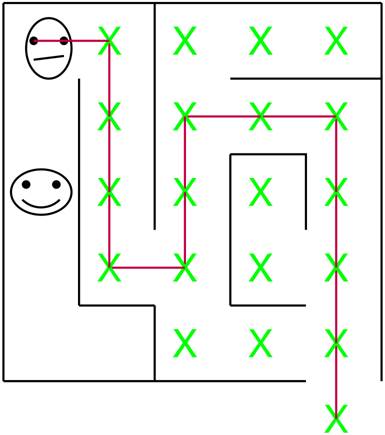
Where is the minatour?



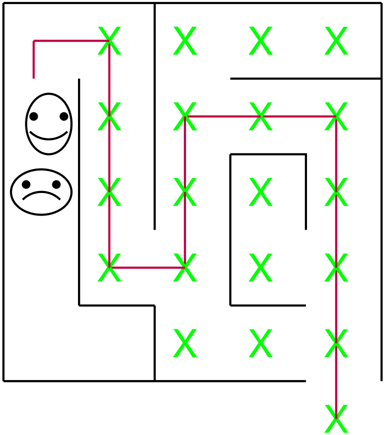
Where is the minatour?



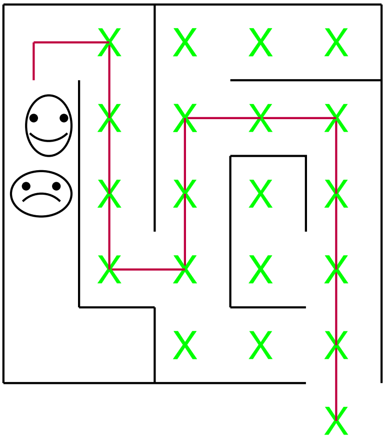
# Where is the minatour?



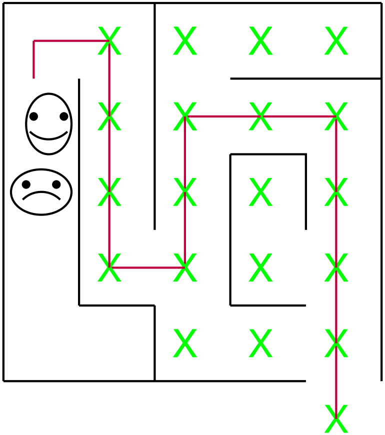
Where is the minatour?



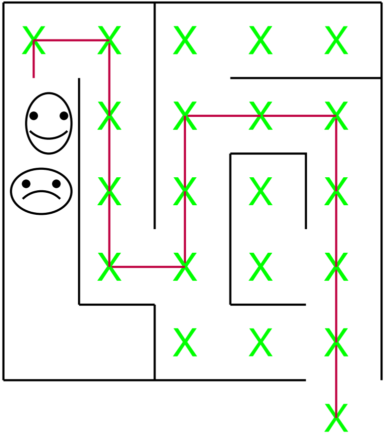
Where is the minatour?



Where is the minatour?



Where is the minatour?



# Searching

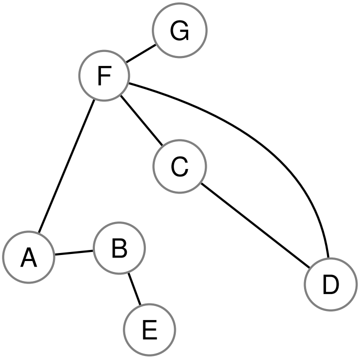
Find a minatour!

# Searching

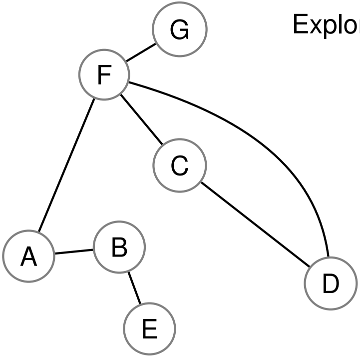
Find a minatour!

Find out which nodes are reachable from $A$.

# Explore.



# Explore.

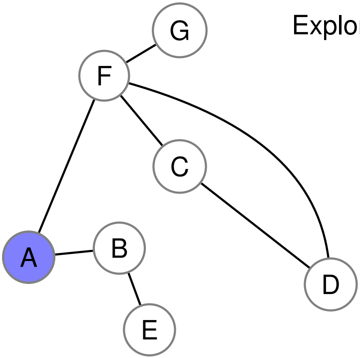



Explore(v):

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.



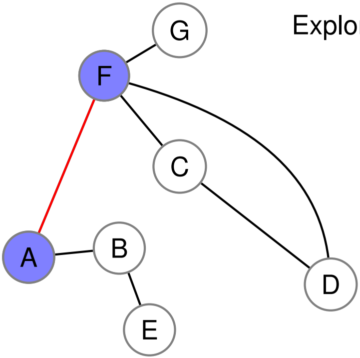

Explore(v):

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.



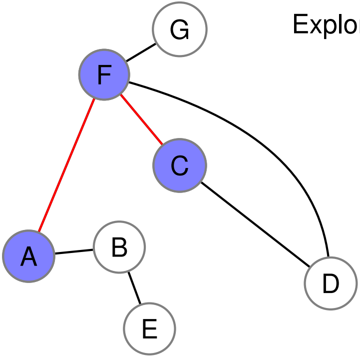

Explore(v):

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.



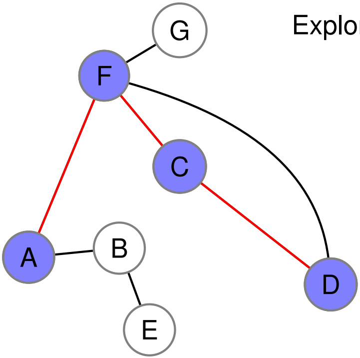

Explore(v):

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.



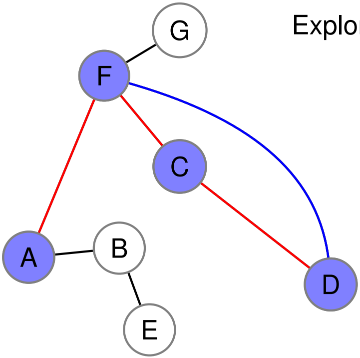

Explore(v):

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.



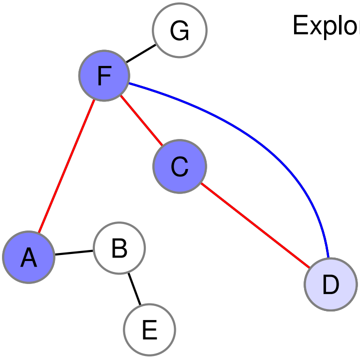

Explore(v):

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.



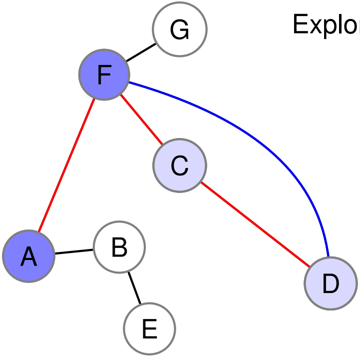

Explore(v):

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.

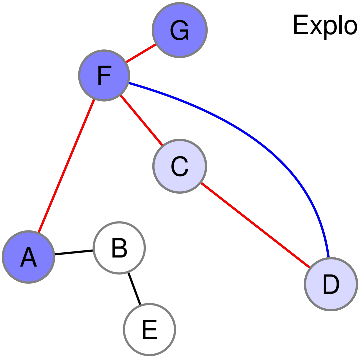



Explore(v):

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.

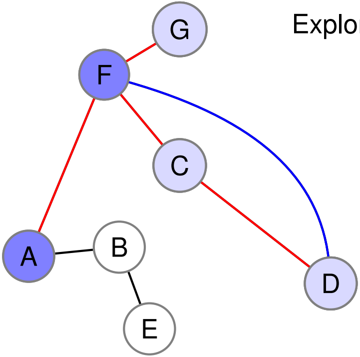



Explore(v):

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.



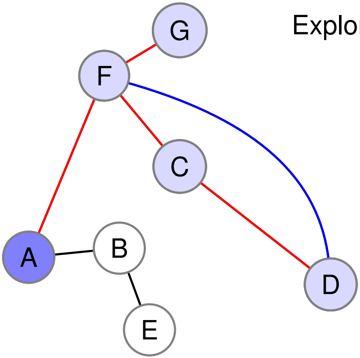

Explore(v):

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.



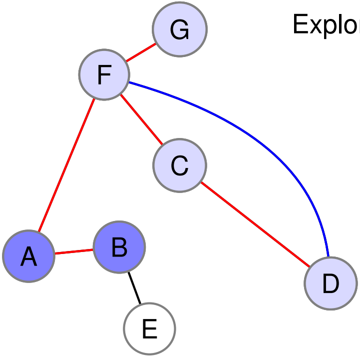

Explore(v):

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.

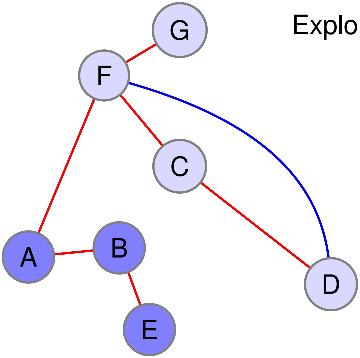



Explore(v):

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.



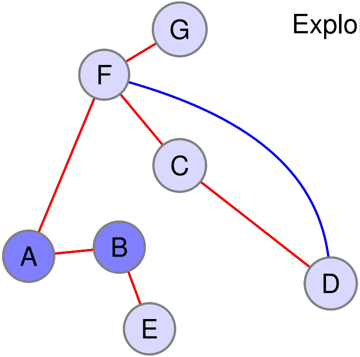

Explore(v):

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.

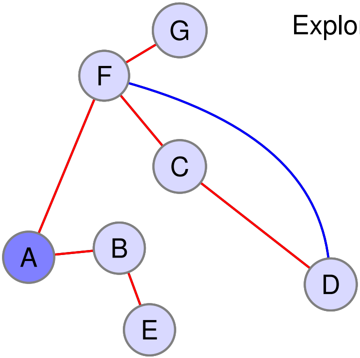



Explore(v):

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.



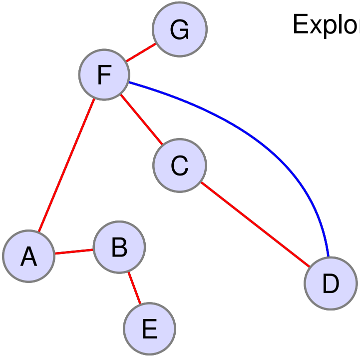

Explore(v):

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.

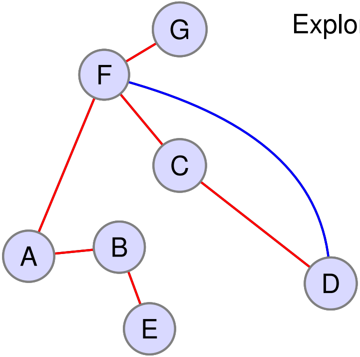



Explore(v):

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)

# 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):**

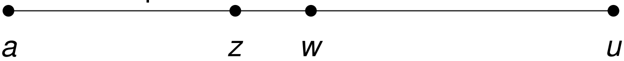
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.     if not visited[w]: Explore(w)

**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):**

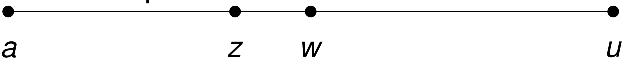
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.     if not visited[w]: Explore(w)

**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):**

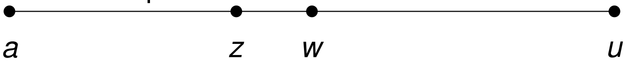
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.     if not visited[w]: Explore(w)

**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):**

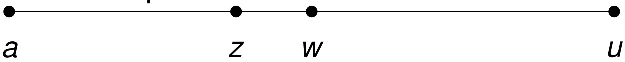
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w)

**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($w$)!

# Correctness.

**Explore(v):**

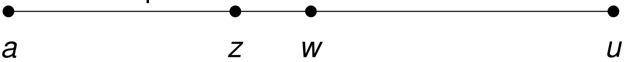
1. Set visited[v] := **true**.
2. For each edge (v,w) in E
3.     if not visited[w]: Explore(w)

**Property:**

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

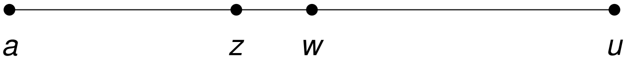
Only: when $u$ visited.
stack contains nodes in a path from $a$ to $u$.

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



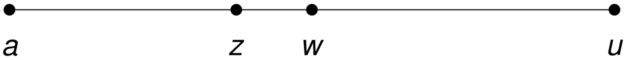

$z$ is explored. $w$ is not!
Explore ($z$) would explore($w$)! Contradiction.

# Proof was induction.




Property: Every node with a path of length $k$ or less is reached.

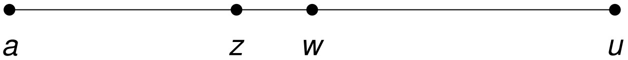
# Proof was induction.




Property: Every node with a path of length $k$ or less is reached.

Induction by Contradiction.

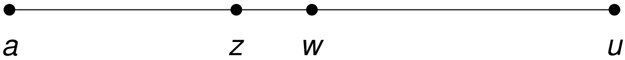
# Proof was induction.




Property: Every node with a path of length $k$ or less is reached.

Induction by Contradiction.
Find smallest $k$ (path length) where property doesn't hold.

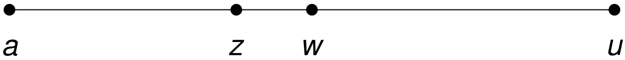
# Proof was induction.




Property: Every node with a path of length $k$ or less is reached.

Induction by Contradiction.
Find smallest $k$ (path length) where property doesn't hold.
It does hold.

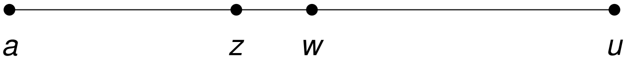
# Proof was induction.




Property: Every node with a path of length $k$ or less is reached.

Induction by Contradiction.
Find smallest $k$ (path length) where property doesn't hold.
It does hold.
No smallest $k$ where it fails.

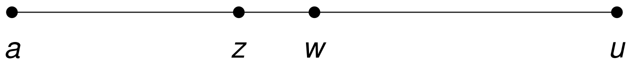
# Proof was induction.




Property: Every node with a path of length $k$ or less is reached.

Induction by Contradiction.
Find smallest $k$ (path length) where property doesn't hold.
It does hold.
No smallest $k$ where it fails.
Must hold for every $k$.

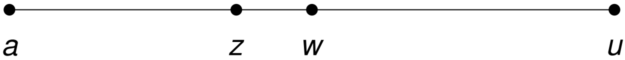
# Proof was induction.




Property: Every node with a path of length $k$ or less is reached.

Induction by Contradiction.
Find smallest $k$ (path length) where property doesn't hold.
It does hold.
No smallest $k$ where it fails.
Must hold for every $k$.
Done joe!!!

# Proof was induction.




Property: Every node with a path of length $k$ or less is reached.

Induction by Contradiction.
Find smallest $k$ (path length) where property doesn't hold.
It does hold.
No smallest $k$ where it fails.
Must hold for every $k$.
Done joe!!! or $\square$.

# 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)**

1: For each node $u$,

2: visited[$u$] = **false**.

3: For each node $u$,

# 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(**$u$**)**

# 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)**

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”.

# 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)**

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!

# 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 cc := 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 cc := 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):**

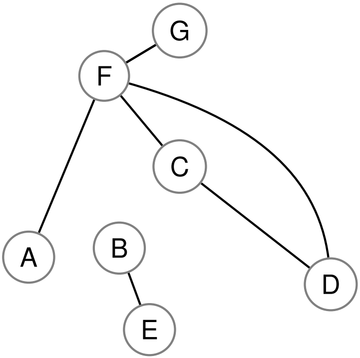
1. Set cc[v] := ccnum.

**DFS(G):**     0. Set cc := 0.

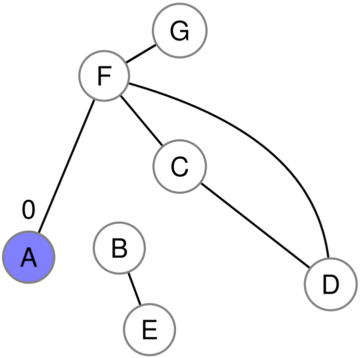
1. for each v in V:
2.   if not visited[v]:
3.       explore(v)
4.       ccnum = ccnum+1

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

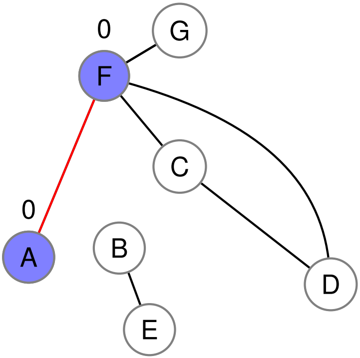
# Connected Components.



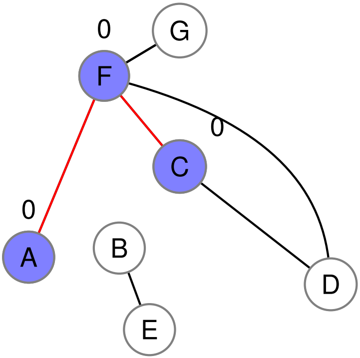
# Connected Components.



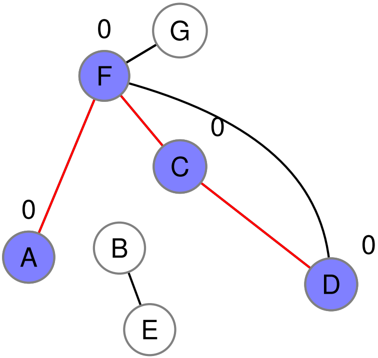
# Connected Components.



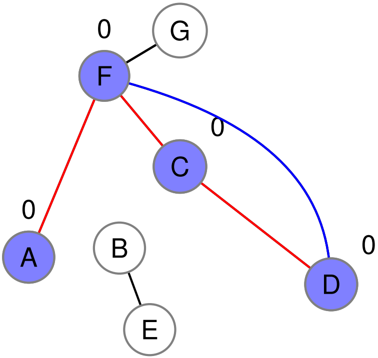
# Connected Components.



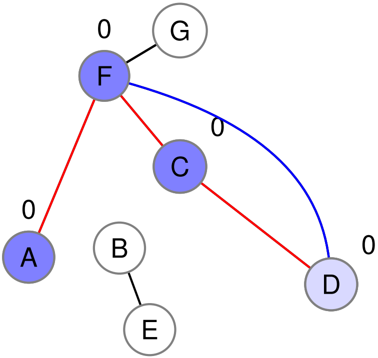
# Connected Components.



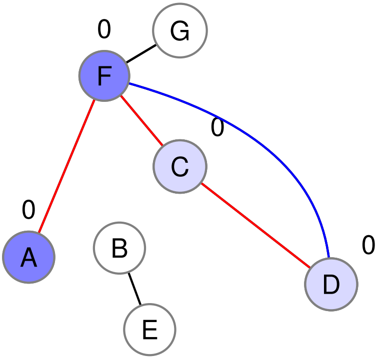
# Connected Components.



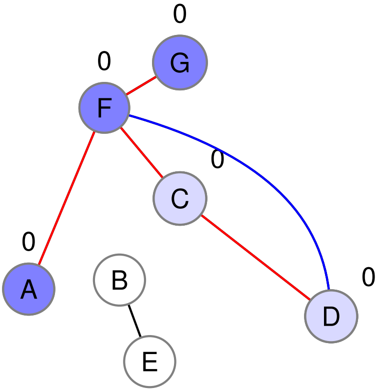
# Connected Components.



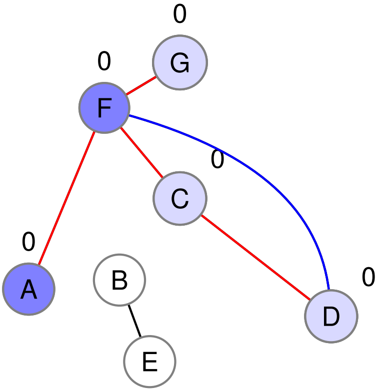
# Connected Components.



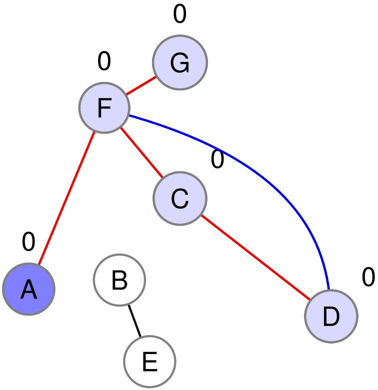
# Connected Components.



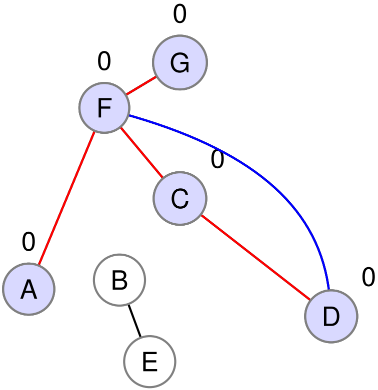
# Connected Components.



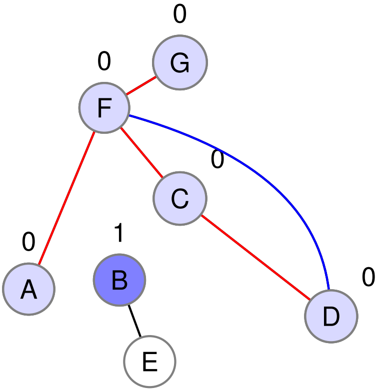
# Connected Components.



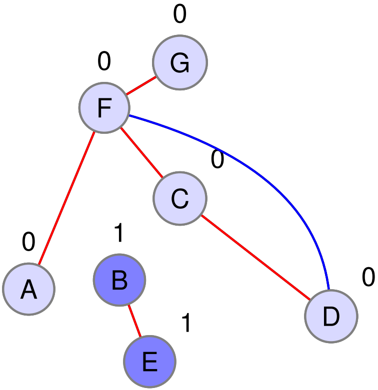
# Connected Components.



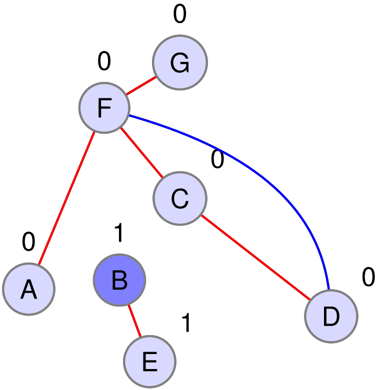
# Connected Components.



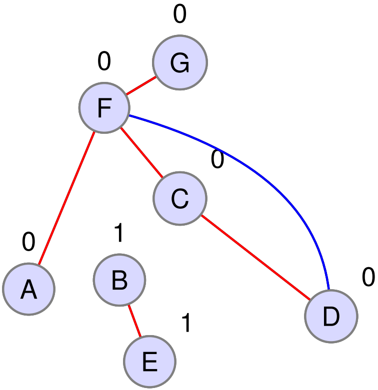
# Connected Components.



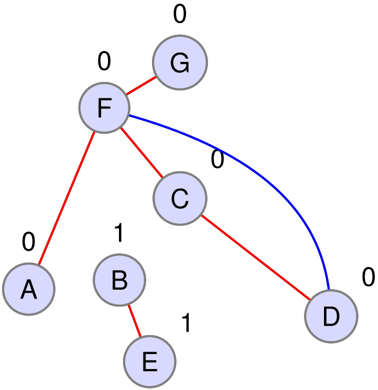
# Connected Components.



# 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.

$\vdots$

Clock: goes up to 2 times number of tree edges.

# 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 tree edges.
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 tree edges.
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 tree edges.
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 tree edges.
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 tree edges.
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.

$\vdots$

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

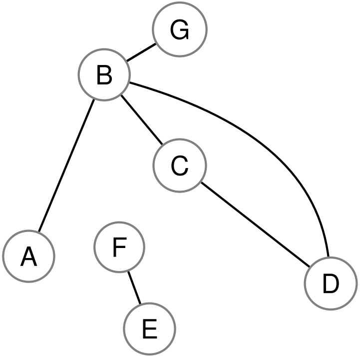
**Property:** For any two nodes, $u$ and $v$, $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained in other.

Interval is "clock interval on stack."

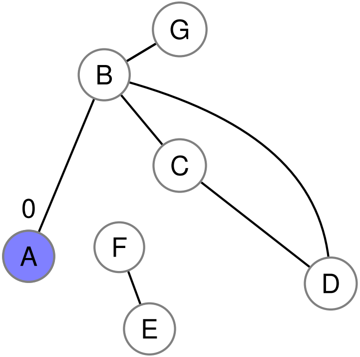
Either both on stack at some point (contained) or not (disjoint.)

Let's just watch it work!

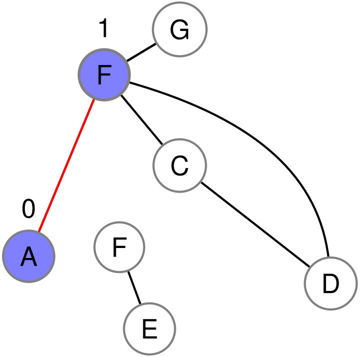
# Example: Pre/Post numbering.



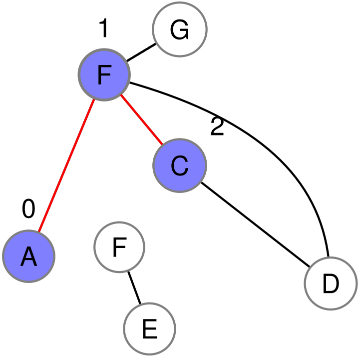
# Example: Pre/Post numbering.



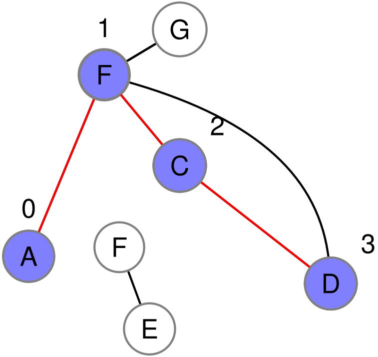
# Example: Pre/Post numbering.



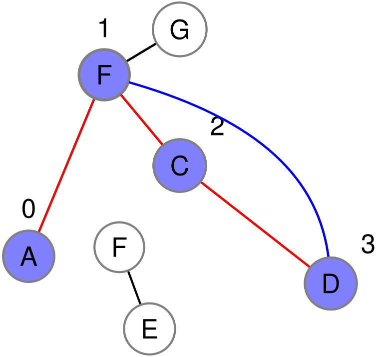
# Example: Pre/Post numbering.



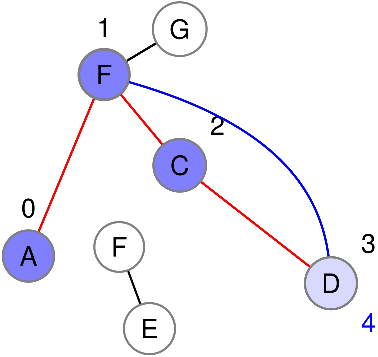
# Example: Pre/Post numbering.



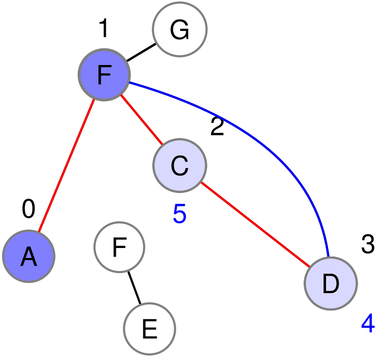
# Example: Pre/Post numbering.



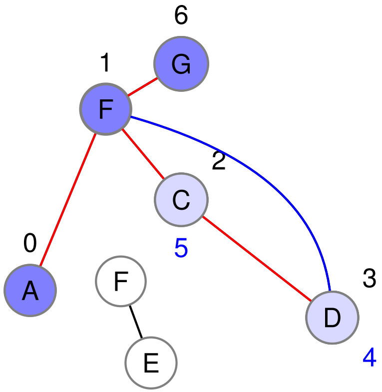
# Example: Pre/Post numbering.



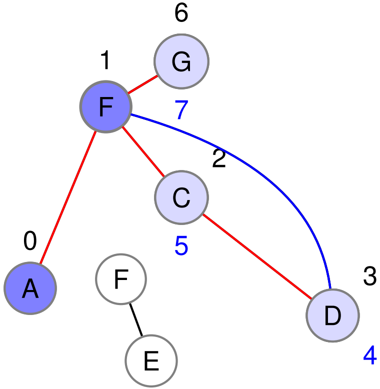
# Example: Pre/Post numbering.



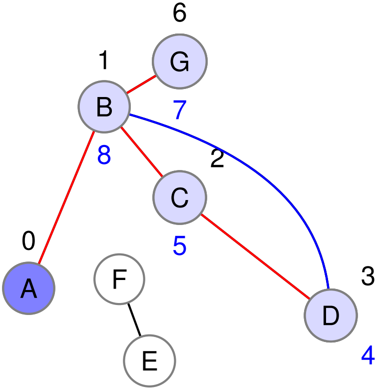
# Example: Pre/Post numbering.



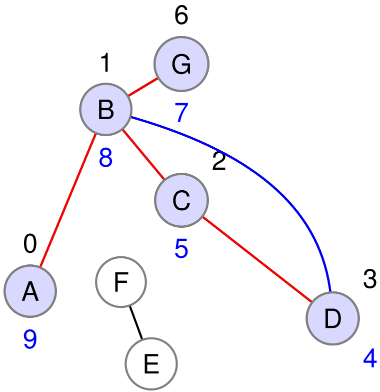
# Example: Pre/Post numbering.



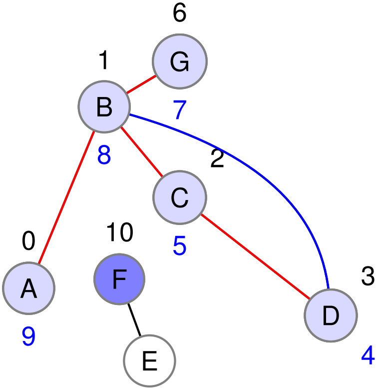
# Example: Pre/Post numbering.



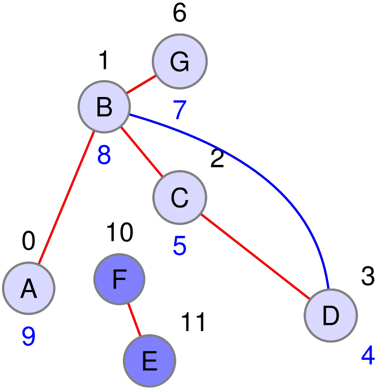
# Example: Pre/Post numbering.



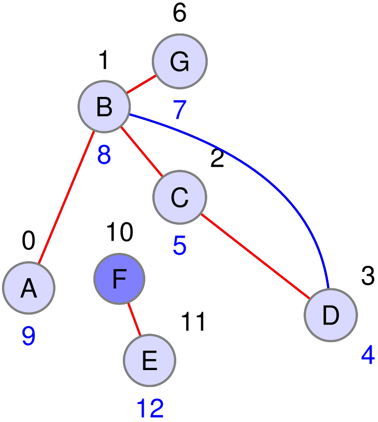
# Example: Pre/Post numbering.



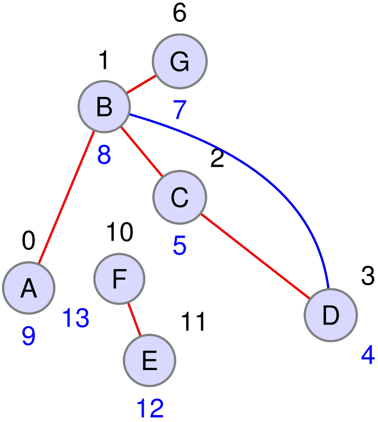
# Example: Pre/Post numbering.



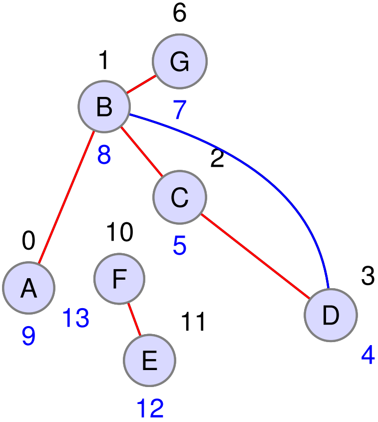
# Example: Pre/Post numbering.



# Example: Pre/Post numbering.



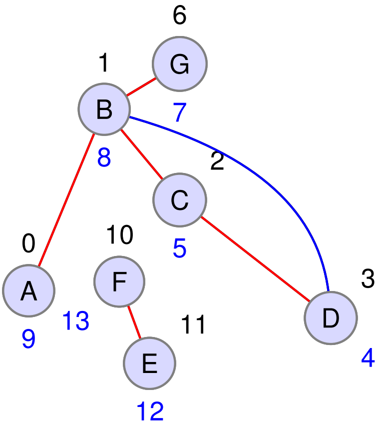
# Example: Pre/Post numbering.




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

$u$ on stack before $v$.

# Example: Pre/Post numbering.



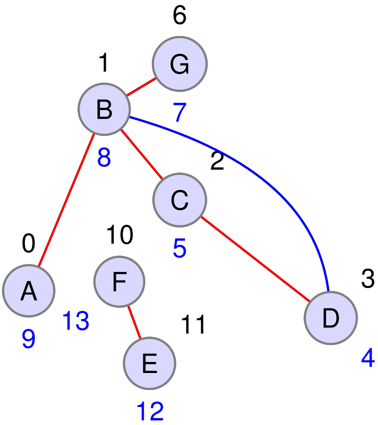

Edge $(u, v)$ is tree edge iff $[pre[v], post[v]] \in [pre[u], post[u]]$.

*u* on stack before *v*.

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

*v* on stack when *v* on stack. Path from *v* to *u*! Cycle!

# Example: Pre/Post numbering.




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

$u$ on stack before $v$.

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

$v$ on stack when $v$ on stack. Path from $v$ to $u$! Cycle!

No edge between $u$ and $v$ if disjoint intervals.

# On Friday/Monday

# On Friday/Monday

Christos Papadimitriou will lecture!