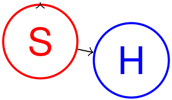
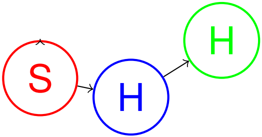
CS 170: Algorithms

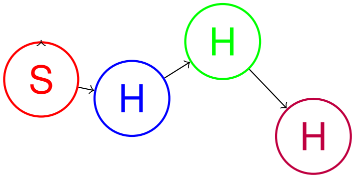
CS 170: Algorithms

CS 170: Algorithms

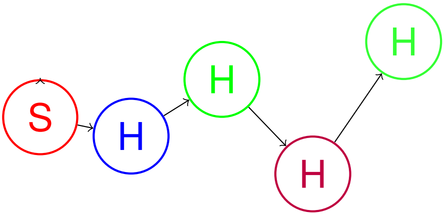


CS 170: Algorithms

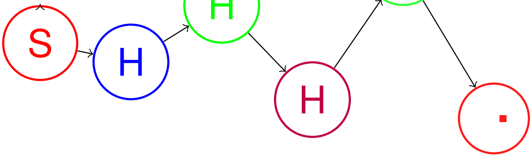



S
H
H
H

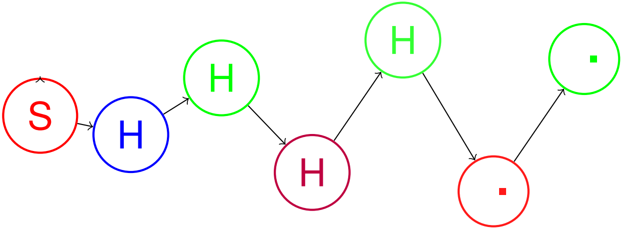
CS 170: Algorithms



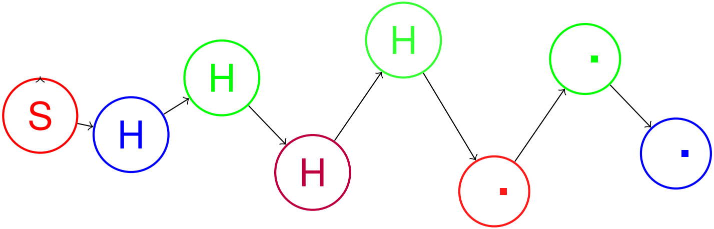
CS 170: Algorithms



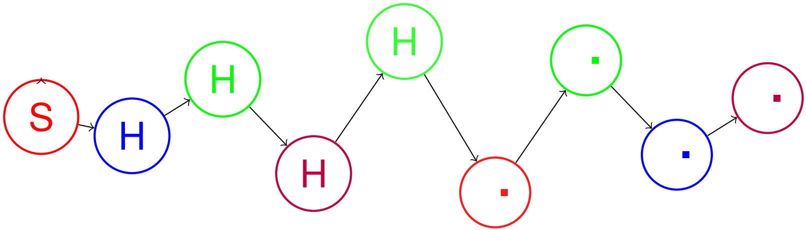
CS 170: Algorithms



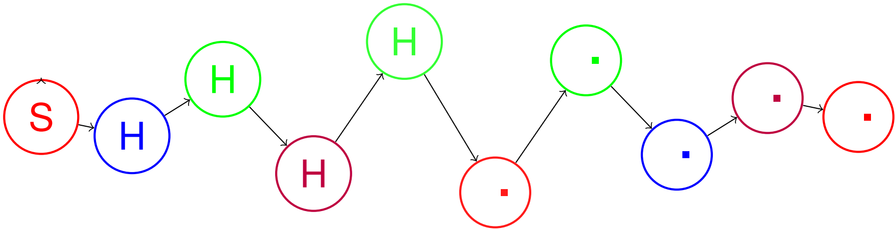
CS 170: Algorithms



CS 170: Algorithms



CS 170: Algorithms



Lecture in a Minute

Depth First Search.
   Call explore until explore the whole graph.
   Connected Components.
   Tree/back edges.
   Back edge $\iff$ cycle

Lecture in a Minute

Depth First Search.
- Call explore until explore the whole graph.
- Connected Components.
- Tree/back edges.
  - Back edge $\iff$ cycle

Pre/Post Ordering.
- Interval of time "on stack".
- Quick cycle test.

Lecture in a Minute

Depth First Search.
- Call explore until explore the whole graph.
- Connected Components.
- Tree/back edges.
  - Back edge $\iff$ cycle

Pre/Post Ordering.
- Interval of time “on stack”.
- Quick cycle test.

Directed Graphs.
- Tree/Back/Forward/Cross edges.
- From pre/post!
- Back Edge $\iff$ cycle!

Lecture in a Minute

Depth First Search.
  Call explore until explore the whole graph.
  Connected Components.
  Tree/back edges.
  Back edge $\iff$ cycle

Pre/Post Ordering.
  Interval of time "on stack".
  Quick cycle test.

Directed Graphs.
  Tree/Back/Forward/Cross edges.
  From pre/post!
  Back Edge $\iff$ cycle!

Topological Sort.
  Alg 1:Inverse Post order number.

Lecture in a Minute

Depth First Search.
  Call explore until explore the whole graph.
  Connected Components.
  Tree/back edges.
   Back edge $\iff$ cycle

Pre/Post Ordering.
  Interval of time "on stack".
  Quick cycle test.

Directed Graphs.
  Tree/Back/Forward/Cross edges.
  From pre/post!
  Back Edge $\iff$ cycle!

Topological Sort.
  Alg 1:Inverse Post order number.
    Inverse order of "stack" pop.

Lecture in a Minute

Depth First Search.
  Call explore until explore the whole graph.
  Connected Components.
  Tree/back edges.
   Back edge $\iff$ cycle

Pre/Post Ordering.
  Interval of time "on stack".
  Quick cycle test.

Directed Graphs.
  Tree/Back/Forward/Cross edges.
  From pre/post!
  Back Edge $\iff$ cycle!

Topological Sort.
  Alg 1:Inverse Post order number.
    Inverse order of "stack" pop.
  Alg 2: Peeling off sources.

Explore.

Graph $G = (V, E)$.

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation?

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation? $O(|V| + |E|)$.

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation? $O(|V| + |E|)$.

Explore(v):

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

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation? $O(|V| + |E|)$.

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.

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation? $O(|V| + |E|)$.

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.

Idea:
Reachable means there is a path,

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation? $O(|V| + |E|)$.

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.

Idea:
Reachable means there is a path,
and there is no first unexplored node

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation? $O(|V| + |E|)$.

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.

Idea:
Reachable means there is a path,
and there is no first unexplored node
since the previous node would explore it.

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation? $O(|V| + |E|)$.

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.

Idea:
Reachable means there is a path,
and there is no first unexplored node
since the previous node would explore it.

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

Explore.

Graph $G = (V, E)$.

Size of adjacency list representation? $O(|V| + |E|)$.

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.

Idea:
Reachable means there is a path,
and there is no first unexplored node
since the previous node would explore it.

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

The time is proportional to total size of the adjacency lists.

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)

```
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)

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.

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)

Connected Components.

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.

Connected Components.

explore(v):

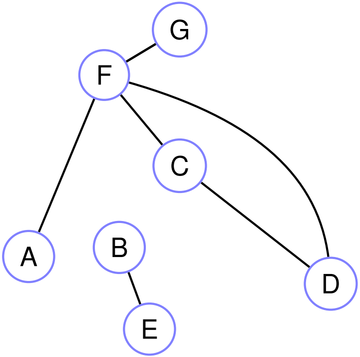
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

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



Connected Components.

explore(v):

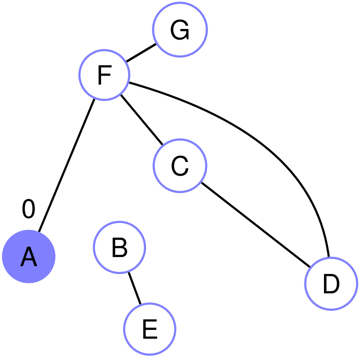
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4. if not visited[w]: explore(w).
5. postvisit(v)

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



Connected Components.

explore(v):

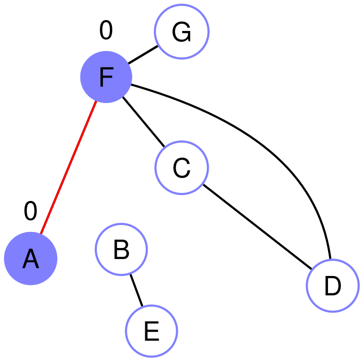
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

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

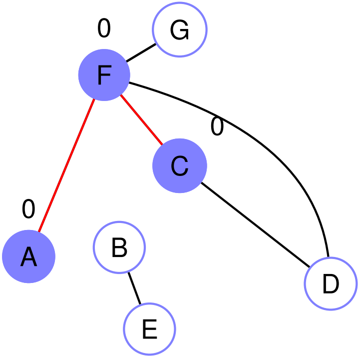


Connected Components.

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

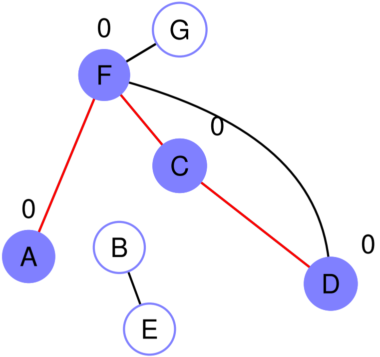


Connected Components.

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

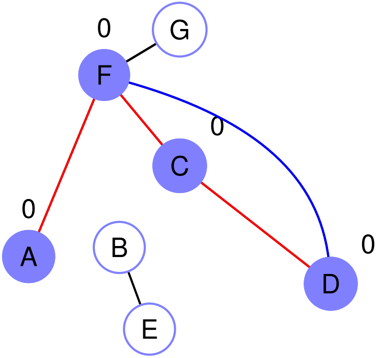


Connected Components.

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



Connected Components.

explore(v):

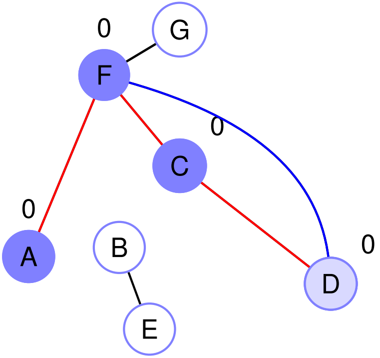
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4. if not visited[w]: explore(w).
5. postvisit(v)

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

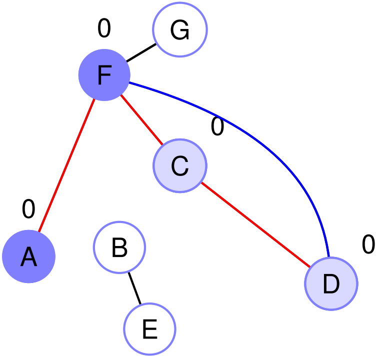


Connected Components.

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



Connected Components.

explore(v):

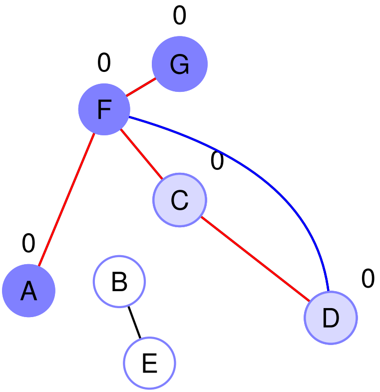
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4. if not visited[w]: explore(w).
5. postvisit(v)

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

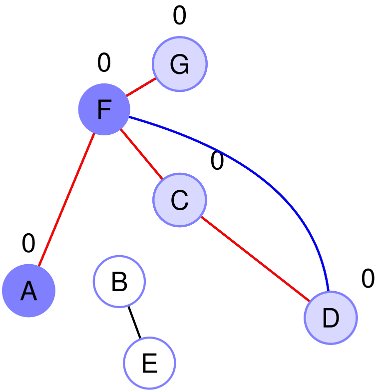


Connected Components.

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



Connected Components.

explore(v):

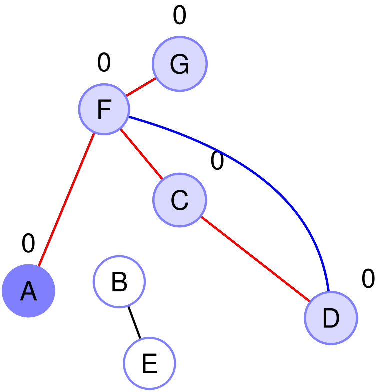
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

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

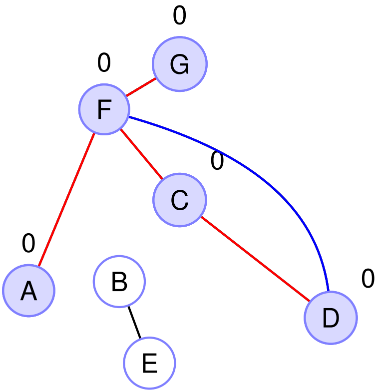


Connected Components.

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



Connected Components.

explore(v):

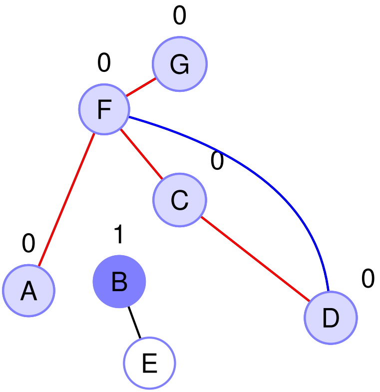
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4. if not visited[w]: explore(w).
5. postvisit(v)

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



Connected Components.

explore(v):

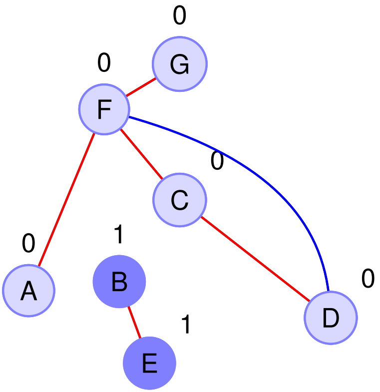
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

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



Connected Components.

explore(v):

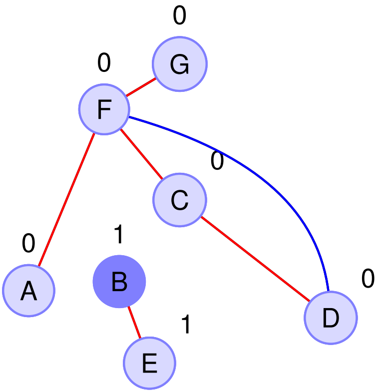
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

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



Connected Components.

explore(v):

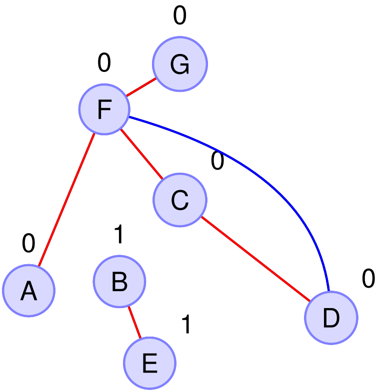
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

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



Connected Components.

explore(v):

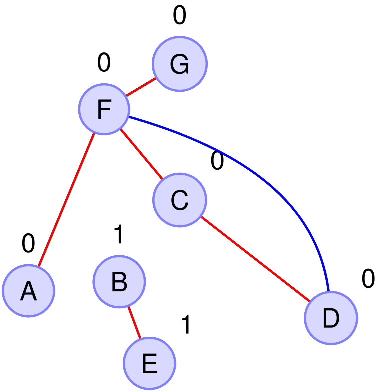
1. Set visited[v] := **true**.
2. previsit(v)
3. For each edge (v,w) in E
4.   if not visited[w]: explore(w).
5. postvisit(v)

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



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

DFS(G):

0. Set clock := 0.

. . .

Postvisit(v):

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

Introspection: pre/post.

Previsit(v):

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

DFS(G):

0. Set clock := 0.

$\ldots$

Clock: goes up to

Postvisit(v):

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

Introspection: pre/post.

Previsit(v):

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

DFS(G):

0. Set clock := 0.

   . . .

Postvisit(v):

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

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

Introspection: pre/post.

Previsit(v):

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

DFS(G):

0. Set clock := 0.

   ...

Postvisit(v):

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

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

DFS(G):

0. Set clock := 0.

 . . .

Postvisit(v):

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

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

DFS(G):

0. Set clock := 0.

...

Postvisit(v):

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

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.

$\ldots$

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):

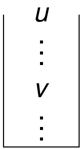
0. Set clock := 0.

. . .

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.

$\dots$

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.”

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

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):

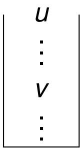
0. Set clock := 0.

   ...

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.

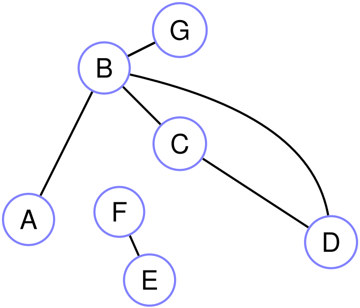
Interval is "clock interval on stack."



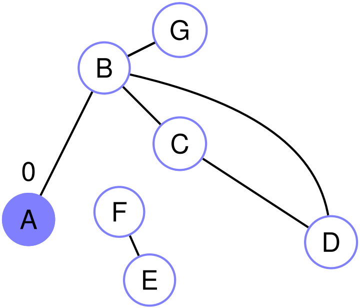
Either both on stack at some point (contained) or not (disjoint.)

Let's just watch it work!

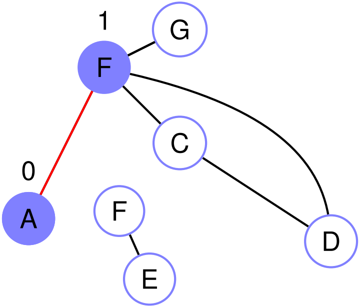
Example: Pre/Post numbering.



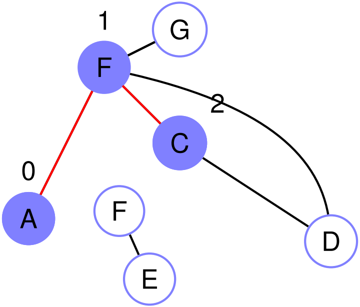
Example: Pre/Post numbering.



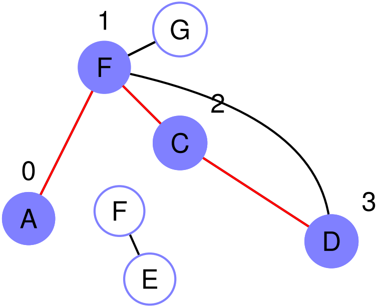
Example: Pre/Post numbering.



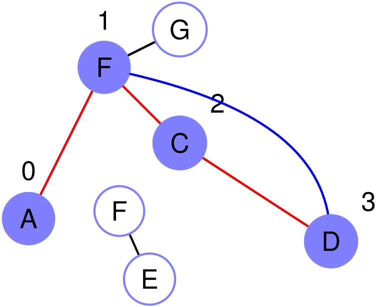
Example: Pre/Post numbering.



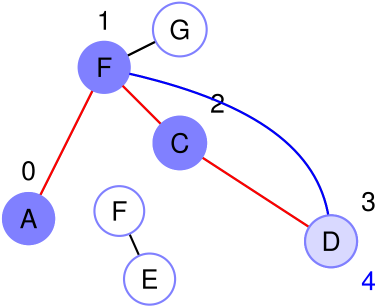
Example: Pre/Post numbering.



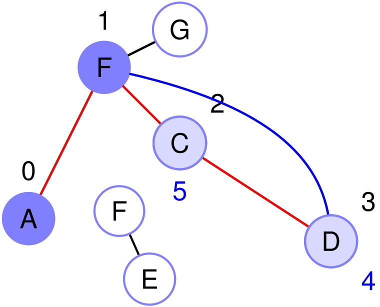
Example: Pre/Post numbering.



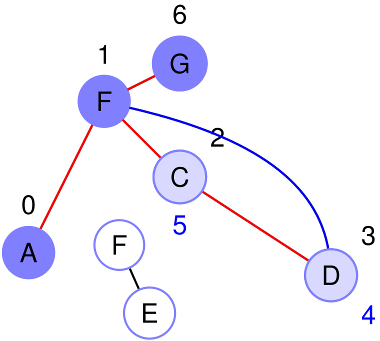
Example: Pre/Post numbering.



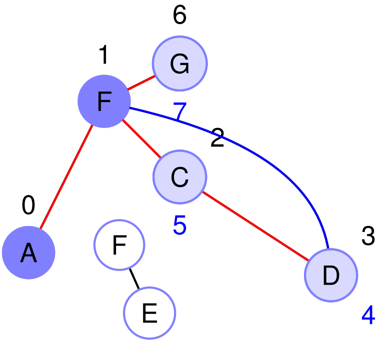
Example: Pre/Post numbering.



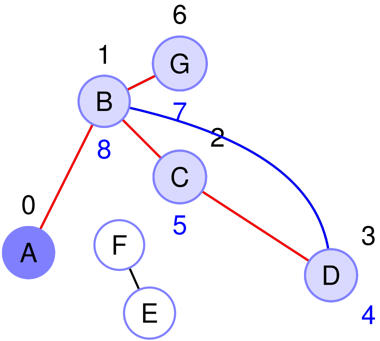
Example: Pre/Post numbering.



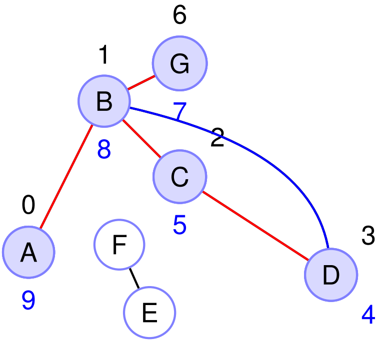
Example: Pre/Post numbering.



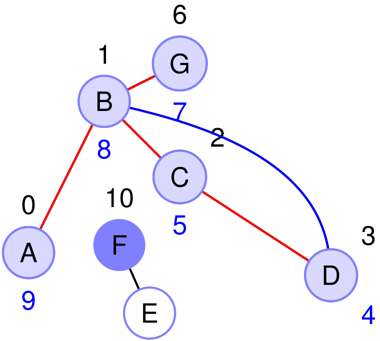
Example: Pre/Post numbering.



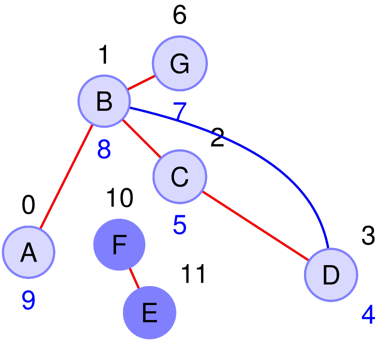
Example: Pre/Post numbering.



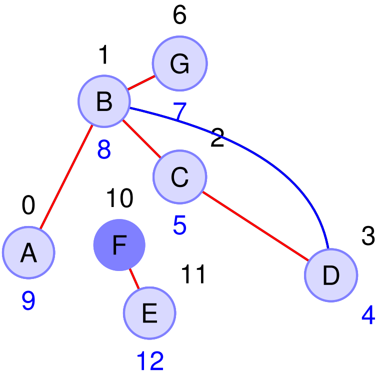
Example: Pre/Post numbering.



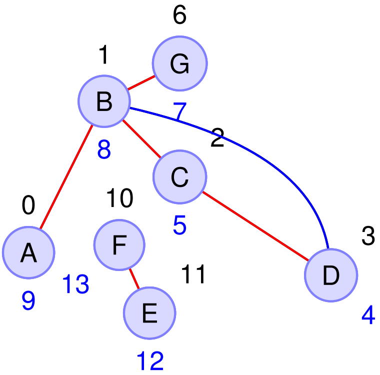
Example: Pre/Post numbering.



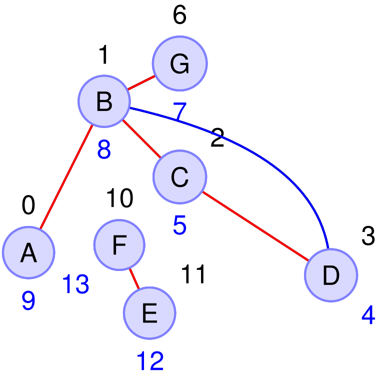
Example: Pre/Post numbering.



Example: Pre/Post numbering.



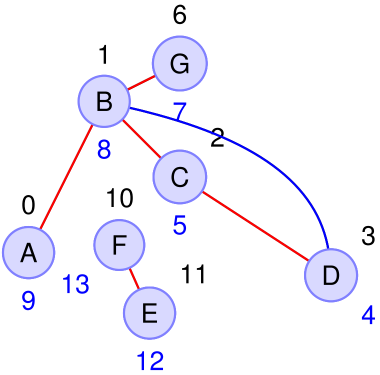
Example: Pre/Post numbering.



Explored edge (u, v) first from u.

Tree edge iff $[pre[v], post[v]] \in [pre[u], post[u]]$.
u on stack before v.

Example: Pre/Post numbering.

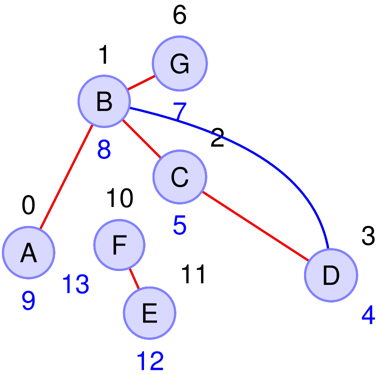


Explored edge (u,v) first from u.

Tree edge iff $[pre[v], post[v]] \in [pre[u], post[u]]$.
u on stack before v.

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.



Explored edge (u, v) first from u.

Tree edge iff $[pre[v], post[v]] \in [pre[u], post[u]]$.
u on stack before v.

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.

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.

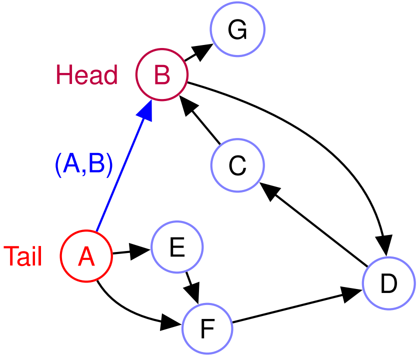
$G = (V, E)$
  vertices V.
  edges $E \subseteq V \times V$.

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



DFS on directed graphs.

Terminology:

DFS on directed graphs.

Terminology:

Root: Starting point.

DFS on directed graphs.

Terminology:

Root: Starting point.

v is ancestor of u:

DFS on directed graphs.

Terminology:

Root: Starting point.

v is ancestor of u:
v on path from/to root.

DFS on directed graphs.

Terminology:

Root: Starting point.

v is ancestor of u:
v on path from/to root.

v is descendant of u:

DFS on directed graphs.

Terminology:

Root: Starting point.

v is ancestor of u:
v on path from/to root.

v is descendant of u:
u is an ancestor of v.

DFS on directed graphs.

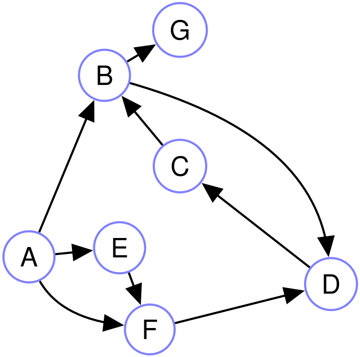
Terminology:

Root: Starting point.

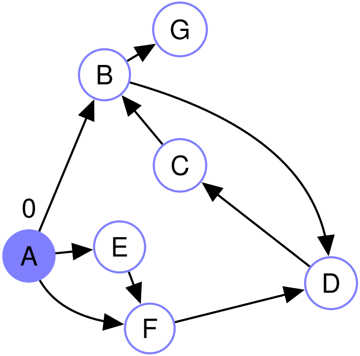
v *is ancestor of* u*:*
v on path from/to root.

v *is descendant of* u*:*
u is an ancestor of v.

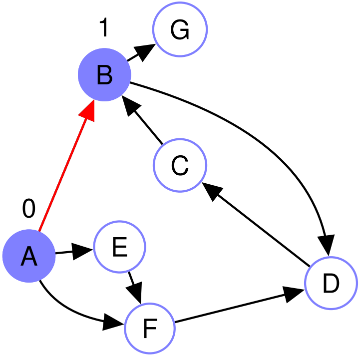
Depth first search: directed.



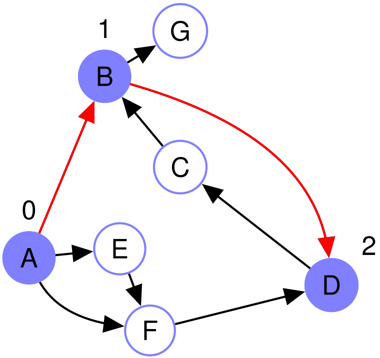
Depth first search: directed.



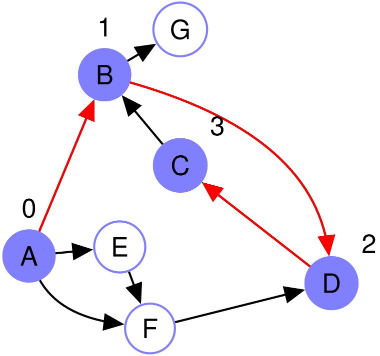
Depth first search: directed.



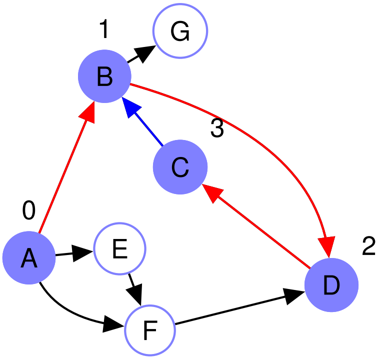
Depth first search: directed.



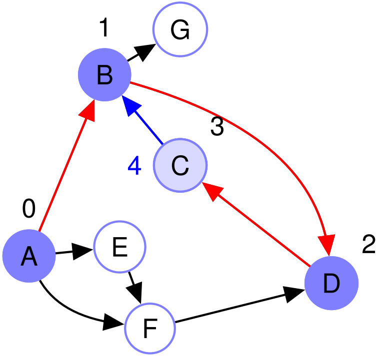
Depth first search: directed.



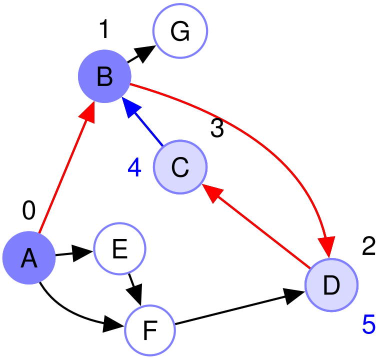
Depth first search: directed.



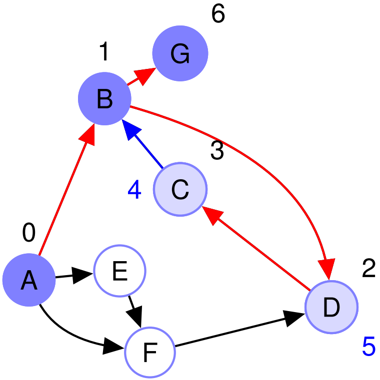
Depth first search: directed.



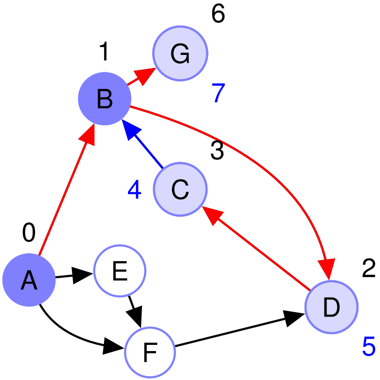
Depth first search: directed.



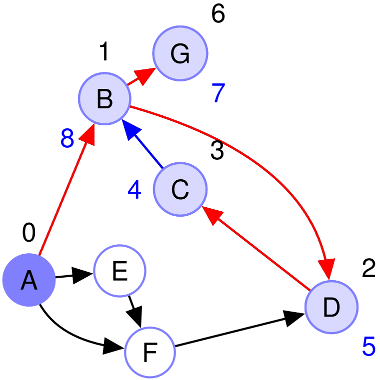
Depth first search: directed.



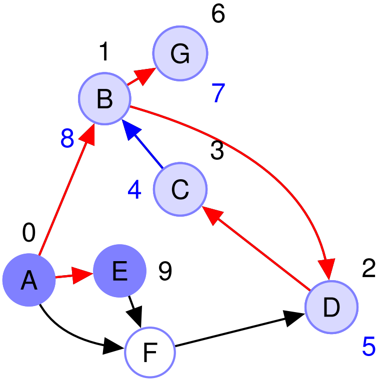
Depth first search: directed.



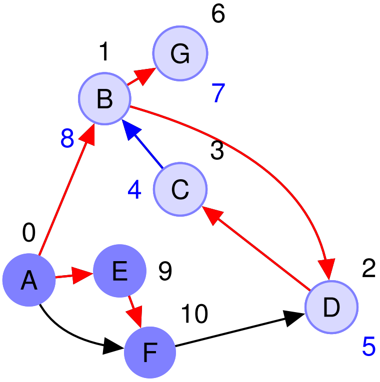
Depth first search: directed.



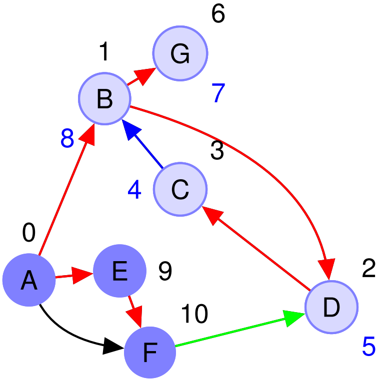
Depth first search: directed.



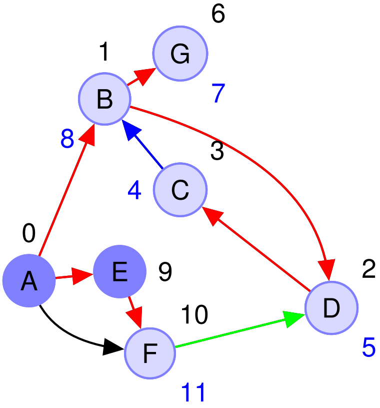
Depth first search: directed.



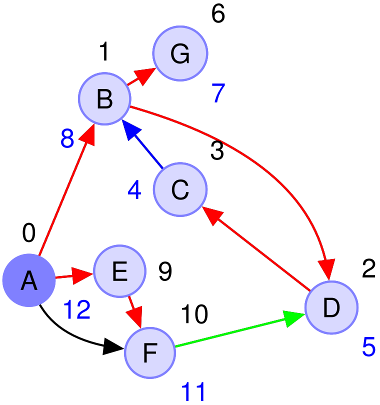
Depth first search: directed.



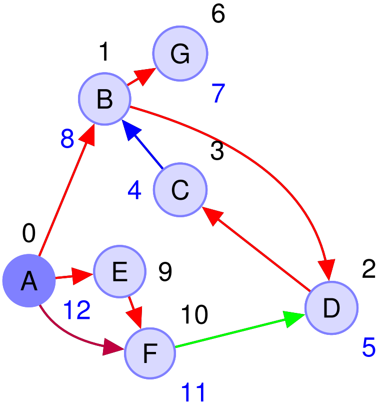
Depth first search: directed.



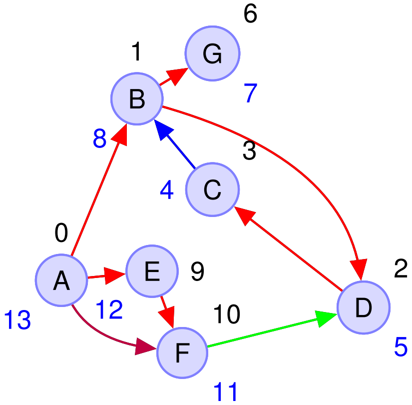
Depth first search: directed.



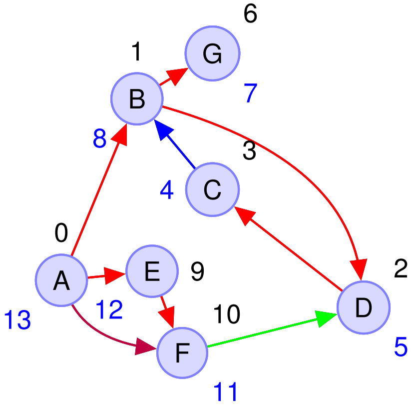
Depth first search: directed.



Depth first search: directed.

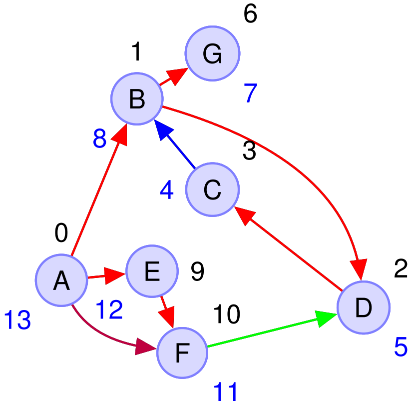


Depth first search: directed.



Tree/forward edge (u, v): int(v) in int(u).

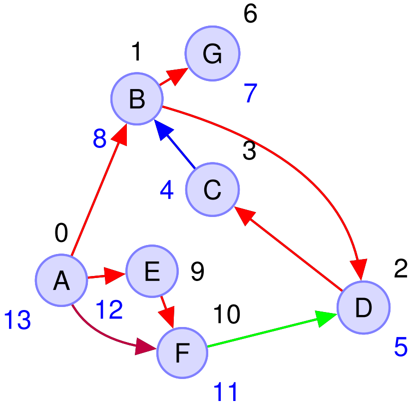
Depth first search: directed.



Tree/forward edge (u, v): int(v) in int(u).

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

Depth first search: directed.

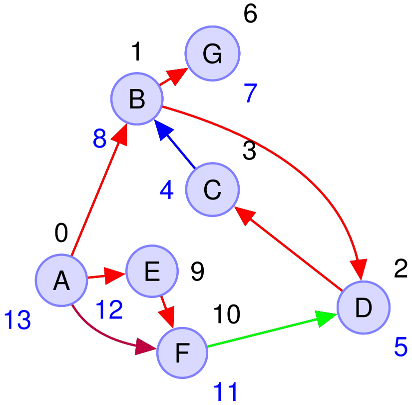


Tree/forward edge (u, v): int(v) in int(u).

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

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

Depth first search: directed.



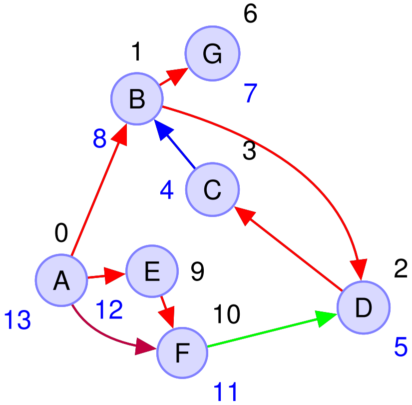
Tree/forward edge (u,v): int(v) in int(u).

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

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

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

Depth first search: directed.



Tree/forward edge (u, v): int(v) in int(u).

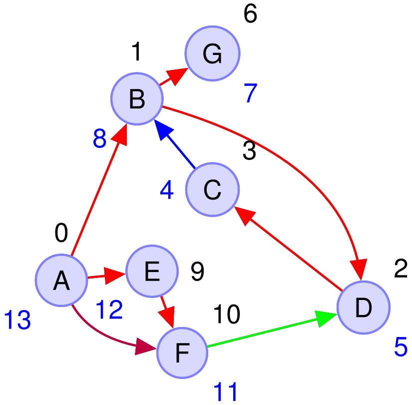
Forward (A, F): [10,11] in [0,13] or [0,[10,11],13]

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

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

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

Depth first search: directed.



Tree/forward edge (u, v): int(v) in int(u).

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

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

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

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

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

Directed Acyclic Graphs: Depth First Search

Edge: (u, v)

Directed Acyclic Graphs: Depth First Search

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

Directed Acyclic Graphs: Depth First Search

Edge: (u, v)

From u to v.

Tail – u

Directed Acyclic Graphs: Depth First Search

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

Directed Acyclic Graphs: Depth First Search

Edge: (u, v)

From u to v.

Tail – u

Head – v

Tree edge – “Direct call tree of explore.”

Directed Acyclic Graphs: Depth First Search

Edge: (u, v)

From u to v.

Tail – u

Head – v

Tree edge – "Direct call tree of explore."

Forward edge – "Edge to descendant (not in tree)"

Directed Acyclic Graphs: Depth First Search

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

Tree edge – "Direct call tree of explore."

Forward edge – "Edge to descendant (not in tree)"

Back edge – "Edge to ancestor"

Directed Acyclic Graphs: Depth First Search

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

Tree edge – "Direct call tree of explore."

Forward edge – "Edge to descendant (not in tree)"

Back edge – "Edge to ancestor"

Cross edge – None of the above.
    v already explored before u is visited.

Directed Acyclic Graphs: Depth First Search

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

Tree edge – "Direct call tree of explore."

Forward edge – "Edge to descendant (not in tree)"

Back edge – "Edge to ancestor"

Cross edge – None of the above.
    v already explored before u is visited.

Directed Acyclic Graph

Directed Graph ...

Directed Acyclic Graph

Directed Graph ...without cycles.

Directed Acyclic Graph

Directed Graph ...without cycles. Cycle: $v_0 \rightarrow v_1 \rightarrow \ldots v_k \rightarrow v_0$.

Directed Acyclic Graph

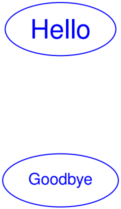
Directed Graph ...without cycles. Cycle: $v_0 \rightarrow v_1 \rightarrow \ldots v_k \rightarrow v_0$.
Why?

Directed Acyclic Graph

Directed Graph ...without cycles. Cycle: $v_0 \to v_1 \to \dots v_k \to v_0$.
Why?

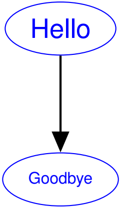
Directed Acyclic Graph

Directed Graph ...without cycles. Cycle: $v_0 \rightarrow v_1 \rightarrow \ldots v_k \rightarrow v_0$.
Why?



Directed Acyclic Graph

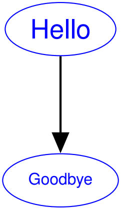
Directed Graph ...without cycles. Cycle: $v_0 \to v_1 \to \ldots v_k \to v_0$.
Why?



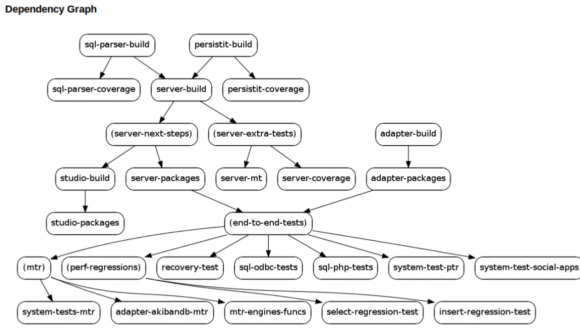
“Hello” before “Goodbye”

Directed Acyclic Graph

Directed Graph ...without cycles. Cycle: $v_0 \rightarrow v_1 \rightarrow \ldots v_k \rightarrow v_0$.
Why?



"Hello" before "Goodbye"



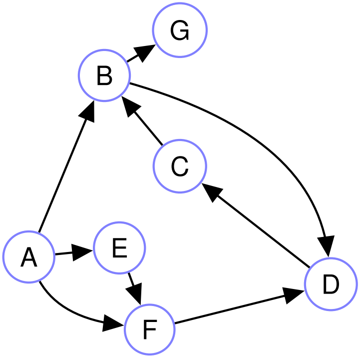
Example.

Example.

Acyclic Graph?

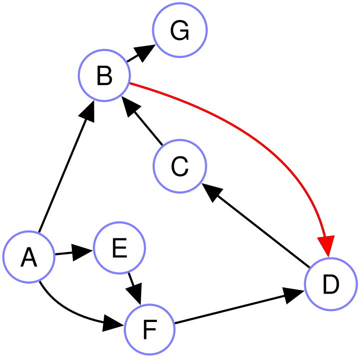
Example.

Acyclic Graph?



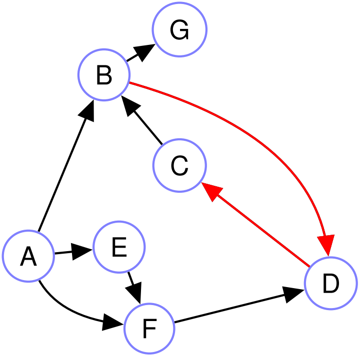
Example.

Acyclic Graph?



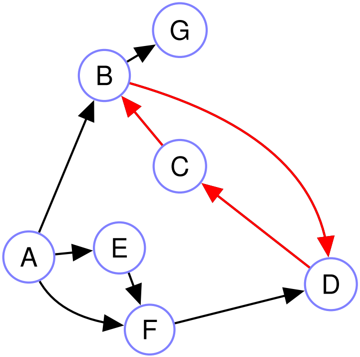
Example.

Acyclic Graph?



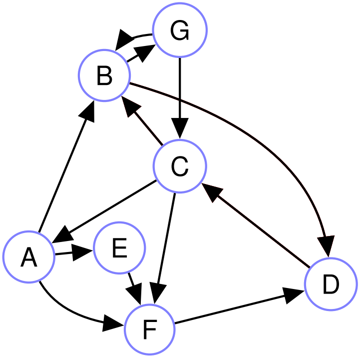
Example.

Acyclic Graph?

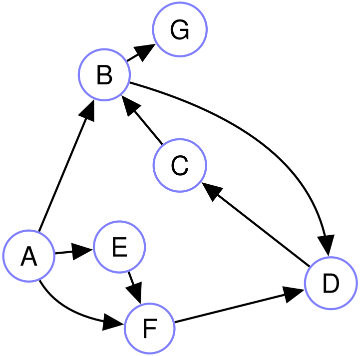


Example.

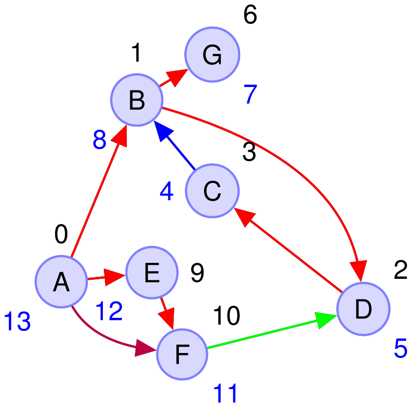
Acyclic Graph?



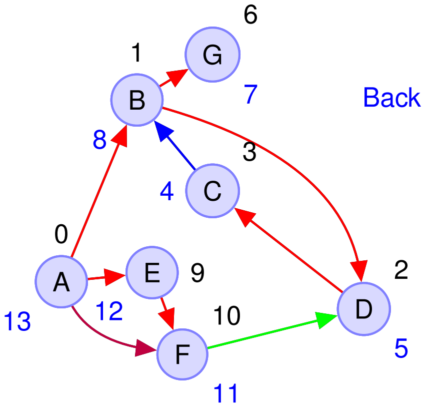
Depth first search: directed.



Depth first search: directed.

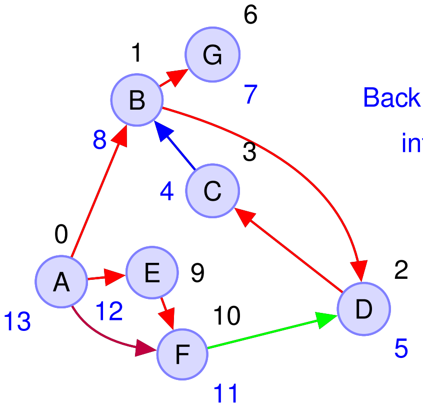


Depth first search: directed.



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

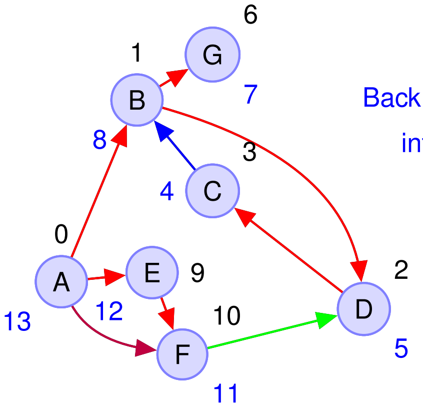
Depth first search: directed.



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

int(C) $= [3, 4]$ and int(B) $= [1, 8]$.

Depth first search: directed.

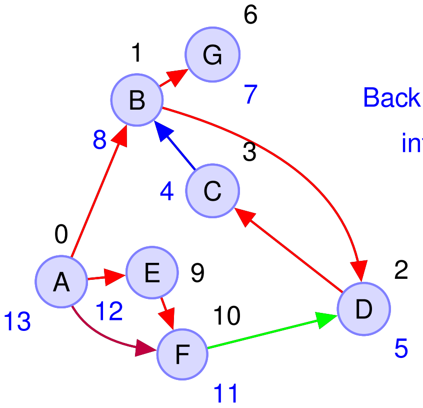


Back edge (u, v): int(v) contains int(u).

int$(C) = [3, 4]$ and int$(B) = [1, 8]$.

Back edge (u, v)

Depth first search: directed.

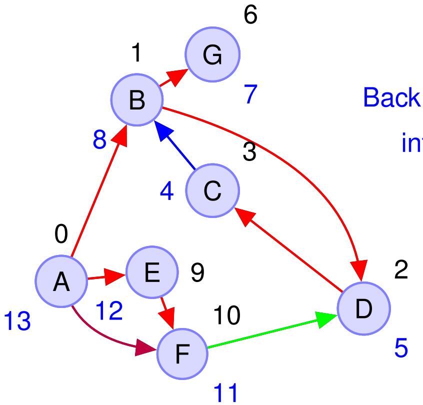


Back edge (u, v): int(v) contains int(u).

int$(C) = [3, 4]$ and int$(B) = [1, 8]$.

Back edge (u, v)
....edge to ancestor

Depth first search: directed.

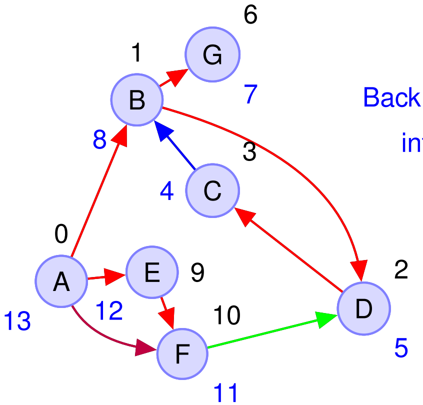


Back edge (u, v): int(v) contains int(u).

int(C) $= [3, 4]$ and int(B) $= [1, 8]$.

Back edge (u, v)
....edge to ancestor
tree edges from v to u.

Depth first search: directed.

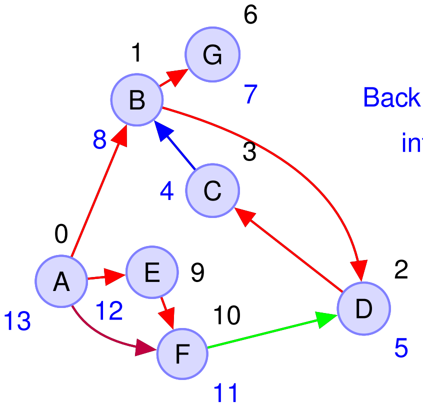


Back edge (u, v): int(v) contains int(u).

int(C) $= [3, 4]$ and int(B) $= [1, 8]$.

Back edge (u, v)
....edge to ancestor
tree edges from v to u.
Back edge means cycle!

Depth first search: directed.



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

int$(C) = [3, 4]$ and int$(B) = [1, 8]$.

Back edge (u, v)
....edge to ancestor
tree edges from v to u.
Back edge means cycle! $\implies$ not acyclic!

Testing for cycle.

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

Testing for cycle.

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

Proof:

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

There is a cycle

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

There is a cycle

$v_0 \to v_1$

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

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 back edge.

Proof:
Back edge $\implies$ cycle!

There is a cycle

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

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

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.
(without loss of generality since can renumber vertices.)

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

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.
(without loss of generality since can renumber vertices.)

All nodes on cycle explored when **explore**(v_0) returns

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

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.
(without loss of generality since can renumber vertices.)

All nodes on cycle explored when **explore**(v_0) returns

For each v_i: int[v_i] $\in$ int[v_0]!

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

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.
(without loss of generality since can renumber vertices.)

All nodes on cycle explored when **explore**(v_0) returns

For each v_i: 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 back edge.

Proof:
Back edge $\implies$ cycle!

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.
(without loss of generality since can renumber vertices.)

All nodes on cycle explored when **explore**(v_0) returns

For each v_i: int[v_i] $\in$ int[v_0]!

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

Cycle $\implies$ back edge!

Testing for cycle.

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

Proof:
Back edge $\implies$ cycle!

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.
(without loss of generality since can renumber vertices.)

All nodes on cycle explored when **explore**(v_0) returns

For each v_i: 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.

Run DFS.

Fast checking algorithm.

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

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

Fast checking algorithm.

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

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.

Run DFS. $O(|V| + |E|)$ time.
For each edge (u, v): is int(u) in int(v). $O(|E|)$ time.

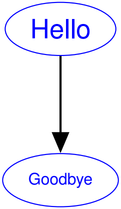
Fast checking algorithm.

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

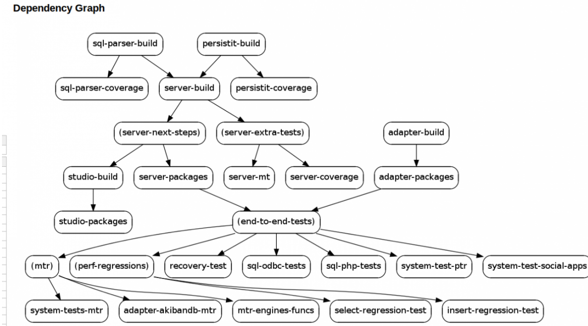
Run DFS. $O(|V|+|E|)$ time.
For each edge (u,v): is int(u) in int(v). $O(|E|)$ time.

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

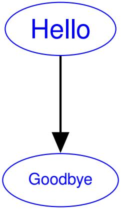
Directed Acyclic Graph

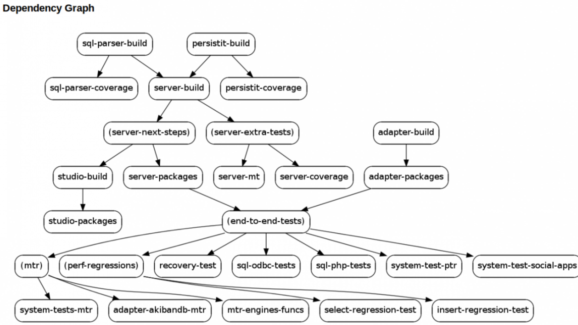


“Hello” before “Goodbye”



Directed Acyclic Graph

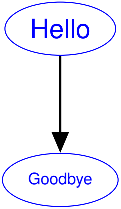


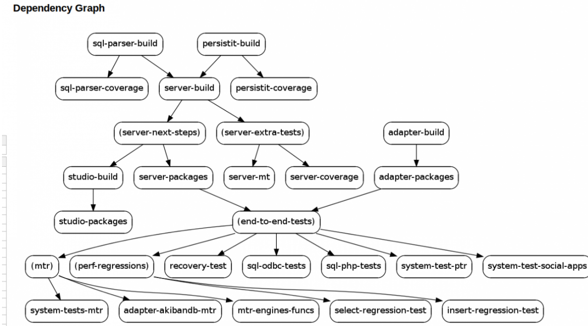


“Hello” before “Goodbye”

No cycles!

Directed Acyclic Graph

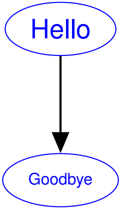




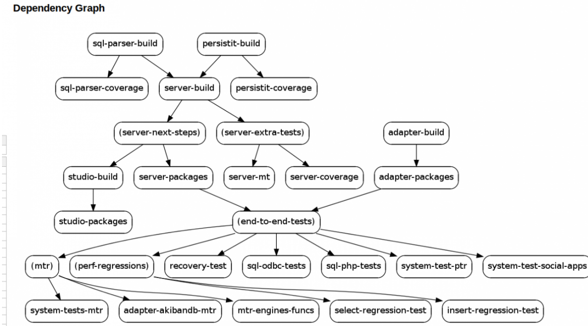
“Hello” before “Goodbye”

No cycles! Can tell in linear time!

Directed Acyclic Graph



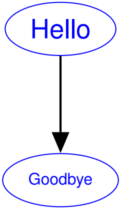
“Hello” before “Goodbye”



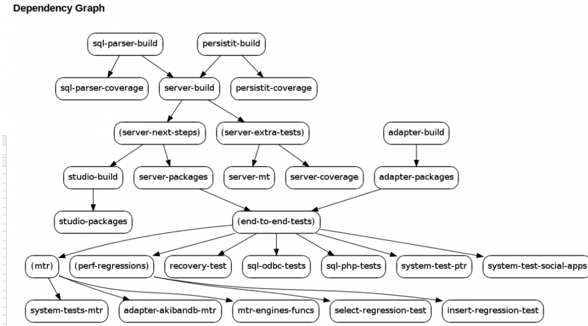
No cycles! Can tell in linear time!

Ohhh...

Directed Acyclic Graph



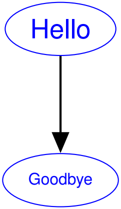
“Hello” before “Goodbye”



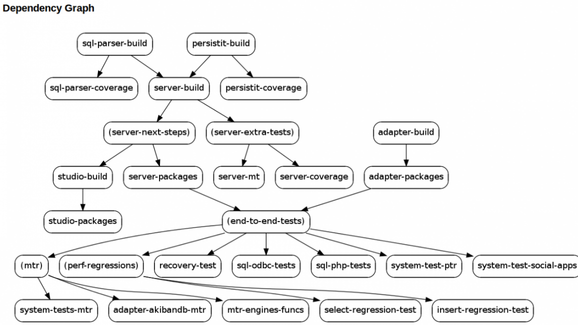
No cycles! Can tell in linear time!

Ohhh...Kayyyy...

Directed Acyclic Graph



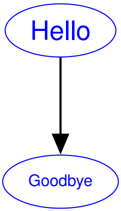
"Hello" before "Goodbye"



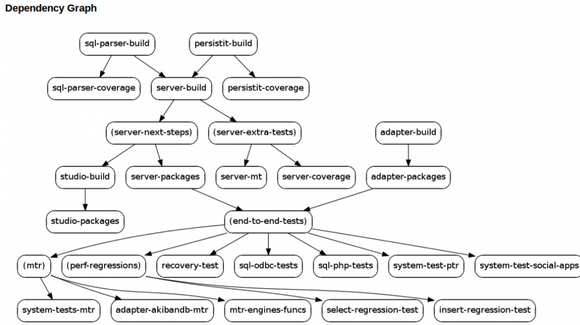
No cycles! Can tell in linear time!

Ohhh...Kayyyy...
Really want to find ordering for build!

Directed Acyclic Graph



"Hello" before "Goodbye"



No cycles! Can tell in linear time!

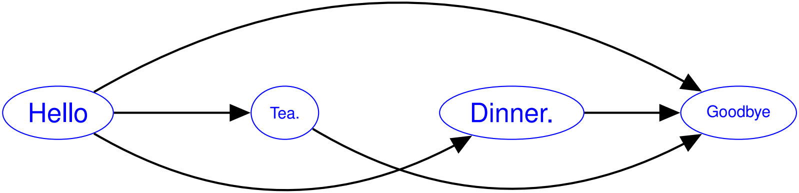
Ohhh...Kayyyy...
Really want to find ordering for build!
Where things are cool!

Linearize.

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

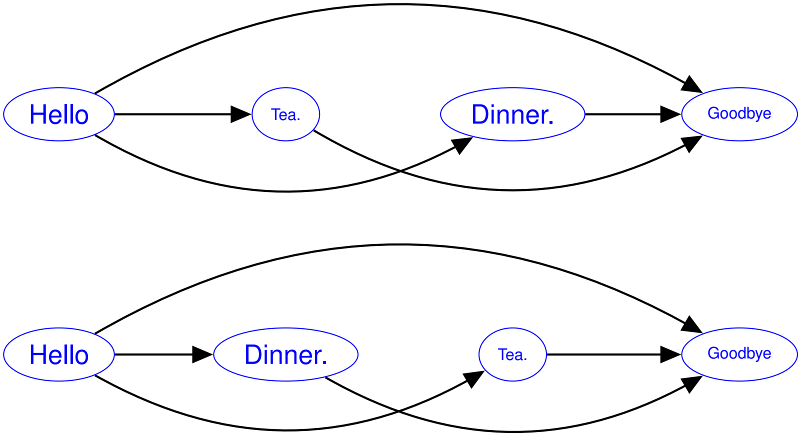
Linearize.

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

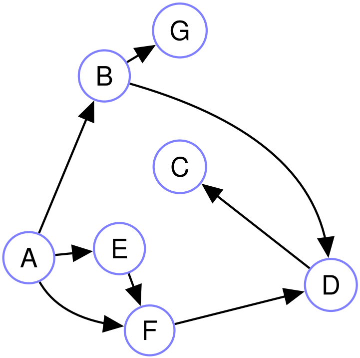


Linearize.

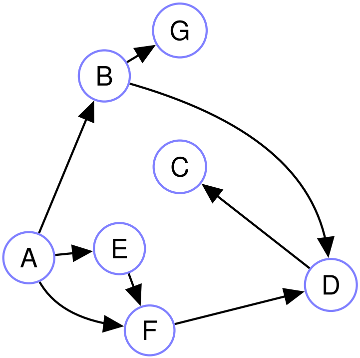
Topological Sort: For $G = (V, E)$, find ordering where each edge goes from earlier vertex to later in acyclic graph.



Topological Sort Example.

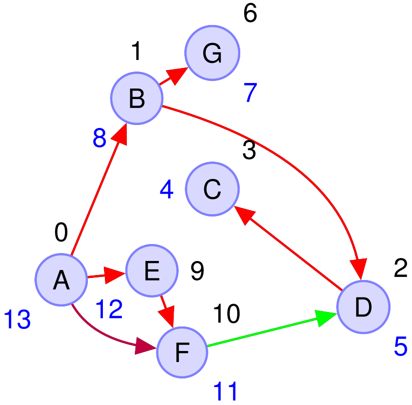


Topological Sort Example.



A linear order:

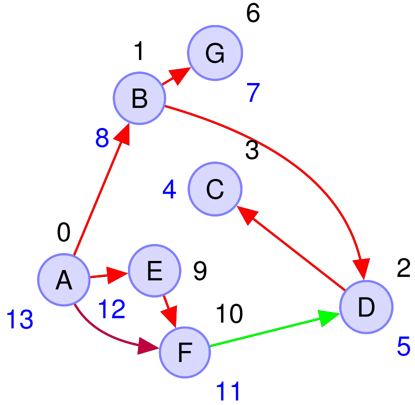
Topological Sort Example.



A linear order:

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

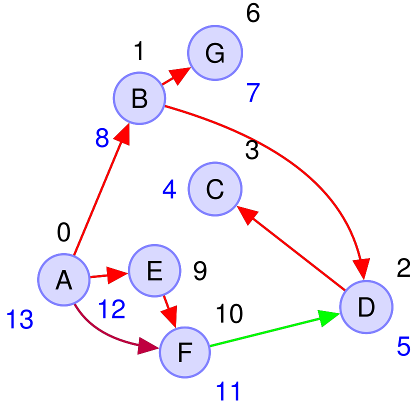
Topological Sort Example.



A linear order:

A, F, E, B, G, D, C? Nope.

Topological Sort Example.

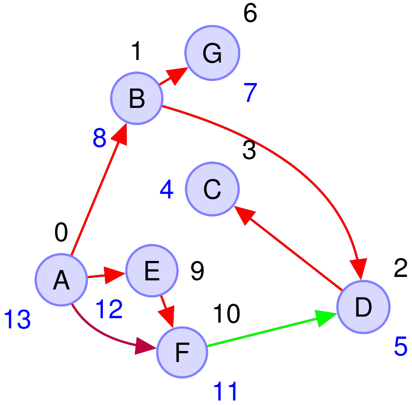


A linear order:

A, F, E, B, G, D, C? Nope.

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

Topological Sort Example.



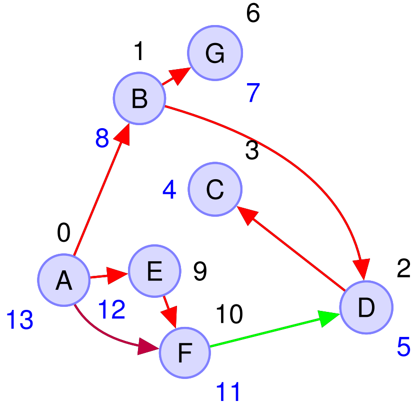
A linear order:

A, F, E, B, G, D, C? Nope.

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

In DFS: When is A popped off stack?

Topological Sort Example.



A linear order:

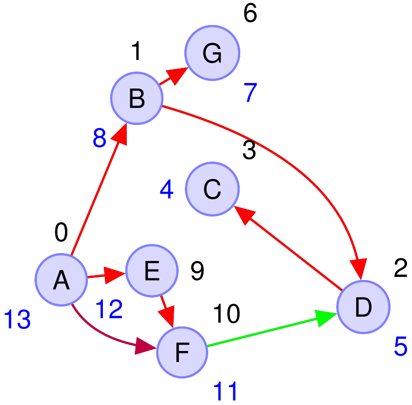
A, F, E, B, G, D, C? Nope.

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

In DFS: When is A popped off stack?

Last!

Topological Sort Example.



A linear order:

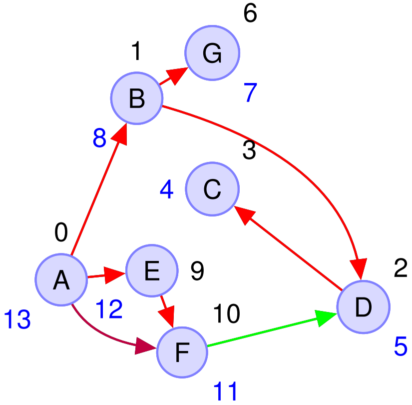
A, F, E, B, G, D, C? Nope.

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

In DFS: When is A popped off stack?

Last! E

Topological Sort Example.



A linear order:

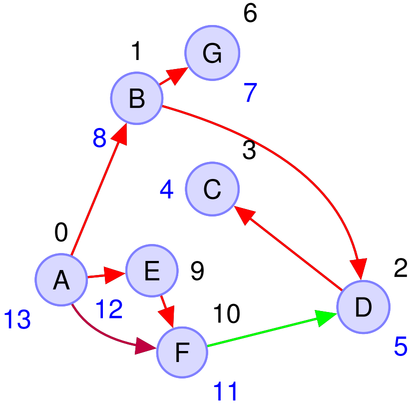
A, F, E, B, G, D, C? Nope.

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

In DFS: When is A popped off stack?

Last! E second to last.

Topological Sort Example.



A linear order:

A, F, E, B, G, D, C? Nope.

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

In DFS: When is A popped off stack?

Last! E second to last. ...

Topological Sort: DFS

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.
No back edges!

Topological Sort: DFS

Property: Every edge in a DAG (u, v) has *post*$(u) >$ *post*(v).
No back edges!
Tree and Forward edge (u, v):
  int(u) contains int(v): *pre*(u), *pre*$[v]$, *post*$[v]$, *post*$[u]$

Topological Sort: DFS

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.
No back edges!
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

Property: Every edge in a DAG (u, v) has *post*$(u) >$ *post*(v).
No back edges!
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)

Top Sort: output in reverse post order number.

Topological Sort: DFS

Property: Every edge in a DAG (u, v) has *post*$(u) >$ *post*(v).
No back edges!
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)

Top Sort: output in reverse post order number.

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

Topological Sort: DFS

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.
No back edges!
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)

Top Sort: output in reverse post order number.

Runtime: $O(|V| + |E|)$.
..procedure PostVisit outputs during DFS

Topological Sort: DFS

Property: Every edge in a DAG (u, v) has $post(u) > post(v)$.
No back edges!
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)

Top Sort: output in reverse post order number.

Runtime: $O(|V| + |E|)$.
  ..procedure PostVisit outputs during DFS
  ..reverse.

Source/sinks in a DAG.

Source is node with no incoming arcs.

Source/sinks in a DAG.

Source is node with no incoming arcs.
Sink is node with no outcoming arcs.

Source/sinks in a DAG.

Source is node with no incoming arcs.
Sink is node with no outcoming arcs.

Highest post order node is source.

Source/sinks in a DAG.

Source is node with no incoming arcs.
Sink is node with no outcoming arcs.

Highest post order node is source.
Lowest post order node is sink.

Source/sinks in a DAG.

Source is node with no incoming arcs.
Sink is node with no outcoming arcs.

Highest post order node is source.
Lowest post order node is sink.

Property: Every DAG has at least one source and sink.

Source/sinks in a DAG.

Source is node with no incoming arcs.
Sink is node with no outcoming arcs.

Highest post order node is source.
Lowest post order node is sink.

Property: Every DAG has at least one source and sink.

Topological Sort Algorithm: Find source, output, repeat.

Source/sinks in a DAG.

Source is node with no incoming arcs.
Sink is node with no outcoming arcs.

Highest post order node is source.
Lowest post order node is sink.

Property: Every DAG has at least one source and sink.

Topological Sort Algorithm: Find source, output, repeat.

Naively: $O(nm)$..

Source/sinks in a DAG.

Source is node with no incoming arcs.
Sink is node with no outcoming arcs.

Highest post order node is source.
Lowest post order node is sink.

Property: Every DAG has at least one source and sink.

Topological Sort Algorithm: Find source, output, repeat.

Naively: $O(nm)$.. there is a better implementation.

Source/sinks in a DAG.

Source is node with no incoming arcs.
Sink is node with no outcoming arcs.

Highest post order node is source.
Lowest post order node is sink.

Property: Every DAG has at least one source and sink.

Topological Sort Algorithm: Find source, output, repeat.

Naively: $O(nm)$.. there is a better implementation.

Useful on Monday.

Lecture in a Minute

Depth First Search.

Call explore until explore the whole graph.

Connected Components.

Tree/back edges.

Back edge $\iff$ cycle

Lecture in a Minute

Depth First Search.
- Call explore until explore the whole graph.
- Connected Components.
- Tree/back edges.
  - Back edge $\iff$ cycle

Pre/Post Ordering.
- Interval of time "on stack".
- Quick cycle test.

Lecture in a Minute

Depth First Search.
- Call explore until explore the whole graph.
- Connected Components.
- Tree/back edges.
  - Back edge $\iff$ cycle

Pre/Post Ordering.
- Interval of time "on stack".
- Quick cycle test.

Directed Graphs.
- Tree/Back/Forward/Cross edges.
- From pre/post!
- Back Edge $\iff$ cycle!

Lecture in a Minute

Depth First Search.
  Call explore until explore the whole graph.
  Connected Components.
  Tree/back edges.
  Back edge $\iff$ cycle

Pre/Post Ordering.
  Interval of time "on stack".
  Quick cycle test.

Directed Graphs.
  Tree/Back/Forward/Cross edges.
  From pre/post!
  Back Edge $\iff$ cycle!

Topological Sort.
  Alg 1:Inverse Post order number.

Lecture in a Minute

Depth First Search.
  Call explore until explore the whole graph.
  Connected Components.
  Tree/back edges.
   Back edge $\iff$ cycle

Pre/Post Ordering.
  Interval of time “on stack”.
  Quick cycle test.

Directed Graphs.
  Tree/Back/Forward/Cross edges.
  From pre/post!
  Back Edge $\iff$ cycle!

Topological Sort.
  Alg 1:Inverse Post order number.
    Inverse order of “stack” pop.

Lecture in a Minute

Depth First Search.
  Call explore until explore the whole graph.
  Connected Components.
  Tree/back edges.
  Back edge $\iff$ cycle

Pre/Post Ordering.
  Interval of time "on stack".
  Quick cycle test.

Directed Graphs.
  Tree/Back/Forward/Cross edges.
  From pre/post!
  Back Edge $\iff$ cycle!

Topological Sort.
  Alg 1:Inverse Post order number.
    Inverse order of "stack" pop.
  Alg 2: Peeling off sources.