Graph Algorithms: Depth-First Searching

Pontus Ekberg

Uppsala University

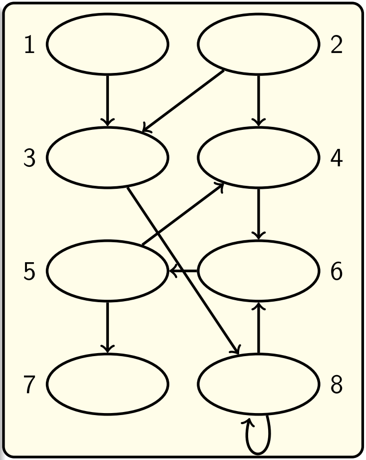
(Based on previous material by Mohamed Faouzi Atig and Parosh Aziz Abdulla)

Depth-First Search

- Input: A graph $G = (V, E)$ and a node $s \in V$
- Output:
  - The set of nodes reachable from s
  - Produce a *depth-first tree* with root s that contains all reachable nodes from s
- The algorithm works on both directed and undirected graphs

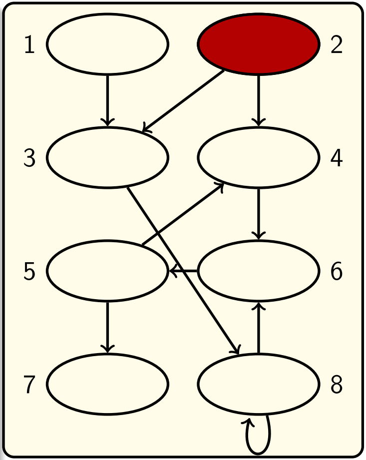
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



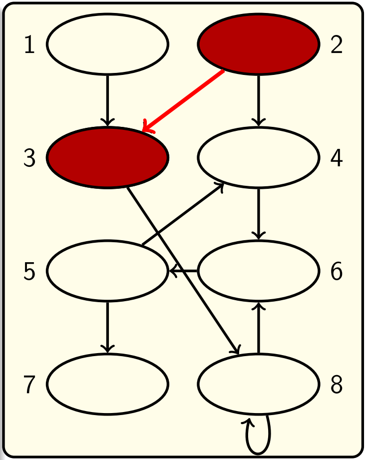
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



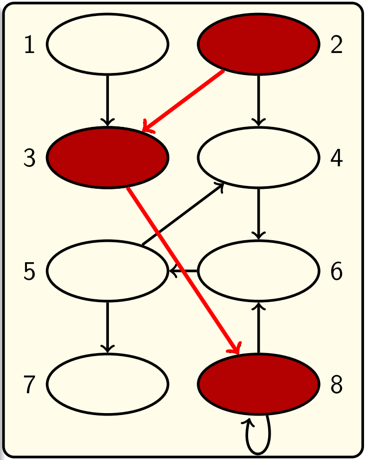
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



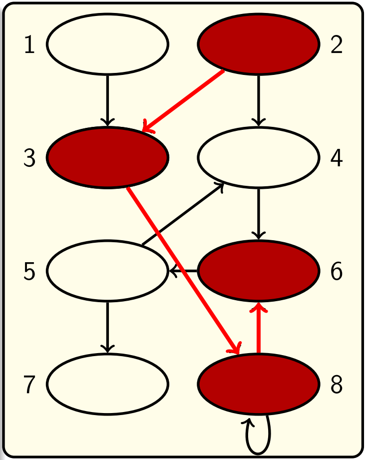
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



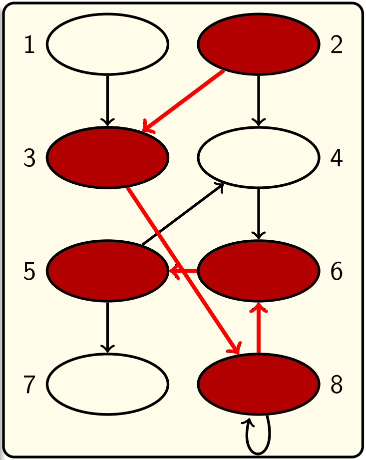
Depth-First Search: Principle

- Initially the source node is the only discovered node

- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.

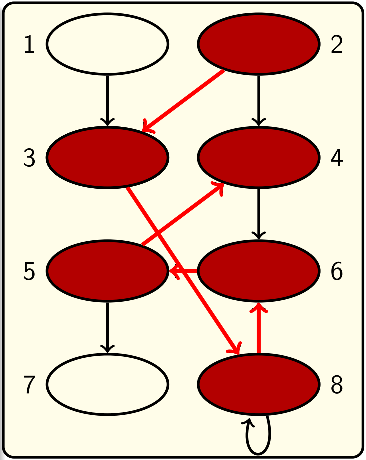
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.

- The process continues until we have discovered all the nodes that are reachable from the original source node.



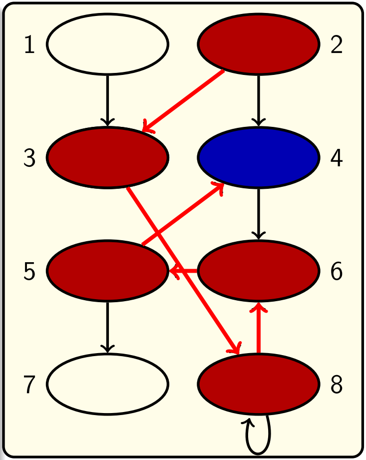
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node *v*. Only edges to unexplored nodes are explored.
- Once all of *v*'edges have been explored, the search *backtracks* to explore edges leaving the node from which *v* was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



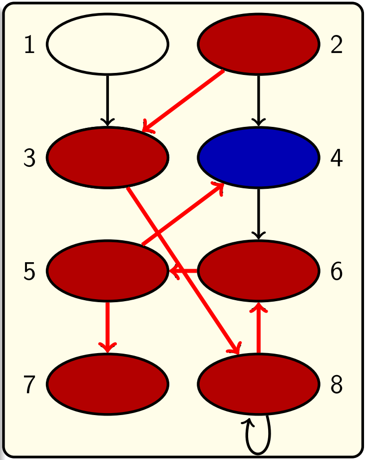
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node *v*. Only edges to unexplored nodes are explored.
- Once all of *v*'edges have been explored, the search *backtracks* to explore edges leaving the node from which *v* was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



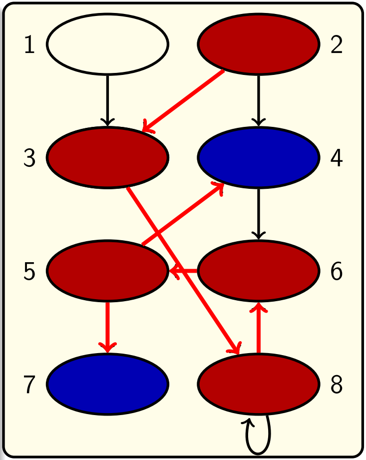
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node *v*. Only edges to unexplored nodes are explored.
- Once all of *v*'edges have been explored, the search *backtracks* to explore edges leaving the node from which *v* was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



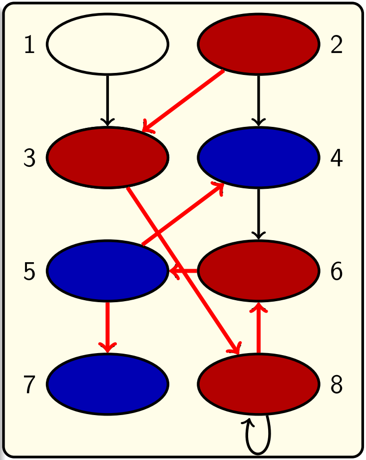
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node *v*. Only edges to unexplored nodes are explored.
- Once all of *v*'edges have been explored, the search *backtracks* to explore edges leaving the node from which *v* was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



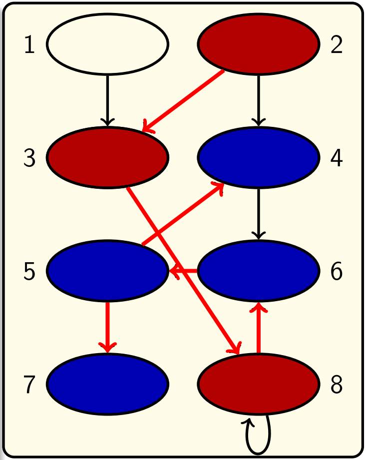
Depth-First Search: Principle

- Initially the source node is the only discovered node

- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.

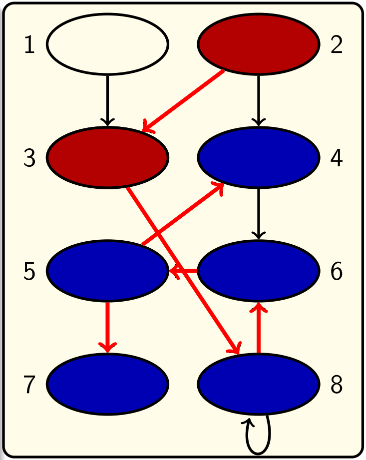
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.

- The process continues until we have discovered all the nodes that are reachable from the original source node.



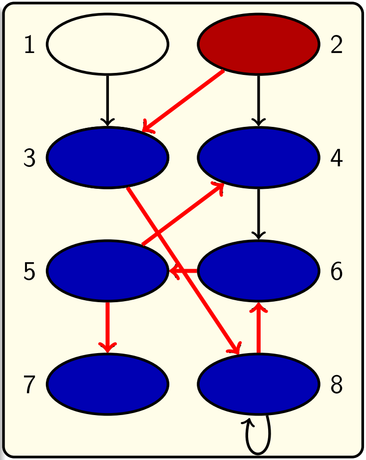
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



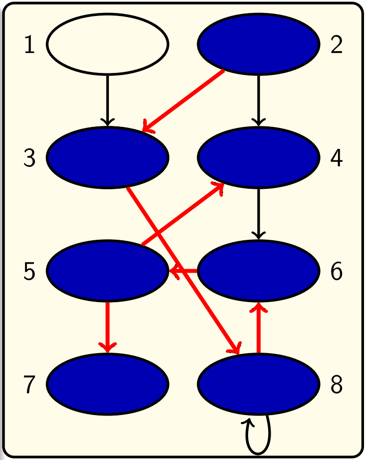
Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node *v*. Only edges to unexplored nodes are explored.
- Once all of *v*'edges have been explored, the search *backtracks* to explore edges leaving the node from which *v* was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



Depth-First Search: Principle

- Initially the source node is the only discovered node
- Explore edges out of the most recently discovered node v. Only edges to unexplored nodes are explored.
- Once all of v'edges have been explored, the search *backtracks* to explore edges leaving the node from which v was discovered.
- The process continues until we have discovered all the nodes that are reachable from the original source node.



Depth-First Search: Algorithm

```
DFS(G, s)
for each vertex u ∈ G.V
    do u.color ← WHITE
       u.π ← NIL
       u.d ← 0
       u.f ← 0
time ← 0
DFS-VISIT(s)
```

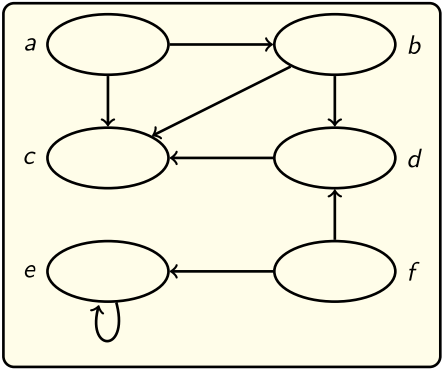
```
DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
    do if v.color = WHITE
          then v.π ← u
               DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```

Each node u has the following attributes:

- $u.color$: the color of each node visited
    - *WHITE*: not discovered
    - *RED*: discovered but not analyzed
    - *BLUE*: finished, i.e., discovered and analyzed
- $u.\pi$: predecessor of u in the analysis
- $u.d$: discovery time, a counter indicating when the node u is discovered
- $u.f$: finishing time, a counter indicating when the processing of u (and all its descendant) is finished.

Depth-First Search: Algorithm

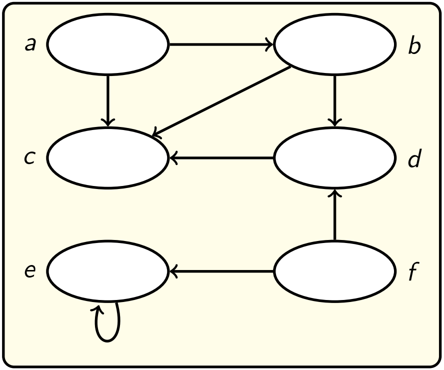
```
DFS(G, s)
for each vertex u ∈ G.V
        do u.color ← WHITE
           u.π ← NIL
           u.d ← 0
           u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
                then v.π ← u
                     DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



Depth-First Search: Algorithm

```
DFS(G, s)
for each vertex u ∈ G.V
    do u.color ← WHITE
       u.π ← NIL
       u.d ← 0
       u.f ← 0
time ← 0
DFS-VISIT(s)
```

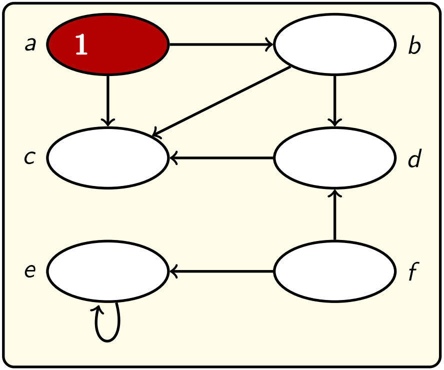
```
DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
    do if v.color = WHITE
         then v.π ← u
              DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 0

Depth-First Search: Algorithm

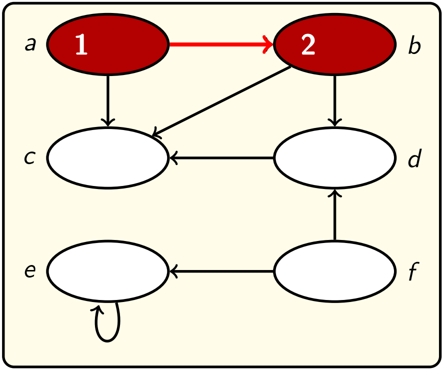
```
DFS(G, s)
for each vertex u ∈ G.V
        do u.color ← WHITE
           u.π ← NIL
           u.d ← 0
           u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
               then v.π ← u
                    DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 1

Depth-First Search: Algorithm

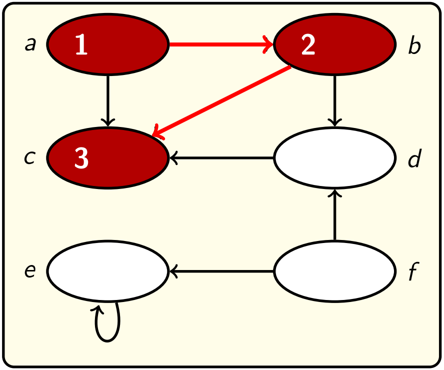
```
DFS(G, s)
for each vertex u ∈ G.V
    do u.color ← WHITE
       u.π ← NIL
       u.d ← 0
       u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
    do if v.color = WHITE
          then v.π ← u
               DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 2

Depth-First Search: Algorithm

```
DFS(G, s)
for each vertex u ∈ G.V
    do u.color ← WHITE
       u.π ← NIL
       u.d ← 0
       u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
    do if v.color = WHITE
          then v.π ← u
               DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```

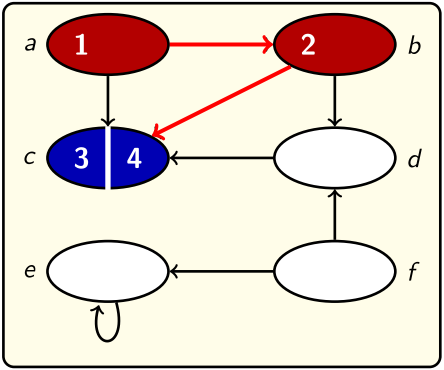


time = 3

Depth-First Search: Algorithm

```
DFS(G, s)
for each vertex u ∈ G.V
    do u.color ← WHITE
       u.π ← NIL
       u.d ← 0
       u.f ← 0
time ← 0
DFS-VISIT(s)
```

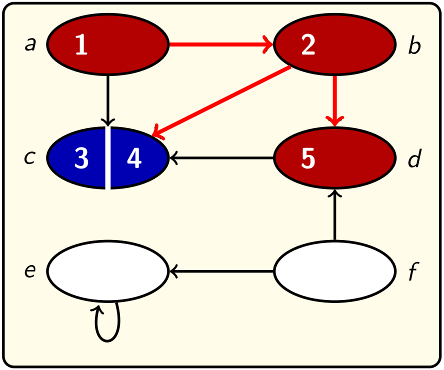
```
DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
    do if v.color = WHITE
          then v.π ← u
               DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 4

Depth-First Search: Algorithm

```
DFS(G, s)
for each vertex u ∈ G.V
        do u.color ← WHITE
           u.π ← NIL
           u.d ← 0
           u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
                then v.π ← u
                     DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```

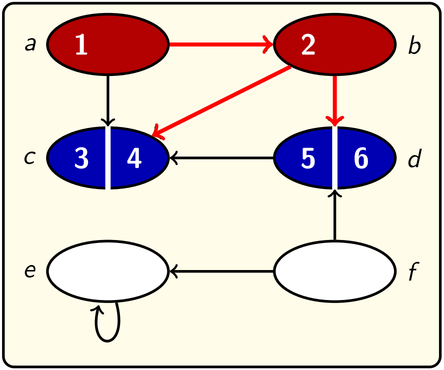


time = 5

Depth-First Search: Algorithm

```
DFS(G, s)
for each vertex u ∈ G.V
        do u.color ← WHITE
            u.π ← NIL
            u.d ← 0
            u.f ← 0
time ← 0
DFS-VISIT(s)
```

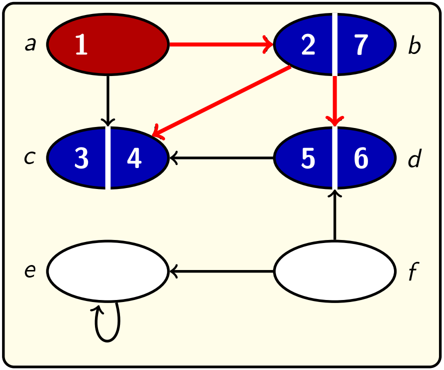
```
DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
                then v.π ← u
                     DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 6

Depth-First Search: Algorithm

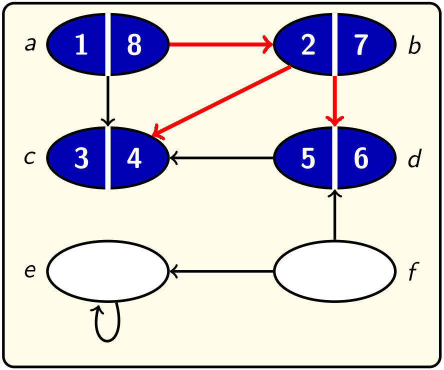
```
DFS(G, s)
for each vertex u ∈ G.V
        do u.color ← WHITE
            u.π ← NIL
            u.d ← 0
            u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
                then v.π ← u
                        DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
```



time = 7

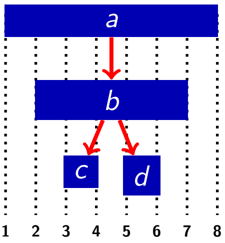
Depth-First Search: Algorithm

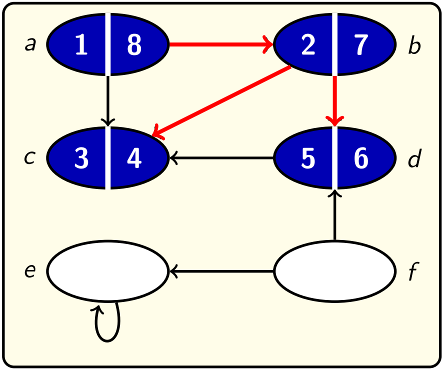
```
DFS(G, s)
for each vertex u ∈ G.V
        do u.color ← WHITE
           u.π ← NIL
           u.d ← 0
           u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 8

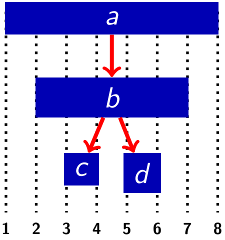
Depth-First Search: Algorithm





Parenthesis Theorem and the Depth-First Tree

Depth-First Search: Parenthesis Theorem



Time-stamp structure

For any two discovered nodes u and v, one of the following properties holds:

- u is a descendant of v if and only if $[u.d, u.f]$ is subinterval of $[v.d, v.f]$
- u is an ancestor of v if and only if $[u.d, u.f]$ contains $[v.d, v.f]$
- u is unrelated to v if and only if $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint.

Depth-First Search: Complexity

```
DFS(G, s)
for each vertex u ∈ G.V
      do u.color ← WHITE
         u.π ← NIL
         u.d ← 0
         u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
      do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```

Depth-First Search: Complexity

```
DFS(G, s)
for each vertex u ∈ G.V
        do u.color ← WHITE
           u.π ← NIL
           u.d ← 0
           u.f ← 0
time ← 0
DFS-VISIT(s)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```

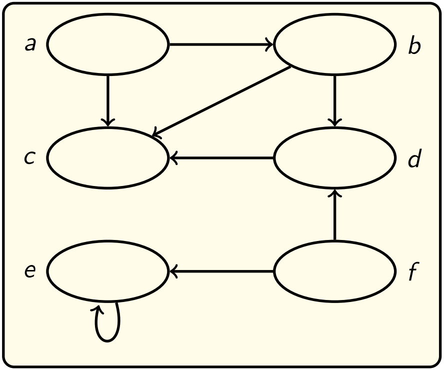
- Initialization costs $O(|V|)$
- The procedure DFS-VISIT is called at most $|V|$ times
- Each edge is considered at most one time along the **for** loop, in all the interations of the **while** loop taken together
- Total time $= O(|V| + |E|)$

Graph Traversal Algorithms

- Breadth-First Search and Depth-First Search explore only the nodes that are reachable from the original source node s
- To traverse all the nodes of the graph:
  - Select a source node v
  - Explore all the nodes that are reachable from v (in depth or breadth)
  - If any undiscovered nodes remain, then one of them is selected as new source node, and the search is repeated from that source node.

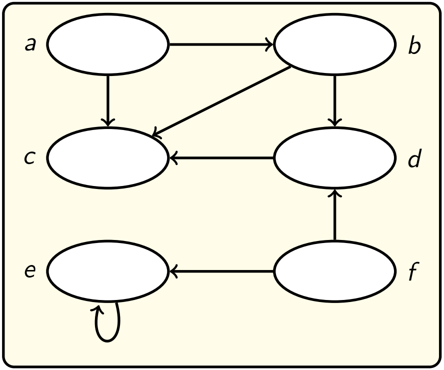
Depth-First Search: Exploring all Nodes

```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
               then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
               then v.π ← u
                    DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



Depth-First Search: Exploring all Nodes

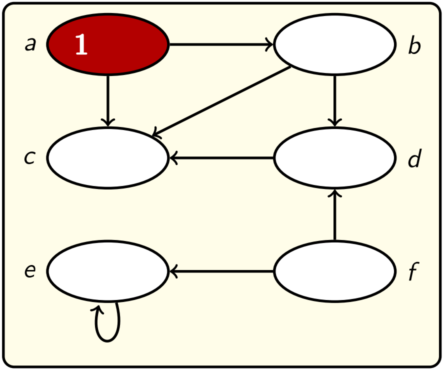
```
DFS(G)
for each vertex u ∈ G : V
      do u.color ← WHITE
         u.π ← NIL
time ← 0
for each vertex u ∈ G.V
      do if u.color = WHITE
            then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
      do if v.color = WHITE
            then v.π ← u
                 DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 0

Depth-First Search: Exploring all Nodes

```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
```

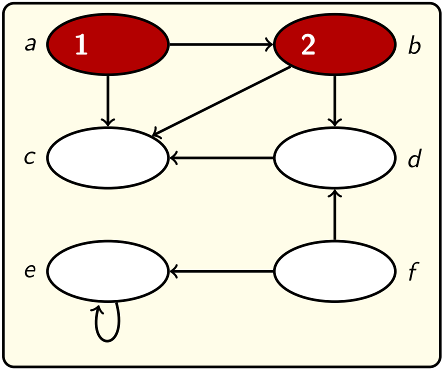


time $= 1$

Depth-First Search: Exploring all Nodes

```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)
```

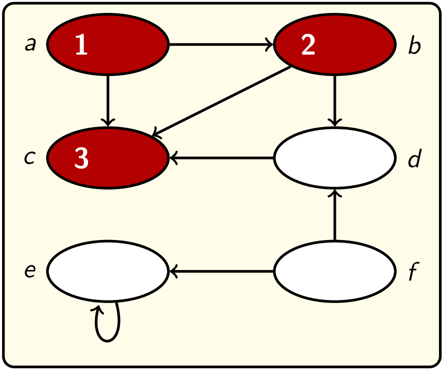
```
DFS-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
```



time = 2

Depth-First Search: Exploring all Nodes

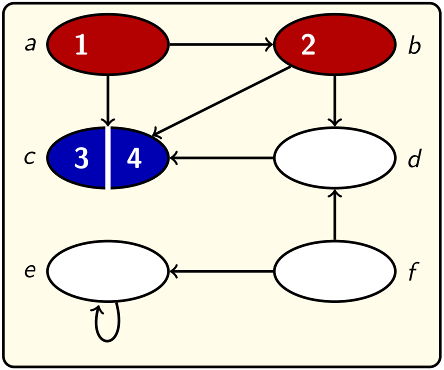
```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 3

Depth-First Search: Exploring all Nodes

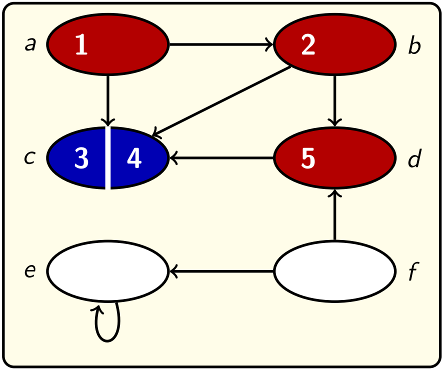
```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
            u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 4

Depth-First Search: Exploring all Nodes

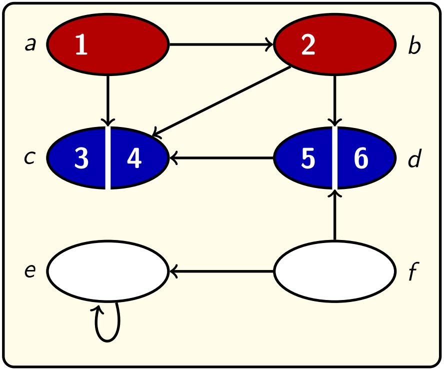
```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
```



time = 5

Depth-First Search: Exploring all Nodes

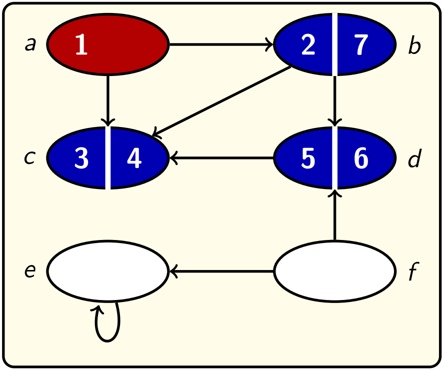
```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```



time = 6

Depth-First Search: Exploring all Nodes

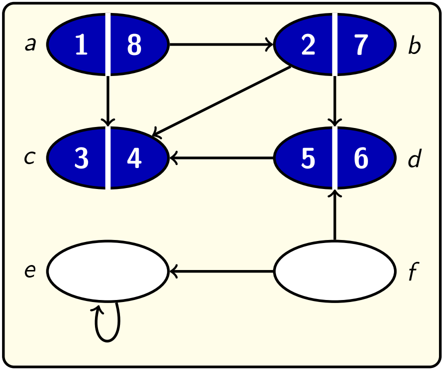
```
DFS(G)
for each vertex u ∈ G : V
        do u. color ← WHITE
            u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u. color = WHITE
                then DFS-VISIT(u)

DFS-VISIT(u)
u. color ← RED
time ← time +1
u.d ← time
for each v ∈ G. Adj[u]
        do if v. color = WHITE
                then v.π ← u
                        DFS-VISIT(v)
u. color ← BLUE
time ← time +1
u.f ← time
```



time = 7

Depth-First Search: Exploring all Nodes

```
DFS(G)
for each vertex u ∈ G : V
        do u. color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u. color = WHITE
              then DFS-VISIT(u)

DFS-VISIT(u)
u. color ← RED
time ← time +1
u.d ← time
for each v ∈ G. Adj[u]
        do if v. color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u. color ← BLUE
time ← time +1
u.f ← time
```



time = 8

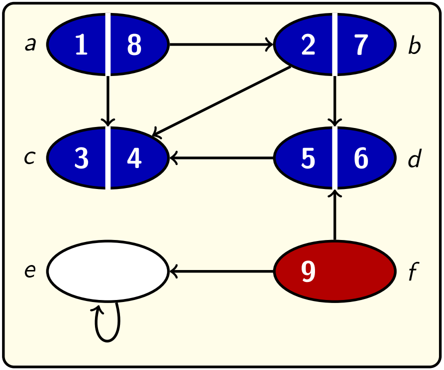
Depth-First Search: Exploring all Nodes

DFS(G)

for each vertex $u \in G : V$
  do $u.color \leftarrow WHITE$
    $u.\pi \leftarrow NIL$
$time \leftarrow 0$
for each vertex $u \in G.V$
  do if $u.color = WHITE$
    then DFS-VISIT(u)

DFS-VISIT(u)

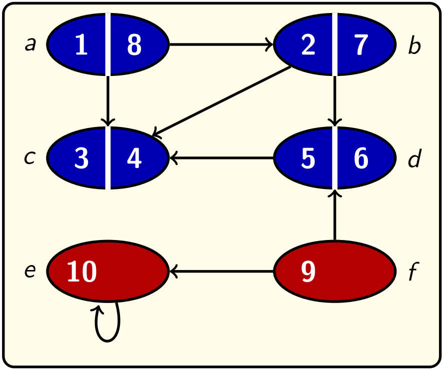
$u.color \leftarrow RED$
$time \leftarrow time + 1$
$u.d \leftarrow time$
for each $v \in G.Adj[u]$
  do if $v.color = WHITE$
    then $v.\pi \leftarrow u$
      DFS-VISIT(v)
$u.color \leftarrow BLUE$
$time \leftarrow time + 1$
$u.f \leftarrow time$



time = 9

Depth-First Search: Exploring all Nodes

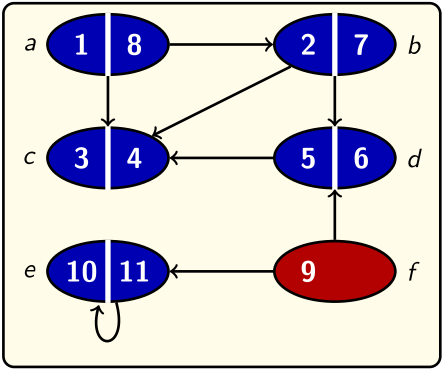
```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
            u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
                then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
                then v.π ← u
                        DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
```



time = 10

Depth-First Search: Exploring all Nodes

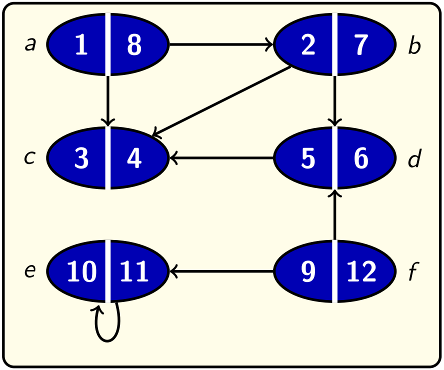
```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
```



time = 11

Depth-First Search: Exploring all Nodes

```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
```

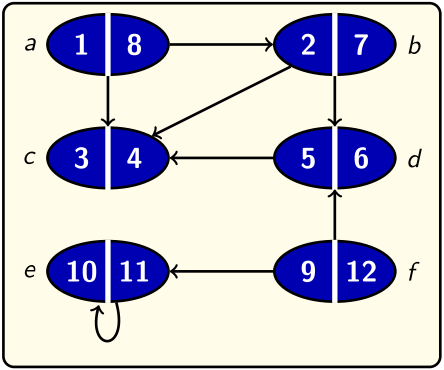


time = 12

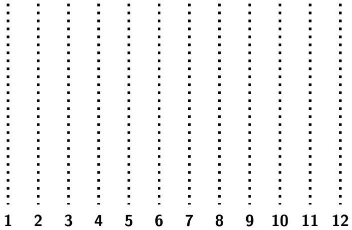
Depth-First Search: Exploring all Nodes

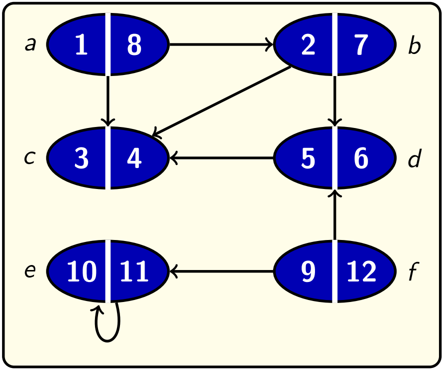
```
DFS(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then DFS-VISIT(u)
```

```
DFS-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
```



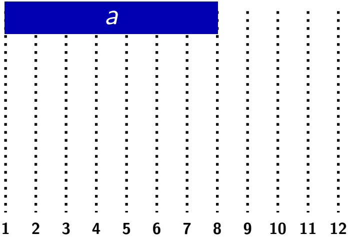
Depth-First Search: Exploring all Nodes

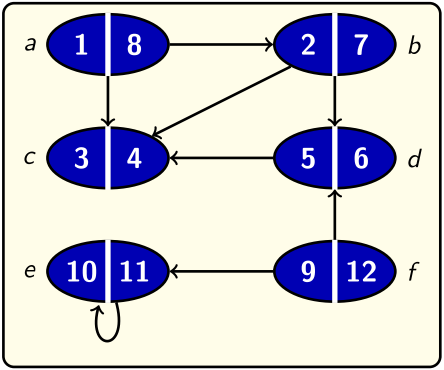




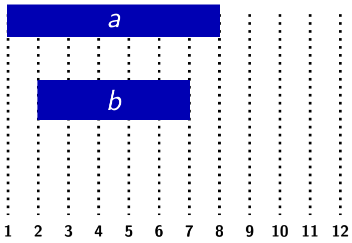
Parenthesis Theorem and the Depth-First Forest

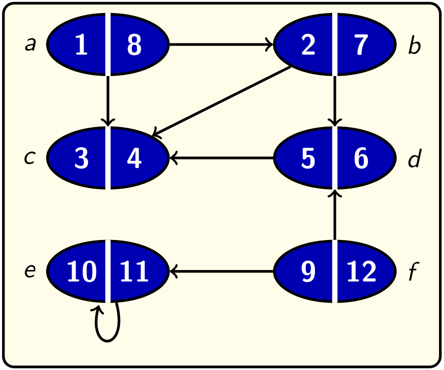
Depth-First Search: Exploring all Nodes





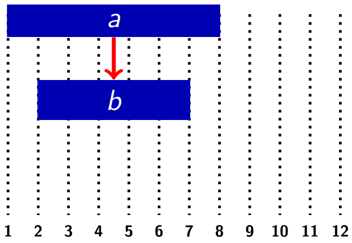
Depth-First Search: Exploring all Nodes

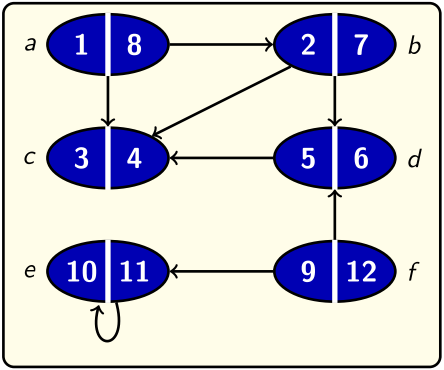




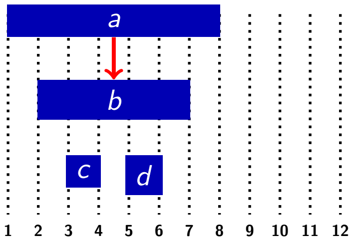
Parenthesis Theorem and the Depth-First Forest

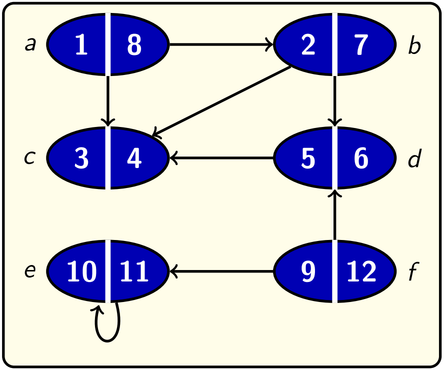
Depth-First Search: Exploring all Nodes



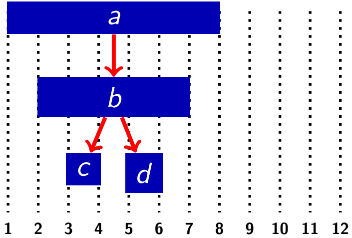


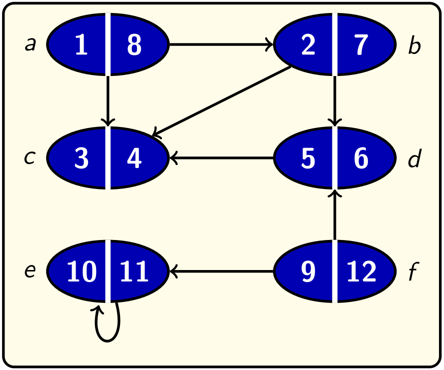
Depth-First Search: Exploring all Nodes





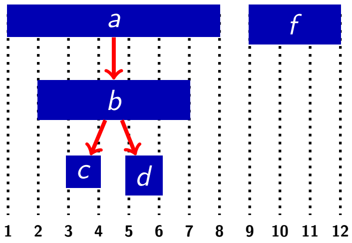
Depth-First Search: Exploring all Nodes

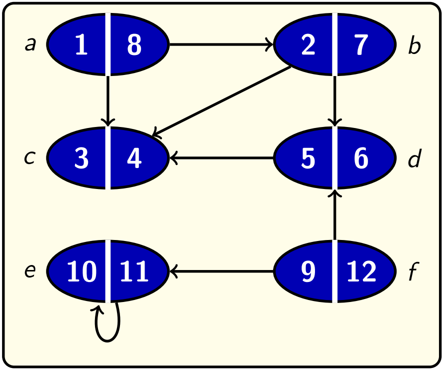




Parenthesis Theorem and the Depth-First Forest

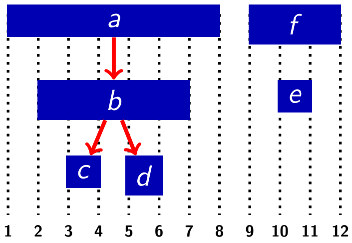
Depth-First Search: Exploring all Nodes

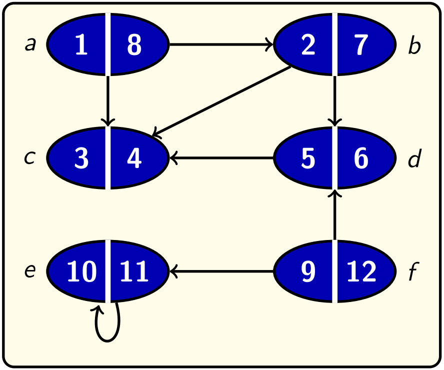




Parenthesis Theorem and the Depth-First Forest

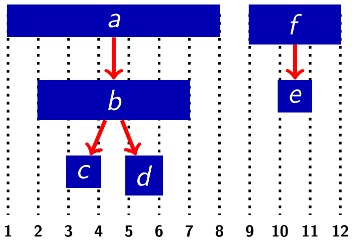
Depth-First Search: Exploring all Nodes

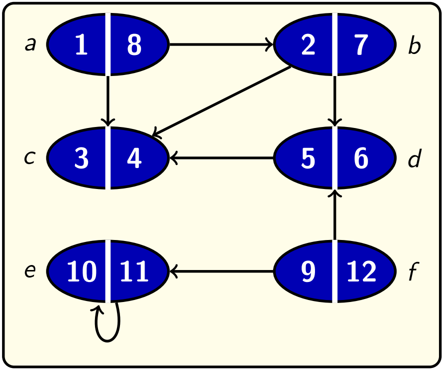




Parenthesis Theorem and the Depth-First Forest

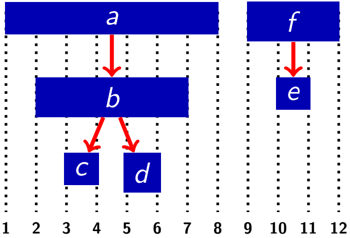
Depth-First Search: Exploring all Nodes





Parenthesis Theorem and the Depth-First Forest

Depth-First Search: Parenthesis Theorem



Time-stamp structure

For any two nodes u and v, one of the following properties holds:

- u is a descendant of v if and only if $[u.d, u.f]$ is subinterval of $[v.d, v.f]$
- v is an ancestor of v if and only if $[u.d, u.f]$ contains $[v.d, v.f]$
- u is unrelated to v if and only if $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint.

Depth-First Search: Complexity

```
DFS(G)
for each vertex u ∈ G : V
      do u.color ← WHITE
         u.π ← NIL
time ← 0
for each vertex u ∈ G.V
      do if u.color = WHITE
            then DFS-VISIT(u)
```

```
DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
      do if v.color = WHITE
            then v.π ← u
                 DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```

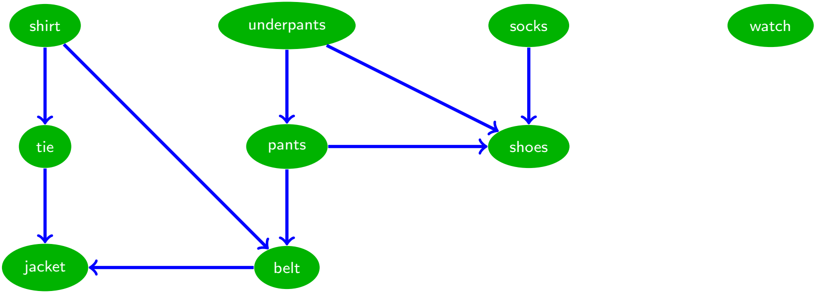
Depth-First Search: Complexity

```
DFS(G)
for each vertex u ∈ G : V
      do u.color ← WHITE
         u.π ← NIL
time ← 0
for each vertex u ∈ G.V
      do if u.color = WHITE
            then DFS-VISIT(u)

DFS-VISIT(u)
u.color ← RED
time ← time + 1
u.d ← time
for each v ∈ G.Adj[u]
      do if v.color = WHITE
            then v.π ← u
                 DFS-VISIT(v)
u.color ← BLUE
time ← time + 1
u.f ← time
```

- Initialization costs $O(|V|)$
- The procedure DFS-VISIT is called exactly once for each node v.
- During an execution of DFS-VISIT(v), the **for** loop executes $|G.Adj[v]|$ times.
- Total time $= O(|V| + |E|)$

Application: Topological Sort

- Directed Acyclic Graph (DAG):



- A topological sort of the graph:

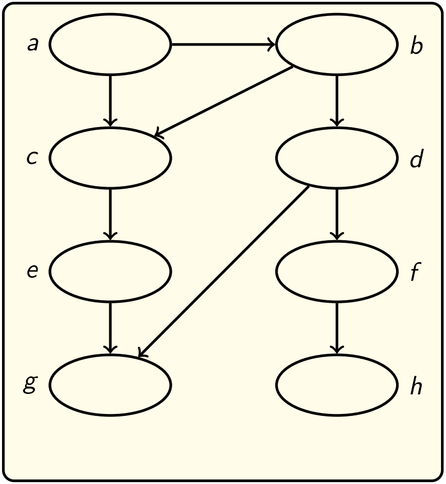
Topological Sort

- Topological sort:
  - Input: Directed Acyclic Graph (DAG) $G = (V, E)$
  - Output: Order the nodes such that if (u, v) is an edge of G then u precedes v
- Examples of Applications:
  - Find an order to follow a set course that takes into account the prerequisites of each course
    - To follow *Algorithms and Data Structures I*, the student must have completed a programming course.
  - Solve the dependencies for installing software
    - Find an order such that each software is installed after all the others softwares on which it depends.

Toplogical Sorting

TOPOLOGICAL-SEARCH(G)

1 call DFS(G)
2 output nodes in order of decreasing finish times



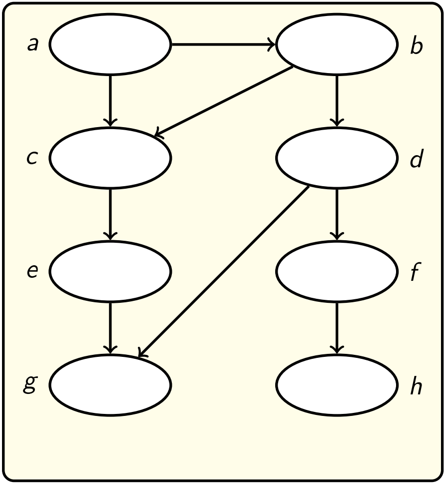
Toplogical Sorting

TOPOLOGICAL-SEARCH(G)

1 call DFS(G)

2 output nodes in order of decreasing finish times

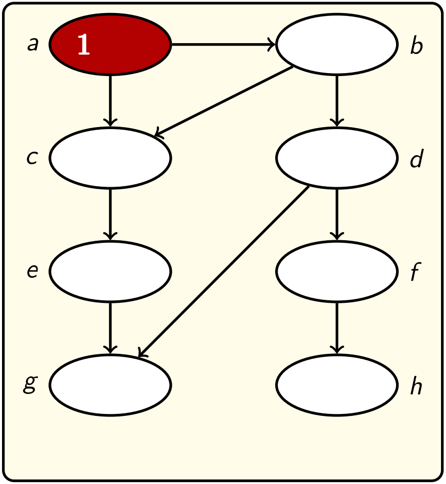
time $= 0$



Toplogical Sorting

TOPOLOGICAL-SEARCH(G)
1 call DFS(G)
2 output nodes in order of decreasing finish times

time = 1

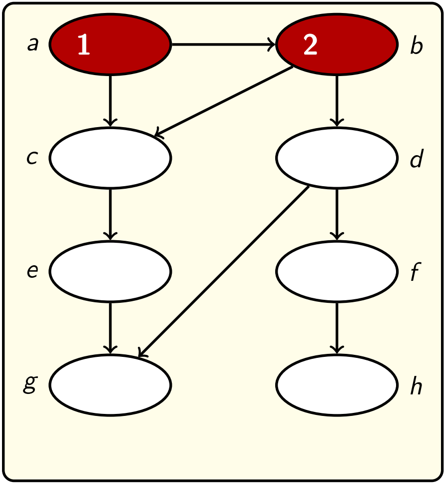


Toplogical Sorting

TOPOLOGICAL-SEARCH(G)

1 call DFS(G)
2 output nodes in order of decreasing finish times

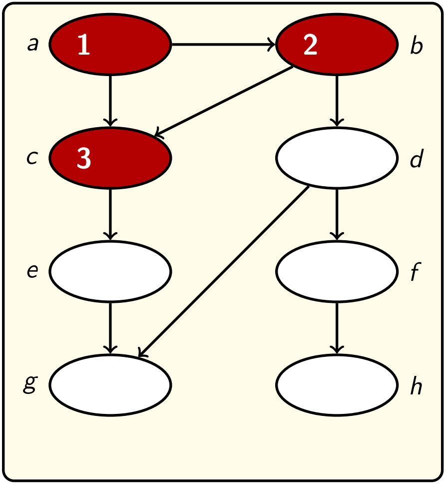
time = 2



Toplogical Sorting

TOPOLOGICAL-SEARCH(G)
1 call DFS(G)
2 output nodes in order
of decreasing finish times

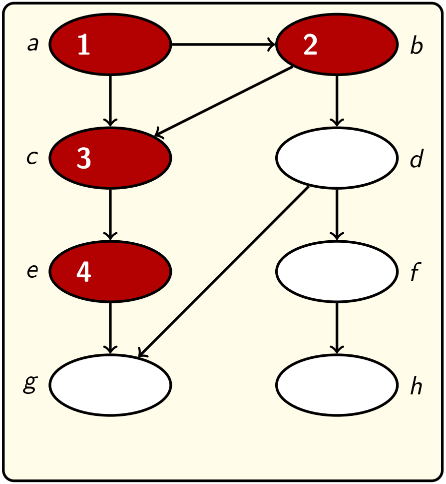
time $= 3$



Toplogical Sorting

TOPOLOGICAL-SEARCH(G)
1 call DFS(G)
2 output nodes in order
of decreasing finish times

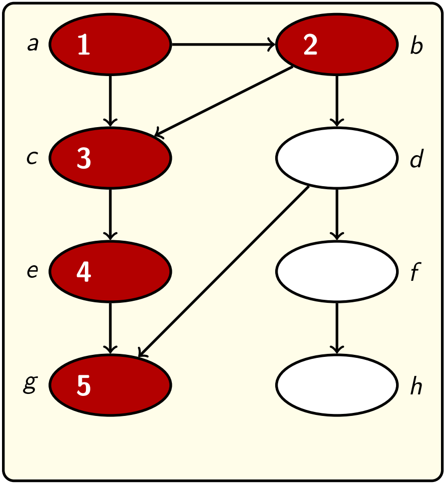
time = 4



Toplogical Sorting

TOPOLOGICAL-SEARCH(G)

1 call DFS(G)
2 output nodes in order of decreasing finish times



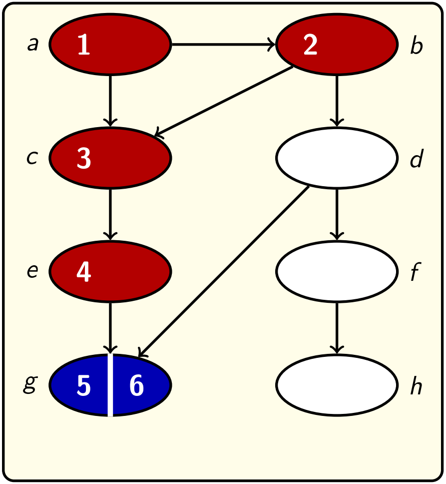
time $= 5$

Toplogical Sorting

TOPOLOGICAL-SEARCH(G)
1 call DFS(G)
2 output nodes in order
of decreasing finish times

g

time = 6

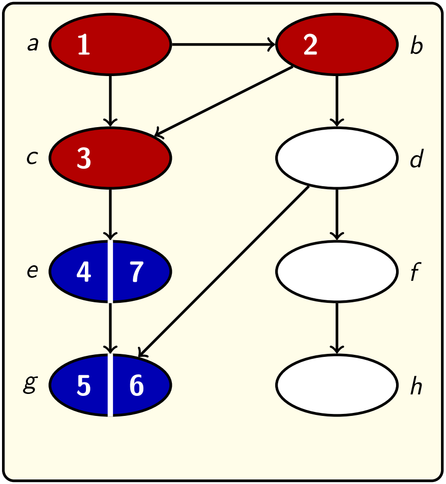


Toplogical Sorting

TOPOLOGICAL-SEARCH(G)
1 call DFS(G)
2 output nodes in order of decreasing finish times

e	g

time = 7

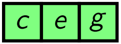


Toplogical Sorting

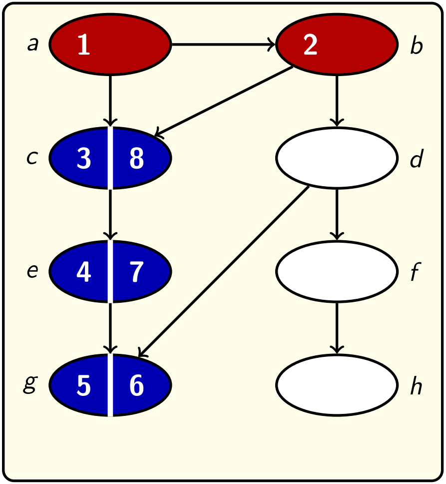
TOPOLOGICAL-SEARCH(G)

1 call DFS(G)

2 output nodes in order of decreasing finish times



time = 8



Toplogical Sorting

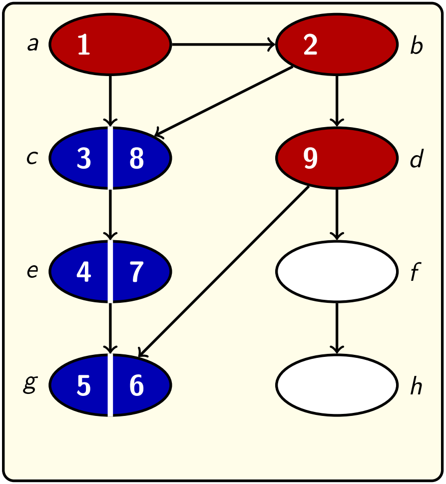
TOPOLOGICAL-SEARCH(G)

1 call DFS(G)

2 output nodes in order of decreasing finish times

c	e	g

time $= 9$



Toplogical Sorting

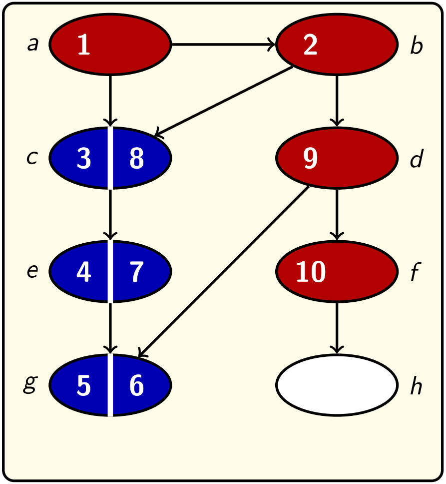
TOPOLOGICAL-SEARCH(G)

1 call DFS(G)

2 output nodes in order of decreasing finish times

c	e	g

time = 10



Toplogical Sorting

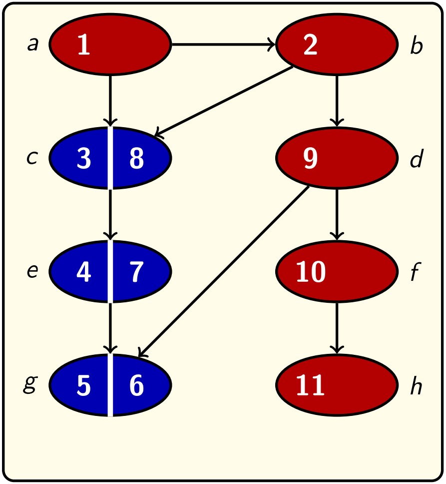
TOPOLOGICAL-SEARCH(G)

1 call DFS(G)

2 output nodes in order of decreasing finish times

c	e	g

time $= 11$



Toplogical Sorting

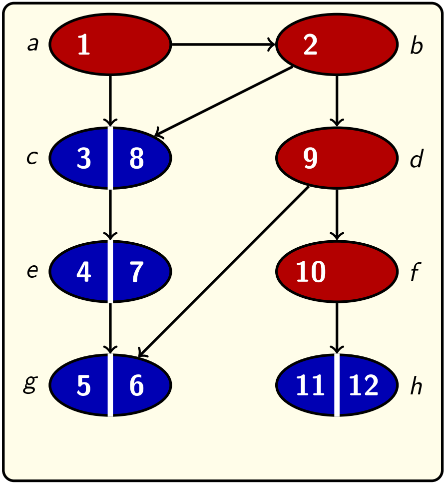
TOPOLOGICAL-SEARCH(G)

1 call DFS(G)

2 output nodes in order of decreasing finish times

h	c	e	g

time = 12



Toplogical Sorting

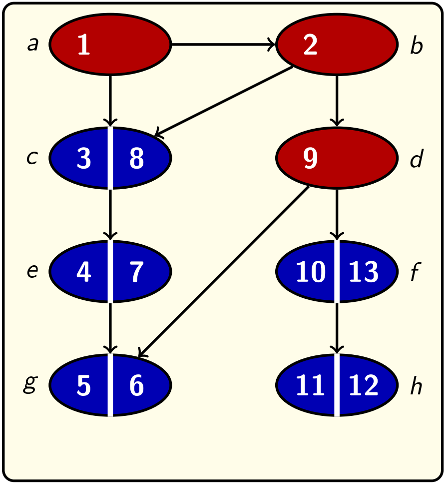
TOPOLOGICAL-SEARCH(G)

1 call DFS(G)

2 output nodes in order of decreasing finish times

f	h	c	e	g

time = 13



Toplogical Sorting

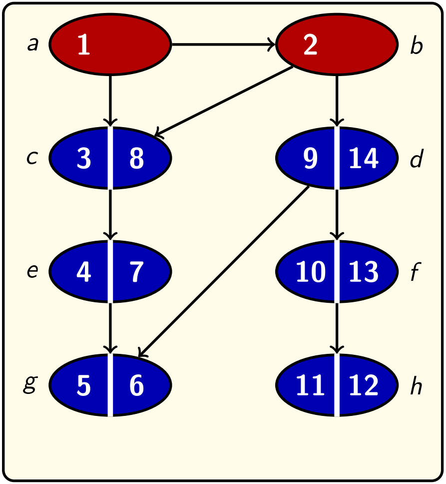
TOPOLOGICAL-SEARCH(G)

1 call DFS(G)

2 output nodes in order of decreasing finish times

d	f	h	c	e	g

time = 14

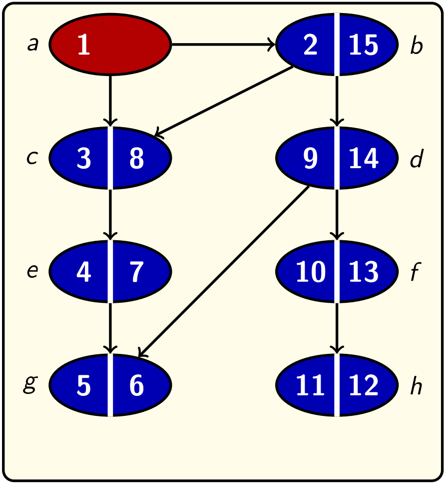


Toplogical Sorting

TOPOLOGICAL-SEARCH(G)
1 call DFS(G)
2 output nodes in order of decreasing finish times

b	d	f	h	c	e	g

time = 15

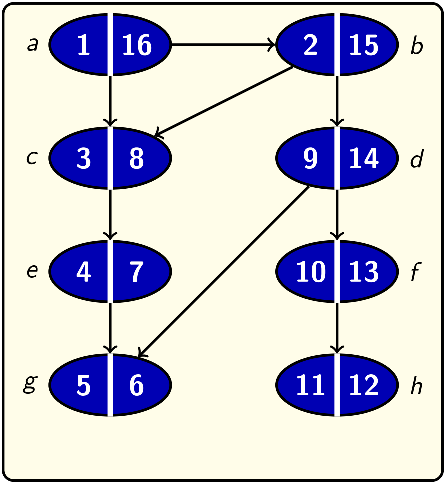


Toplogical Sorting

TOPOLOGICAL-SEARCH(G)
1 call DFS(G)
2 output nodes in order
of decreasing finish times

a	b	d	f	h	c	e	g

time = 16

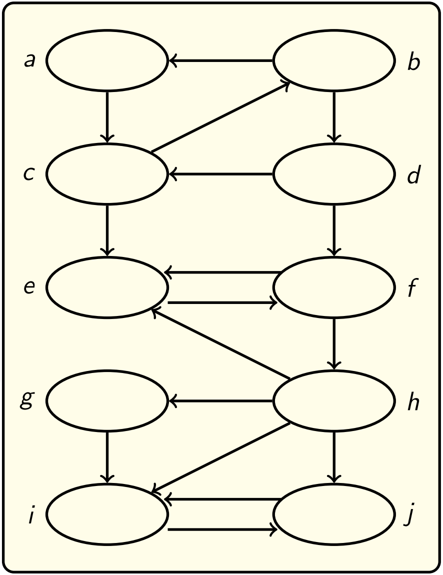


Topological Sort: Complexity

```
TOP-SORT(G)
for each vertex u ∈ G : V
        do u.color ← WHITE
           u.π ← NIL
time ← 0
for each vertex u ∈ G.V
        do if u.color = WHITE
              then TOP-SORT-VISIT(u)

TOP-SORT-VISIT(u)
u.color ← RED
time ← time +1
u.d ← time
for each v ∈ G.Adj[u]
        do if v.color = WHITE
              then v.π ← u
                   DFS-VISIT(v)
u.color ← BLUE
time ← time +1
u.f ← time
print u
```

- Initialization costs $O(|V|)$
- The procedure TOP-SORT-VISIT is called exactly once for each node v.
- During an execution of TOP-SORT-VISIT(v), the **for** loop executes $|G.Adj[v]|$ times.
- Total time $= O(|V| + |E|)$

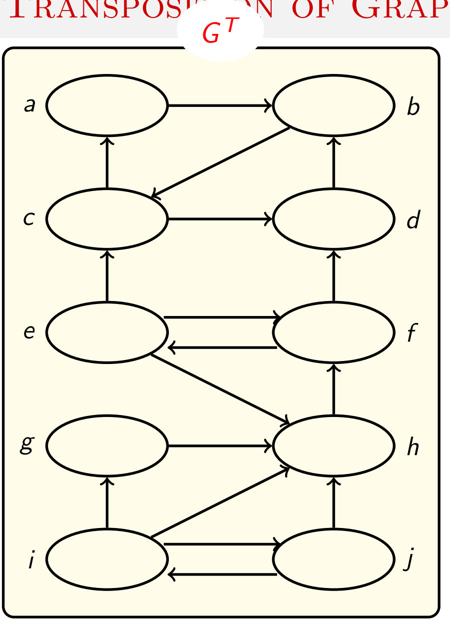
Transposition of Graphs

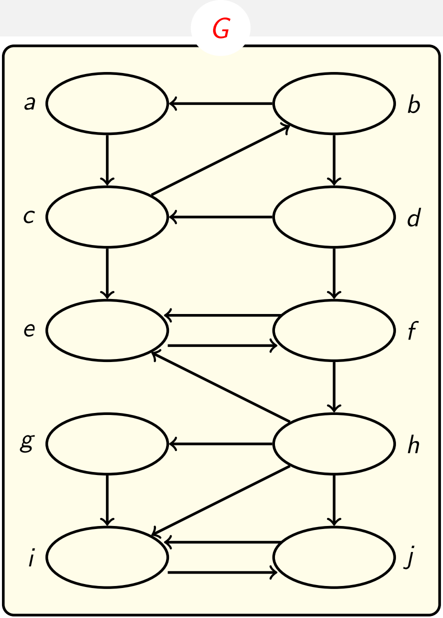


Transposition of Graphs

The transposition of a graph $G = (V, E)$ is a graph $G^T = (V, E^T)$ where $E^T = \{(u, v) \mid (v, u) \in E\}$ (i.e., all the edges are reversed)

Transposition of Graphs





Strongly Connected Components

Strongly connected components

A Strongly Connected Component (SCC) of a graph $G = (V, E)$ is a maximal set of nodes $C \subseteq V$ such that every two nodes $u, v \in C$ are reachable from each other

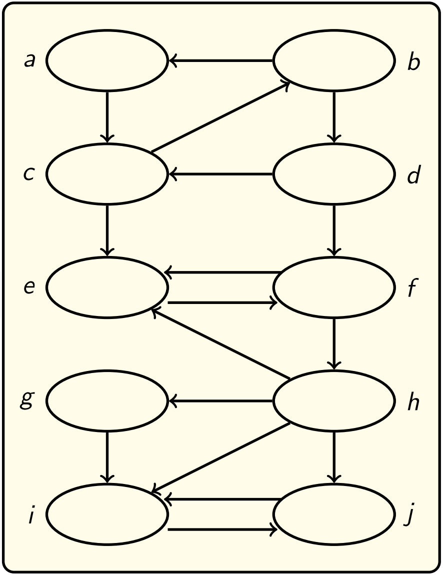
Observation

G^T and G have the same set of SCC's

Component Graph

The component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ of G is defined as follows:

- V^{SCC} has one node for each SCC in G
- E^{SCC} has an edge if there is an edge between the two corresponding SCC's in G



Strongly Connected Components

Strongly connected components

A Strongly Connected Component (SCC) of a graph $G = (V, E)$ is a maximal set of nodes $C \subseteq V$ such that every two nodes $u, v \in C$ are reachable from each other

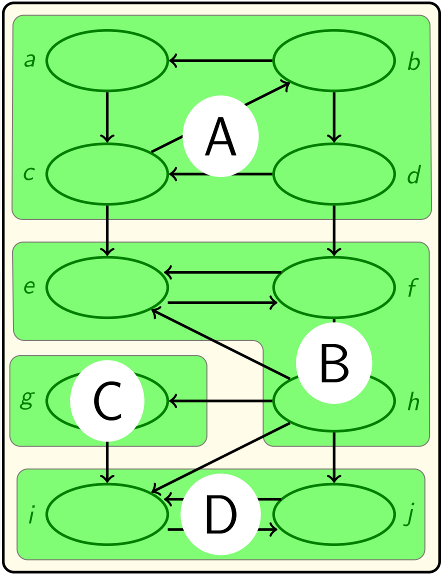
Observation

G^T and G have the same set of SCC's

Component Graph

The component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ of G is defined as follows:

- V^{SCC} has one node for each SCC in G
- E^{SCC} has an edge if there is an edge between the two corresponding SCC's in G



Strongly Connected Components

Strongly connected components

A Strongly Connected Component (SCC) of a graph $G = (V, E)$ is a maximal set of nodes $C \subseteq V$ such that every two nodes $u, v \in C$ are reachable from each other

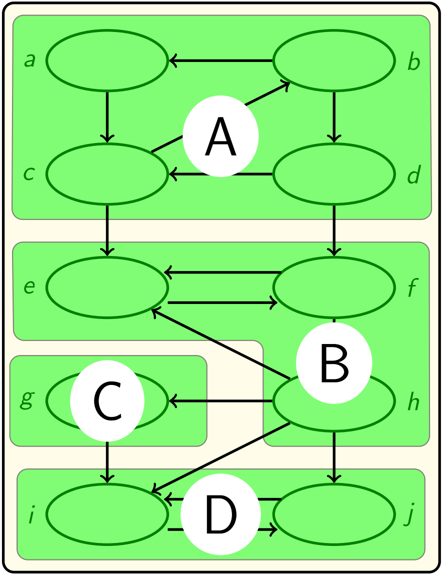
Observation

G^T and G have the same set of SCC's

Component Graph

The component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ of G is defined as follows:

- V^{SCC} has one node for each SCC in G
- E^{SCC} has an edge if there is an edge between the two corresponding SCC's in G



Strongly Connected Components

Strongly connected components

A Strongly Connected Component (SCC) of a graph $G = (V, E)$ is a maximal set of nodes $C \subseteq V$ such that every two nodes $u, v \in C$ are reachable from each other

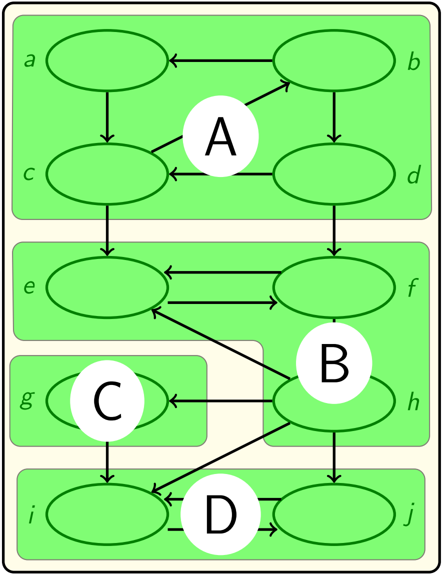
Observation

G^T and G have the same set of SCC's

Component Graph

The component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ of G is defined as follows:

- V^{SCC} has one node for each SCC in G
- E^{SCC} has an edge if there is an edge between the two corresponding SCC's in G



Strongly Connected Components

Strongly connected components

A Strongly Connected Component (SCC) of a graph $G = (V, E)$ is a maximal set of nodes $C \subseteq V$ such that every two nodes $u, v \in C$ are reachable from each other

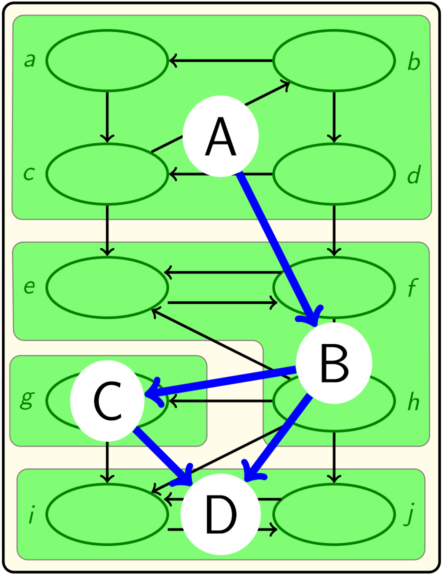
Observation

G^T and G have the same set of SCC's

Component Graph

The component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ of G is defined as follows:

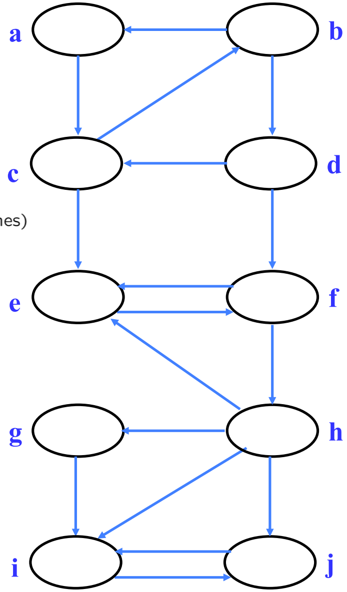
- V^{SCC} has one node for each SCC in G
- E^{SCC} has an edge if there is an edge between the two corresponding SCC's in G



Strongly Connected Components

SCC(G)

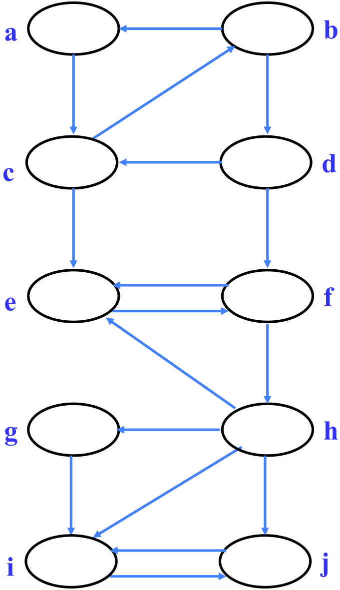
1 call $DFS(G)$ to compute finishing times
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 each tree in depth-first forest = SCC



Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

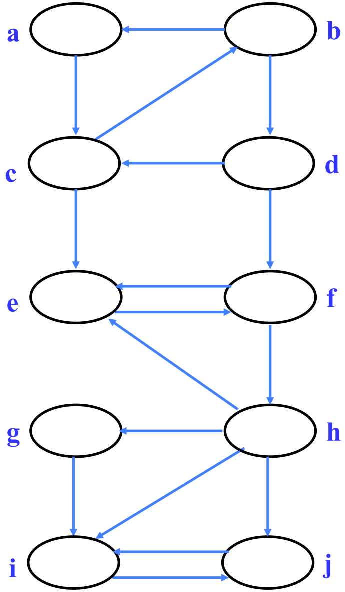


Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 0



Strongly Connected Components

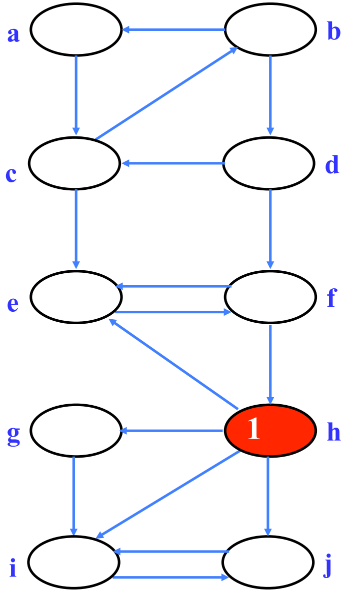
SCC(G)

1 call $DFS(G)$ to compute finishing times

2 .....

3 .....

time = 1

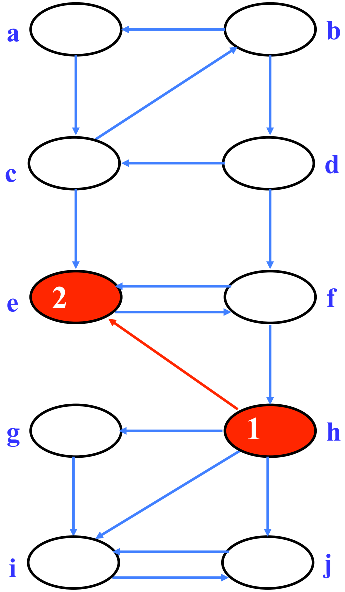


Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 2

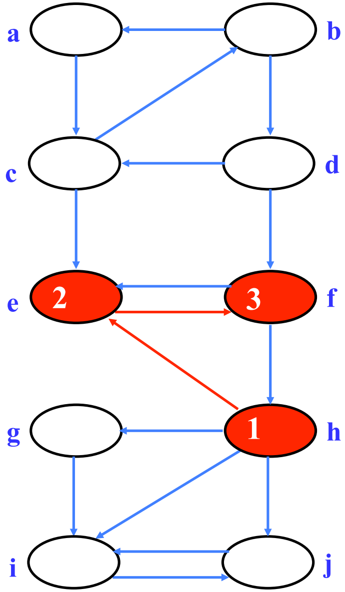


Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 3



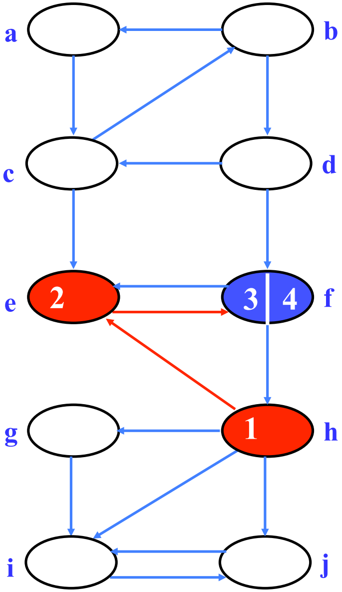
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 4

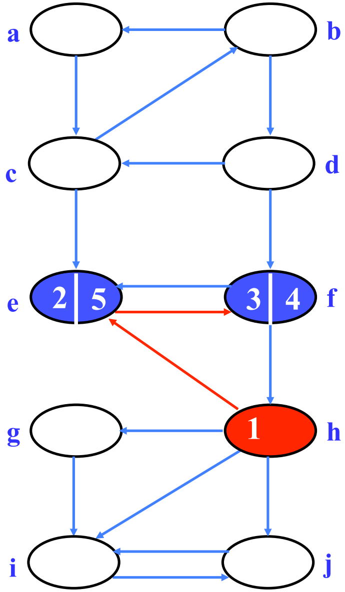
f



Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....



time = 5

e f

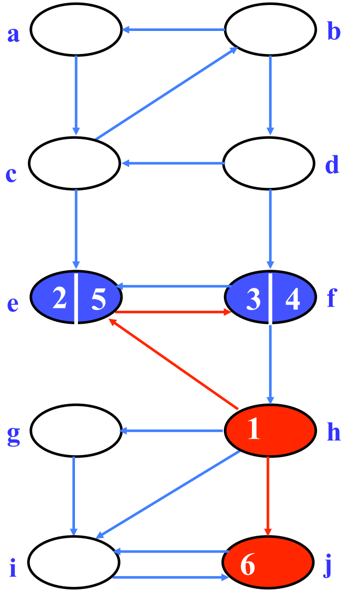
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 6

e f



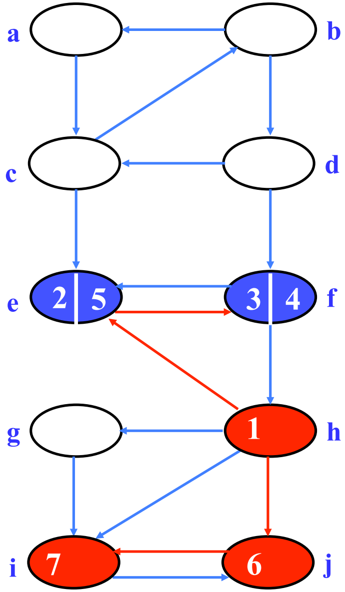
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 7

e f



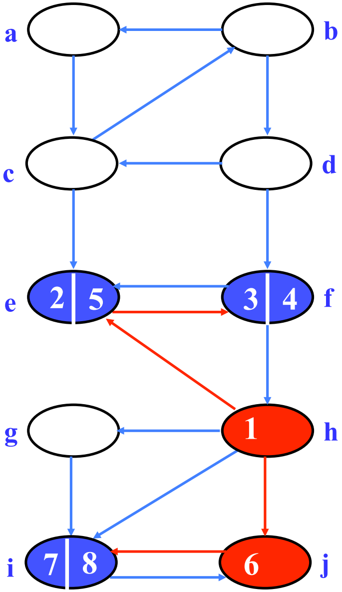
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 8

i e f



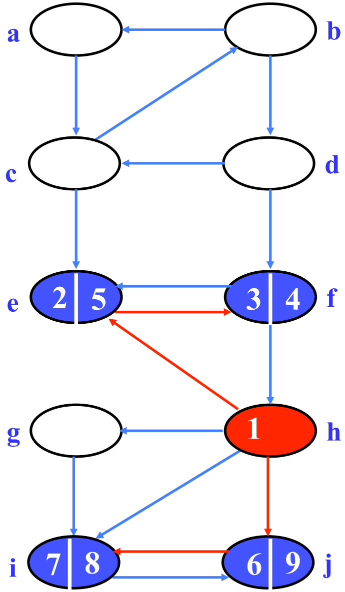
Strongly Connected Components

$\text{SCC}(G)$

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 9

j i e f



Strongly Connected Components

SCC(G)

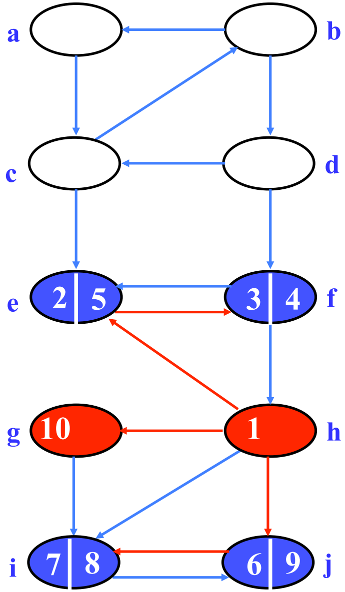
call $DFS(G)$ to compute finishing times

.....

.....

time = 10

j i e f



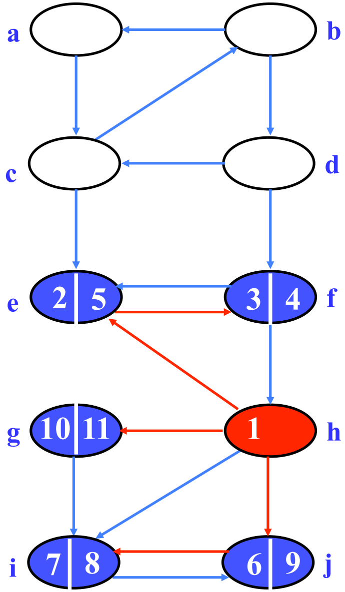
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 11

g j i e f



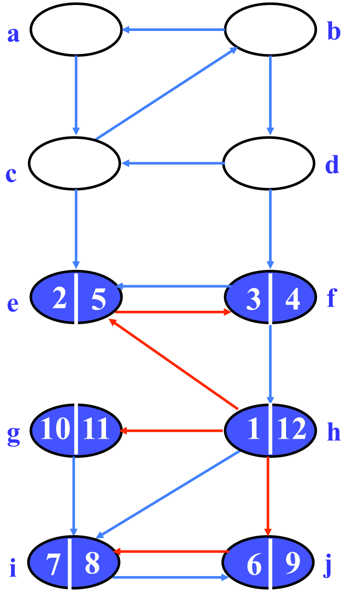
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 12

h g j i e f



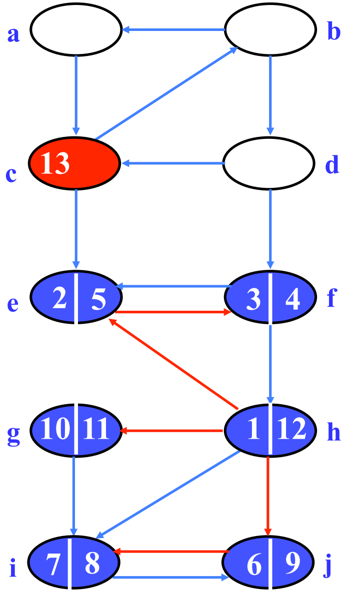
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 13

h g j i e f



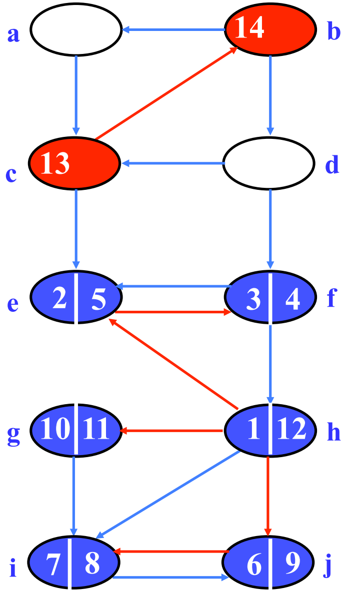
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 14

h g j i e f



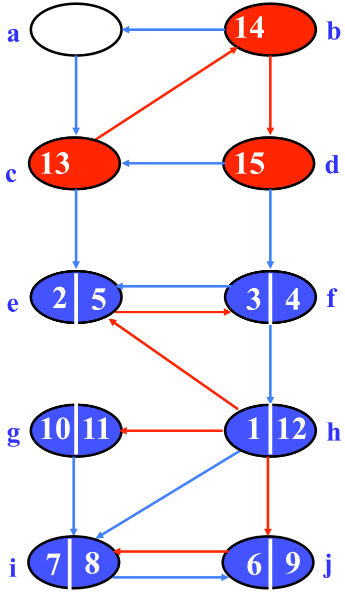
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 15

h g j i e f



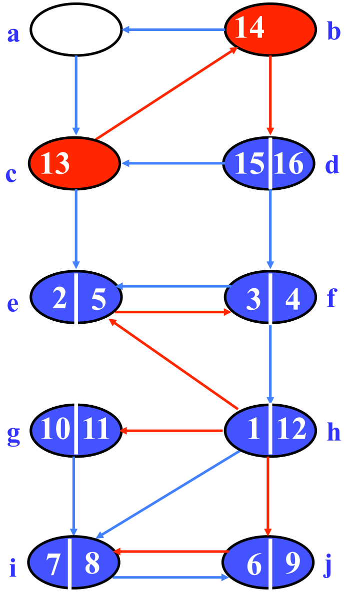
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 16

d h g j i e f



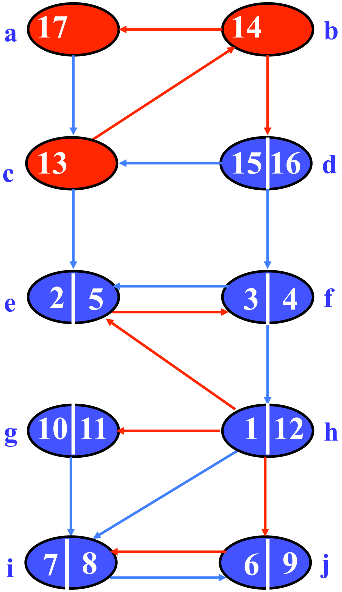
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 17

d h g j i e f



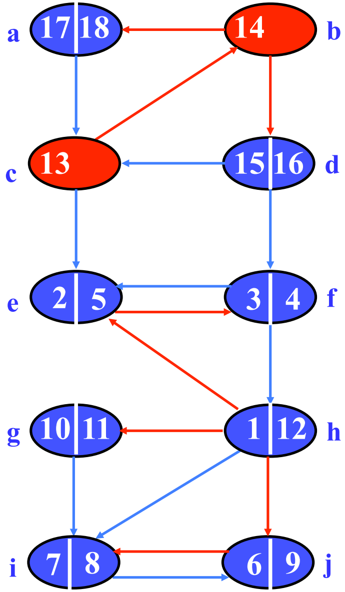
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 18

a d h g j i e f



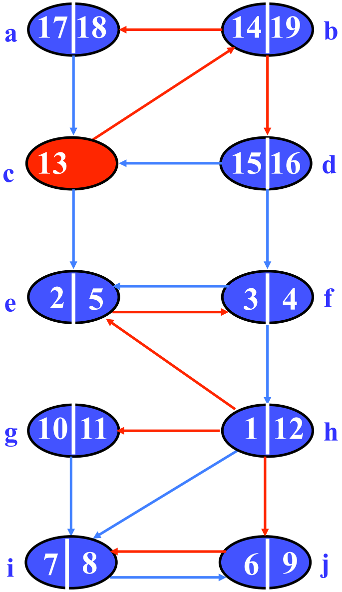
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 19

b a d h g j i e f



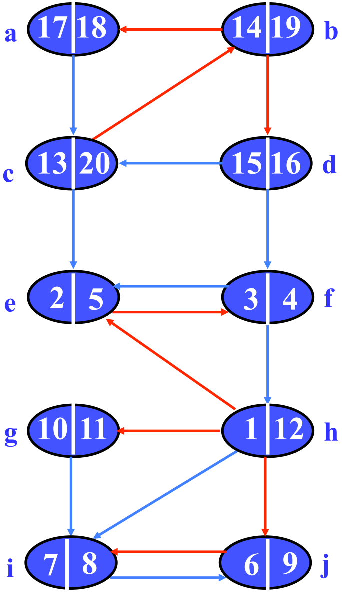
Strongly Connected Components

SCC(G)

1 call $DFS(G)$ to compute finishing times
2 .....
3 .....

time = 20

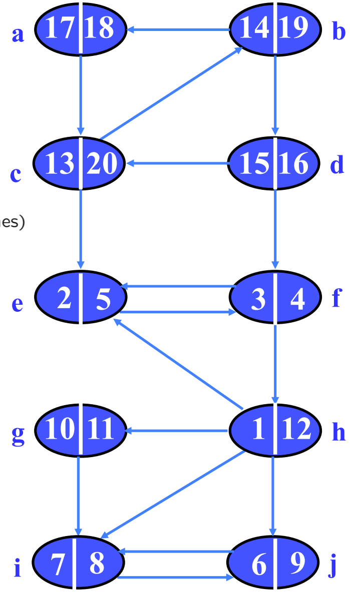
c b a d h g j i e f



Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....



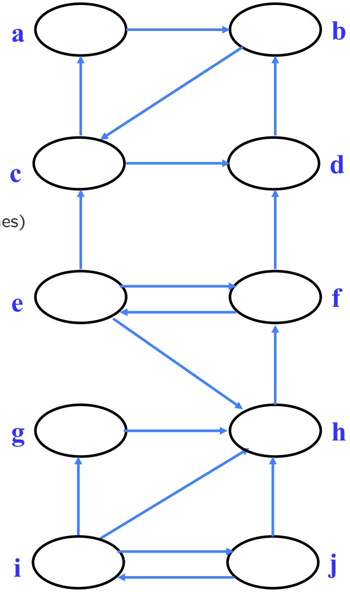
c b a d h g j i e f

Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

c b a d h g j i e f



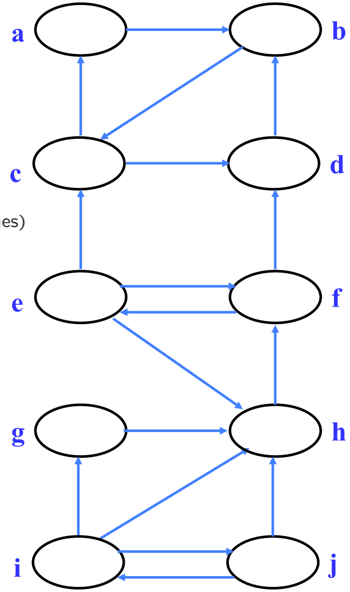
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 0

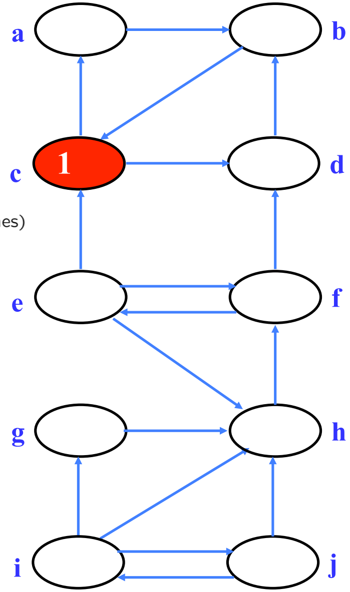
c b a d h g j i e f



Strongly Connected Components

SCC(G)

```
.....
Call DFS(G^T)
(call nodes in order of decreasing finishing times)
.....
```



time = 1

~~c~~ b a d h g j i e f

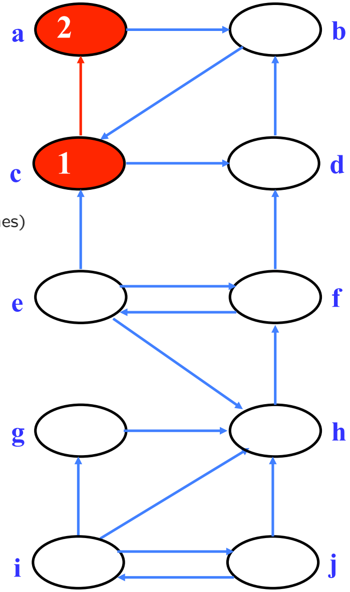
Strongly Connected Components

SCC(G)

```
.....
Call DFS(G^T)
(call nodes in order of decreasing finishing times)
.....
```

time = 2

~~c~~ b ~~a~~ d h g j i e f



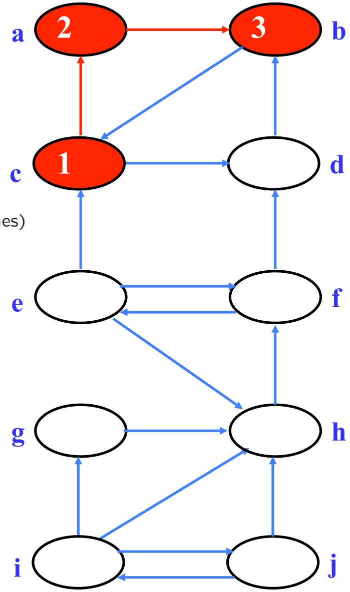
Strongly Connected Components

SCC(G)

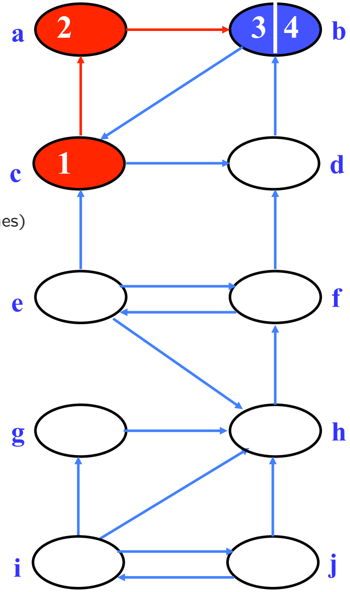
1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 3

~~c~~ ~~b~~ ~~a~~ d h g j i e f



Strongly Connected Components



SCC(G)

```
.....
Call DFS(G^T)
(call nodes in order of decreasing finishing times)
.....
```

time = 4

~~c~~ ~~b~~ ~~a~~ d h g j i e f

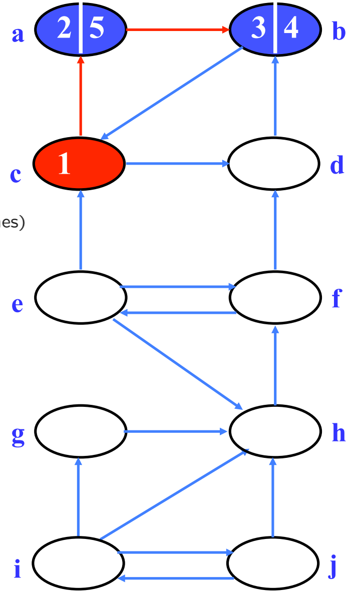
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 5

~~c~~ ~~b~~ ~~a~~ d h g j i e f



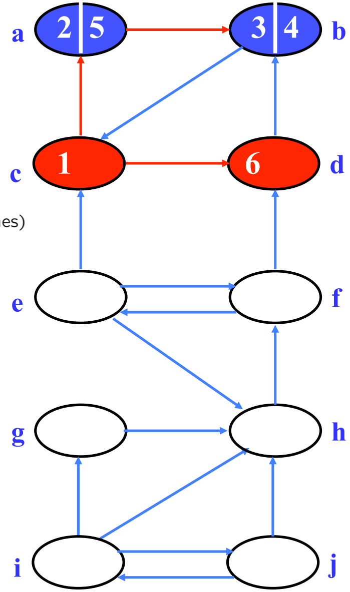
Strongly Connected Components

SCC(G)

```
.....
Call DFS(G^T)
(call nodes in order of decreasing finishing times)
.....
```

time = 6

~~c~~ ~~b~~ ~~a~~ ~~d~~ h g j i e f



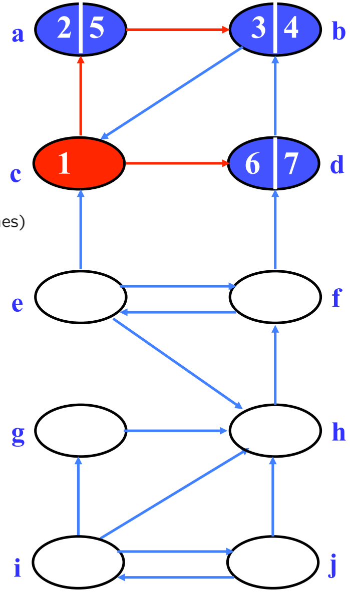
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 7

~~c~~ ~~b~~ ~~a~~ ~~d~~ h g j i e f



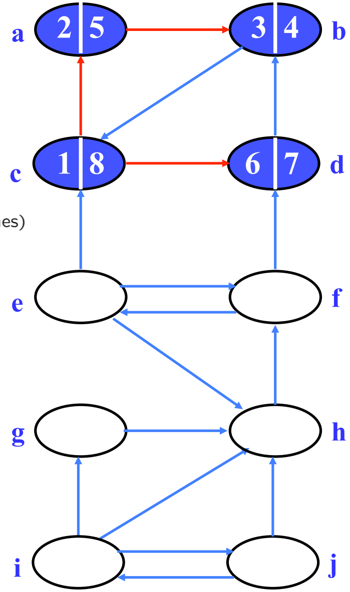
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 8

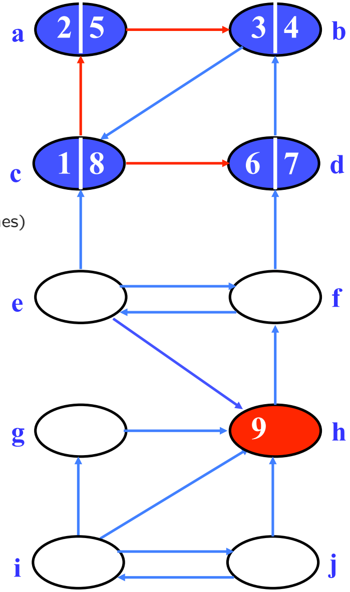
~~c~~ ~~b~~ ~~a~~ ~~d~~ h g j i e f



Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....



time = 9

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ g j i e f

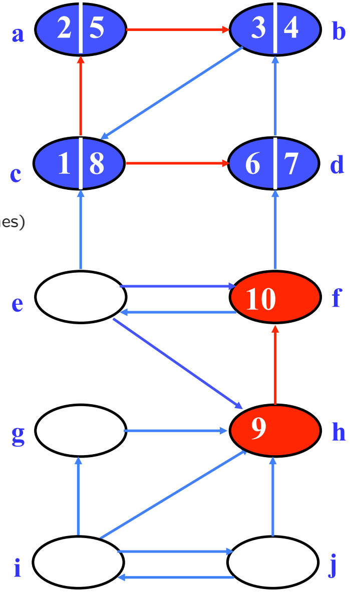
Strongly Connected Components

SCC(G)

```
.....
Call DFS(G^T)
(call nodes in order of decreasing finishing times)
.....
```

time = 10

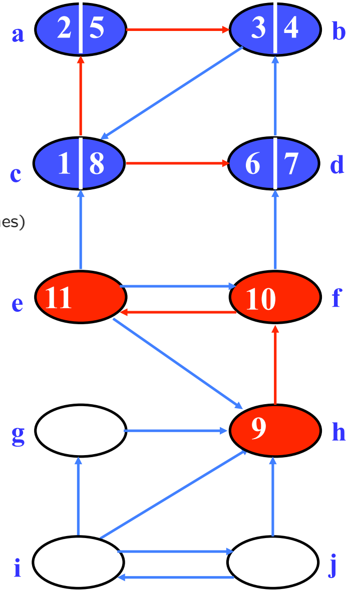
~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ g j i e ~~f~~



Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....



time = 11

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ g j i ~~e~~ ~~f~~

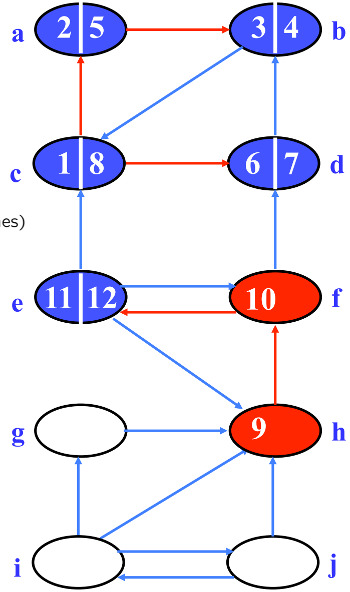
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 12

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ g j i ~~e~~ ~~f~~



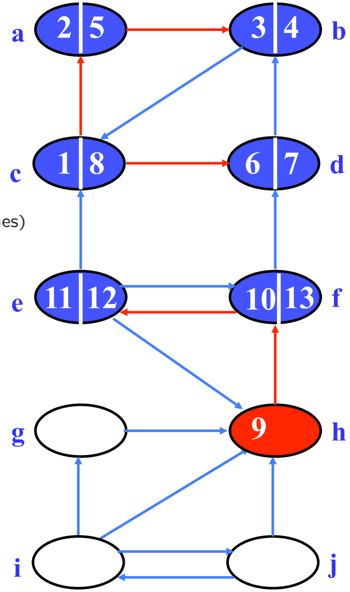
Strongly Connected Components

SCC(G)

```
.....
Call DFS(G^T)
(call nodes in order of decreasing finishing times)
.....
```

time = 13

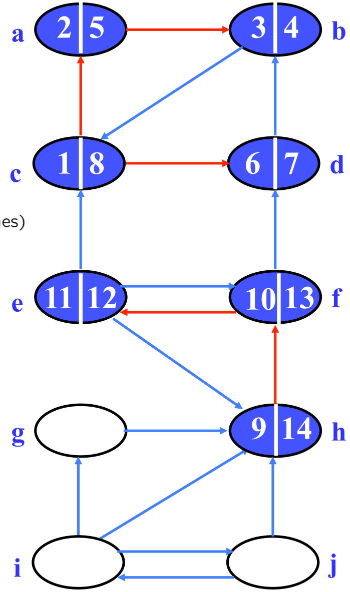
~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ g j i ~~e~~ ~~f~~



Strongly Connected Components

SCC(G)

```
.....
Call DFS(G^T)
(call nodes in order of decreasing finishing times)
.....
```



time = 14

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ g j i ~~e~~ ~~f~~

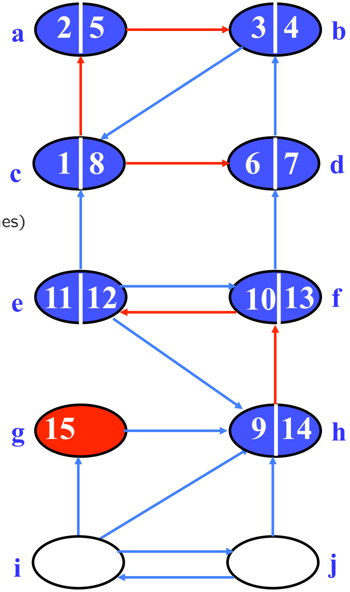
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 15

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ ~~g~~ j i ~~e~~ ~~f~~



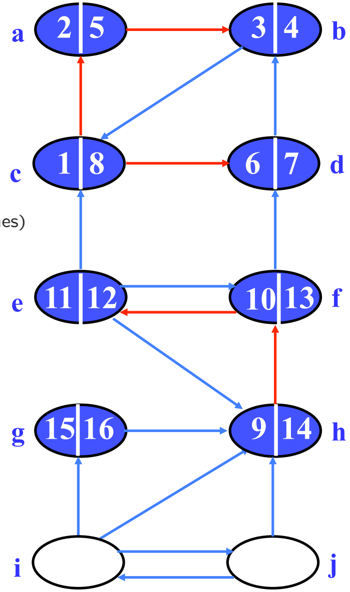
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
  (call nodes in order of decreasing finishing times)
3 .....

time = 16

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ ~~g~~ j i ~~e~~ ~~f~~



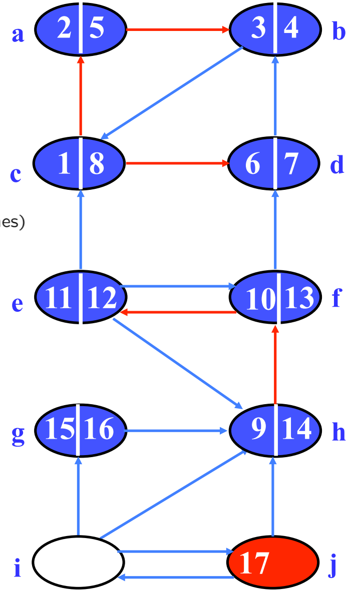
Strongly Connected Components

SCC(G)

.....
Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
.....

time = 17

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ ~~g~~ ~~j~~ i ~~e~~ ~~f~~



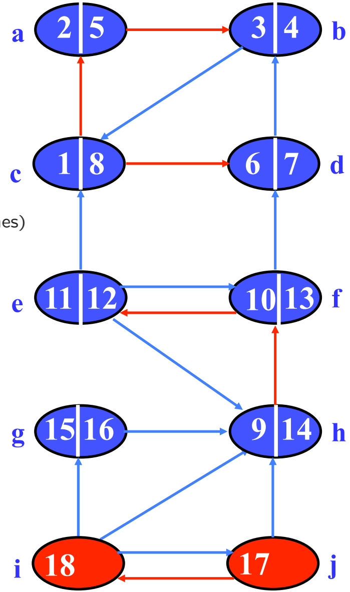
Strongly Connected Components

SCC(G)

.....
Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
.....

time = 18

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ ~~g~~ ~~j~~ ~~i~~ ~~e~~ ~~f~~



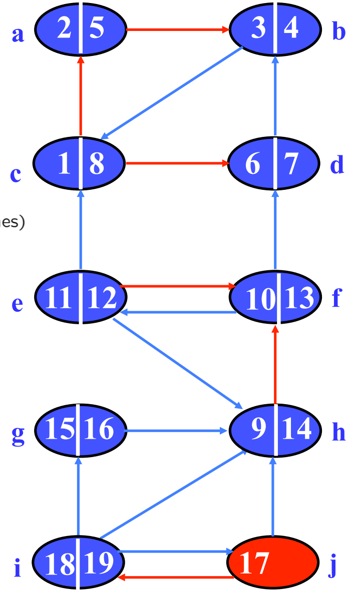
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 19

~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ ~~g~~ ~~j~~ ~~i~~ ~~e~~ ~~f~~



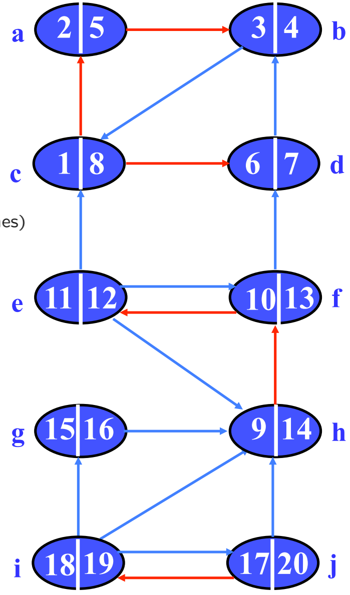
Strongly Connected Components

SCC(G)

1 .....
2 Call $DFS(G^T)$
(call nodes in order of decreasing finishing times)
3 .....

time = 20

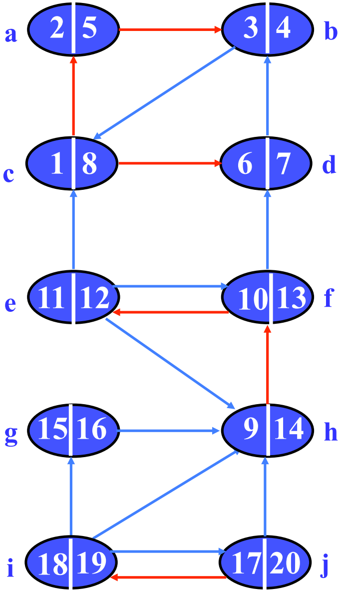
~~c~~ ~~b~~ ~~a~~ ~~d~~ ~~h~~ ~~g~~ ~~j~~ ~~i~~ ~~e~~ ~~f~~



Strongly Connected Components

SCC(G)

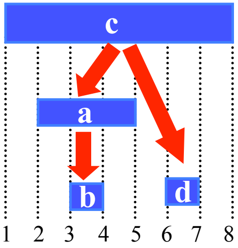
1 .....
2 .....
3 each tree in depth-first forest = SCC

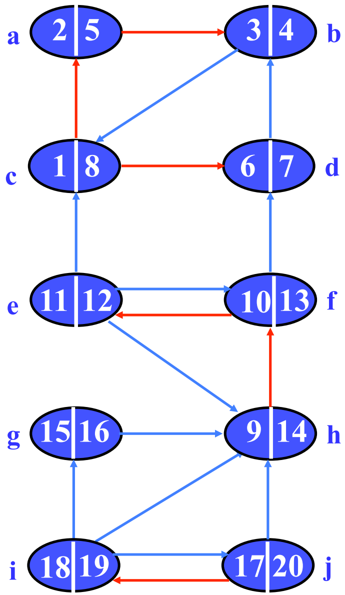


Strongly Connected Components

SCC(G)

1 .....
2 .....
3 each tree in depth-first forest = SCC

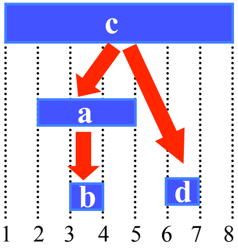


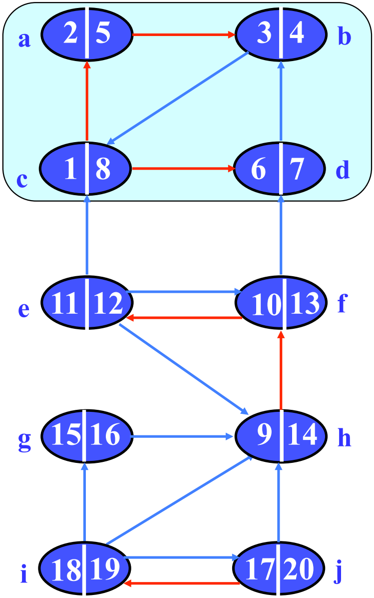


Strongly Connected Components

SCC(G)

1 .....
2 .....
3 each tree in depth-first forest = SCC

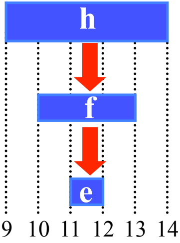


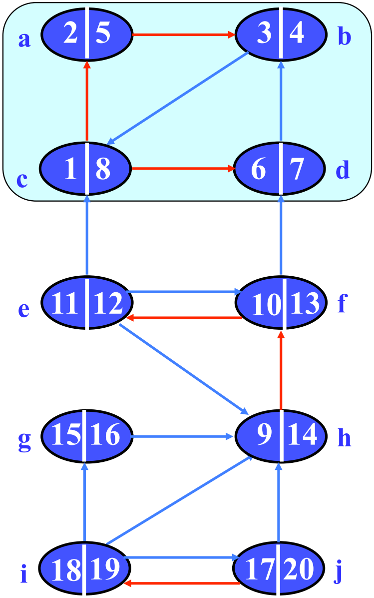


Strongly Connected Components

SCC(G)

1 .....
2 .....
3 each tree in depth-first forest = SCC

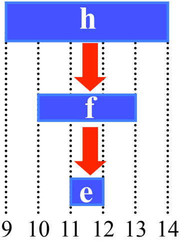


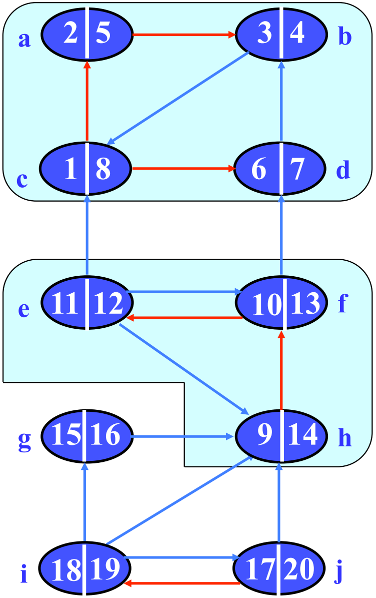


Strongly Connected Components

SCC(G)

1 .....
2 .....
3 each tree in depth-first forest = SCC

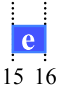


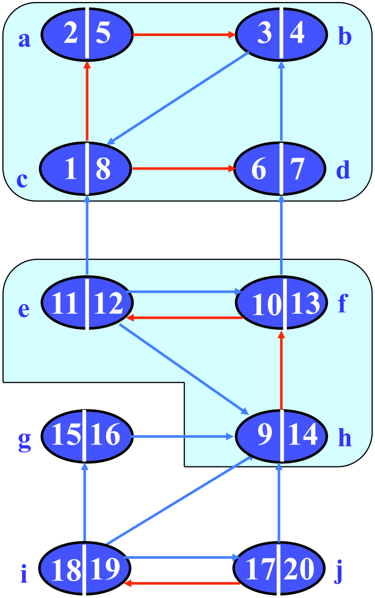


Strongly Connected Components

SCC(G)

1 .....
2 .....
3 each tree in depth-first forest = SCC

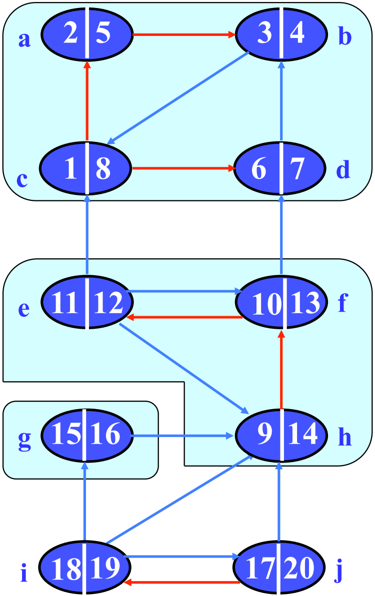




Strongly Connected Components

$\text{SCC}(G)$

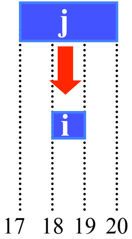
```
.....
.....
each tree in depth-first forest = SCC
```

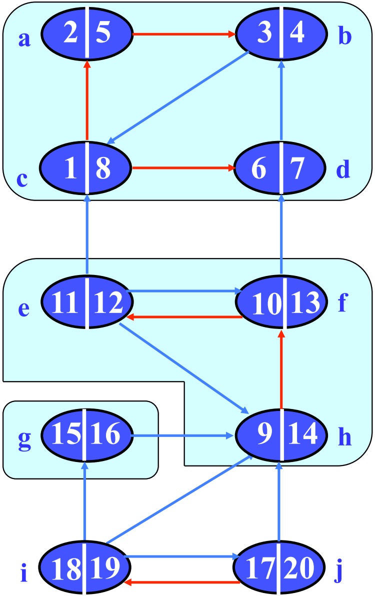


Strongly Connected Components

SCC(G)

1 .....
2 .....
3 each tree in depth-first forest = SCC





Strongly Connected Components

SCC(G)

1 .....
2 .....
3 each tree in depth-first forest = SCC