

CS 170

Efficient Algorithms and Intractable Problems

Lecture 5: Graph Decompositions (Graphs part 1)

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

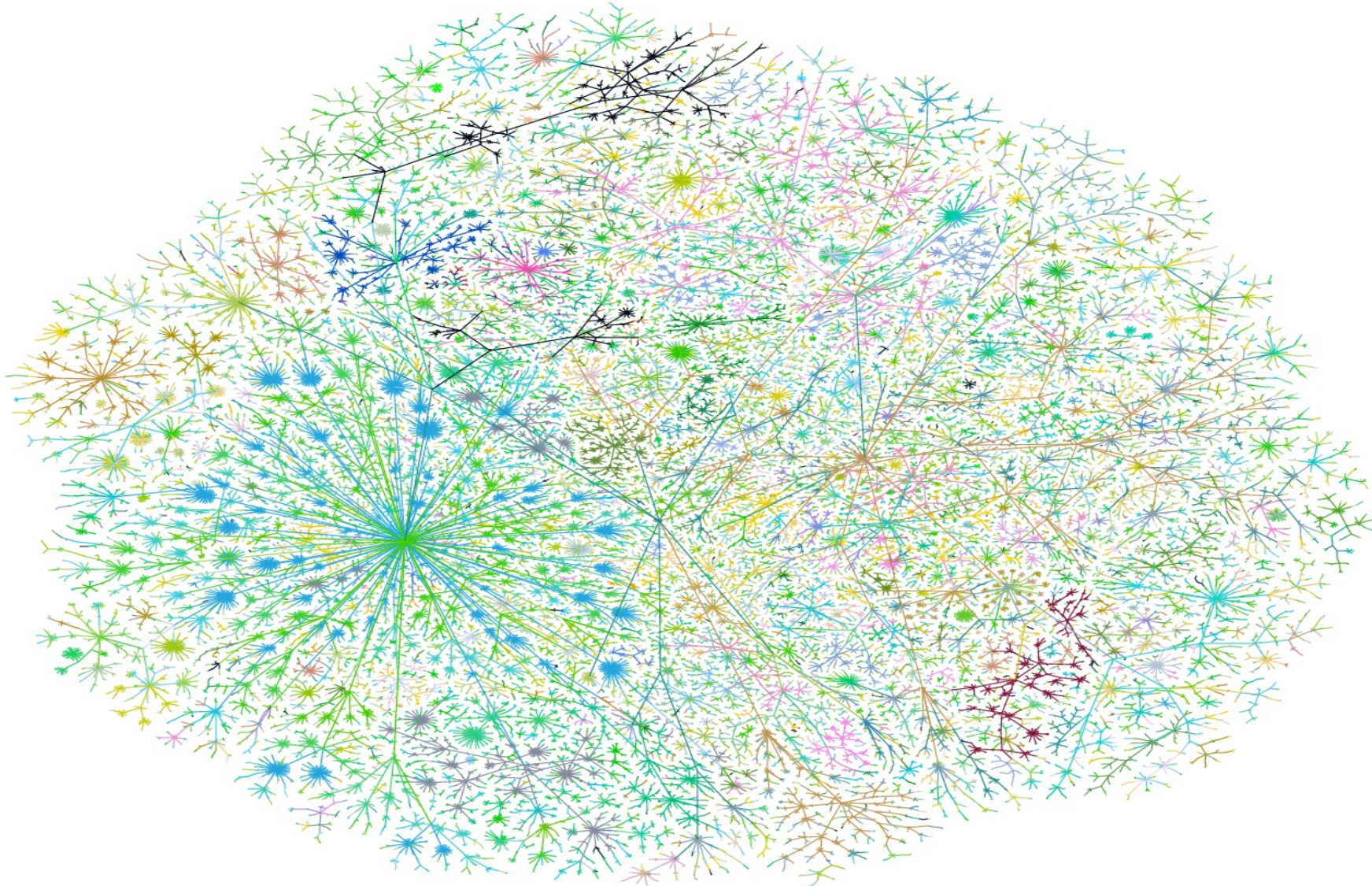
HW3 will be released today

Go to the discussion sections this week!

→ They help a lot with HW3 😊

Graphs

Graph of the internet, around 1999.



Graphs

Graph of Friendships on Facebook



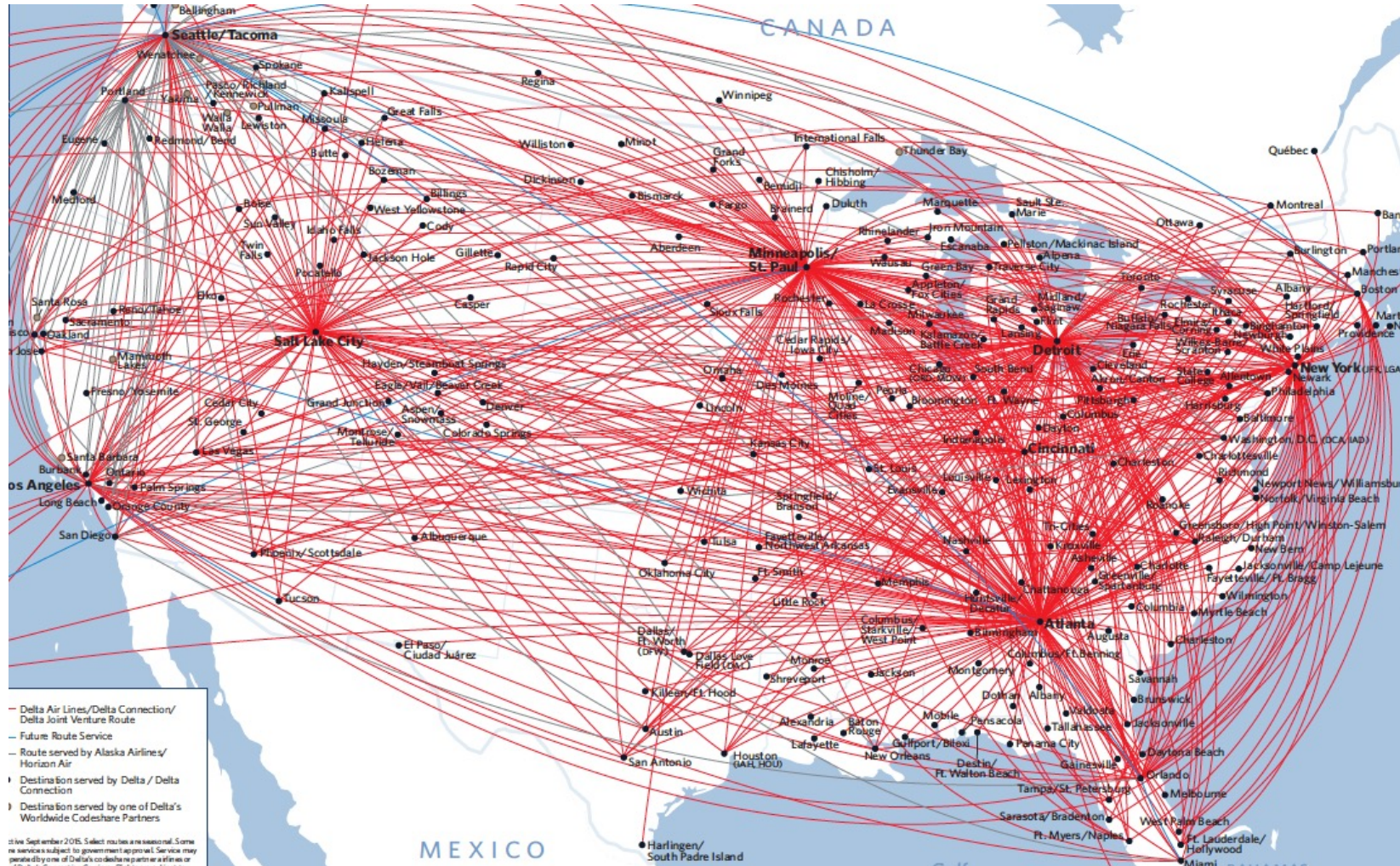
Graphs

Graph of neurological connections in the brain



Graphs

Graph of Delta Flights in the US



Graph Algorithms

There are lots of graphs everywhere. Many of them are very large!

Somethings we may want to do with graphs:

- Storing and accessing the graphs
- Find structures in them: like communities, friend groups, ...
- Find routes: how to get from one point to another
- ...

How to design algorithms on graphs

- Are they correct? Are they fast? Can we do better?

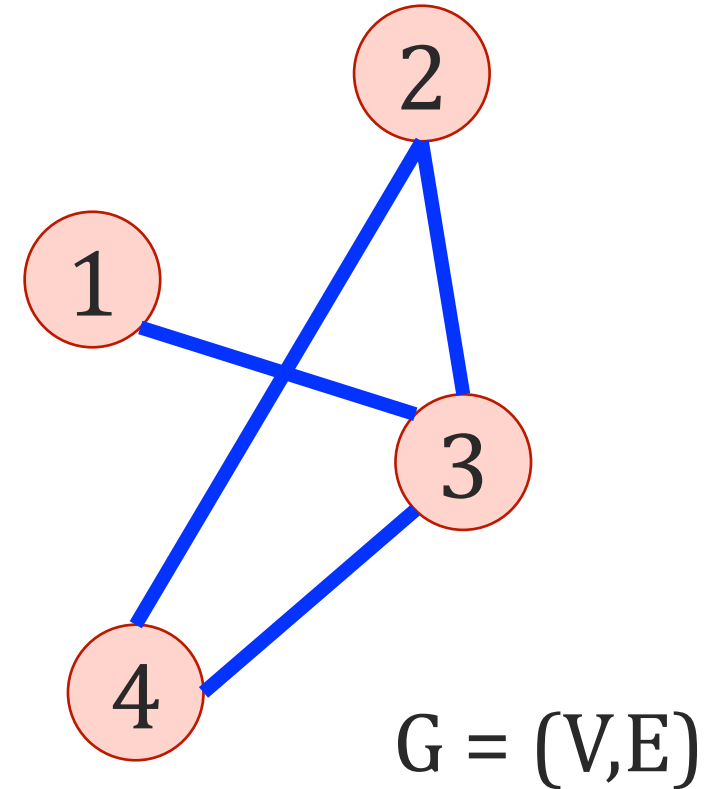
Undirected Graphs

An undirected graph G has:

- A set V of vertices/nodes
- A set E of edges
- Formally, $G = (V,E)$

Example

- $V = \{1,2,3,4\}$
- $E = \{ \{1,3\}, \{2,4\}, \{3,4\}, \{2,3\} \}$



Example of undirected graphs:

- Facebook friendship graph ($V \approx 3.03$ billion, $E \approx 338 \times 3.03$ billion)

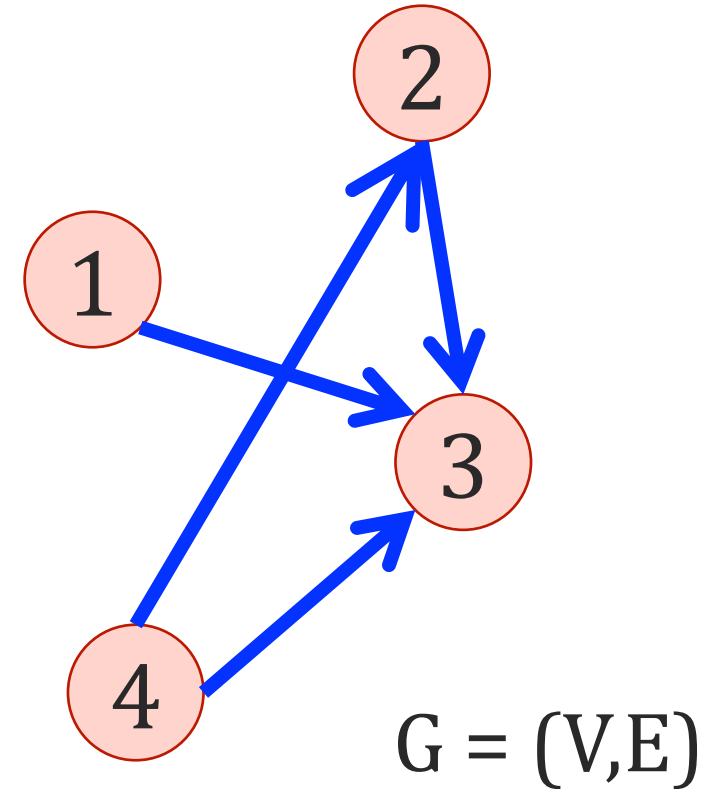
Directed Graphs

An **directed** graph G has:

- A set V of vertices/nodes
- A set E of **directed** edges
- Formally, $G = (V,E)$

Example

- $V = \{1, 2, 3, 4\}$
- $E = \{ (1,3), (2,3), (4,2), (4,3) \}$



Example of directed graphs:

- ~~Twitter “follow” graph ($V \approx 450$ million, $E \approx 707 \times 450$ million)~~
X!

Parameters Representing Graphs

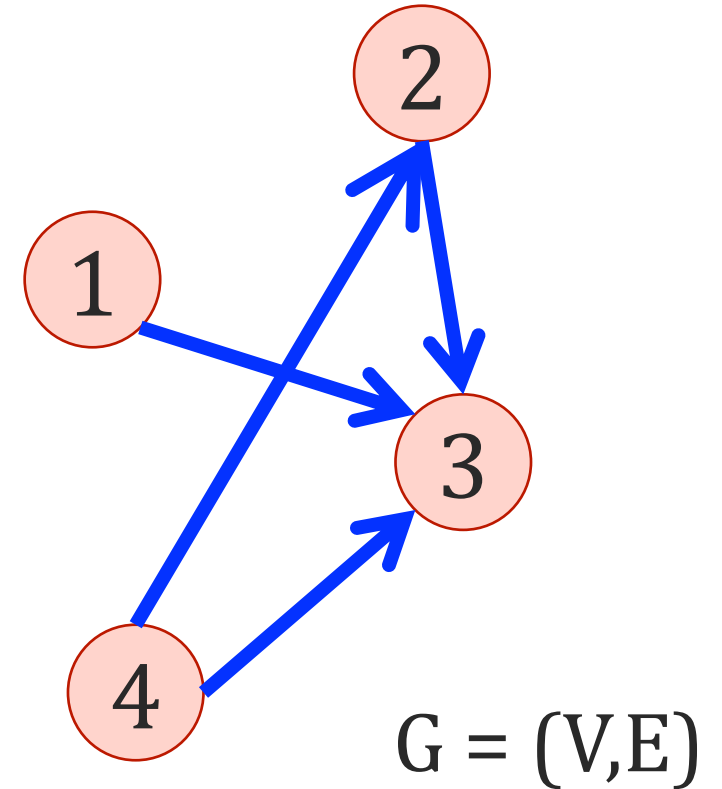
Number of nodes: $|V| = n$

Number of Edges: $|E| = m$

- $m \in O(n^2)$
- In many cases, $m \ll n^2$

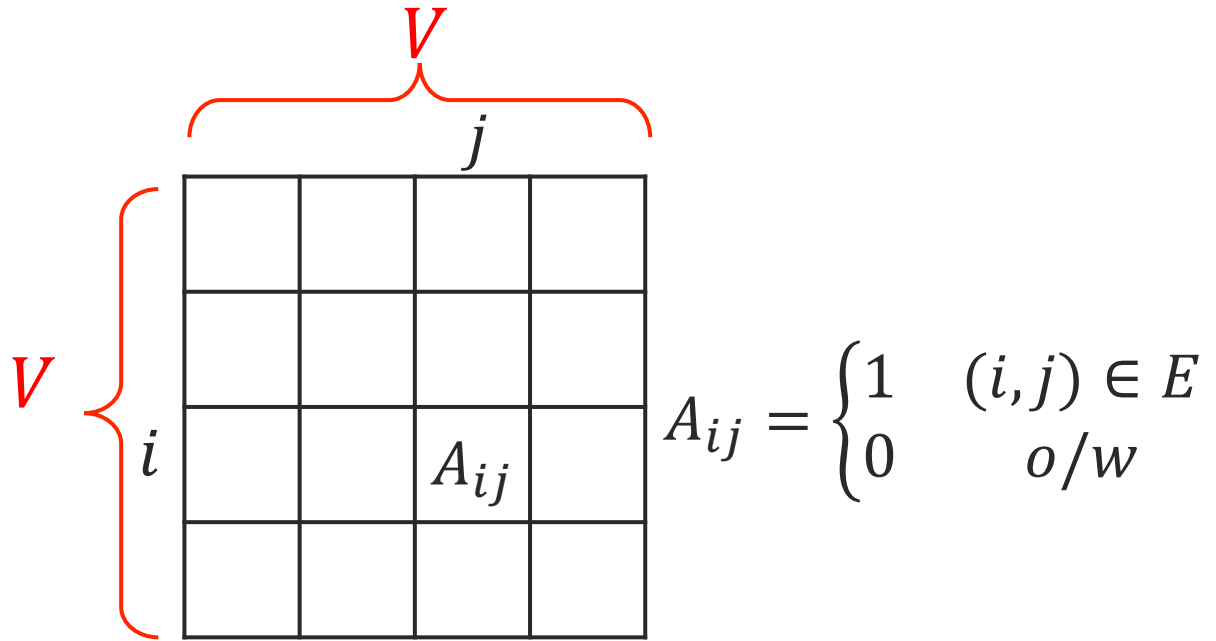
Degree (*deg*)

- Undirected (**deg(v)**): Number of edges on a node v
- Directed:
 - **in-deg(v)**: # of edges coming into v
 - **out-deg(v)**: # of edges coming out of v (also called **deg(v)** sometimes)

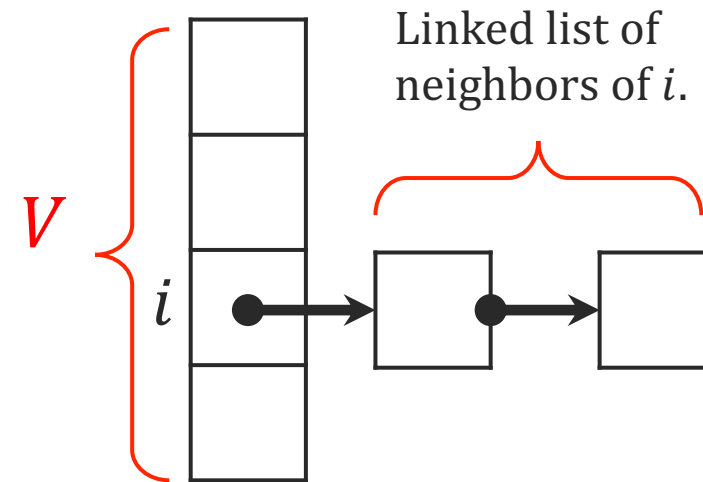


Representing Graphs on Computers

An **adjacency matrix**



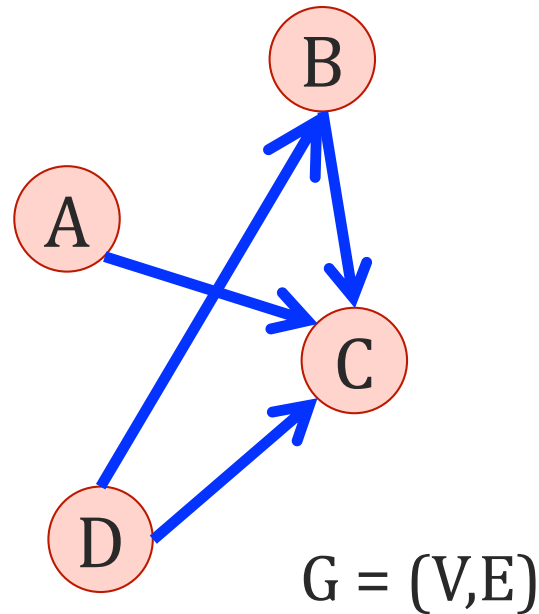
An **adjacency list**



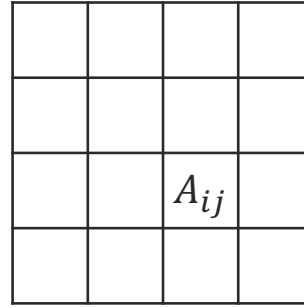
Representing Graphs on Computers: Example

An **adjacency** matrix

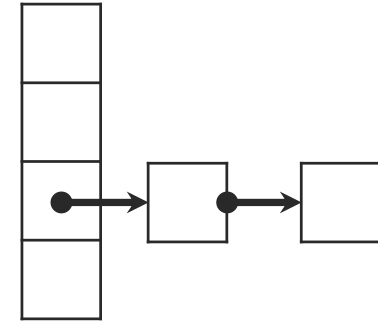
An **adjacency** list



Tradeoffs



Adjacency matrix



Adjacency list

Storage size		
Is $(u, v) \in E$?		
Enumerate all u 's neighbors		

Recall, n is the number of nodes, m is the number of edges.

Graph Questions

1. Is there a path from u to v ?
2. Is graph G connected?
3. What are G 's connected components?

Maze/Labyrinth

How do you explore a graph?

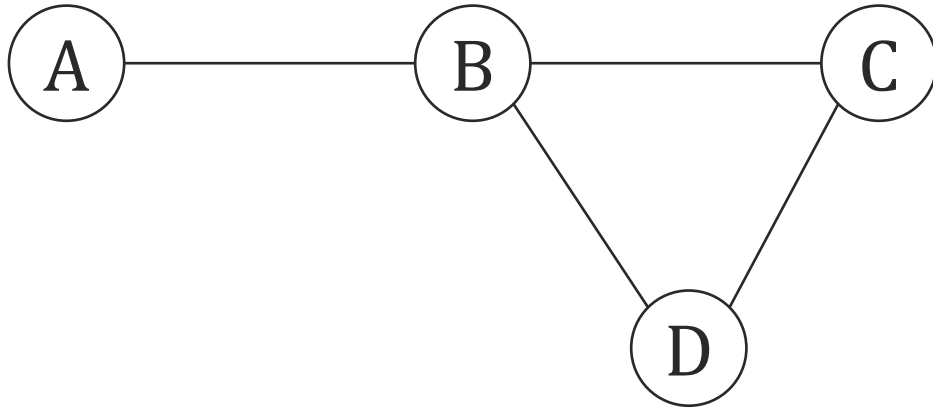
With a piece of chalk and a string!




Mark where
you've visited

Trace your way back, if
you've seen everything



Exploring: Connectivity in undirected graphs



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

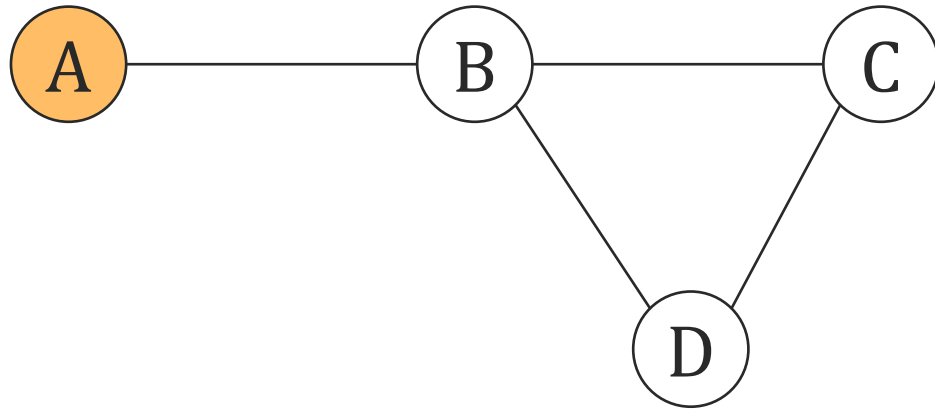
For v such that $\{u, v\} \in E$ //alphabetic order




If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

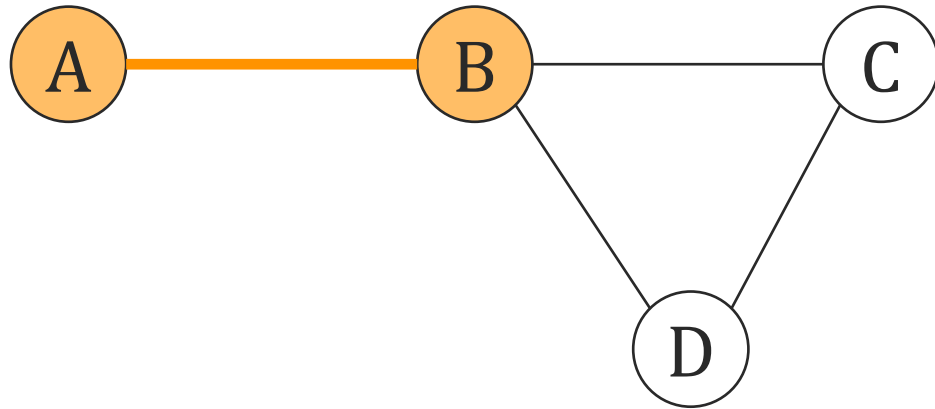
For v such that $\{u, v\} \in E$ //alphabetic order




If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

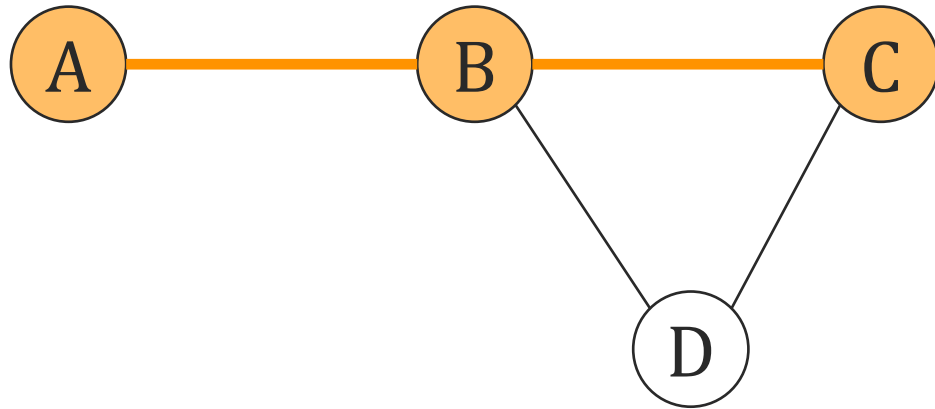
For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

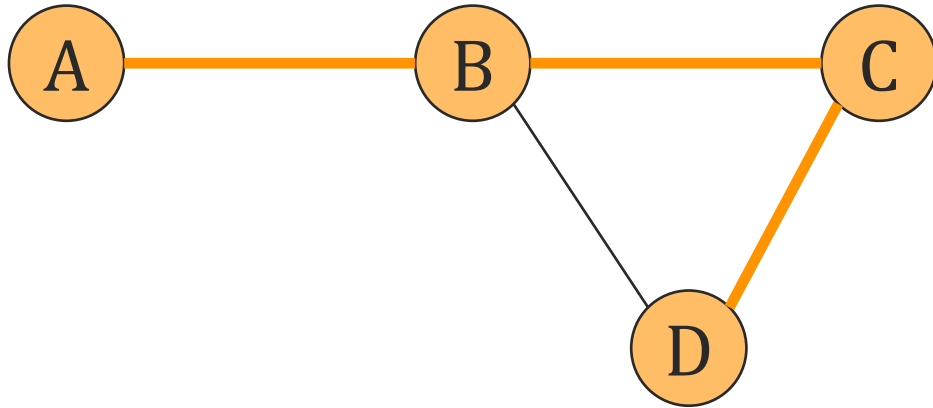
For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

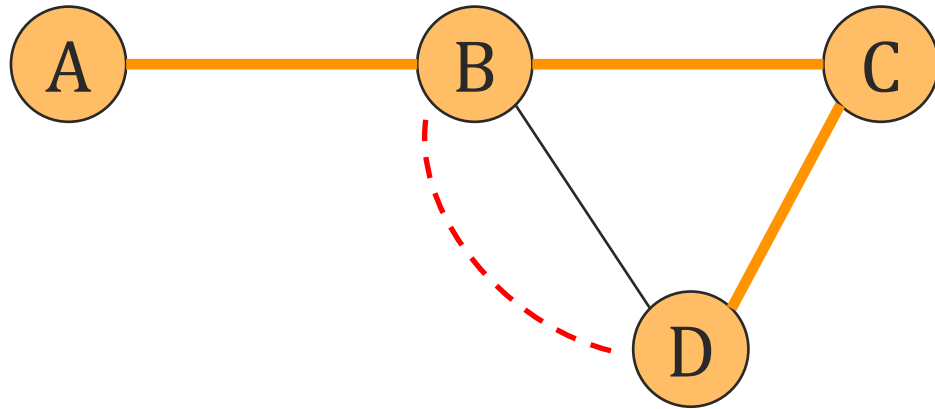
For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

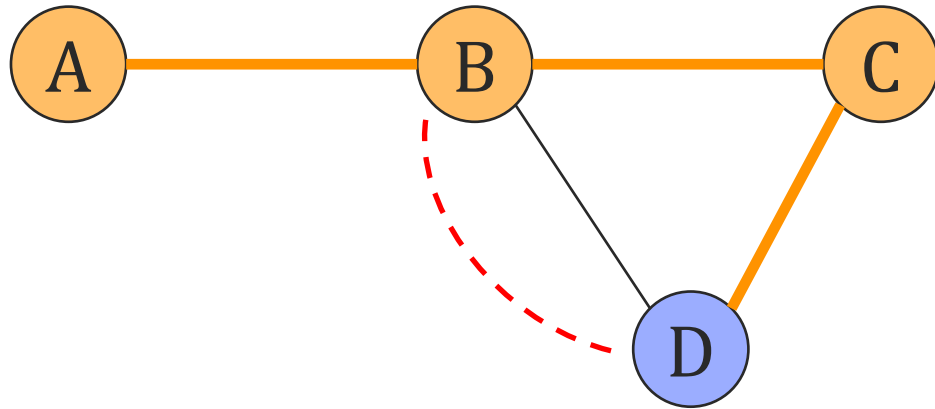
For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

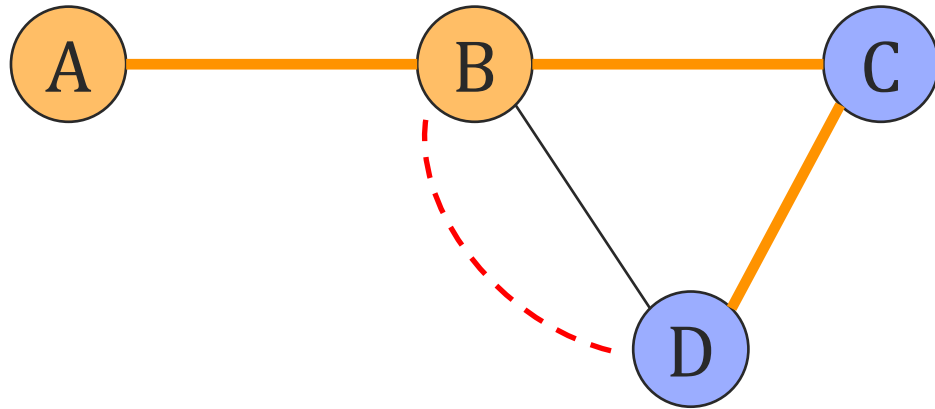
For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

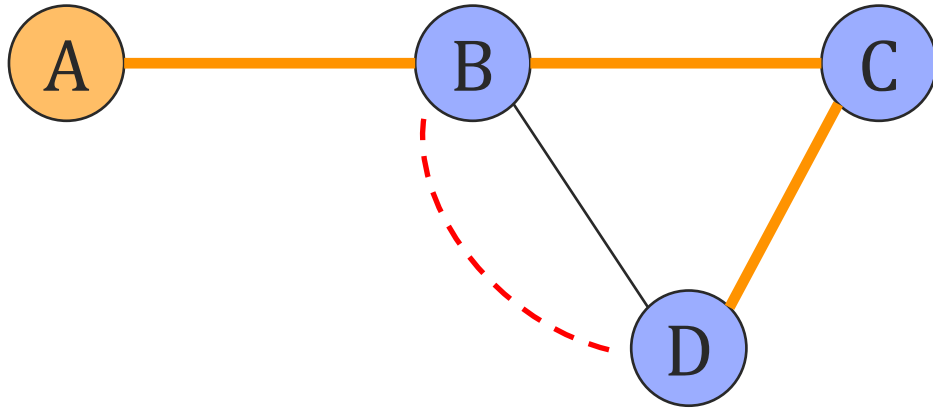
For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

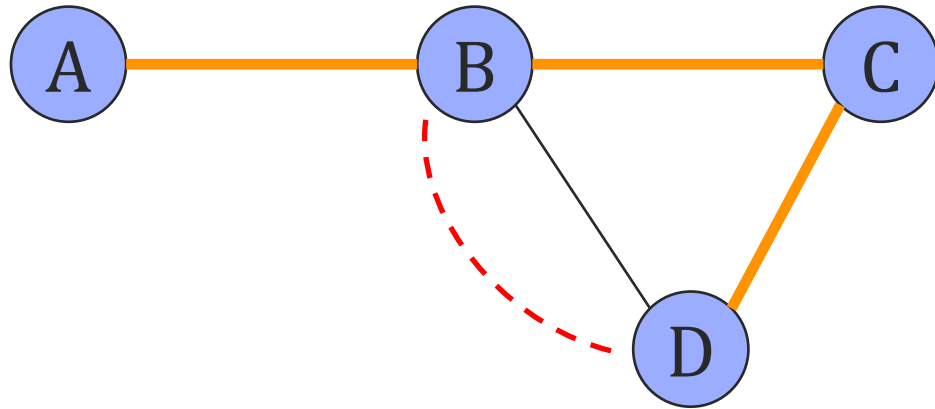
For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Exploring: Connectivity in undirected graphs



- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

boolean array *visited*(n)

// initialize to all false.

Explore's guarantees

Property: *explore*(G, u) visits exactly the vertices v s.t., G has a path from u to v .

Proof:

- v is visited \longrightarrow G has a path from u to v



- G has a path from u to v \longrightarrow v is visited

explore(G, u)

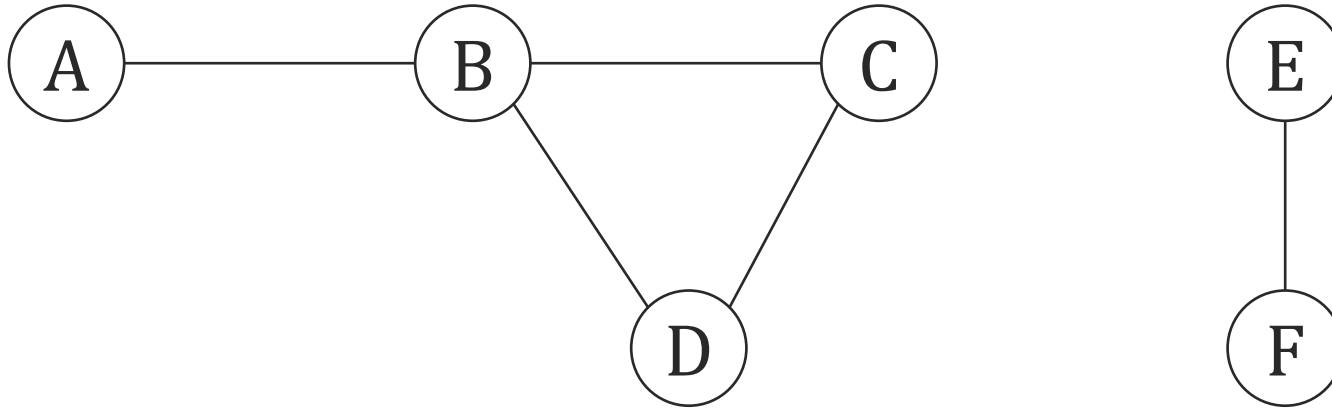
visited[u] = *true*

For v such that $\{u, v\} \in E$

If *visited*[v] = *false* then *explore*(G, v)

3 min break!
(Please close the doors)

Depth-First Search (DFS) – undirected graphs



explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

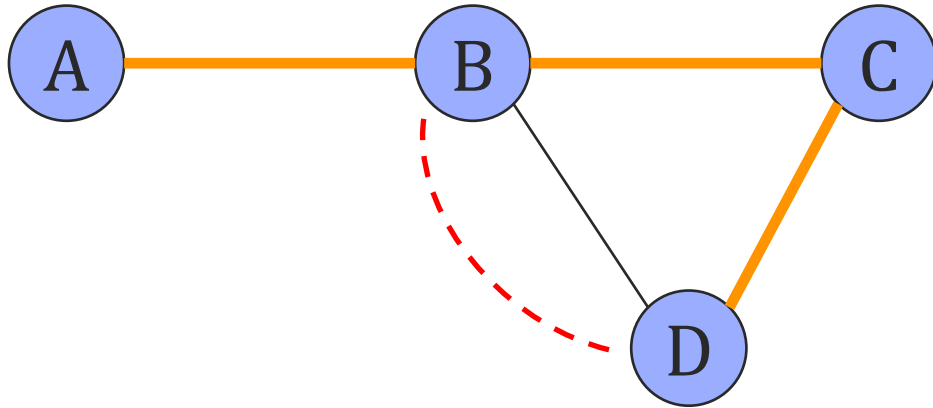
If *visited*[v] = *false* then *explore*(G, v)

dfs(G)

boolean array *visited*(n)

// initialize to all false.

Depth-First Search (DFS) – undirected graphs



explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

dfs(G)

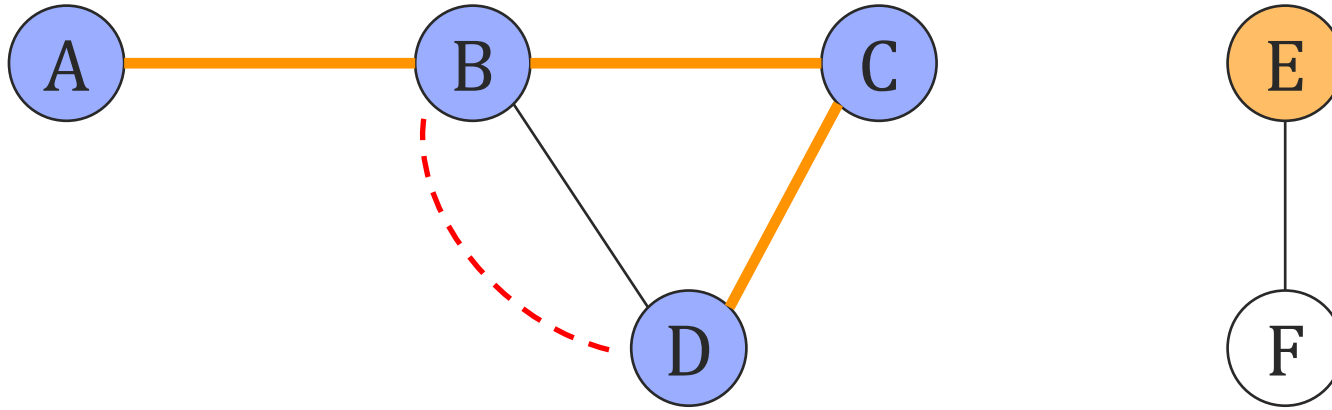
boolean array *visited*(n)

// initialize to all false.

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – undirected graphs



explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

dfs(G)

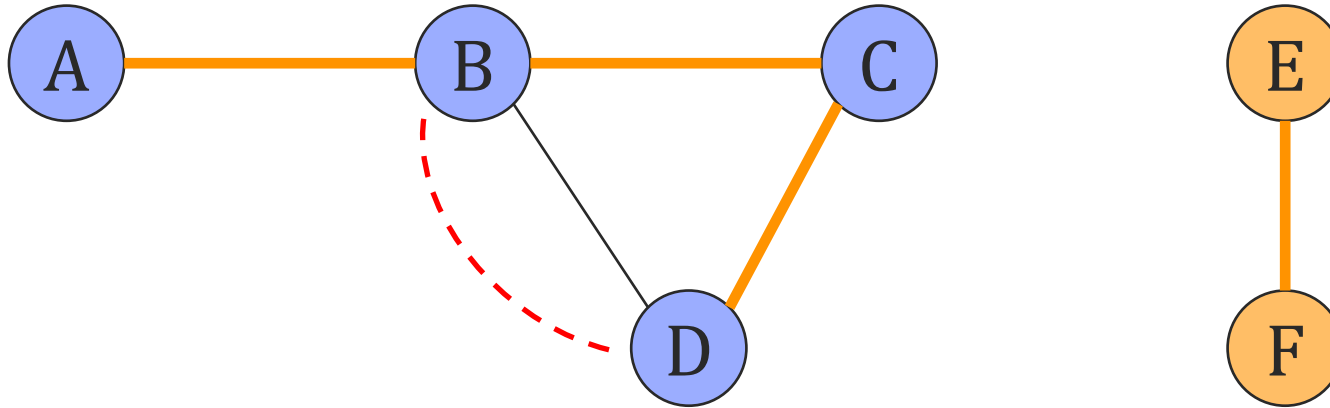
boolean array *visited*(n)

// initialize to all false.

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – undirected graphs



explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

dfs(G)

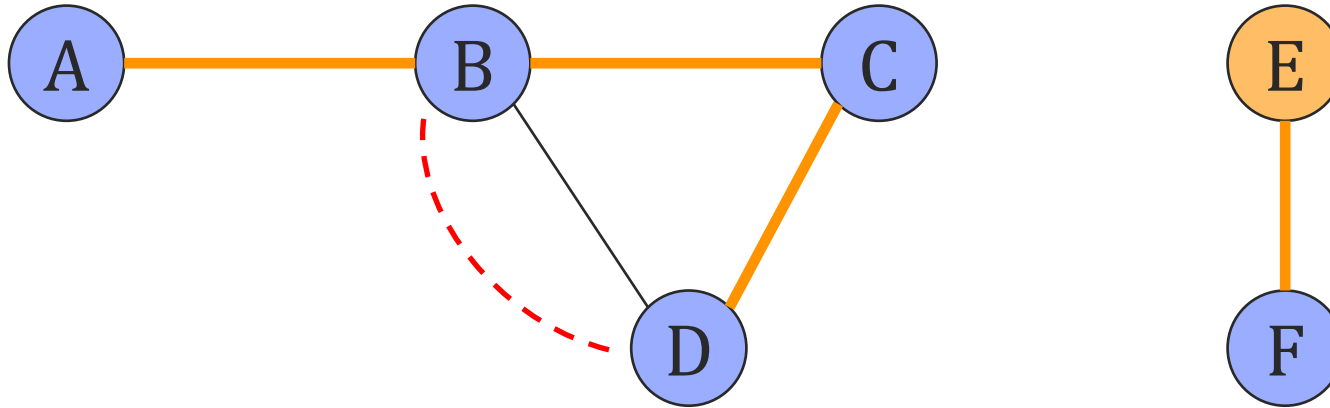
boolean array *visited*(n)

// initialize to all false.

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – undirected graphs



explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

dfs(G)

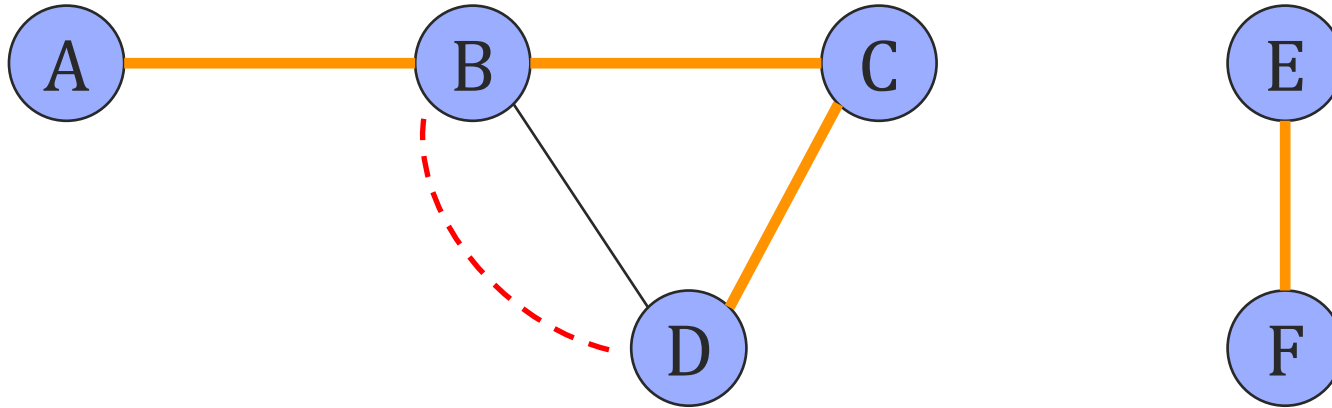
boolean array *visited*(n)

// initialize to all false.

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – undirected graphs



explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

dfs(G)

boolean array *visited*(n)

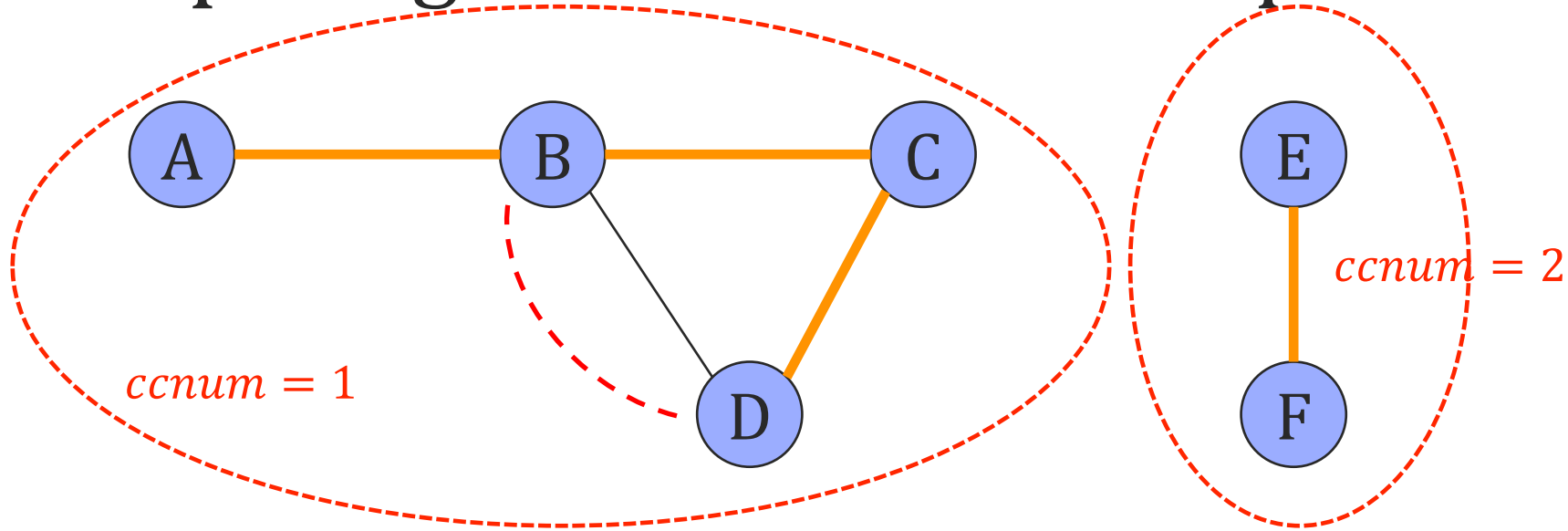
// initialize to all false.

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Using DFS to find the connected components of a graph!

Computing G 's connected components (undirected)



explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

dfs(G)

boolean array *visited*(n)

// initialize to all false.

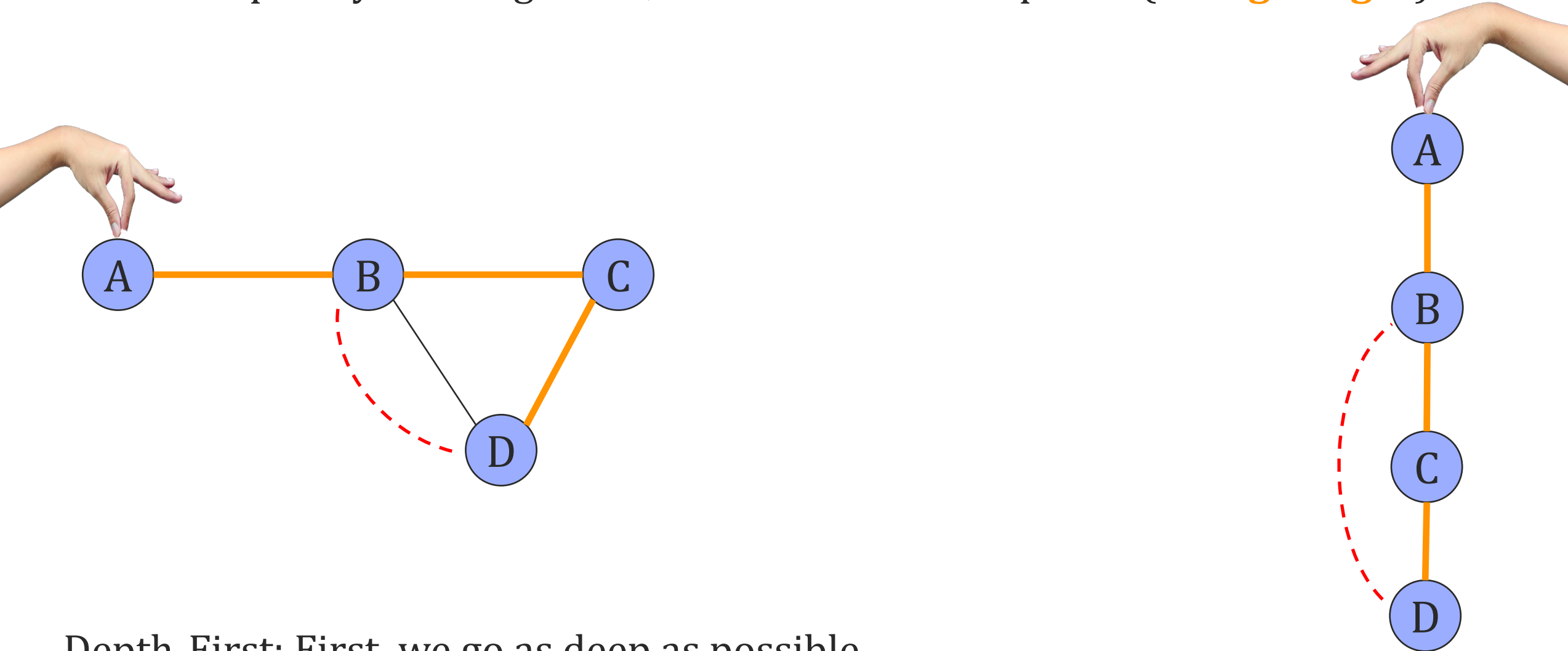
For $v \in V$ //alphabetic order

If *visited*[v] = *false*

then *explore*(G, v);

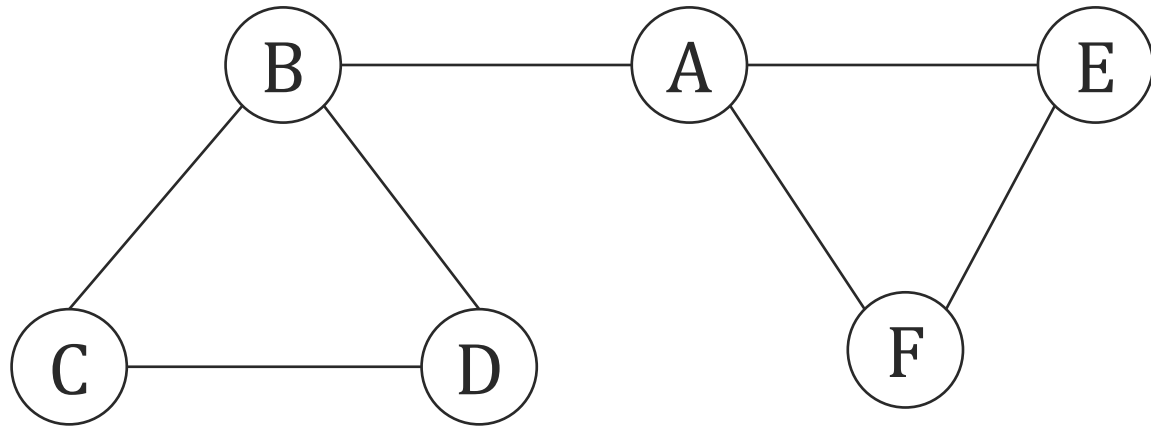
The DFS Tree / Forest!

We are implicitly building a tree, with the calls to “explore” (**orange edges**)



Depth-First: First, we go as deep as possible

A DFS Tree Example (Undirected)



DFS Facts

Cross Edge:

An edge between two vertices u, v , from different branches of the DFS tree (neither descendent or ancestor)

Fact: In an undirected graph, there are no cross edges!

DFS Runtime

We call $explore(G, u)$ exactly once for each $u \in V$.

Runtime of $explore(G, u)$:

$explore(G, u)$

$visited[u] = true$

$\rightarrow O(1)$

For v such that $\{u, v\} \in E$

$\rightarrow O(\deg(u))$: enumerate u 's neighbors

If $visited[v] = false$ then $explore(G, v)$

Total DFS Runtime:

$$\sum_{u \in V} O(1 + \deg(u)) = O(n + m)$$

DFS for Directed Graphs

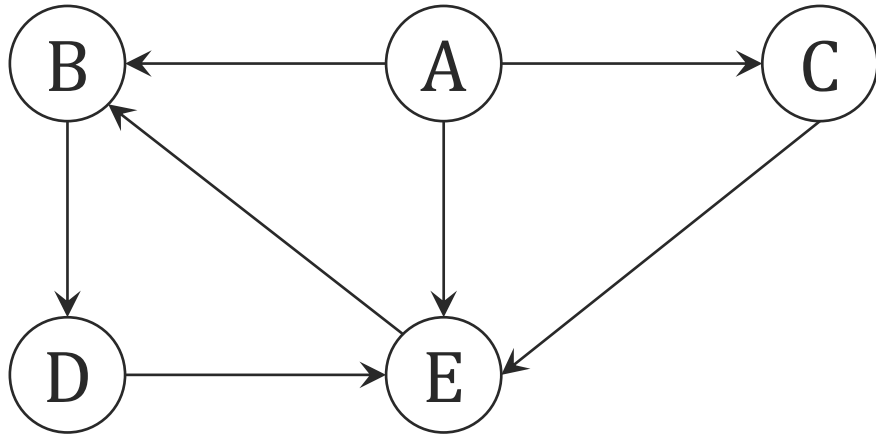
The same principle:

- Keep track of *visited*[u].
 - At every vertex, “*explore*” any unvisited neighbors.
- This time only *outgoing edges*

Because of the directedness of edges, the DFS tree/forest looks a bit different!

Let's do some more book-keeping around when a node is “*explored*”

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

dfs(G)

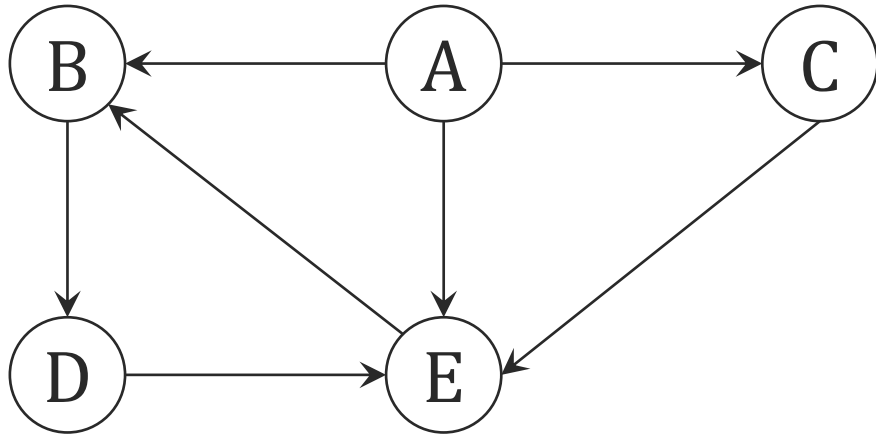
boolean array *visited*(n)

// initialize to all false.

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

pre[u] = *clock*; *clock*++

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

post[u] = *clock*; *clock*++

dfs(G)

boolean array *visited*(n)

// initialize to all false.

clock = 1

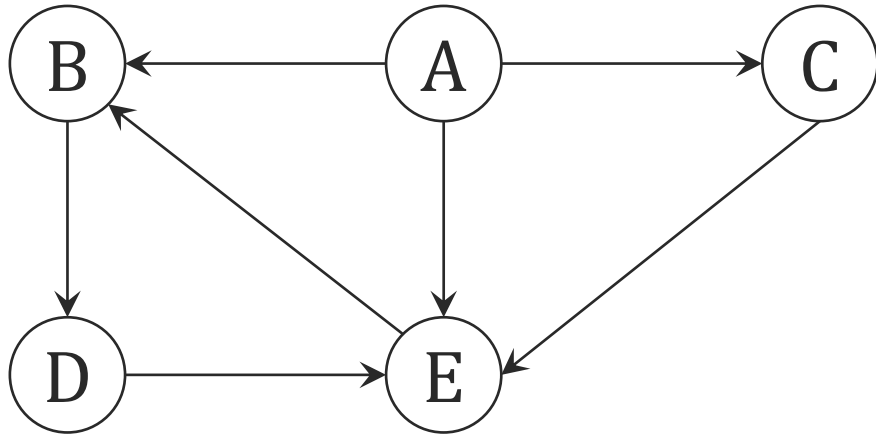
int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs

Ⓐ $pre[A] = 1$



explore(G, u)

$visited[u] = true$

$pre[u] = clock; clock++$

For v such that $\{u, v\} \in E$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

$post[u] = clock; clock++$

dfs(G)

boolean array $visited(n)$

// initialize to all false.

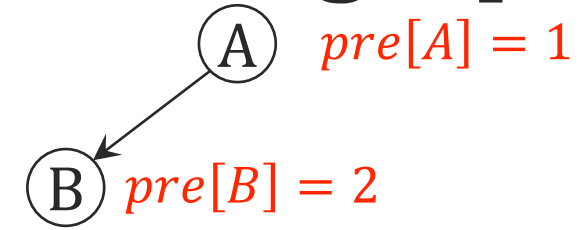
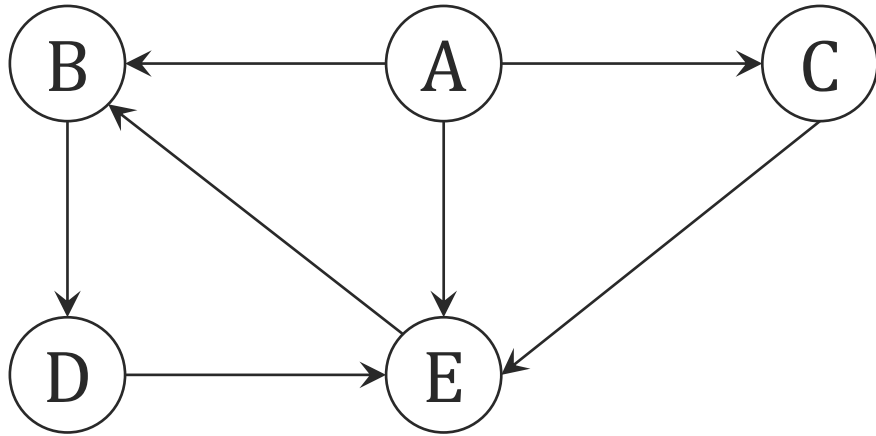
$clock = 1$

int array $pre(n), post(n)$

For $v \in V$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

$visited[u] = true$

$pre[u] = clock; clock++$

For v such that $\{u, v\} \in E$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

$post[u] = clock; clock++$

dfs(G)

boolean array $visited(n)$

// initialize to all false.

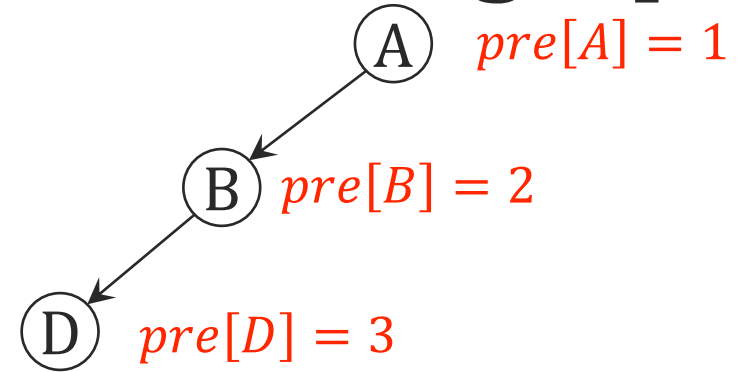
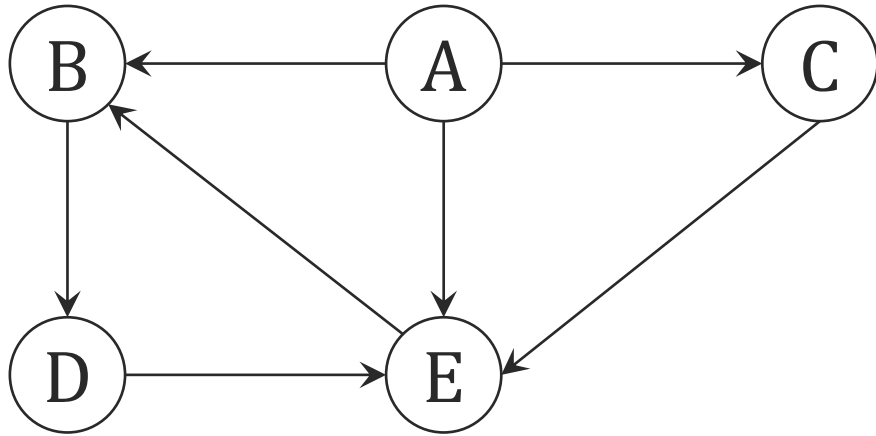
$clock = 1$

int array $pre(n), post(n)$

For $v \in V$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

pre[u] = *clock*; *clock*++

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

post[u] = *clock*; *clock*++

dfs(G)

boolean array *visited*(n)

// initialize to all false.

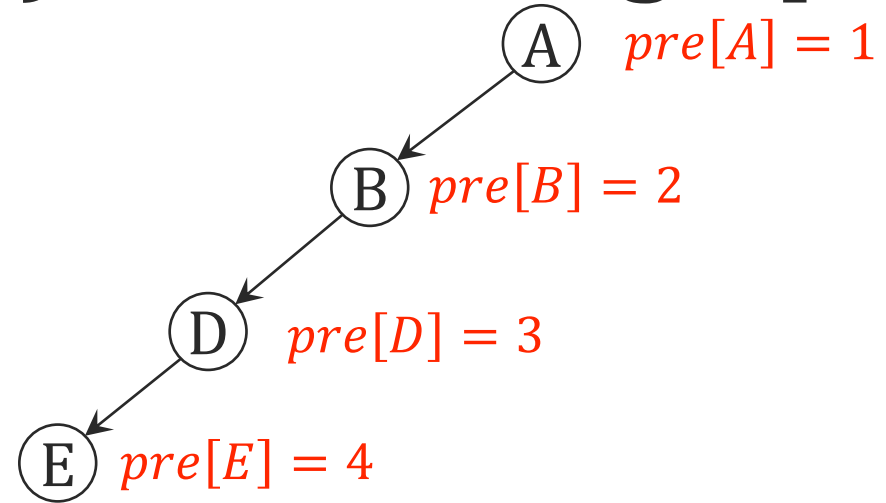
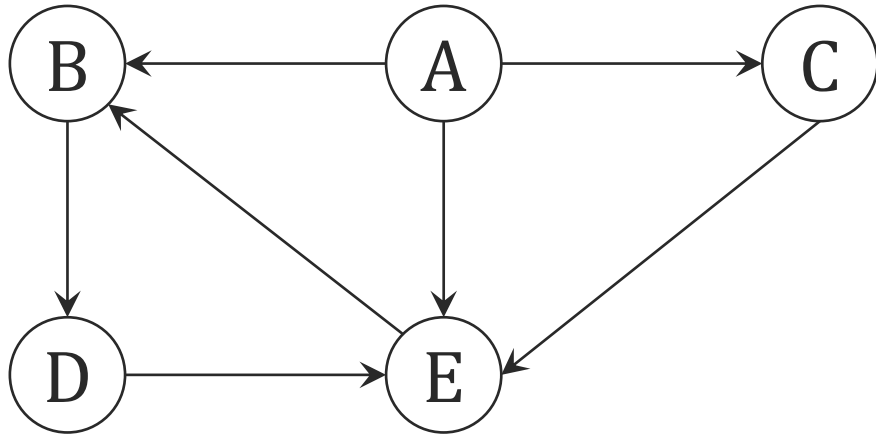
clock = 1

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

$visited[u] = true$

$pre[u] = clock; clock++$

For v such that $\{u, v\} \in E$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

$post[u] = clock; clock++$

dfs(G)

boolean array $visited(n)$

// initialize to all false.

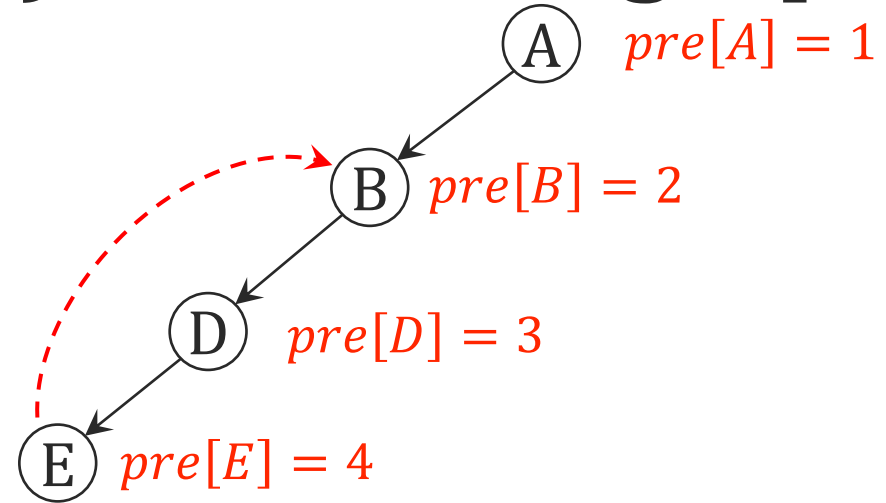
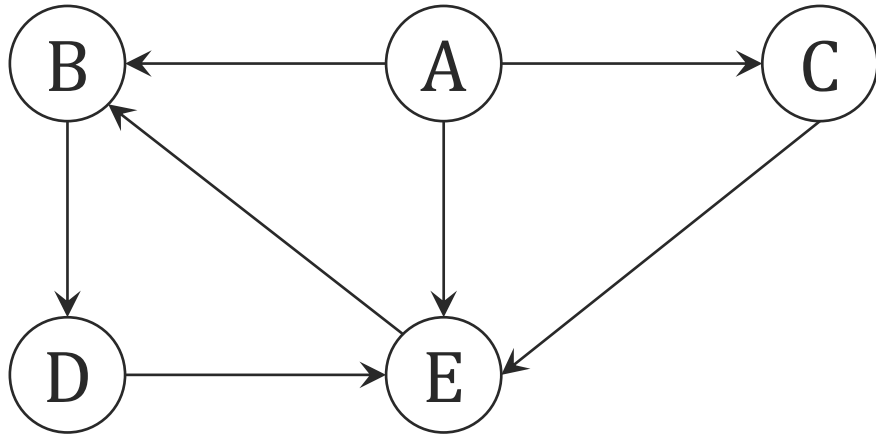
$clock = 1$

int array $pre(n), post(n)$

For $v \in V$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

pre[u] = *clock*; *clock*++

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

post[u] = *clock*; *clock*++

dfs(G)

boolean array *visited*(n)

// initialize to all false.

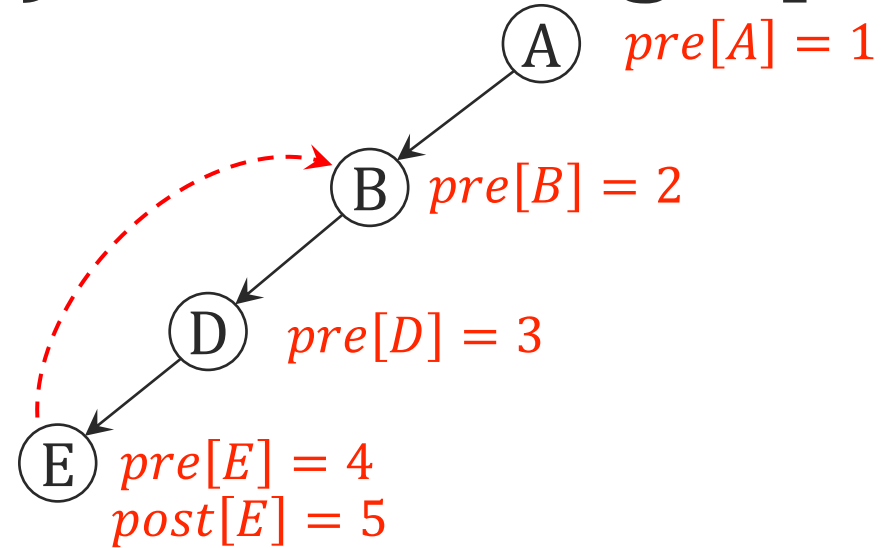
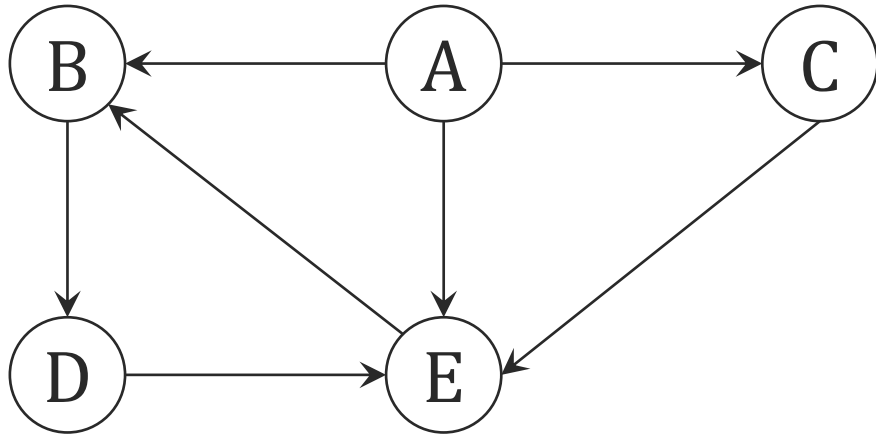
clock = 1

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

$visited[u] = true$

$pre[u] = clock; clock++$

For v such that $\{u, v\} \in E$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

$post[u] = clock; clock++$

dfs(G)

boolean array $visited(n)$

// initialize to all false.

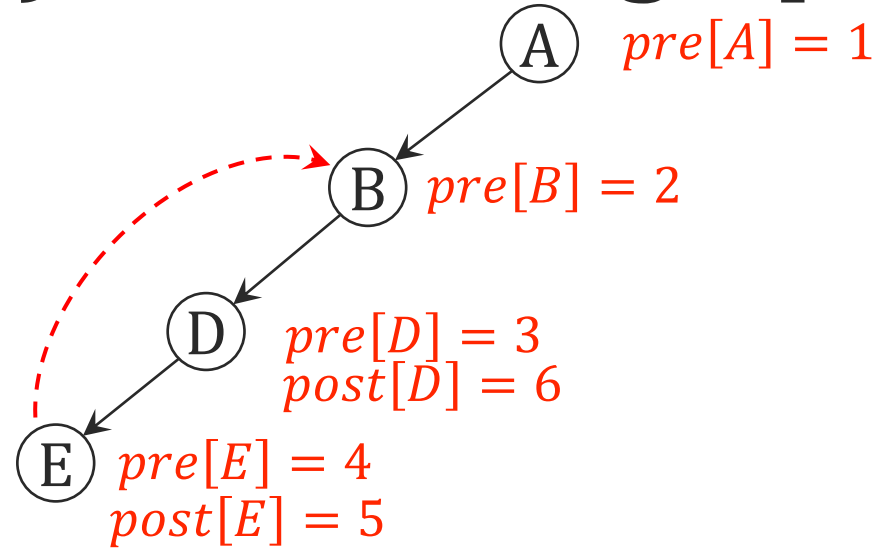
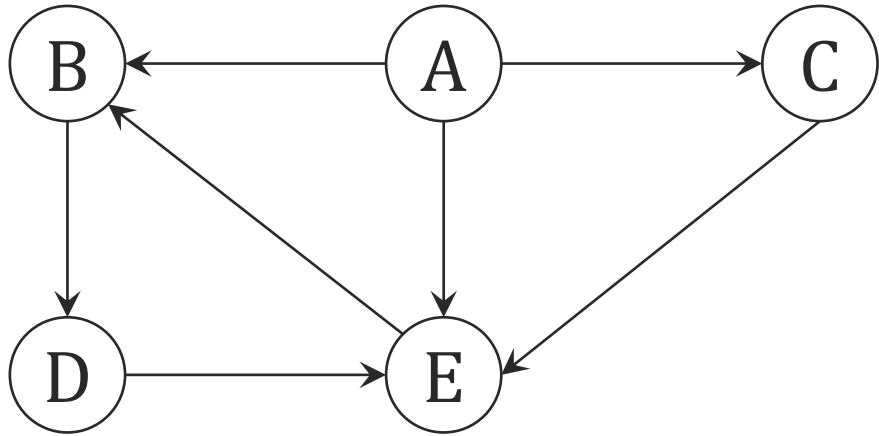
$clock = 1$

int array $pre(n), post(n)$

For $v \in V$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

pre[u] = *clock*; *clock*++

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

post[u] = *clock*; *clock*++

dfs(G)

boolean array *visited*(n)

// initialize to all false.

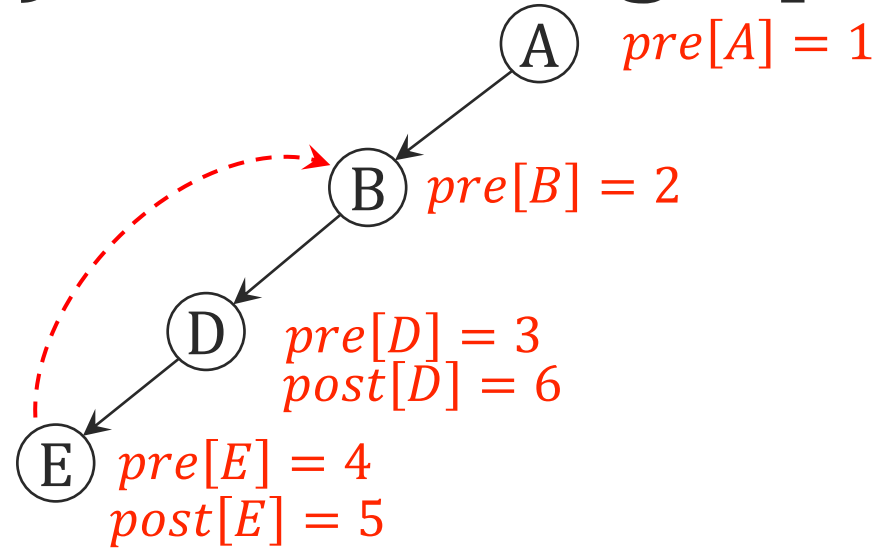
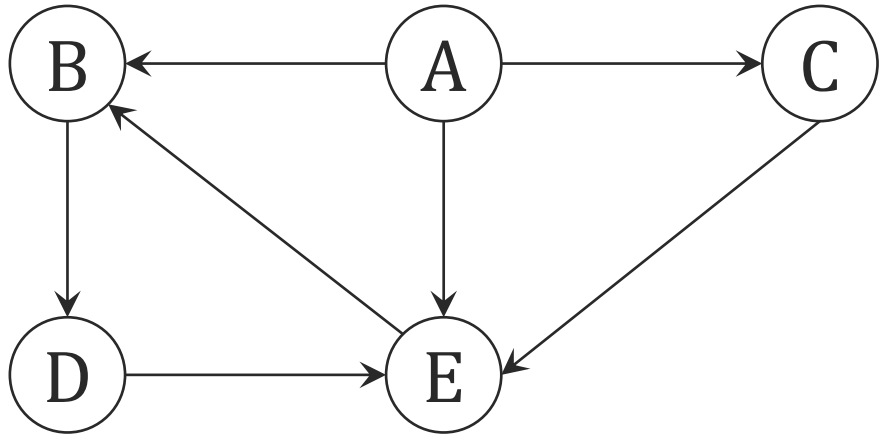
clock = 1

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

pre[u] = *clock*; *clock*++

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

post[u] = *clock*; *clock*++

dfs(G)

boolean array *visited*(n)

// initialize to all false.

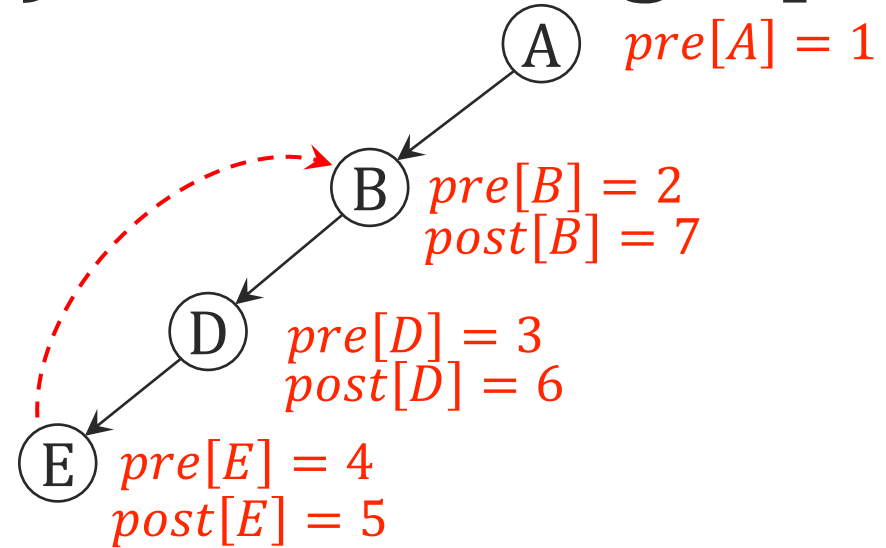
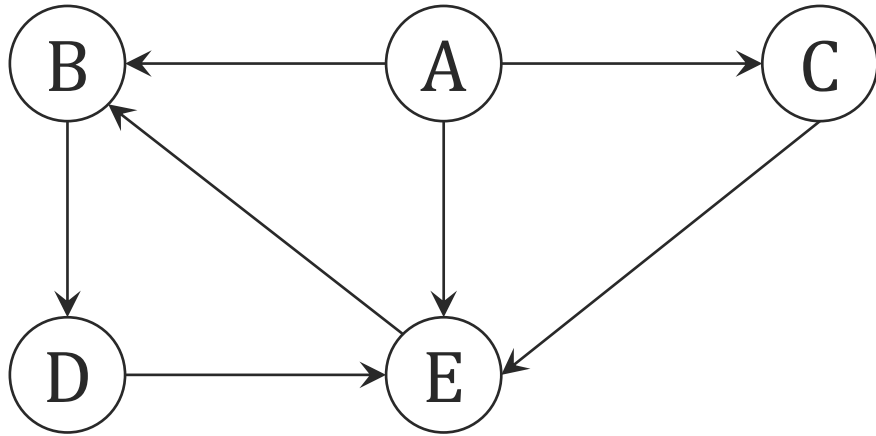
clock = 1

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

pre[u] = *clock*; *clock*++

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

post[u] = *clock*; *clock*++

dfs(G)

boolean array *visited*(n)

// initialize to all false.

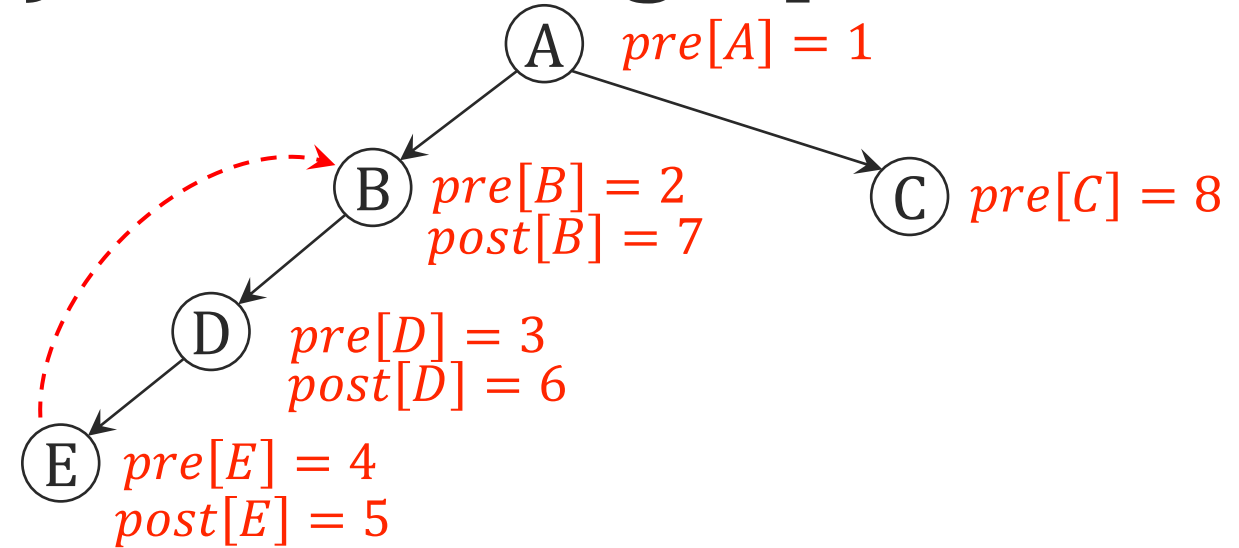
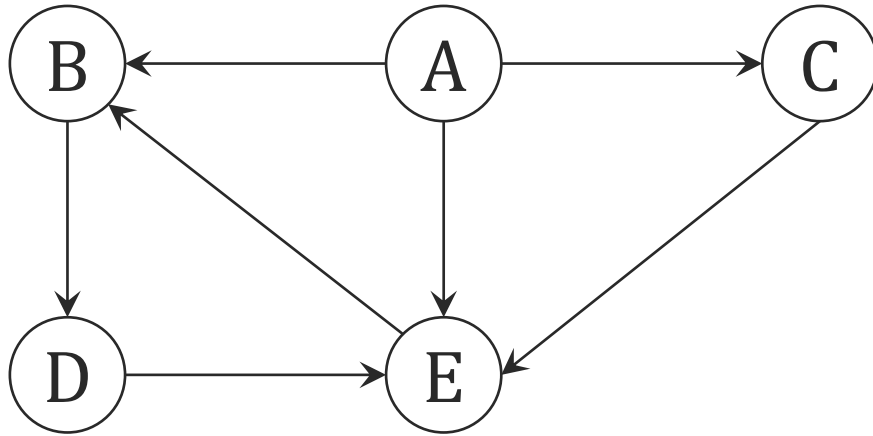
clock = 1

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = true

pre[u] = *clock*; *clock*++

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = false then *explore*(G, v)

post[u] = *clock*; *clock*++

dfs(G)

boolean array *visited*(n)

// initialize to all false.

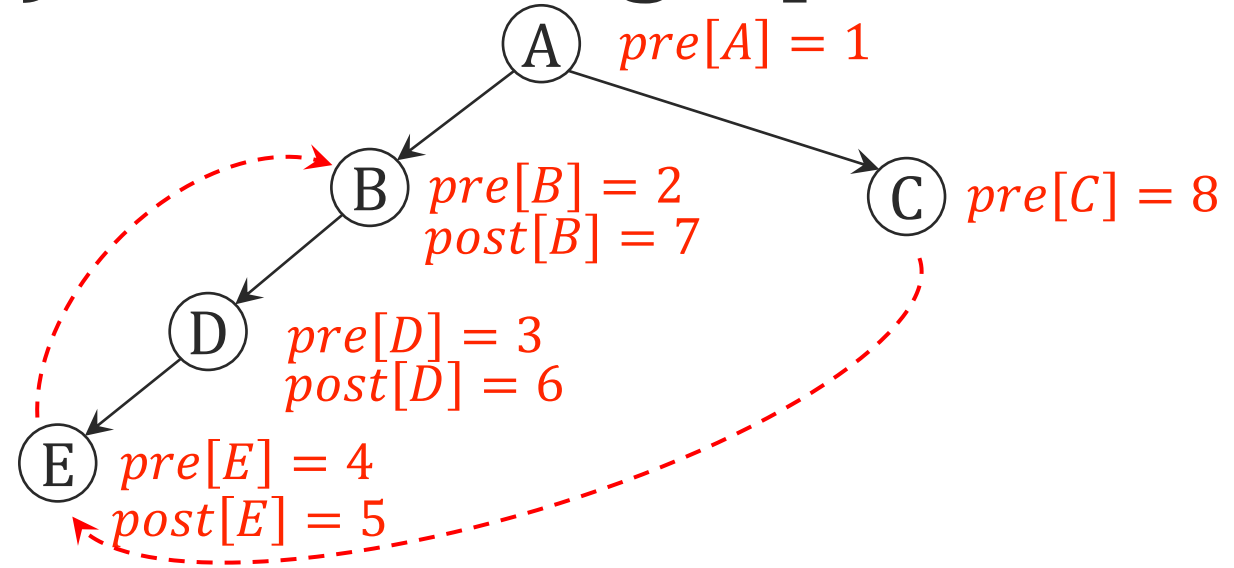
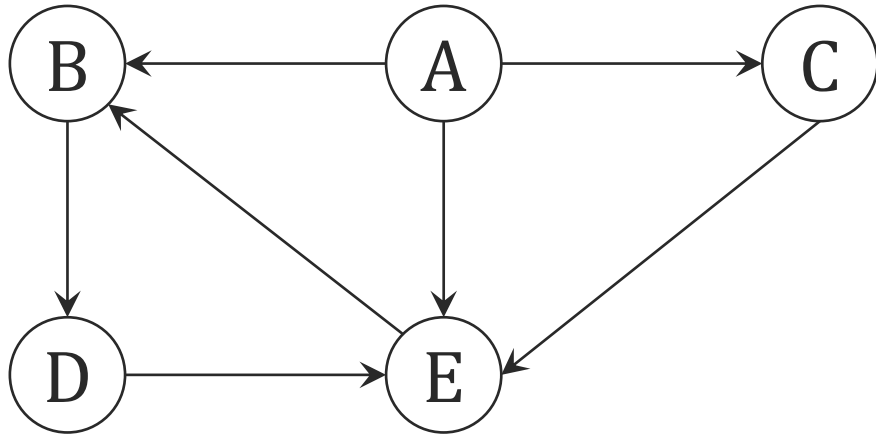
clock = 1

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = false then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

$visited[u] = true$

$pre[u] = clock; clock++$

For v such that $\{u, v\} \in E$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

$post[u] = clock; clock++$

dfs(G)

boolean array $visited(n)$

// initialize to all false.

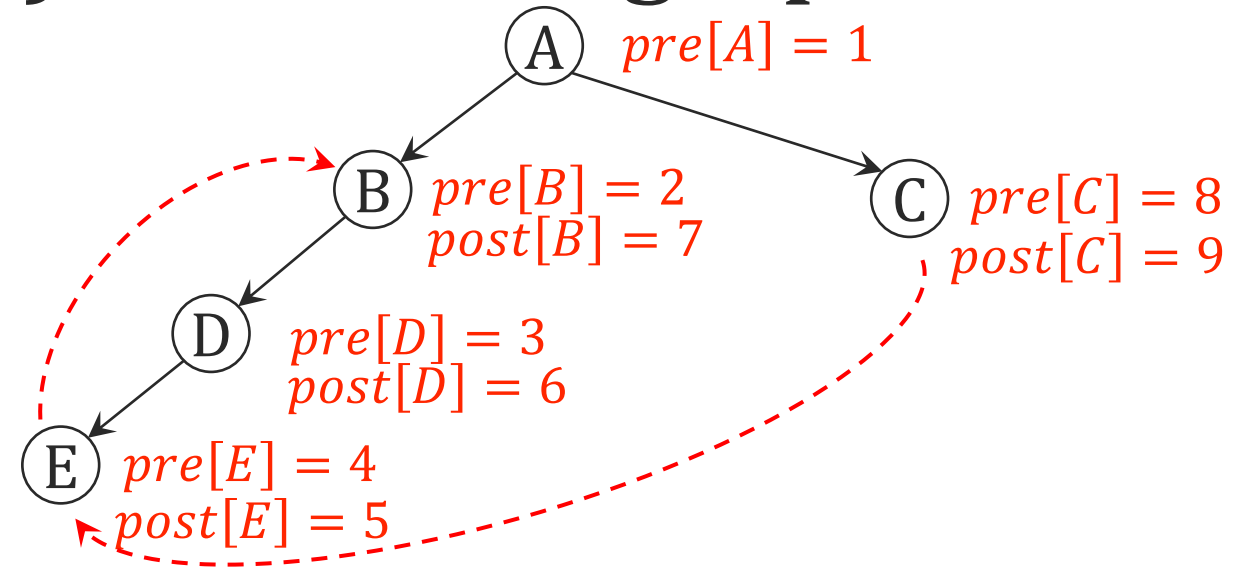
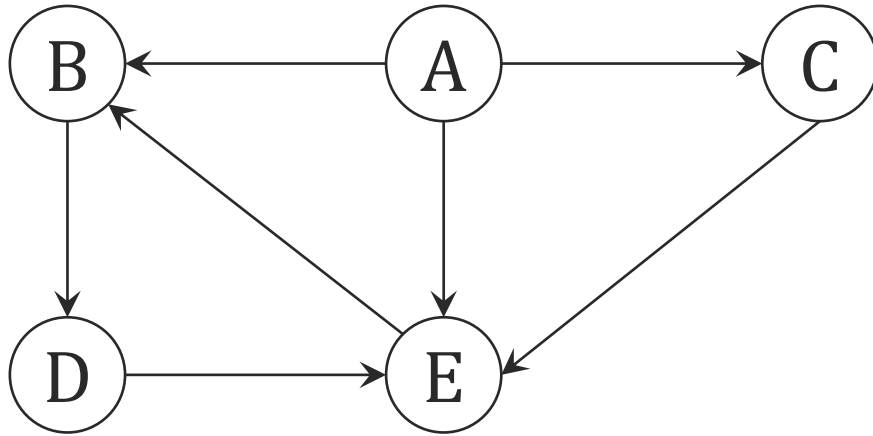
$clock = 1$

int array $pre(n), post(n)$

For $v \in V$ //alphabetic order

If $visited[v] = false$ then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

pre[u] = *clock*; *clock*++

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

post[u] = *clock*; *clock*++

dfs(G)

boolean array *visited*(n)

// initialize to all false.

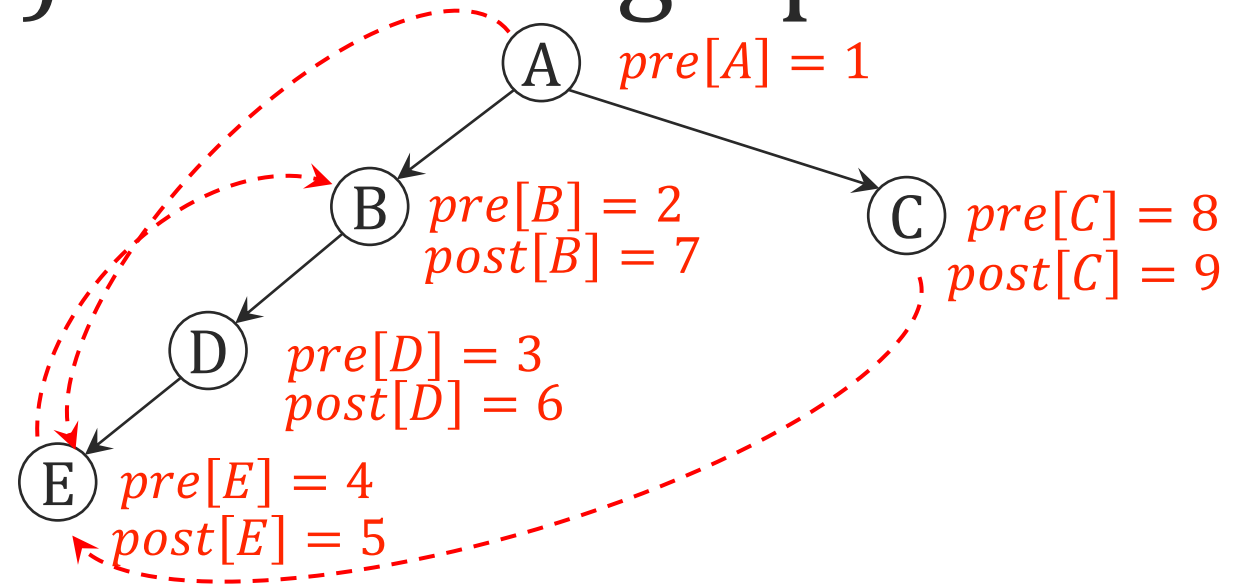
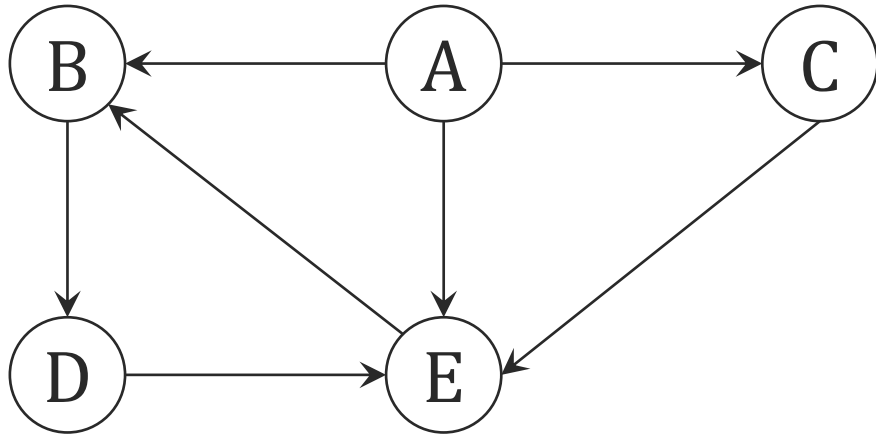
clock = 1

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

$pre[u] = clock; clock++$

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

$post[u] = clock; clock++$

dfs(G)

boolean array *visited*(n)

// initialize to all false.

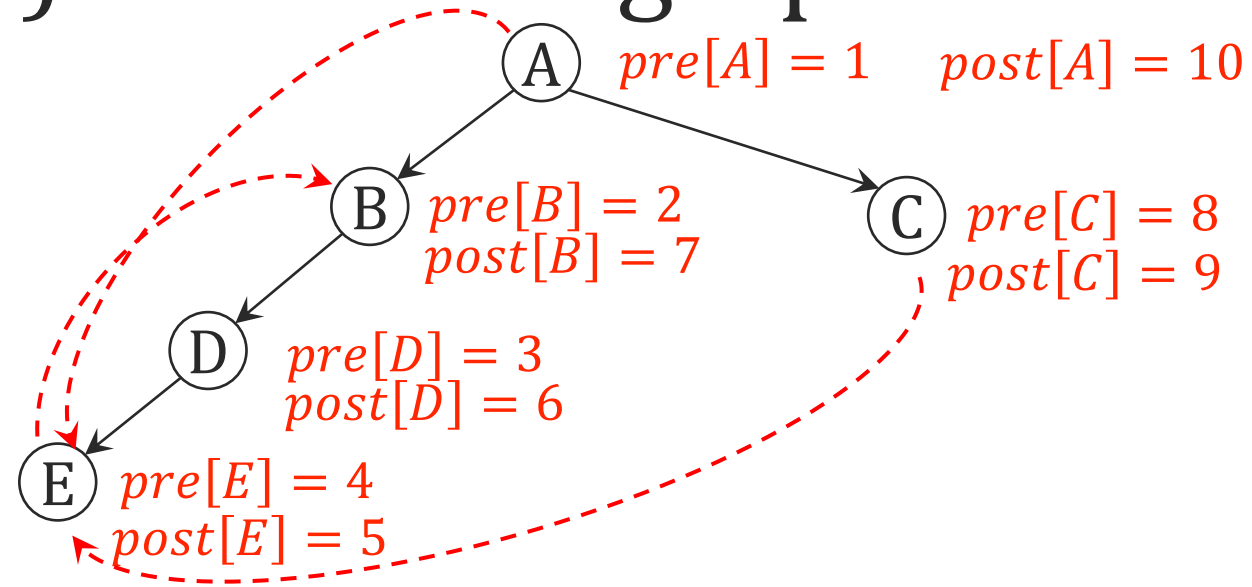
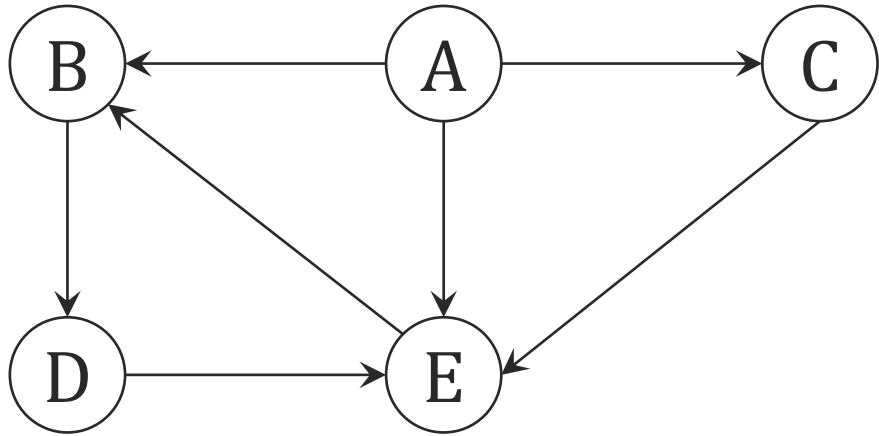
$clock = 1$

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

Depth-First Search (DFS) – directed graphs



explore(G, u)

visited[u] = *true*

$pre[u] = clock; clock++$

For v such that $\{u, v\} \in E$ //alphabetic order

If *visited*[v] = *false* then *explore*(G, v)

$post[u] = clock; clock++$

dfs(G)

boolean array *visited*(n)

// initialize to all false.

$clock = 1$

int array *pre*(n), *post*(n)

For $v \in V$ //alphabetic order

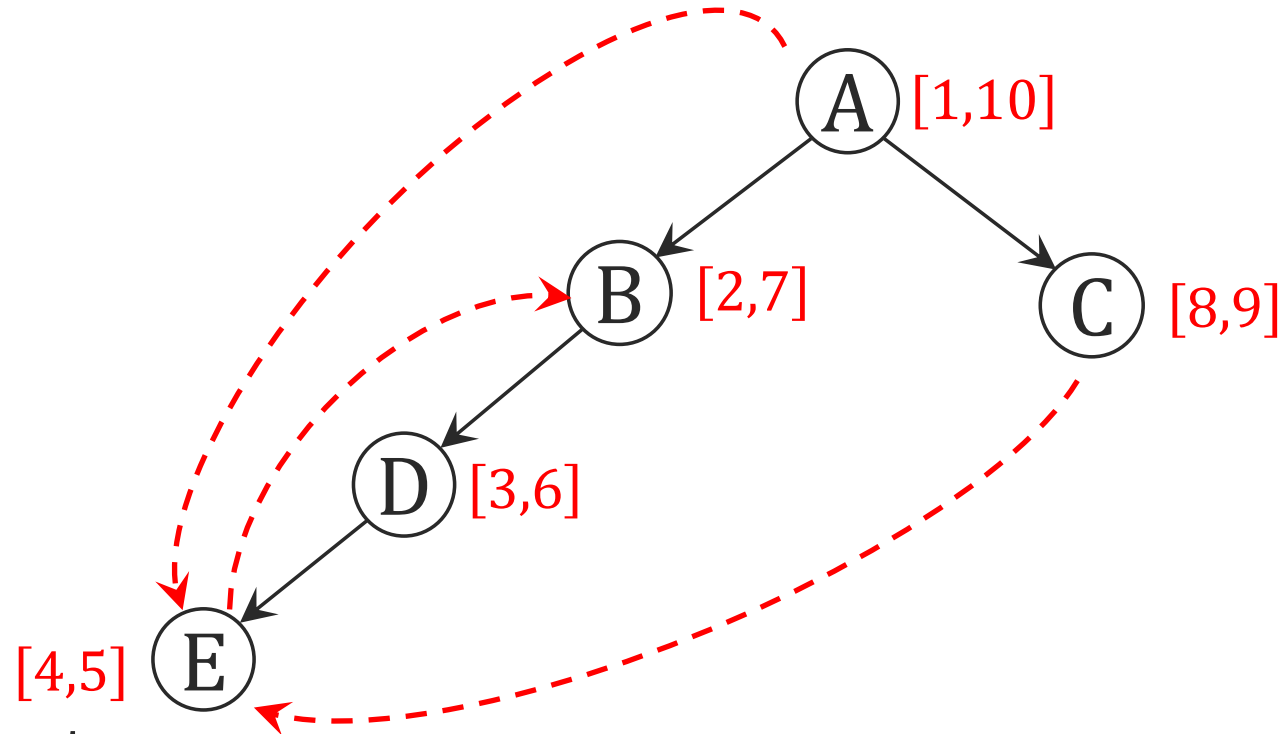
If *visited*[v] = *false* then *explore*(G, v)

DFS Tree/Forest for Directed Graphs

More concise notations: Interval $[pre[u], post[u]]$.

Types of edges:

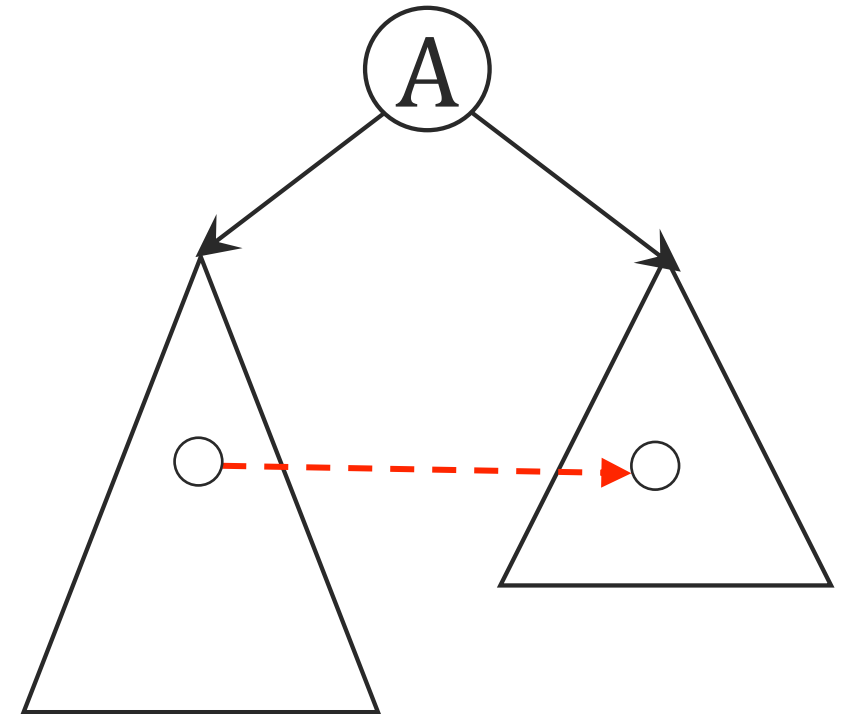
- Tree edge
Recursive **explore** calls
- **Back edge:**
From descendent to ancestor
- **Forward edge:**
From ancestor to non-child descendent
- **Cross edge:**
Between neither descendent or ancestor



More on Cross Edges

We saw that a **cross edge** can go from the “right” (later) branch to the “left” (earlier branch).

Can a **cross edge** go from the “left” (earlier) branch to the “right” (later) branch.



Tree and Forward Edges

Imagine $(u, v) \in E$ is a tree edge or a forward edge.

What is the relationship between $[pre[v], post[v]]$ and $[pre[u], post[u]]$?

$$pre[u] < pre[v] < post[v] < post[u]$$

Back Edges

Imagine $(u, v) \in E$ is a back edge.

What is the relationship between $[pre[v], post[v]]$ and $[pre[u], post[u]]$?

$$pre[v] < pre[u] < post[u] < post[v]$$

Cross Edge

Imagine $(u, v) \in E$ is a cross edge.

What is the relationship between $[pre[v], post[v]]$ and $[pre[u], post[u]]$?

$$pre[v] < post[v] < pre[u] < post[u]$$


Edges Types and Intervals Summary

Edge $(u, v) \in E$	
Tree / Forward edge	$pre[u] < pre[v] < post[v] < post[u]$
Back edge:	$pre[v] < pre[u] < post[u] < post[v]$
Cross edge:	$pre[v] < post[v] < pre[u] < post[u]$

All other relationships between intervals are impossible!

Back Edges and Post-times are special!

Edge $(u, v) \in E$	
Tree / Forward edge	$pre[u] < pre[v] < post[v] < post[u]$
Back edge:	$pre[v] < pre[u] < post[u] < post[v]$
Cross edge:	$pre[v] < post[v] < pre[u] < post[u]$



How to detect a back edge?

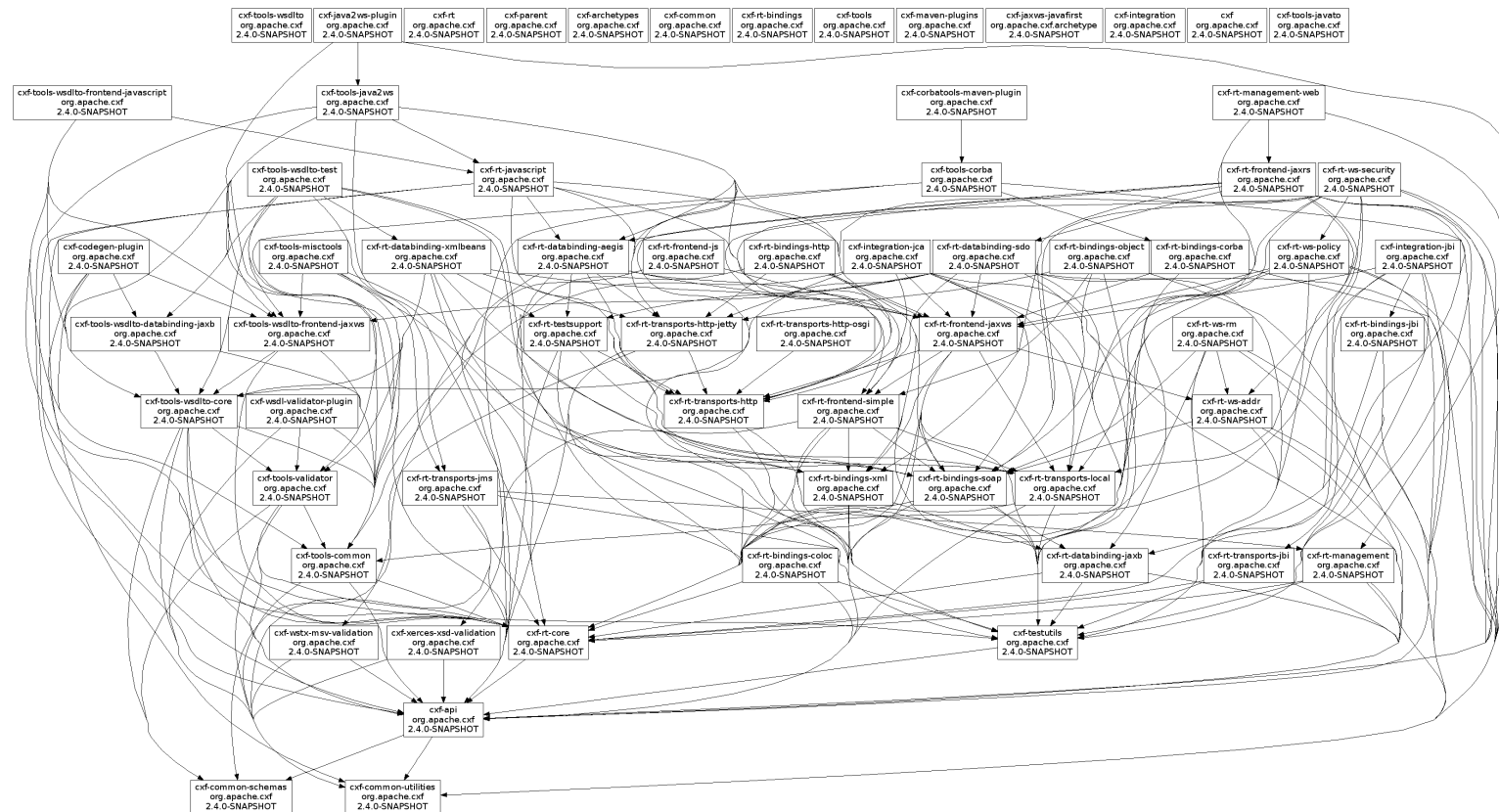
→ An edge $(u, v) \in E$ is a back edge if and only if $post[u] < post[v]$.

Back Edges and Topological Sort

Find an ordering of vertices so that no edges go backward.

→ i.e., If u comes before v in the ordering, there is no edge (v, u) .

E.g., software package dependency



Topological Sort and DAGS

Definition: a directed acyclic graph (DAG) is a graph with no directed cycles.

Claim: Suppose we run a DFS on on G. G is a **DAG** if and only if it has **no back edges**!

Topological Sort and Back Edges

In what order should I install packages?

- The vertex that finishes later (higher post number) is higher up in the DFS tree (or in a later branch).
- Install in the reverse order of the post-times of the vertices.
- Verify this at home!



Wrap up

Graphs are awesome!

DFS is useful!

- Edge types are important
- Simple book keeping tells us about edge types too.

Next time

- More with graphs
- Paths and strongly connected components