CMPSC 465: LECTURE XII

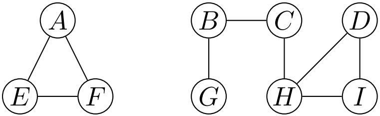
DFS on Directed Graphs

Ke Chen

September 26, 2025

Recall from the previous lecture ...

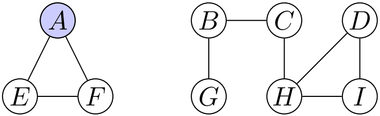
On an undirected graph:



Recall from the previous lecture ...

On an undirected graph:

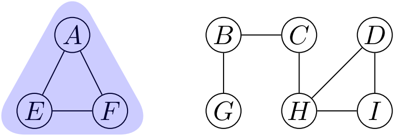
- Explore a node:



Recall from the previous lecture ...

On an undirected graph:

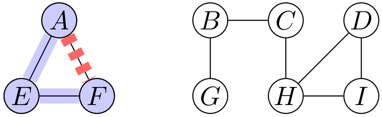
- Explore a node:
  - reveals one connected component



Recall from the previous lecture ...

On an undirected graph:

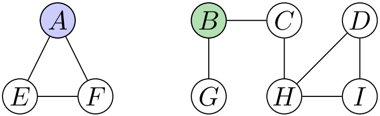
- Explore a node:
  - reveals one connected component
  - produces a DFS tree $\rightsquigarrow$ tree edges, back edges



Recall from the previous lecture ...

On an undirected graph:

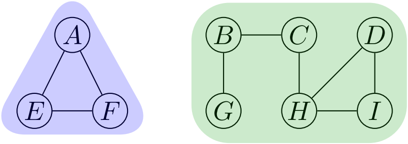
- Explore a node:
  - reveals one connected component
  - produces a DFS tree $\rightsquigarrow$ tree edges, back edges
- DFS repeatedly calls Explore:



Recall from the previous lecture ...

On an undirected graph:

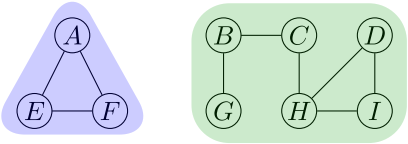
- Explore a node:
  - reveals one connected component
  - produces a DFS tree $\rightsquigarrow$ tree edges, back edges
- DFS repeatedly calls Explore:
  - finds all connected components



Recall from the previous lecture ...

On an undirected graph:

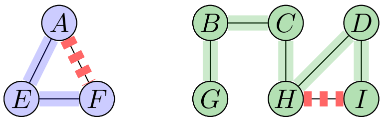
- Explore a node:
  - reveals one connected component
  - produces a DFS tree $\rightsquigarrow$ tree edges, back edges
- DFS repeatedly calls Explore:
  - finds all connected components $\rightsquigarrow$ "Can I go from A to B?"



Recall from the previous lecture ...

On an undirected graph:

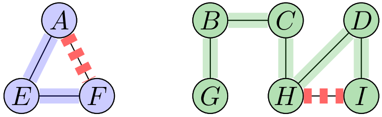
- Explore a node:
  - reveals one connected component
  - produces a DFS tree ⇝ tree edges, back edges
- DFS repeatedly calls Explore:
  - finds all connected components ⇝ "Can I go from A to B?"
  - yields a DFS forest



Recall from the previous lecture ...

On an undirected graph:

- Explore a node:
  - reveals one connected component
  - produces a DFS tree $\rightsquigarrow$ tree edges, back edges
- DFS repeatedly calls Explore:
  - finds all connected components $\rightsquigarrow$ "Can I go from A to B?"
  - yields a DFS forest $\rightsquigarrow$ cycle detection



Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

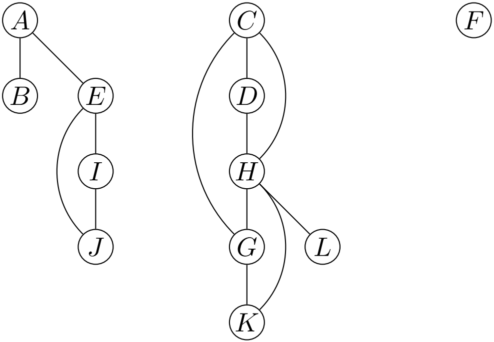
```
// pre and post are integer arrays of size |V|
// clock is a global integer counter starting at 1
Explore(G, s, color)
    visited[s] = color
    pre[s] = clock
    clock = clock + 1
    foreach edge {s, v} ∈ E do
        if visited[v] == 0 then
            Explore(G, v, color)
    post[s] = clock
    clock = clock + 1
```

Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

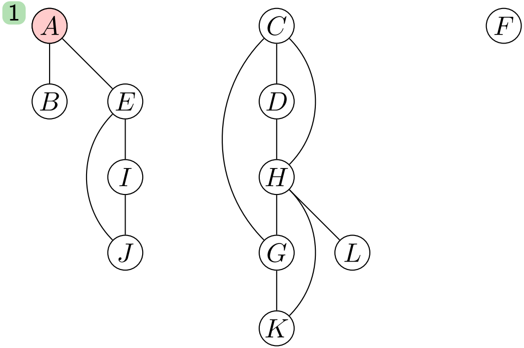


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

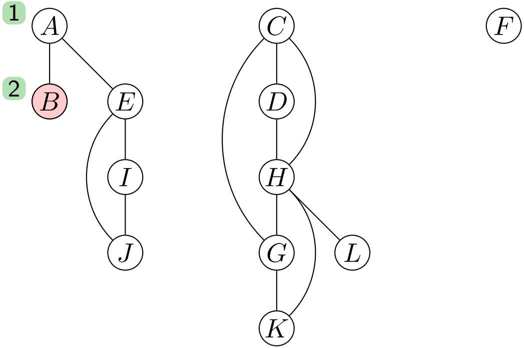


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

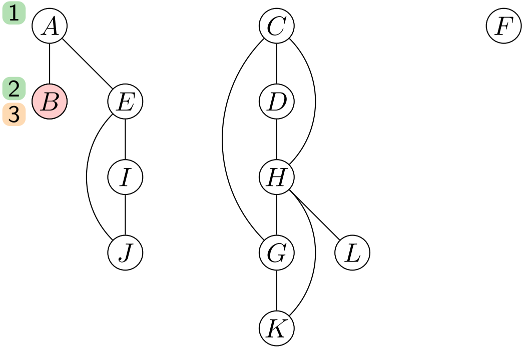


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

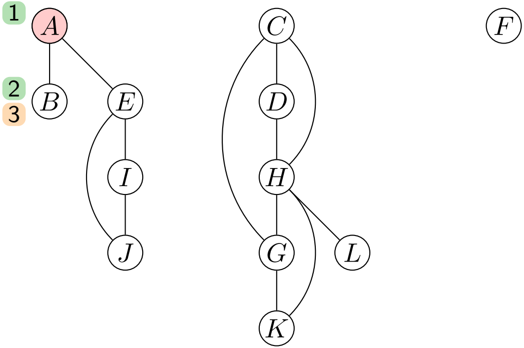


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

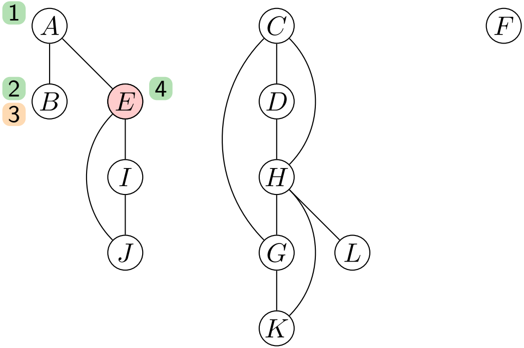


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

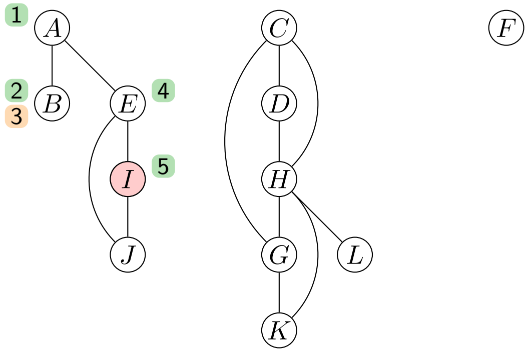


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

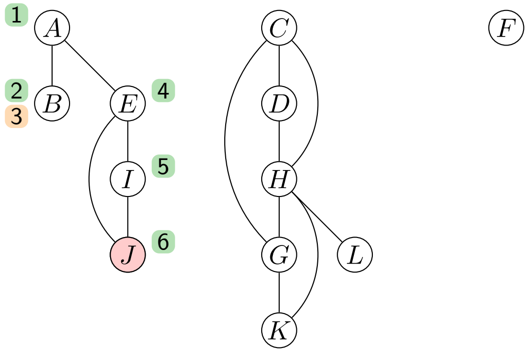


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

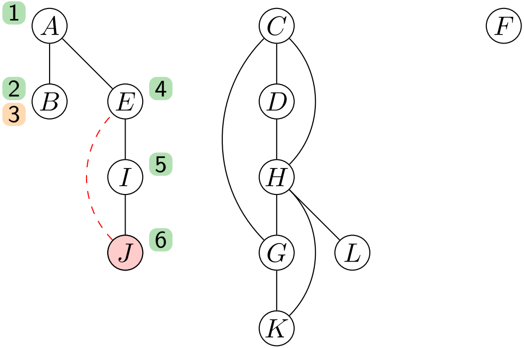


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

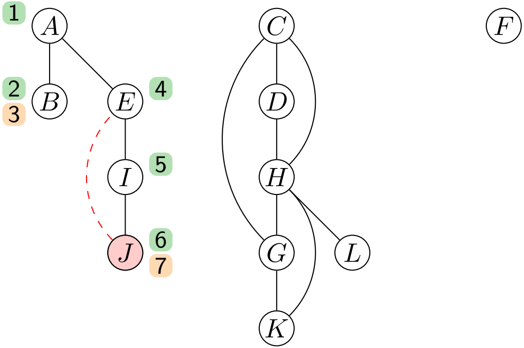


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

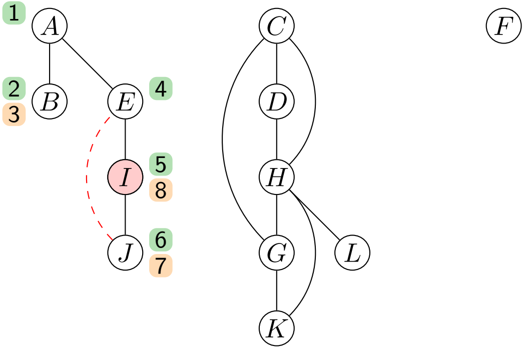


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

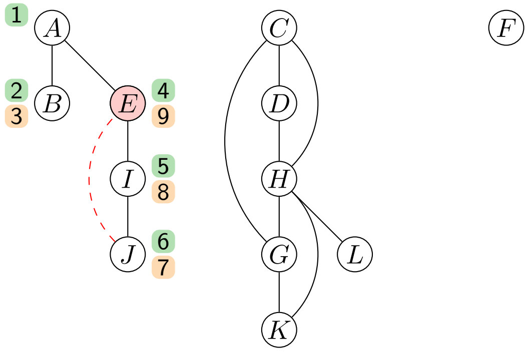


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

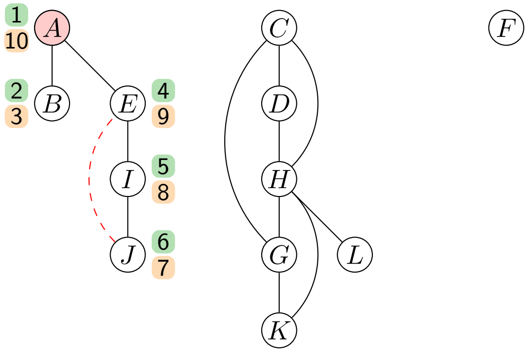


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

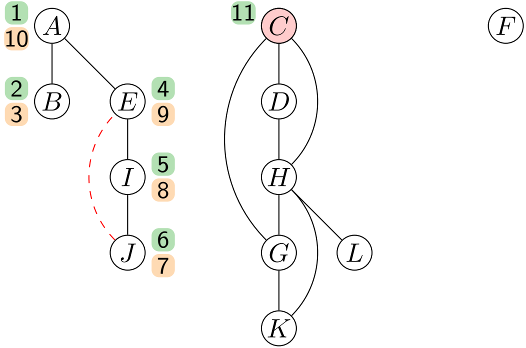


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

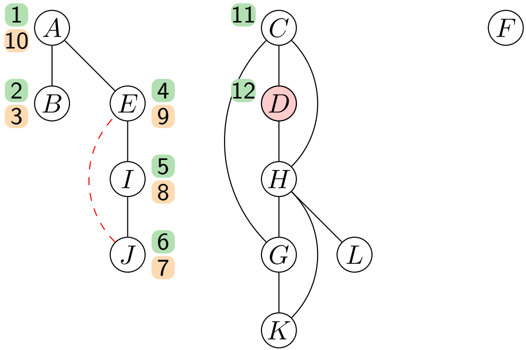


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

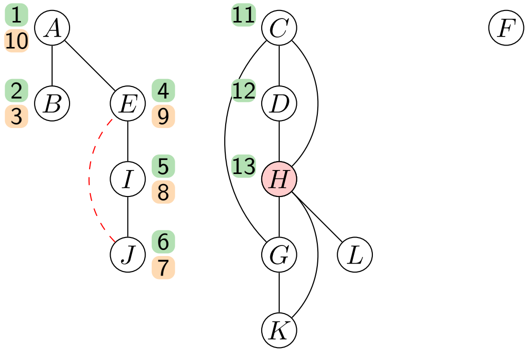


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

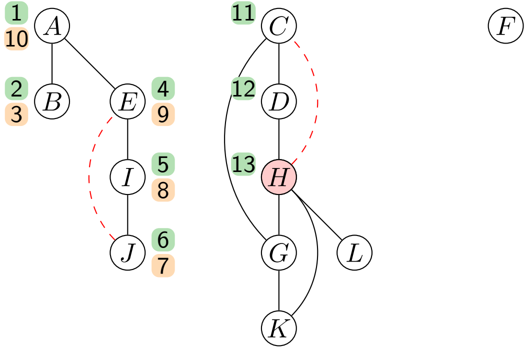


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

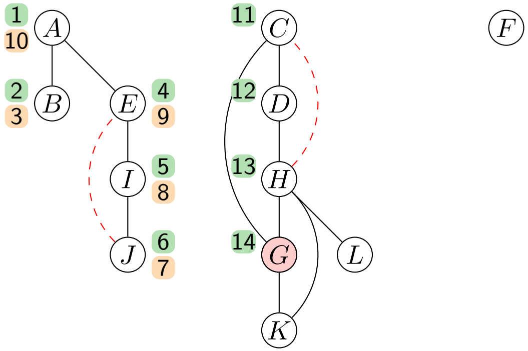


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

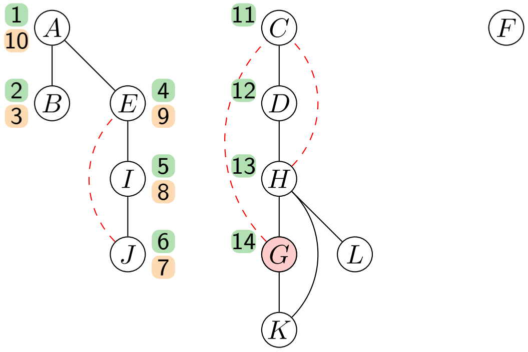


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

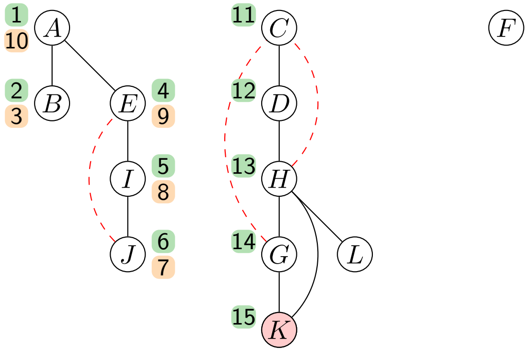


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

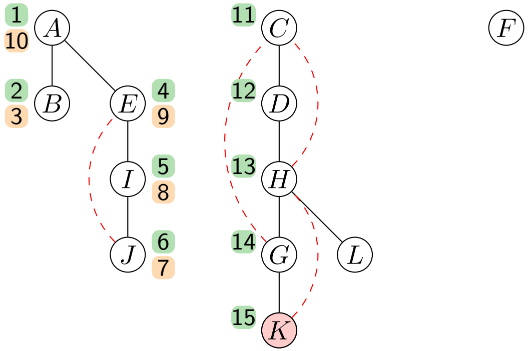


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

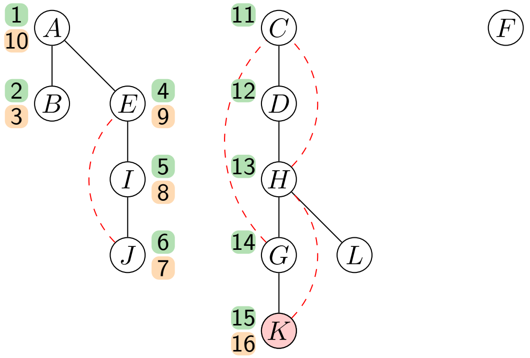


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

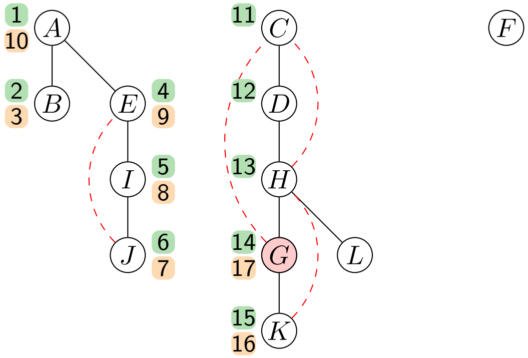


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

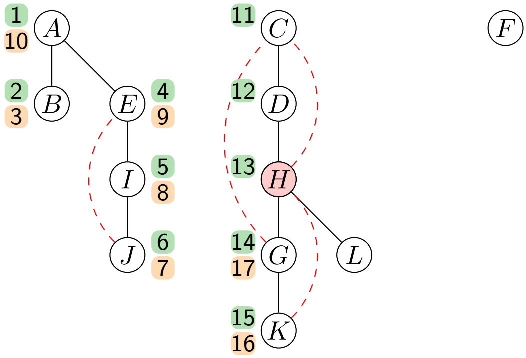


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

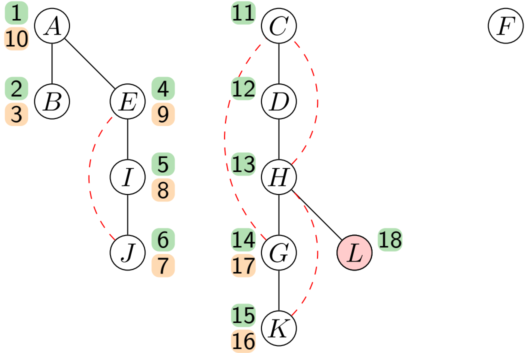


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

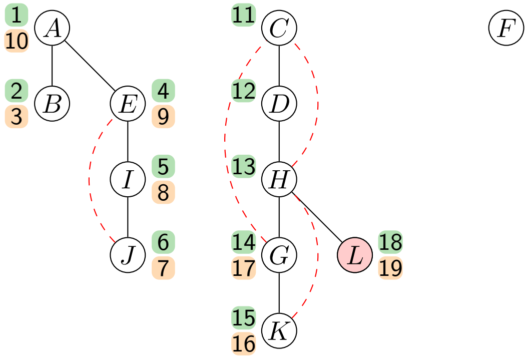


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

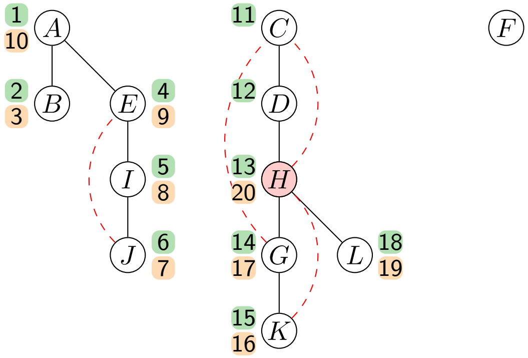


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

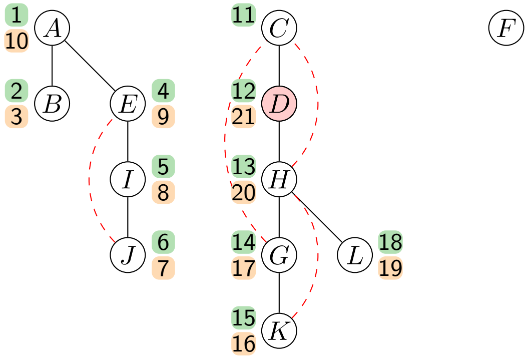


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

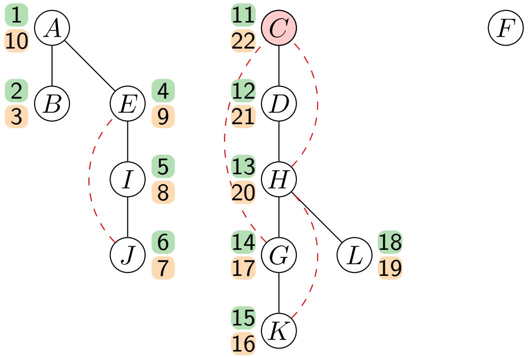


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

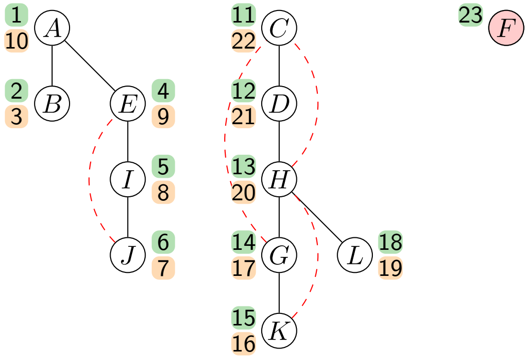


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

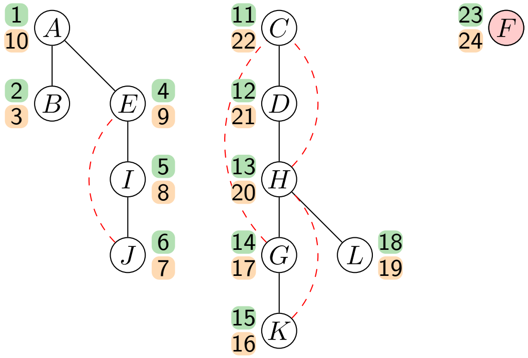


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

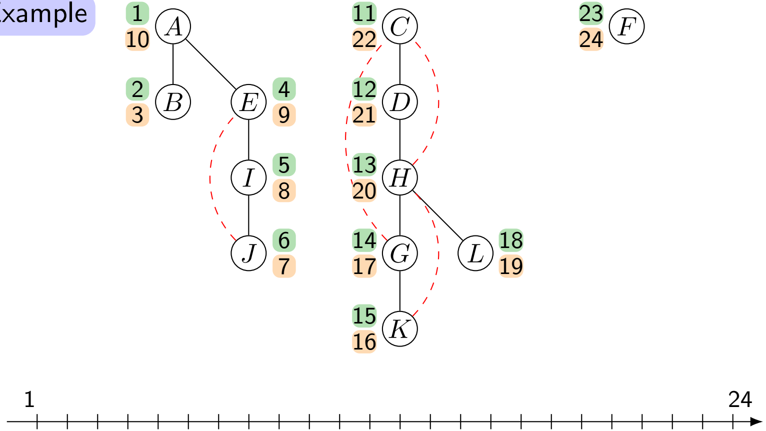


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

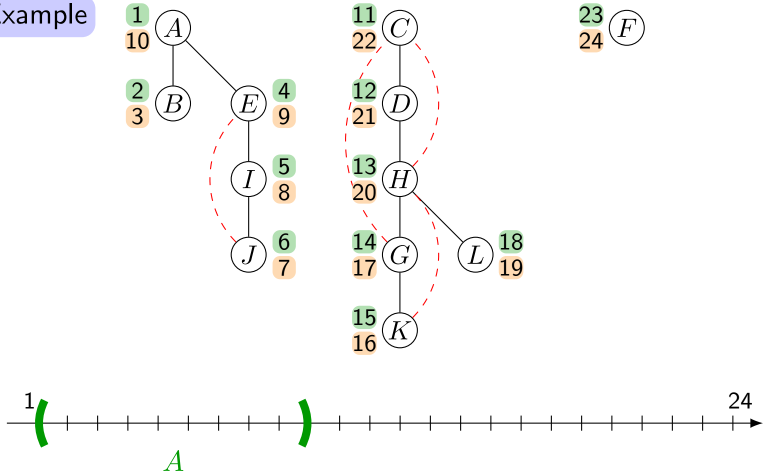


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

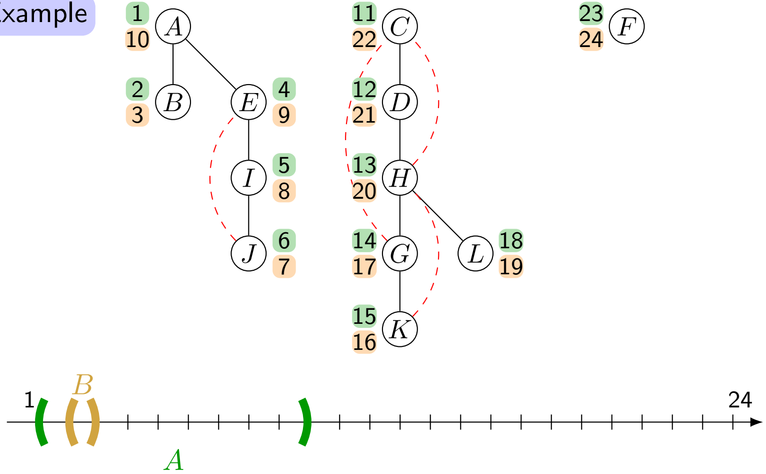


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

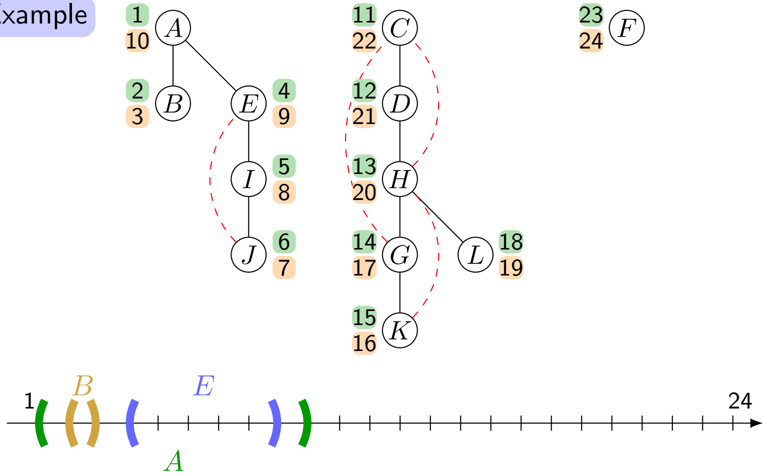


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

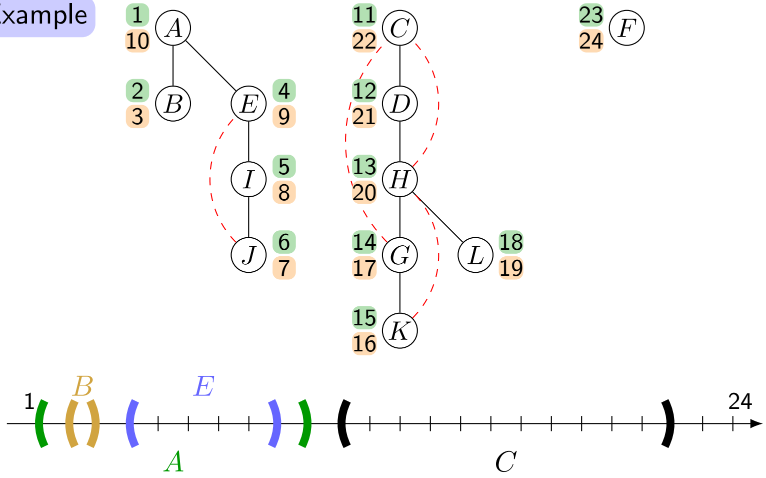


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

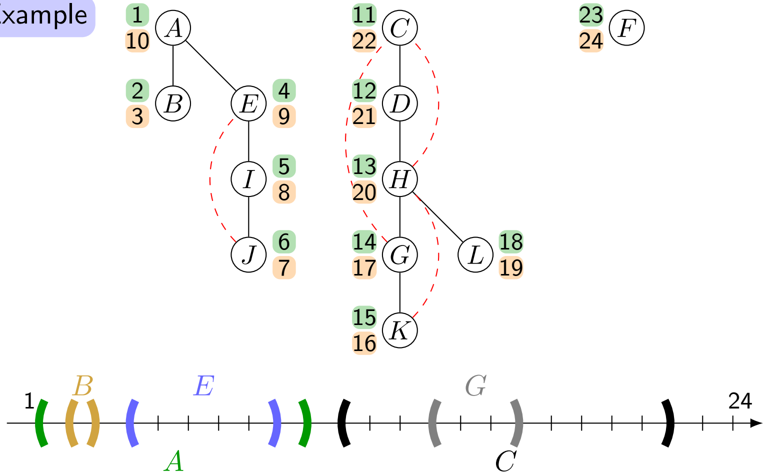


Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

- visit a node for the first time, and
- leave a node for good.

Example

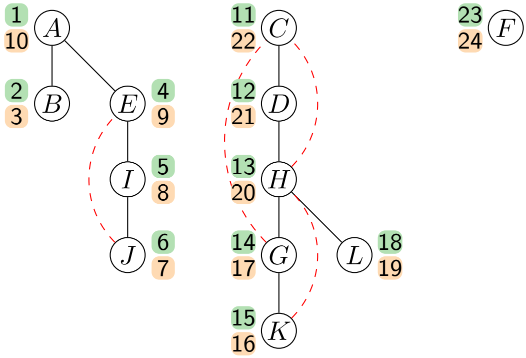


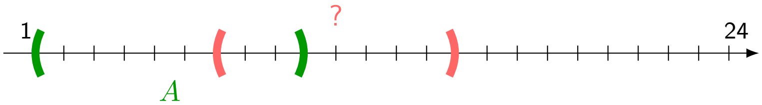
Pre- and post-visit timestamps (undirected)

We can collect more information during Explore by keeping a global clock that ticks every time we:

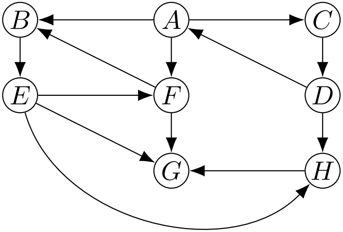
- visit a node for the first time, and
- leave a node for good.

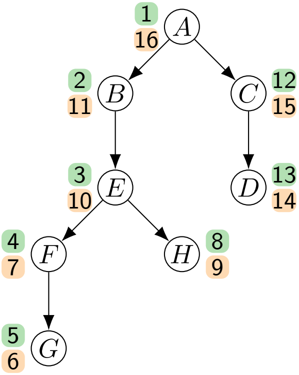
Example





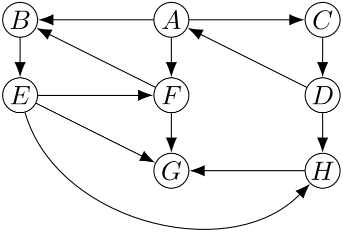
DFS in directed graphs

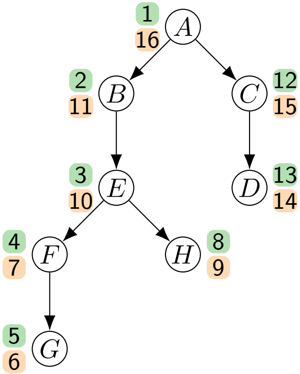




DFS in directed graphs

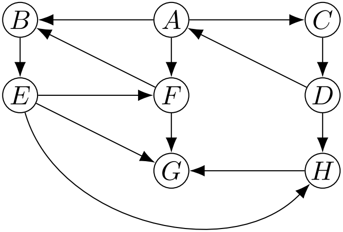
Exactly the same algorithm works!

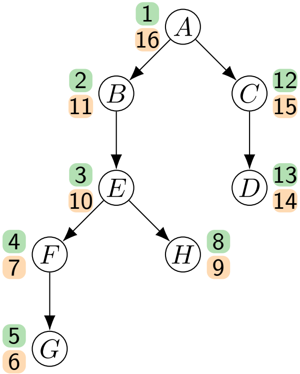




DFS in directed graphs

Exactly the same algorithm works!

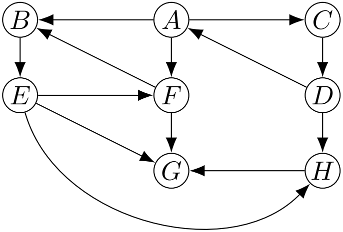


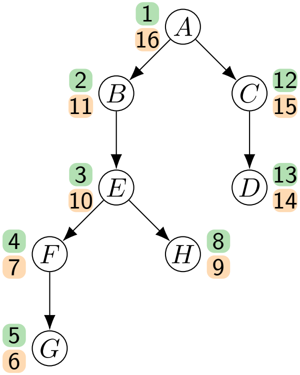


DFS in directed graphs

Exactly the same algorithm works!

But we have more edge types:



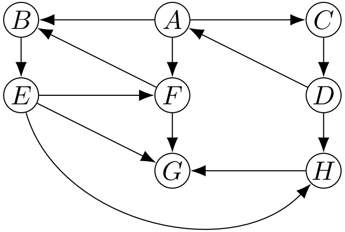


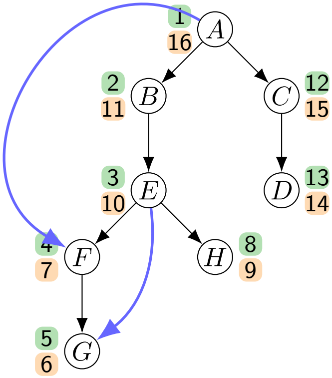
DFS in directed graphs

Exactly the same algorithm works!

But we have more edge types:

- **forward edges** lead to a non-child descendant.



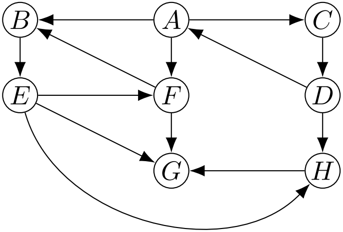


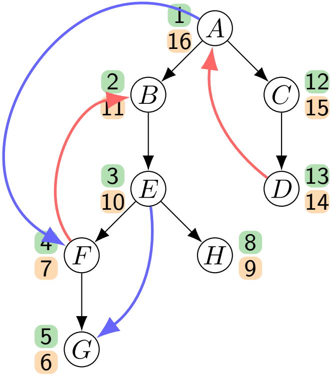
DFS in directed graphs

Exactly the same algorithm works!

But we have more edge types:

- forward edges lead to a non-child descendant.
- back edges lead to an ancestor.



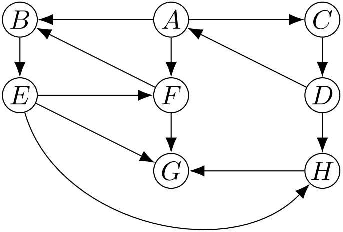


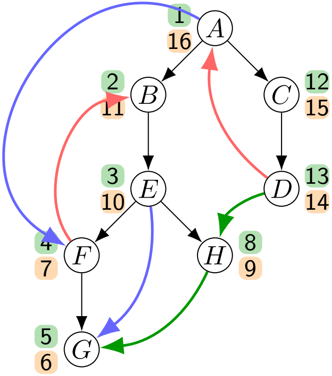
DFS in directed graphs

Exactly the same algorithm works!

But we have more edge types:

- forward edges lead to a non-child descendant.
- back edges lead to an ancestor.
- cross edges lead to neither an ancestor nor a descendant.

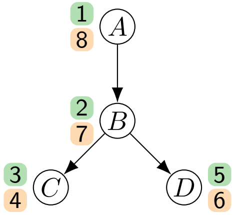




Pre-, post-visit numbers and edge types

Fact If vertex w is an ancestor of vertex v in the DFS tree, then

$$pre[w] < pre[v] < post[v] < post[w]$$



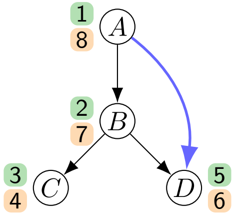
Pre-, post-visit numbers and edge types

Fact If vertex w is an ancestor of vertex v in the DFS tree, then

$$pre[w] < pre[v] < post[v] < post[w]$$

For an edge (w, v) in the graph:

- if $pre[w] < pre[v] < post[v] < post[w]$, then (w, v) is a tree or forward edge.



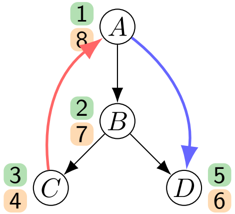
Pre-, post-visit numbers and edge types

Fact If vertex w is an ancestor of vertex v in the DFS tree, then

$$pre[w] < pre[v] < post[v] < post[w]$$

For an edge (w, v) in the graph:

- if $pre[w] < pre[v] < post[v] < post[w]$, then (w, v) is a tree or forward edge.
- if $pre[v] < pre[w] < post[w] < post[v]$, then (w, v) is a back edge.



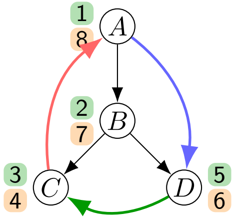
Pre-, post-visit numbers and edge types

Fact If vertex w is an ancestor of vertex v in the DFS tree, then

$$pre[w] < pre[v] < post[v] < post[w]$$

For an edge (w, v) in the graph:

- if $pre[w] < pre[v] < post[v] < post[w]$, then (w, v) is a tree or forward edge.
- if $pre[v] < pre[w] < post[w] < post[v]$, then (w, v) is a back edge.
- if $pre[v] < post[v] < pre[w] < post[w]$, then (w, v) is a cross edge.



Cycle detection (directed)

Fact A directed graph has a cycle if and only if its DFS forest has a back edge.

Cycle detection (directed)

Fact A directed graph has a cycle if and only if its DFS forest has a back edge.

Algorithm for cycle detection

1. Run DFS and assign pre and $post$ numbers.
2. Iterate through all edges (w, v) and check if $pre[v] < pre[w] < post[w] < post[v]$.
3. If found, output "found cycle"; otherwise return "no cycle".

Cycle detection (directed)

Fact A directed graph has a cycle if and only if its DFS forest has a back edge.

Algorithm for cycle detection

1. Run DFS and assign pre and $post$ numbers.
2. Iterate through all edges (w, v) and check if $pre[v] < pre[w] < post[w] < post[v]$.
3. If found, output "found cycle"; otherwise return "no cycle".

Time complexity?

Topological sort

A directed graph without cycles is called a DAG (directed acyclic graph).

Topological sort

A directed graph without cycles is called a DAG (directed acyclic graph).

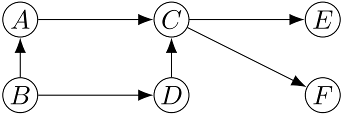
DAGs are very useful and are much easier to work with, because we can order the vertices such that all edges point from an earlier vertex to a later vertex.

Topological sort

A directed graph without cycles is called a DAG (directed acyclic graph).

DAGs are very useful and are much easier to work with, because we can order the vertices such that all edges point from an earlier vertex to a later vertex.

Example

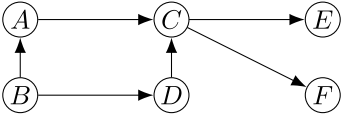


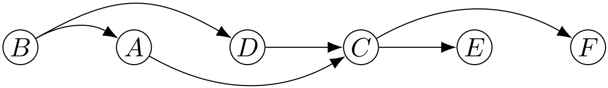
Topological sort

A directed graph without cycles is called a DAG (directed acyclic graph).

DAGs are very useful and are much easier to work with, because we can order the vertices such that all edges point from an earlier vertex to a later vertex.

Example



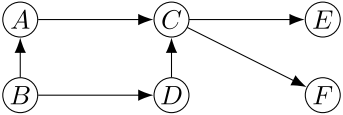


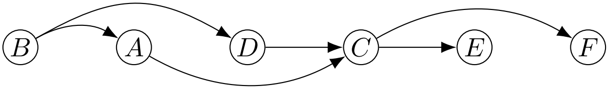
Topological sort

A directed graph without cycles is called a DAG (directed acyclic graph).

DAGs are very useful and are much easier to work with, because we can order the vertices such that all edges point from an earlier vertex to a later vertex.

Example





Such an ordering is called a topological order or a linearization of the DAG.

Topological sort

Fact All DAGs have at least one topological order.

Topological sort

Fact All DAGs have at least one topological order.

Fact In a DAG, every edge leads to a vertex with a smaller post number.

Topological sort

Fact All DAGs have at least one topological order.

Fact In a DAG, every edge leads to a vertex with a smaller post number.
Proof: DAGs do not have back edges, all other edge types satisfies this.

Topological sort

Fact All DAGs have at least one topological order.

Fact In a DAG, every edge leads to a vertex with a smaller post number.
Proof: DAGs do not have back edges, all other edge types satisfies this.

Algorithm for topological sort

1. Run DFS and assign pre and $post$ numbers.
2. Sort vertices in decreasing order of post numbers.

Topological sort

Fact All DAGs have at least one topological order.

Fact In a DAG, every edge leads to a vertex with a smaller post number.
Proof: DAGs do not have back edges, all other edge types satisfies this.

Algorithm for topological sort

1. Run DFS and assign pre and $post$ numbers.
2. Sort vertices in decreasing order of post numbers.

Time complexity?

Topological sort

Fact All DAGs have at least one topological order.

Fact In a DAG, every edge leads to a vertex with a smaller post number.
Proof: DAGs do not have back edges, all other edge types satisfies this.

Algorithm for topological sort

1. Run DFS and assign pre and $post$ numbers.
2. Sort vertices in decreasing order of post numbers.

Time complexity? $O(|V|\log|V| + |E|)$.

Topological sort

Fact All DAGs have at least one topological order.

Fact In a DAG, every edge leads to a vertex with a smaller post number.
Proof: DAGs do not have back edges, all other edge types satisfies this.

Algorithm for topological sort

1. Run DFS and assign pre and $post$ numbers.
2. Sort vertices in decreasing order of post numbers.

Time complexity? $O(|V| \log |V| + |E|)$.

Can we do better?

Topological sort (faster)

Idea Sort at the same time as we assign post numbers.

Topological sort (faster)

Idea Sort at the same time as we assign post numbers.

```
// top_order will contain the ordered vertices
// idx = |V|
Explore(G, s, color)
    visited[s] = color
    pre[s] = clock,    clock = clock + 1
    foreach edge {s, v} ∈ E do
        if visited[v] == 0 then
            Explore(G, v, color)
    post[s] = clock,    clock = clock + 1
    top_order[idx] = s,    idx = idx − 1
```

Topological sort (faster)

Idea Sort at the same time as we assign post numbers.

```
// top_order will contain the ordered vertices
// idx = |V|
Explore(G, s, color)
    visited[s] = color
    pre[s] = clock,    clock = clock + 1
    foreach edge {s, v} ∈ E do
        if visited[v] == 0 then
            Explore(G, v, color)
    post[s] = clock,    clock = clock + 1
    top_order[idx] = s,    idx = idx − 1
```

Time complexity?

Topological sort (faster)

Idea Sort at the same time as we assign post numbers.

```
// top_order will contain the ordered vertices
// idx = |V|
Explore(G, s, color)
    visited[s] = color
    pre[s] = clock,    clock = clock + 1
    foreach edge {s, v} ∈ E do
        if visited[v] == 0 then
            Explore(G, v, color)
    post[s] = clock,    clock = clock + 1
    top_order[idx] = s,    idx = idx − 1
```

Time complexity? $O(|V| + |E|)$.