CMPSC 465: LECTURE XI

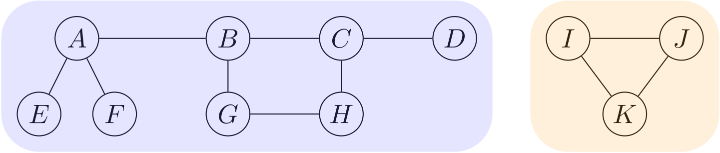
DFS on Undirected Graphs

Ke Chen

September 24, 2025

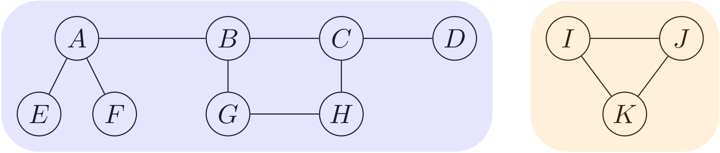
Connected component

Recall that the connected component of an undirected graph is defined as a maximal set of connected vertices.



Connected component

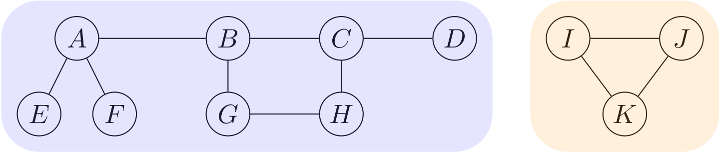
Recall that the connected component of an undirected graph is defined as a maximal set of connected vertices.



- Finding the connected components helps answer queries like "is node v connected to node w?".

Connected component

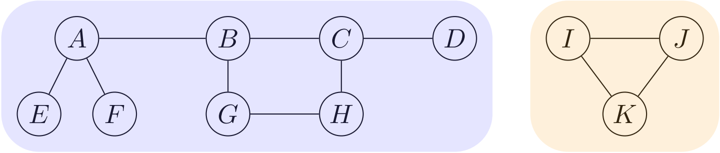
Recall that the connected component of an undirected graph is defined as a maximal set of connected vertices.



- Finding the connected components helps answer queries like "is node v connected to node w?".
- How?

Connected component

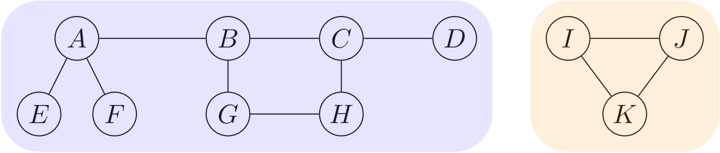
Recall that the connected component of an undirected graph is defined as a maximal set of connected vertices.



- Finding the connected components helps answer queries like "is node v connected to node w?".
- How?

Connected component

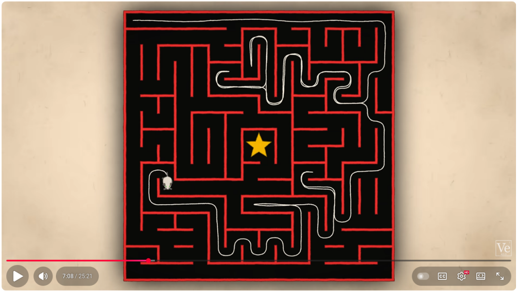
Recall that the connected component of an undirected graph is defined as a maximal set of connected vertices.



- Finding the connected components helps answer queries like "is node v connected to node w?".
- How? Just explore!

Explore by Depth First Search (DFS)

Intuition Explore a maze with a chalk and a string.



The Fastest Maze-Solving Competition On Earth by Veritasium

Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

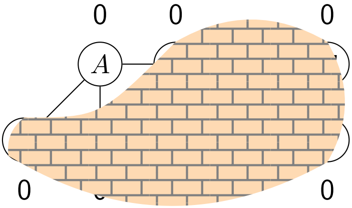
- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

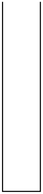
Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example



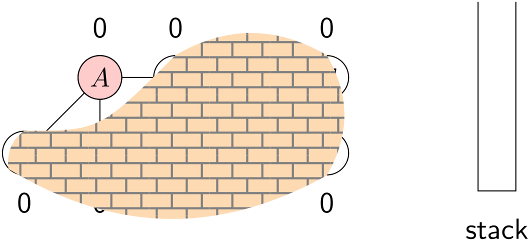


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

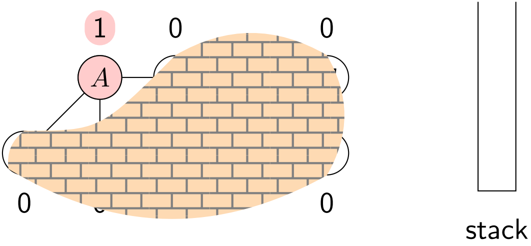


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

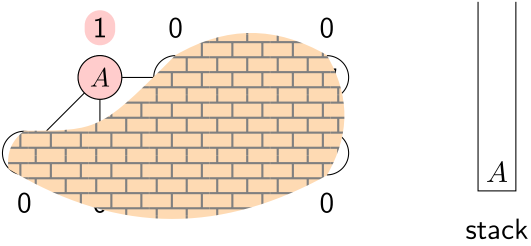


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

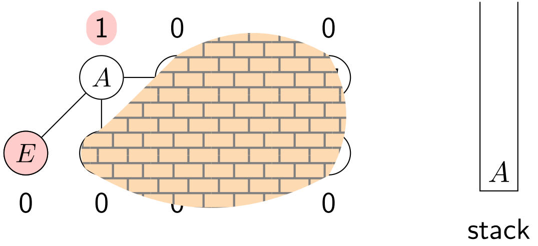


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

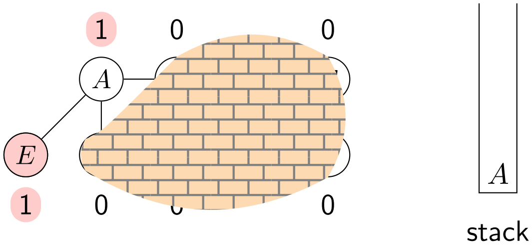


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

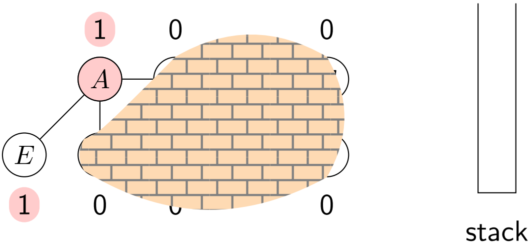


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

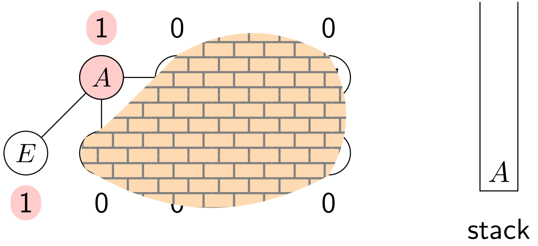


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

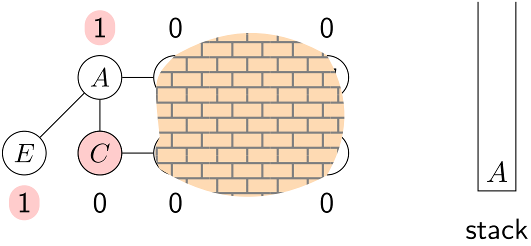


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

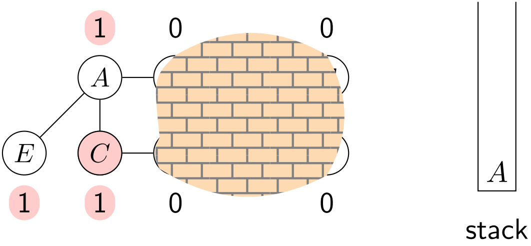


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

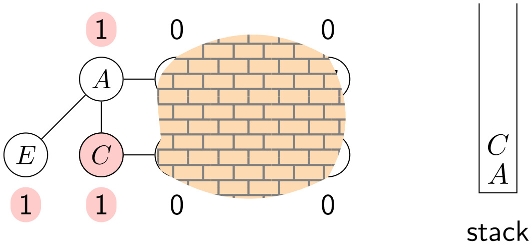


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

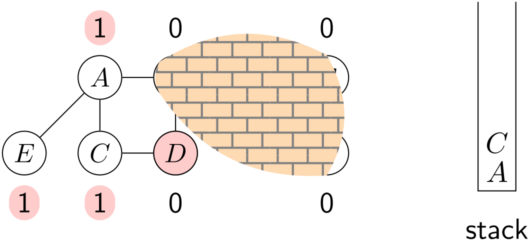


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

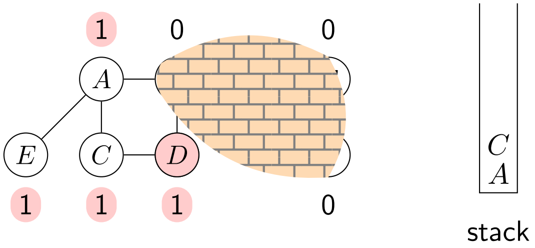


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

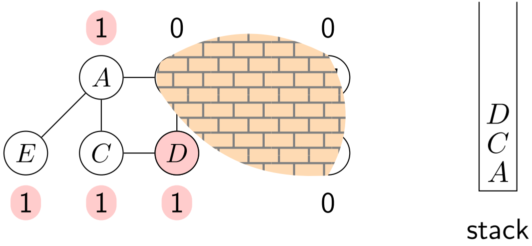


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

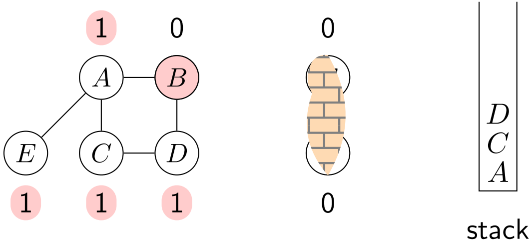


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

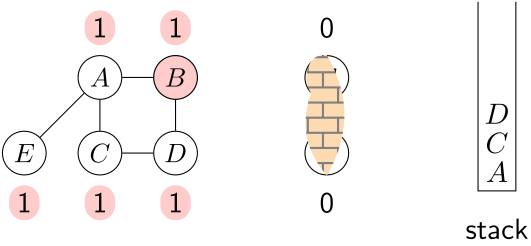


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

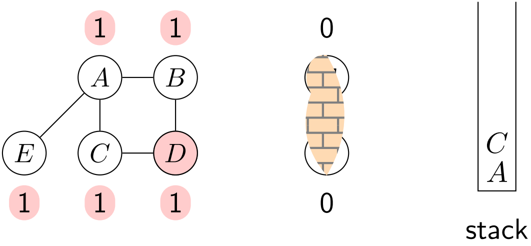


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

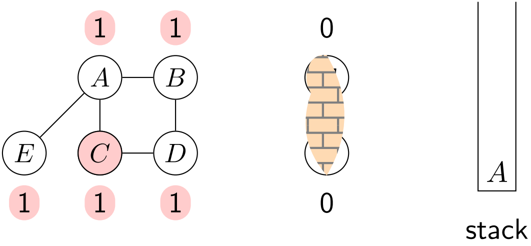


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example

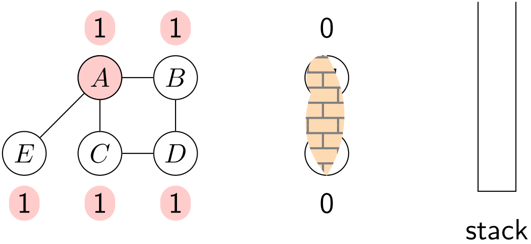


Explore by Depth First Search (DFS)

DFS on a graph follows the same idea:

- The graph is the maze, with vertices corresponding to intersections.
- Integer array (one int per vertex) as the colored cyber-chalk.
- String can be modeled by a stack, to backtrack we pop from the stack.

Example



Explore by Depth First Search (DFS)

Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

Explore(G, s, $color$)

Explore by Depth First Search (DFS)

Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

`//` $visited$ `is an array of length` $|V|$`, filled with 0's`

Explore(G, s, $color$)

Explore by Depth First Search (DFS)

Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

`//` $visited$ `is an array of length` $|V|$`, filled with 0's`

Explore(G, s, $color$)

$visited[s] = color$

Explore by Depth First Search (DFS)

Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

```
// visited is an array of length |V|, filled with 0's
Explore(G, s, color)
    visited[s] = color
    foreach edge {s, v} ∈ E do
        if visited[v] == 0 then
            Explore(G, v, color)
```

Explore by Depth First Search (DFS)

Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

`//` $visited$ `is an array of length` $|V|$`, filled with 0's`

Explore(G, s, $color$)

- $visited[s] = color$
- **foreach** edge $\{s, v\} \in E$ **do**
  - **if** $visited[v] == 0$ **then**
    - Explore(G, v, $color$)

Correctness?

Explore by Depth First Search (DFS)

Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

```
// visited is an array of length |V|, filled with 0's
Explore(G, s, color)
    visited[s] = color
    foreach edge {s, v} ∈ E do
        if visited[v] == 0 then
            Explore(G, v, color)
```

Correctness?

Explore by Depth First Search (DFS)

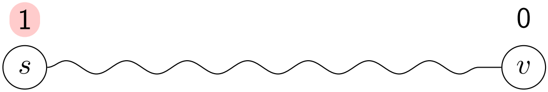
Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

`//` $visited$ `is an array of length` $|V|$`, filled with 0's`

Explore(G, s, $color$)

- $visited[s] = color$
- **foreach** edge $\{s, v\} \in E$ **do**
  - **if** $visited[v] == 0$ **then**
    - Explore(G, v, $color$)

Correctness?



Explore by Depth First Search (DFS)

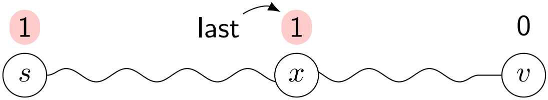
Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

```
// visited is an array of length |V|, filled with 0's
```

Explore(G, s, $color$)

- $visited[s] = color$
- **foreach** edge $\{s, v\} \in E$ **do**
  - **if** $visited[v] == 0$ **then**
    - Explore(G, v, $color$)

Correctness?



Explore by Depth First Search (DFS)

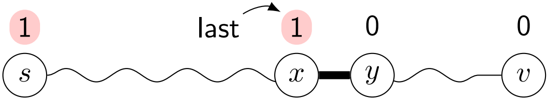
Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

```
// visited is an array of length |V|, filled with 0's
```

Explore(G, s, $color$)

$visited[s] = color$
foreach edge $\{s, v\} \in E$ **do**
  if $visited[v] == 0$ **then**
    Explore(G, v, $color$)

Correctness?



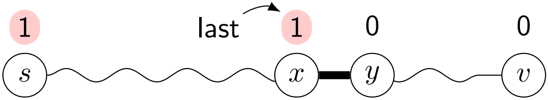
Explore by Depth First Search (DFS)

Input: Graph $G = (V, E)$, starting vertex s, chalk $color$
Output: Mark all nodes reachable from s with $color$

```
// visited is an array of length |V|, filled with 0's
```

Explore(G, s, $color$)
- $visited[s] = color$
- **foreach** edge $\{s, v\} \in E$ **do**
  - **if** $visited[v] == 0$ **then**
    - Explore(G, v, $color$)

Correctness?



The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

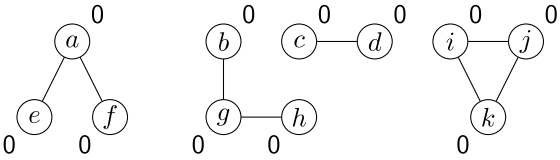
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

- $visited$ is an array of length $|V|$, filled with 0's
- $color = 1$
- **foreach** $s \in V$ **do**
  - **if** $visited[s] == 0$ **then**
    - Explore(G, s, $color$)
    - $color = color + 1$

Example



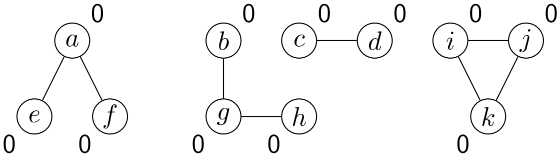
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

Example Explore(G, a, 1)



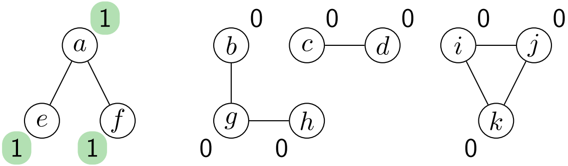
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

$visited$ is an array of length $|V|$, filled with 0's
$color = 1$
foreach $s \in V$ **do**
  if $visited[s] == 0$ **then**
    Explore(G, s, $color$)
    $color = color + 1$

Example Explore(G, a, 1)



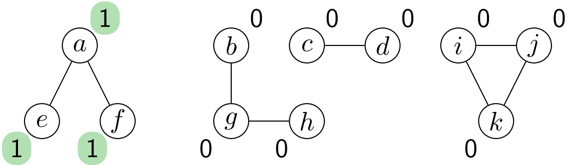
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

Example Explore(G, b, 2)



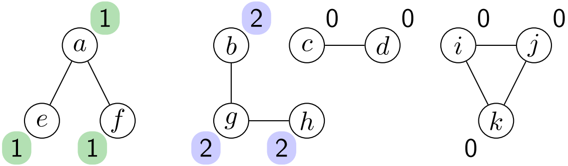
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

<u>DFS($G = (V, E)$)</u>

> $visited$ is an array of length $|V|$, filled with 0's
> $color = 1$
> **foreach** $s \in V$ **do**
>   **if** $visited[s] == 0$ **then**
>   Explore(G, s, $color$)
> $color = color + 1$

Example Explore(G, b, 2)



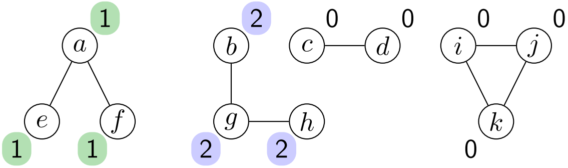
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

<u>DFS($G = (V, E)$)</u>

> $visited$ is an array of length $|V|$, filled with 0's
> $color = 1$
> **foreach** $s \in V$ **do**
>   **if** $visited[s] == 0$ **then**
>   Explore(G, s, $color$)
> $color = color + 1$

Example Explore(G, c, 3)



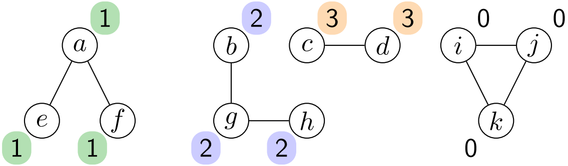
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

$visited$ is an array of length $|V|$, filled with 0's
$color = 1$
foreach $s \in V$ **do**
  if $visited[s] == 0$ **then**
    Explore(G, s, $color$)
    $color = color + 1$

Example Explore(G, c, 3)



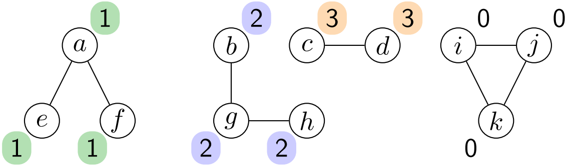
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

$visited$ is an array of length $|V|$, filled with 0's
$color = 1$
foreach $s \in V$ **do**
  if $visited[s] == 0$ **then**
    Explore(G, s, $color$)
    $color = color + 1$

Example   Explore(G, i, 4)



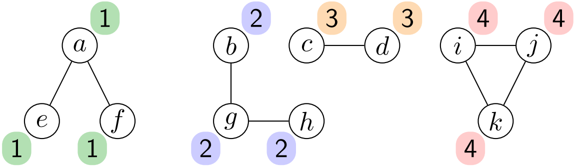
The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

Example Explore(G, i, 4)



The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
    visited is an array of length |V|, filled with 0's
    color = 1
    foreach s ∈ V do
        if visited[s] == 0 then
            Explore(G, s, color)
            color = color + 1
```

Time complexity?

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

Time complexity?

Explore is called on each vertex in V.

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

> *visited* is an array of length $|V|$, filled with 0's
> $color = 1$
> **foreach** $s \in V$ **do**
>> **if** $visited[s] == 0$ **then**
>>> Explore(G, s, $color$)
>>> $color = color + 1$

Time complexity?

Explore is called on each vertex in V.

G in adjacency matrix representation:

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

Time complexity?

Explore is called on each vertex in V.

G in adjacency matrix representation:

- Explore checks all neighbors of a vertex.

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

> $visited$ is an array of length $|V|$, filled with 0's
> $color = 1$
> **foreach** $s \in V$ **do**
>   **if** $visited[s] == 0$ **then**
>   Explore(G, s, $color$)
> $color = color + 1$

Time complexity?

Explore is called on each vertex in V.

G in adjacency matrix representation:

- Explore checks all neighbors of a vertex.

$O\left(|V|^2\right)$

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

    $visited$ is an array of length $|V|$, filled with 0's
    $color = 1$
    foreach $s \in V$ **do**
        if $visited[s] == 0$ **then**
            Explore(G, s, $color$)
            $color = color + 1$

Time complexity?

Explore is called on each vertex in V.

G in adjacency list representation:

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

$visited$ is an array of length $|V|$, filled with 0's
$color = 1$
foreach $s \in V$ **do**
  if $visited[s] == 0$ **then**
    Explore(G, s, $color$)
    $color = color + 1$

Time complexity?

Explore is called on each vertex in V.

G in adjacency list representation:

- Explore checks each edge twice.

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

Time complexity?

Explore is called on each vertex in V.

G in adjacency list representation:

- Explore checks each edge twice.

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

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

Can we do better?

The Depth First Search (DFS) algorithm

- The Explore procedure reveals one connected component.
- DFS works by repeatedly calling Explore.

DFS($G = (V, E)$)

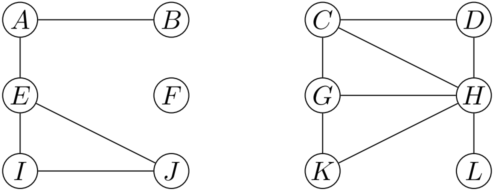
```
visited is an array of length |V|, filled with 0's
color = 1
foreach s ∈ V do
    if visited[s] == 0 then
        Explore(G, s, color)
        color = color + 1
```

Can we do better?

$O(|V| + |E|)$ is the best possible, since we need to read the graph.

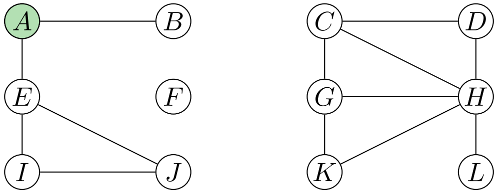
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



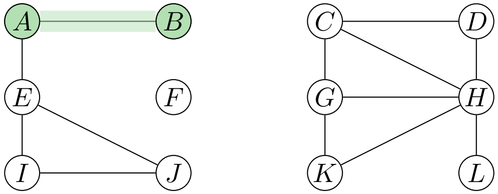
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



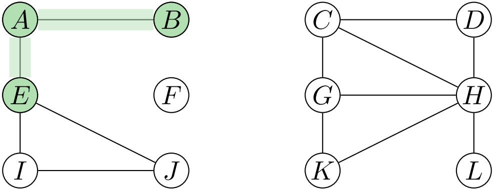
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



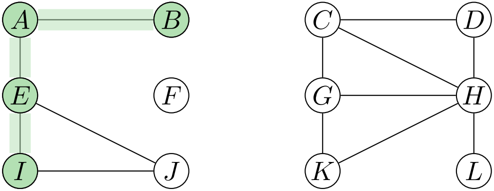
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



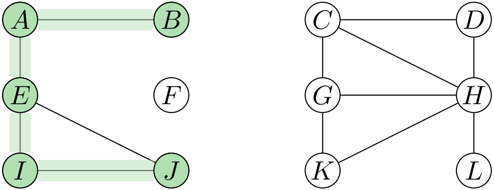
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



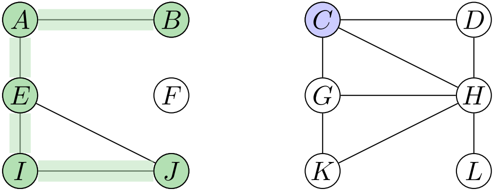
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



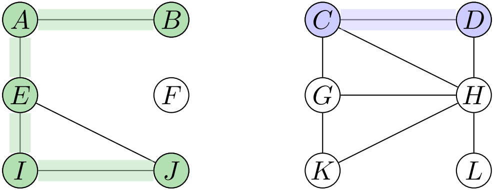
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



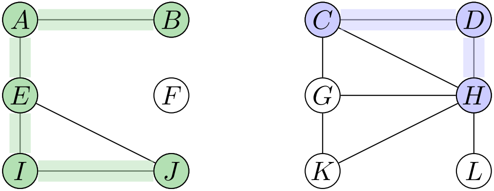
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



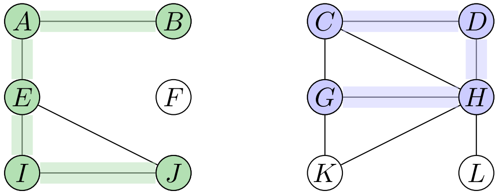
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



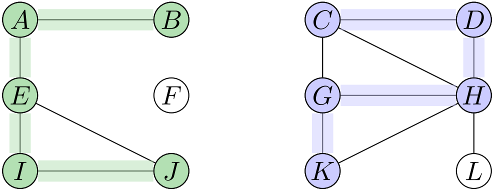
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



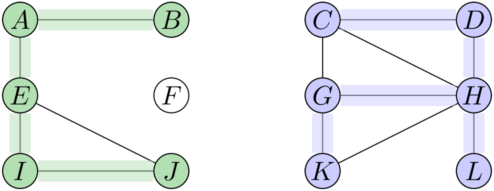
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



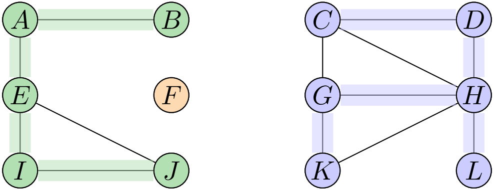
Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



Edge types in DFS forest (undirected)

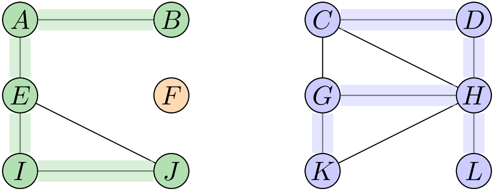
Example Suppose nodes are visited in lexicographical



- This is a DFS forest.

Edge types in DFS forest (undirected)

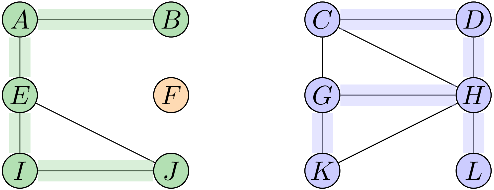
Example Suppose nodes are visited in lexicographical



- This is a DFS forest.
- Colored edges are called tree edges.

Edge types in DFS forest (undirected)

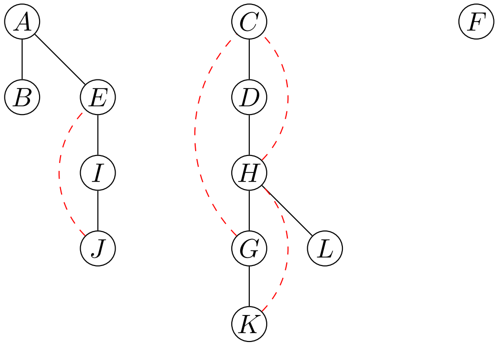
Example Suppose nodes are visited in lexicographical



- This is a DFS forest.
- Colored edges are called tree edges.
- Unused edges are called back edges.

Edge types in DFS forest (undirected)

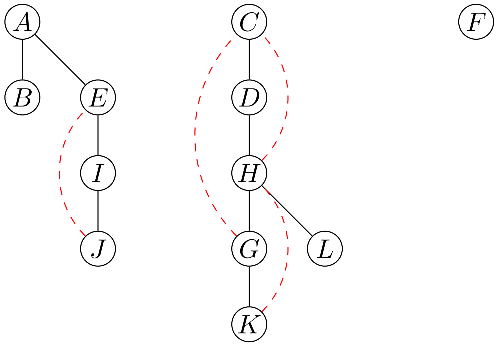
Example Suppose nodes are visited in lexicographical



- This is a DFS forest.
- Solid edges are called tree edges.
- Dashed edges are called back edges.

Edge types in DFS forest (undirected)

Example Suppose nodes are visited in lexicographical



- This is a DFS forest.
- Solid edges are called tree edges.
- Dashed edges are called back edges.
- Back edges correspond to cycles.

Cycle detection (undirected)

To find cycles, it is sufficient to find back edges, which can be done with a simple modification to the Explore procedure.

Explore(G, s, $color$, $previous$)

```
visited[s] = color
foreach edge {s, v} ∈ E do
    if visited[v] == 0 then
        Explore(G, v, color, s)
    if visited[v] ≠ 0  and  v ≠ previous then
        Output {v, s} is a back edge or "found cycle"
```

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.

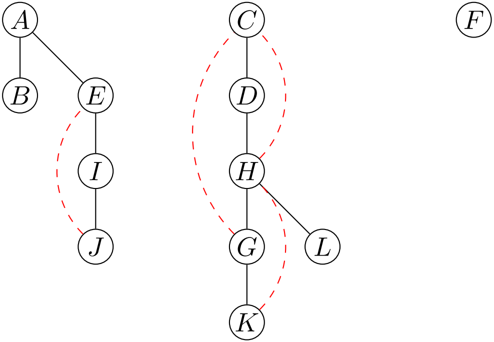
```
// pre and post are integer arrays of size |V|
// clock is an integer counter starting at 1
Explore(G, s, color)
    visited[s] = color
    pre[s] = clock
    clock = clock + 1
    foreach edge {s, v} ∈ E do
        if visited[v] == 0 then
            Explore(G, v, color)
    post[s] = clock
   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

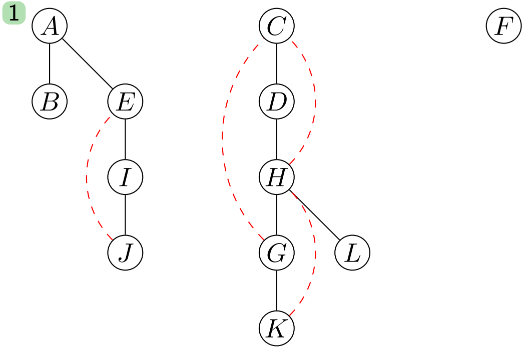


Pre- and post-visit timestamps (undirected)

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

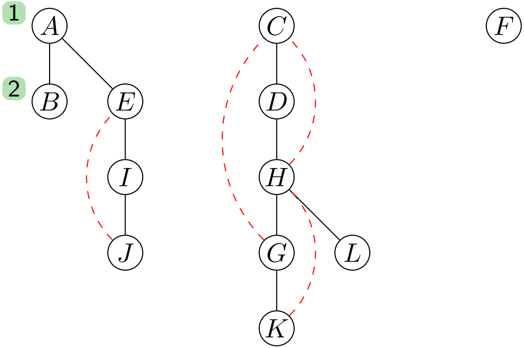


Pre- and post-visit timestamps (undirected)

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

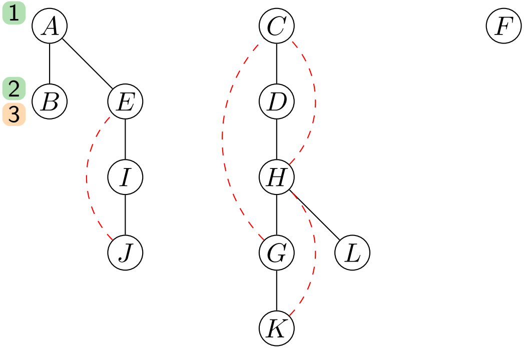


Pre- and post-visit timestamps (undirected)

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

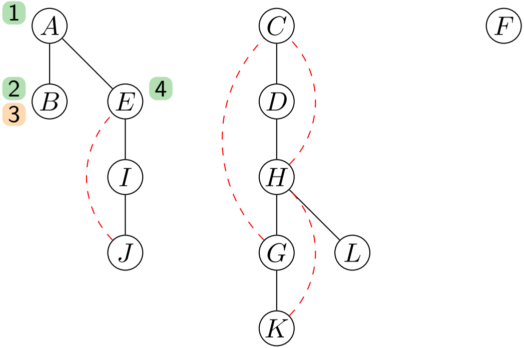


Pre- and post-visit timestamps (undirected)

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

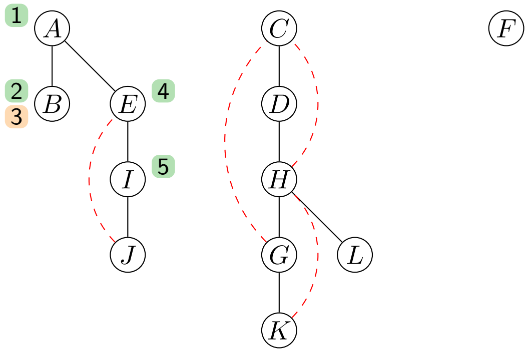


Pre- and post-visit timestamps (undirected)

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

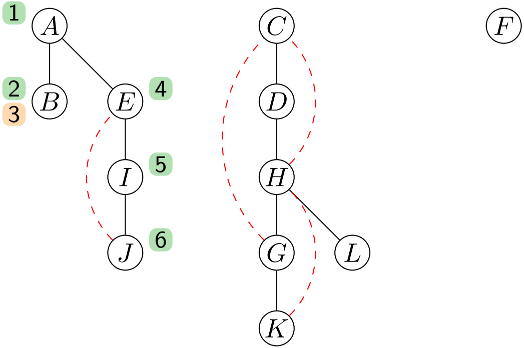


Pre- and post-visit timestamps (undirected)

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

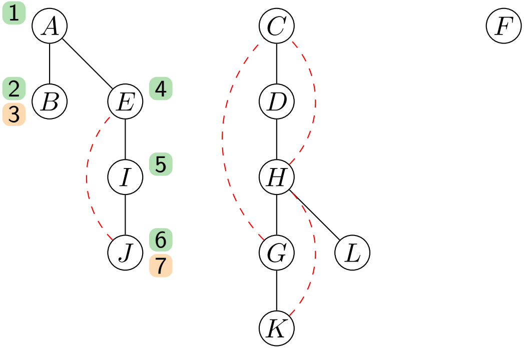


Pre- and post-visit timestamps (undirected)

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

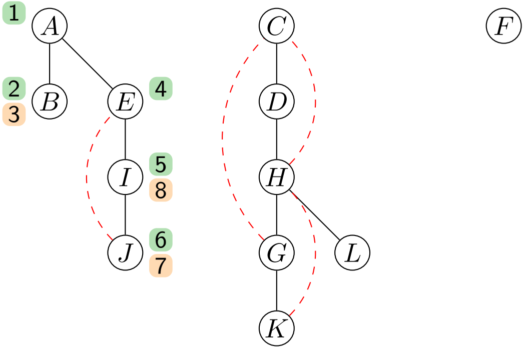


Pre- and post-visit timestamps (undirected)

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

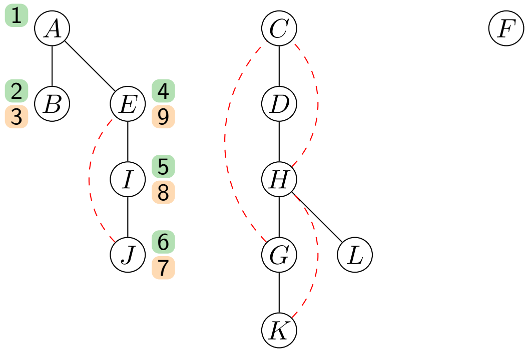


Pre- and post-visit timestamps (undirected)

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

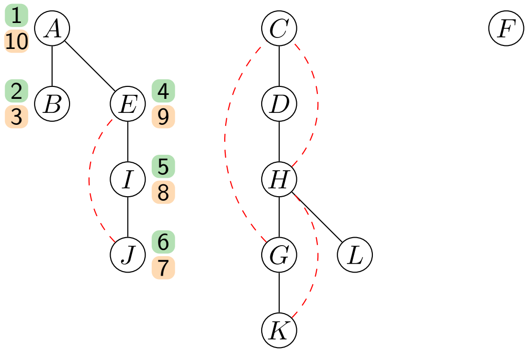


Pre- and post-visit timestamps (undirected)

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

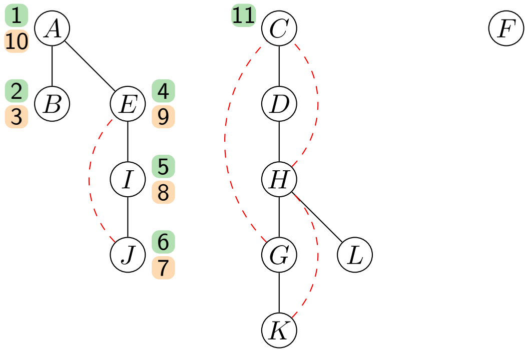


Pre- and post-visit timestamps (undirected)

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

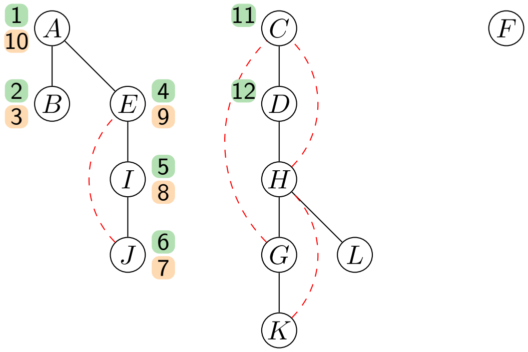


Pre- and post-visit timestamps (undirected)

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

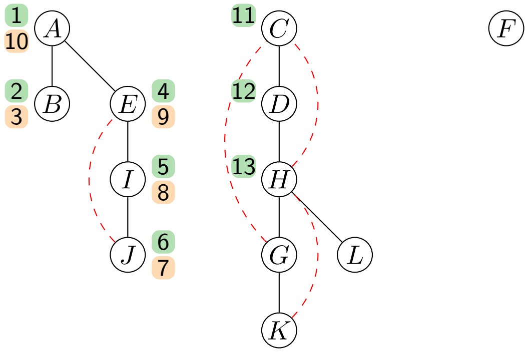


Pre- and post-visit timestamps (undirected)

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

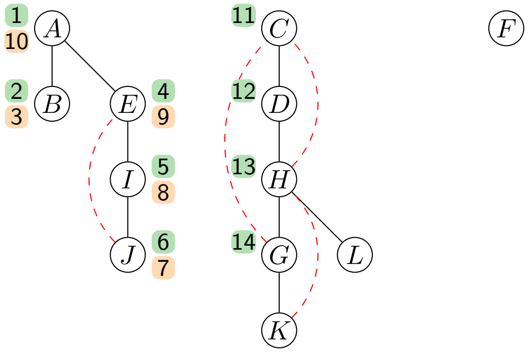


Pre- and post-visit timestamps (undirected)

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

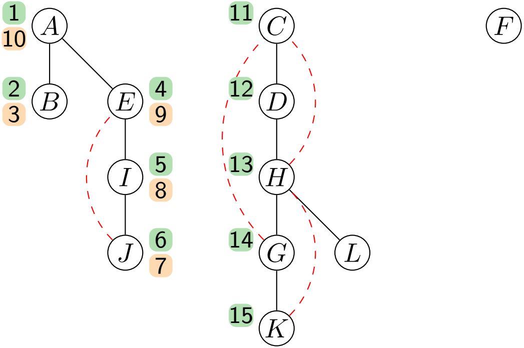


Pre- and post-visit timestamps (undirected)

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

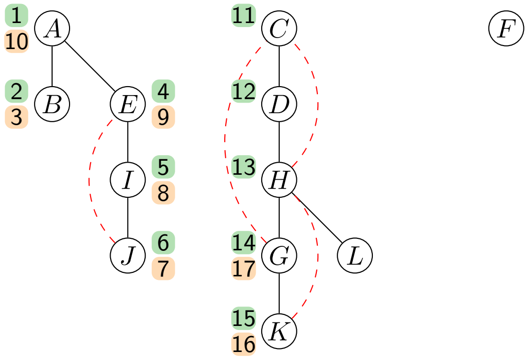


Pre- and post-visit timestamps (undirected)

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

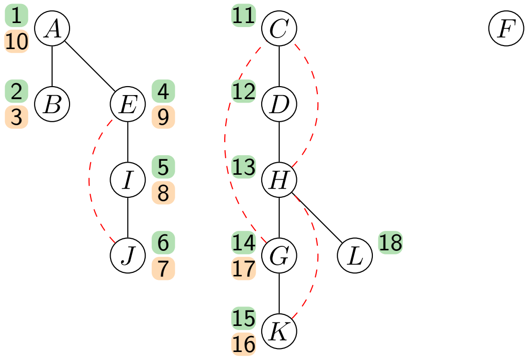


Pre- and post-visit timestamps (undirected)

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

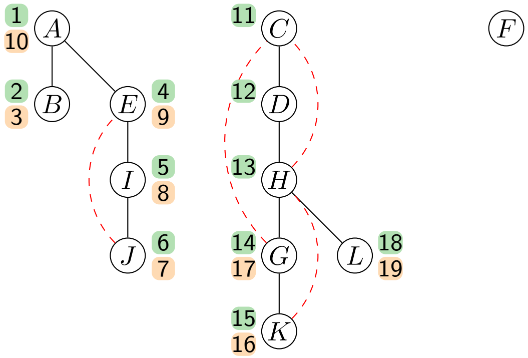


Pre- and post-visit timestamps (undirected)

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

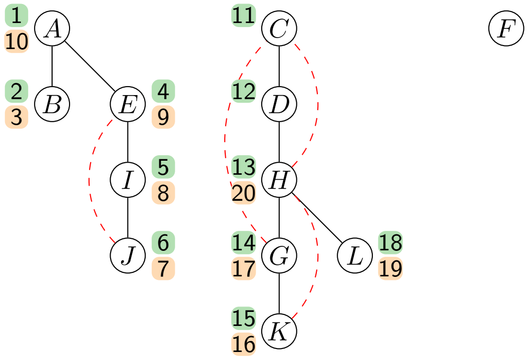


Pre- and post-visit timestamps (undirected)

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

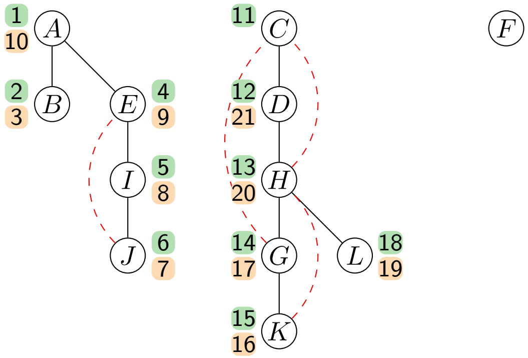


Pre- and post-visit timestamps (undirected)

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

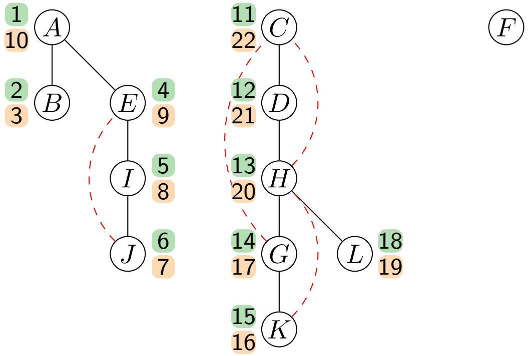


Pre- and post-visit timestamps (undirected)

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

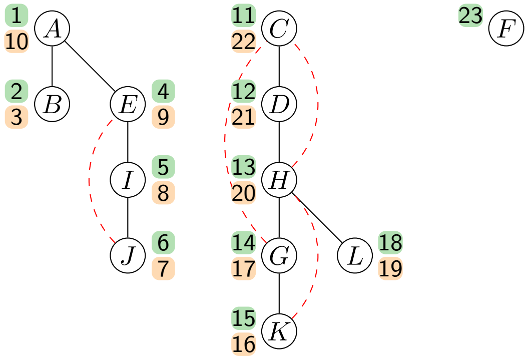


Pre- and post-visit timestamps (undirected)

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