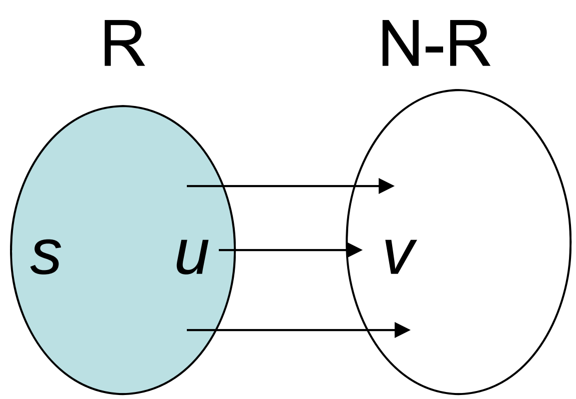
Depth First Search
Acyclicity
Graph Components

CS 4231, Fall 2020 Mihalis Yannakakis

Depth-First Search from a source s



- Policy: Choose edge (u,v) from R (reached nodes) to N-R (unreached) where u is latest node added to reachable set R

- Can write as a recursive algorithm, or

  implement using a stack S for nodes that have been reached and are not completely processed.

Depth-First Search from a source s

Simple version of DFS:

Depth-First-Search(G,s)

for each $u \in N$ do {mark[u]=0; p[u]=$\perp$}
DFS(s)

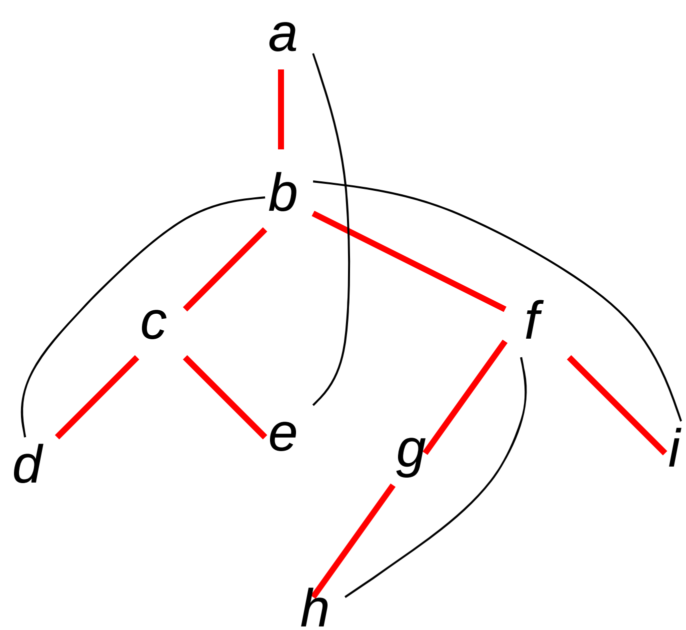
DFS(u)

mark[u]=1;
  for each $v \in$ Adj[u] do
    if mark[v]=0 then { p[v]=u; DFS(v)}

Can keep track of other information for various purposes

Time Complexity: O(n+e)

DFS Example



DFS Tree = Recursion Tree

Connected Components of an Undirected Graph

- Connected(u,v) = $\exists$ path connecting u,v
- Equivalence relation between nodes
  - reflexive, symmetric, transitive
- Equivalence classes = connected components

Computing Connected Components of an Undirected Graph

COMP(G)

```
c=0
for each u∈N do {mark[u]=0; p[u]=⊥}
for each u∈N do
      if mark[u]=0 then { c=c+1; DFSC(u)}
```

DFSC(u)

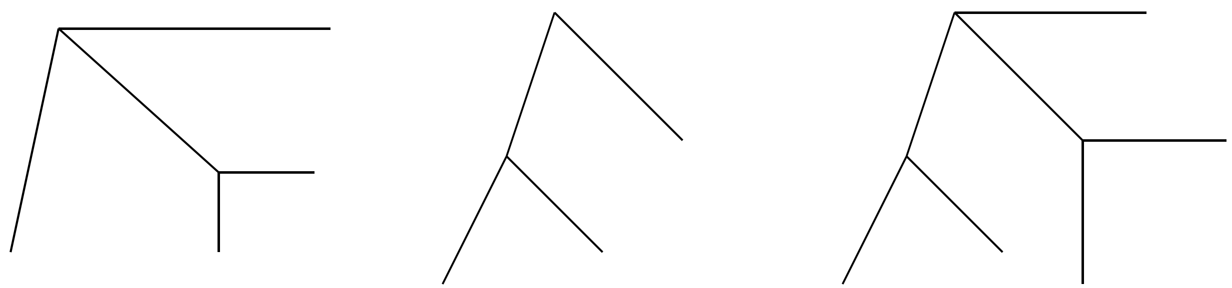
```
mark[u]=1; comp[u]=c;
   for each v ∈ Adj[u] do
     if mark[v]=0 then { p[v]=u;  DFSC(v)}
```

Time Complexity: O(n+e)

Instead of DFS, could use BFS or any other Search

Spanning Forest

- One tree for every connected component
- Forest has n-c edges, where c = #components



If there is only one connected component →
Spanning Tree: a tree that spans all the nodes

Testing Undirected Graph Acyclicity

- Graph acyclic ⇔ no other edges besides the spanning forest
- Modification to algorithm:

  if Search finds an edge (u,v) with mark[v]=1 and v≠p[u] then stop and return "cyclic"
- O(n) time
- Can trace the cycle using the parent information

Depth-First Search of a Graph

Depth-First-Search(G)

for each $u \in N$ do {mark[u]=0; p[u]=$\perp$; color[u]=white}

time=0

for each $u \in N$ do if mark[u]=0 then DFS(u)

DFS(u)

mark[u]=1; color[u] = gray;

time=time+1; d[u]=time; [discovery time of u]

for each $v \in$ Adj[u] do

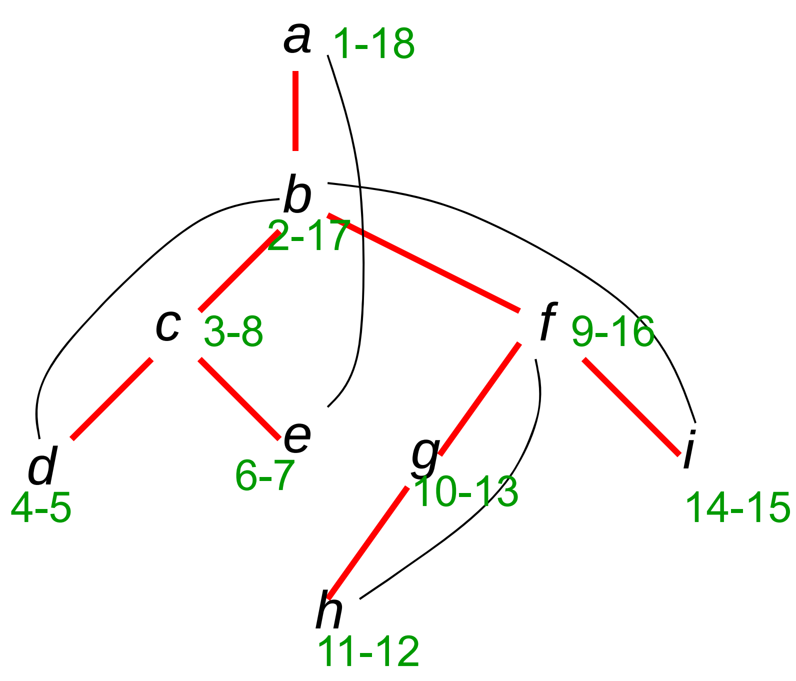
if mark[v]=0 then { p[v]=u; DFS(v)}

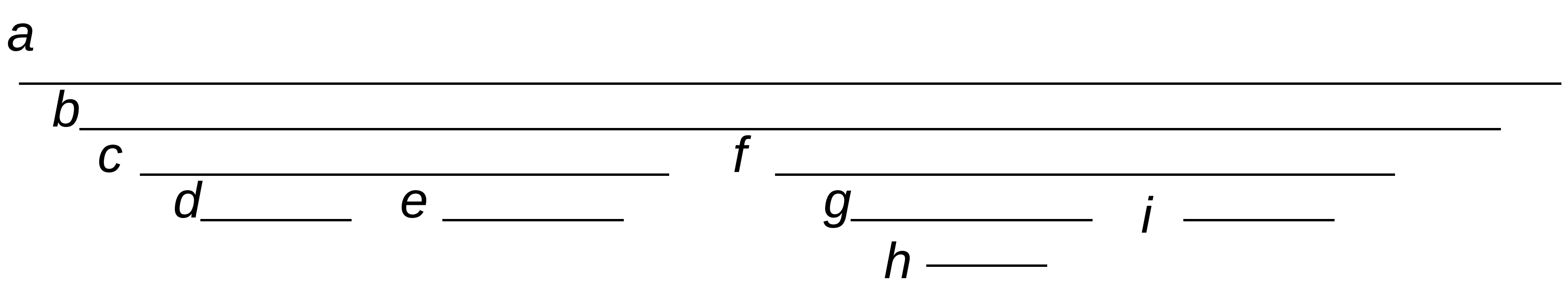
color[u]=black;

time=time+1; f[u]=time [finish time of u]

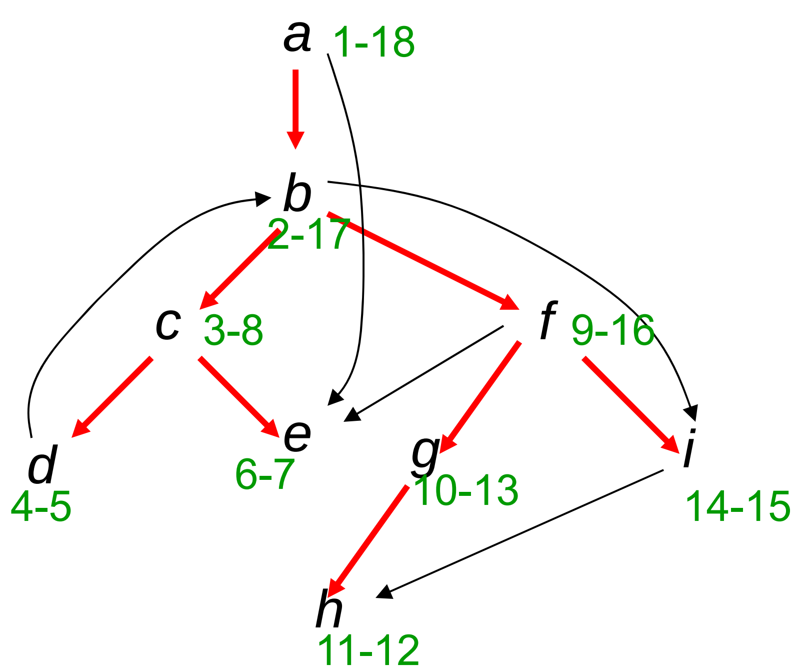
Time Complexity: O(n+e)

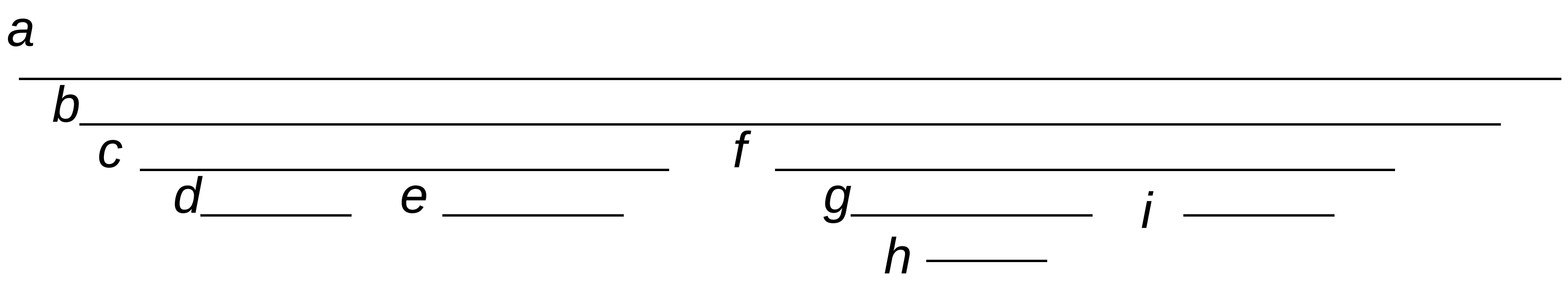
DFS Example





Example



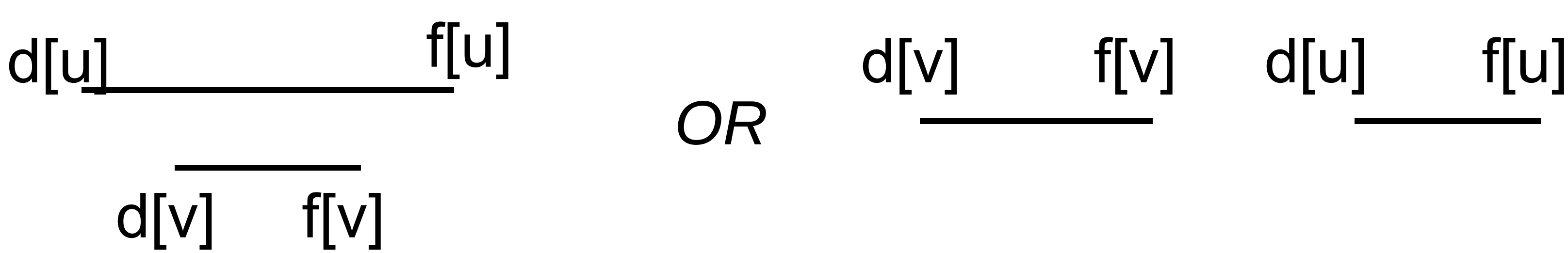


DFS TREE = Recursion Tree

- Preorder $\leftrightarrow$ d[.]   discovery times of nodes
- Postorder $\leftrightarrow$ f[.]   finish times of nodes

- u ancestor of v $\Rightarrow$ d[u] < d[v] < f[v] < f[u]
- u unrelated, after v $\Rightarrow$ d[v] < f[v] < d[u] < f[u]

- Intervals (d[.],f[.]) nested or disjoint:



Edge Classification

- Tree edge (u,v): u parent of v
- Forward edge: u ancestor of v
- Back edge: u descendant of v
- Cross edge: u unrelated to v

- Directed Graphs

  - Cross edges $u \rightarrow v$ go right to left:

    $d[v]<f[v]<d[u]<f[u]$

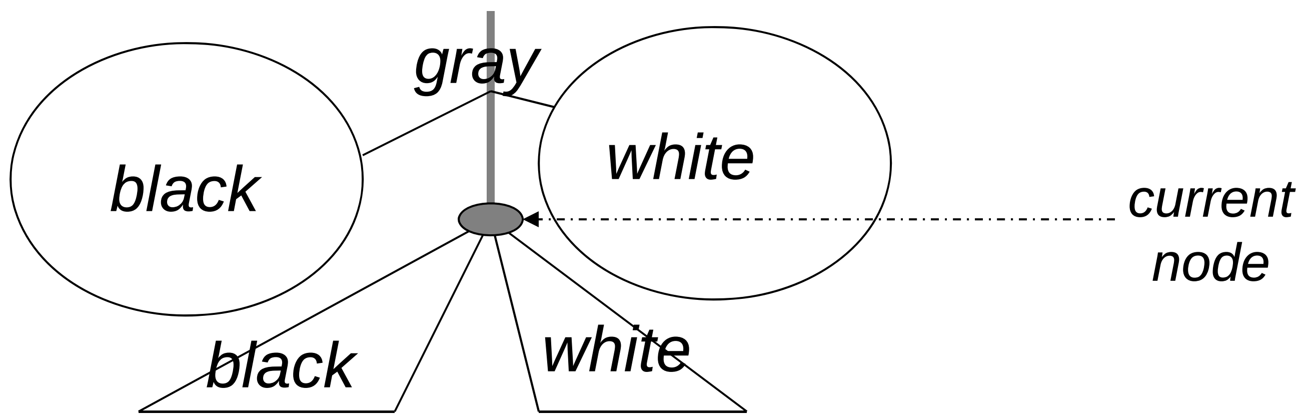
- Undirected Graphs:

  - No cross edges

    every edge is either tree edge or back edge

DFS Invariants

- At each point, nodes partitioned into *white* (unreached: N-R), *gray* (reached active: stack S), *black* (Done: R-S)
- Current stack S (gray nodes) = path of tree from root to current node
- Done (black) nodes: left of path
- Unreached (white) nodes: below and right of path in final DFS tree



DFS Invariants (ctd.)

- At each point:
- No black→white edges
- Every black ---→ white path has to go through a gray node, i.e., gray (active) nodes separate Done nodes from Unreached nodes

- White Path Theorem: u is ancestor of v $\Leftrightarrow$ at time d[u], $\exists$ white path from u to v

Proof: Color changes from time d[u] to f[u]:

Descendants of u change from white to black

Other nodes stay same color

Directed Acyclic Graphs (DAG)

- Applications:
- Scheduling tasks with precedence constraints in an order consistent with constraints
- Recursive calls (incl. arguments): will they run forever?
- Deadlock detection
- Circular definitions (for example in spreadsheet)

Directed Acyclic Graphs (DAG)

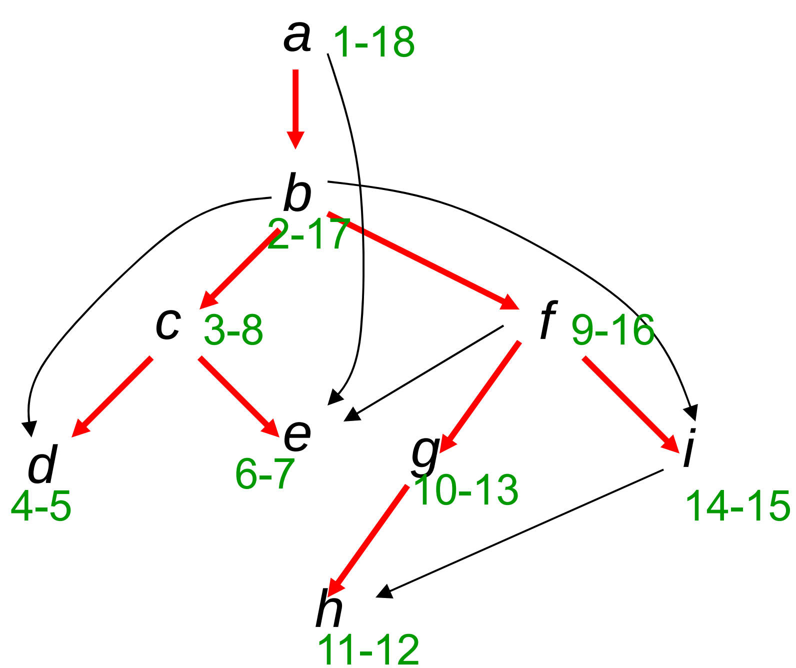
- Topological Sort: linear ordering of nodes so that all edges go left to right - in same direction

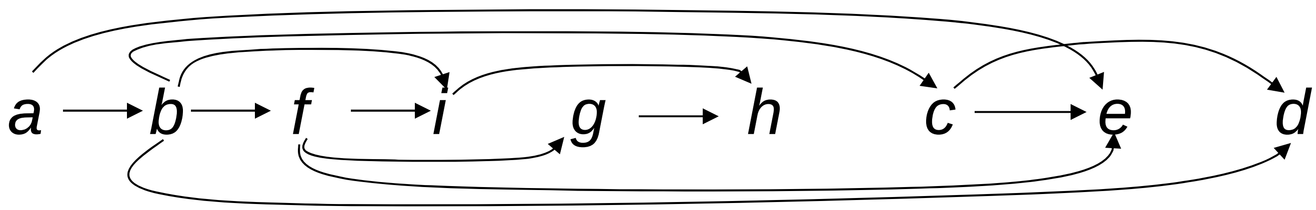
- Graph acyclic $\Leftrightarrow \exists$ topological sort
- Proof:
- topological sort => acyclic

  (cycle must use a right to left edge)

- acyclic => top sort: will give algorithmic proof with DFS

Example





Directed Acyclic Graphs (DAG)

Thm: Directed graph is acyclic $\Leftrightarrow$ no back edges in DFS

Proof:

- If back edge then cycle
- Suppose no back edge in DFS of graph.
- Reverse finishing (post) ordering of nodes in DFS of graph is a topological sort :

  ($u \rightarrow v \Rightarrow f[u] > f[v]$ for tree, forward, cross edges)

  => acyclic

- Detection of back edges: Back edge = edge to a gray node

$\Rightarrow$ Can test if a directed graph is acyclic and compute a topological sort in $O(n+e)$ time

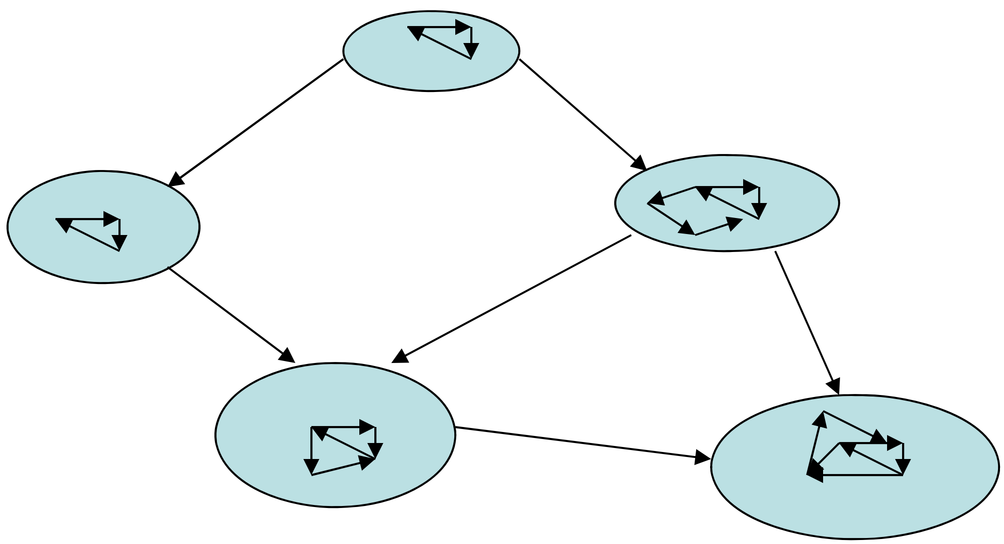
Strongly Connected Directed Graph

- "mutually reachable" relation: u - -→ v and v - - → u
- Strongly connected graph: All nodes mutually reachable
- Testing for strong connectivity:
  1. Pick any source node s and Search(G,s)
  2. Construct reverse graph Gr
  3. Search(Gr,s)
- Graph G is strongly connected iff s reaches all nodes in both G and Gr

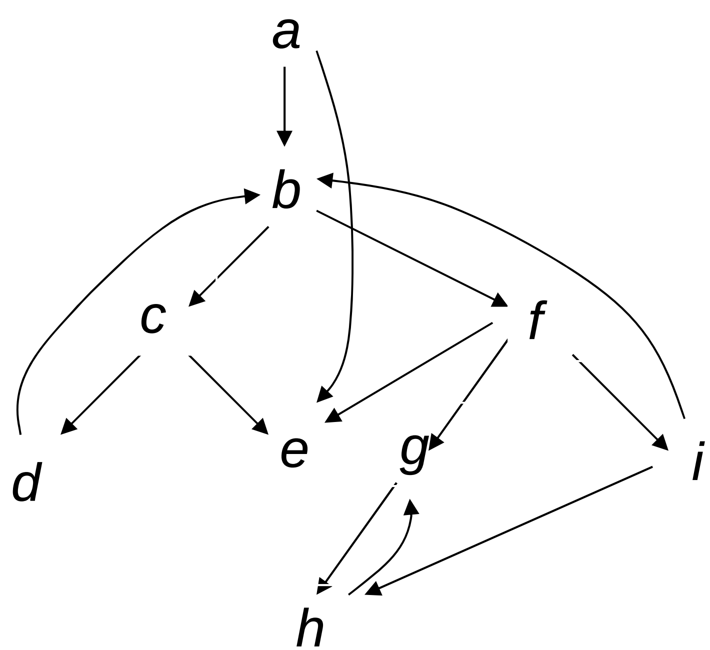
Strongly Connected Components of a Directed Graph

- "mutually reachable" relation is an equivalence relation
- Strongly Connected Components (SCC's) = equivalence classes
- Every cycle is contained in some SCC

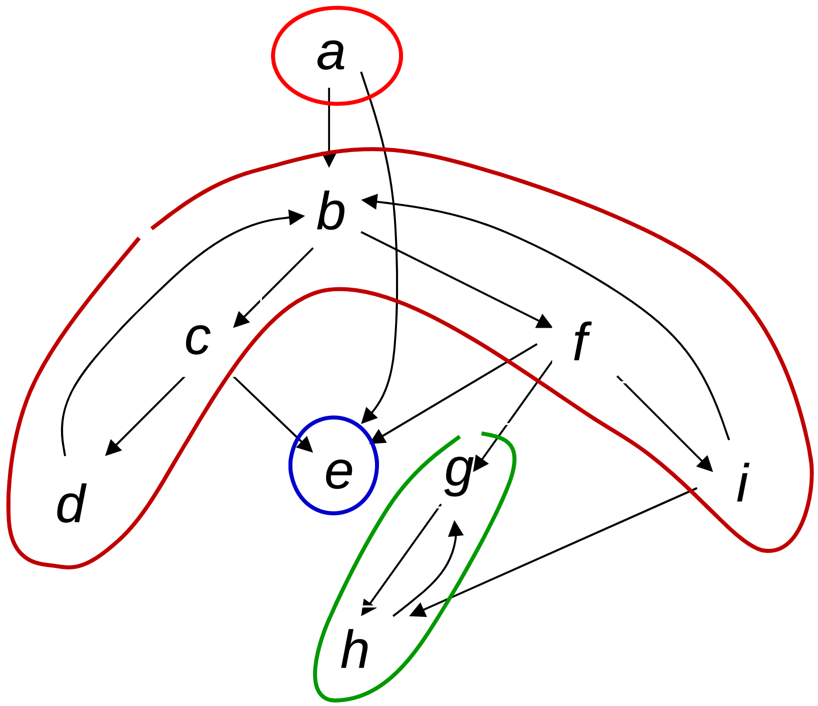
- Structure of a Digraph: DAG of SCC's



Example

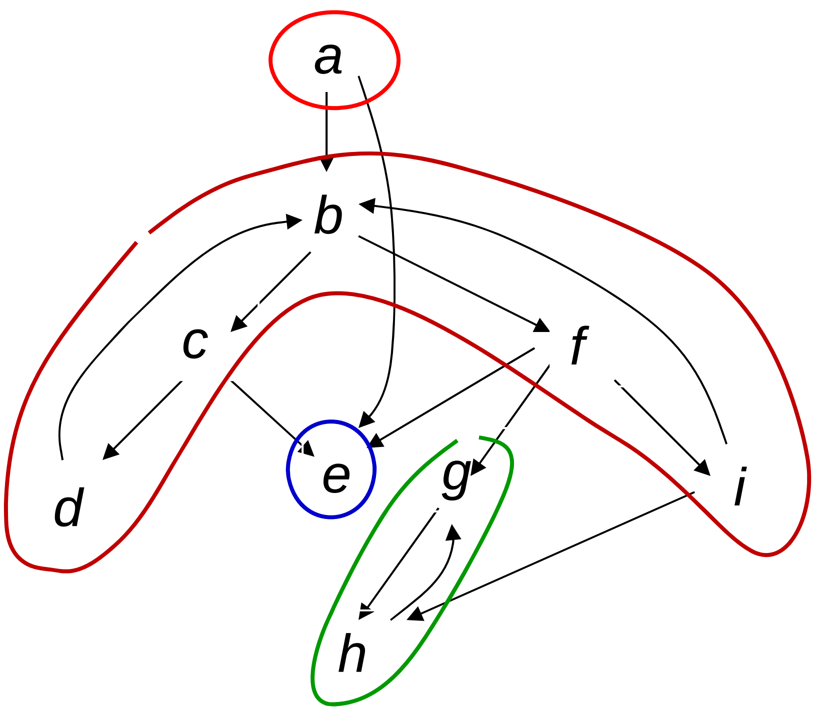


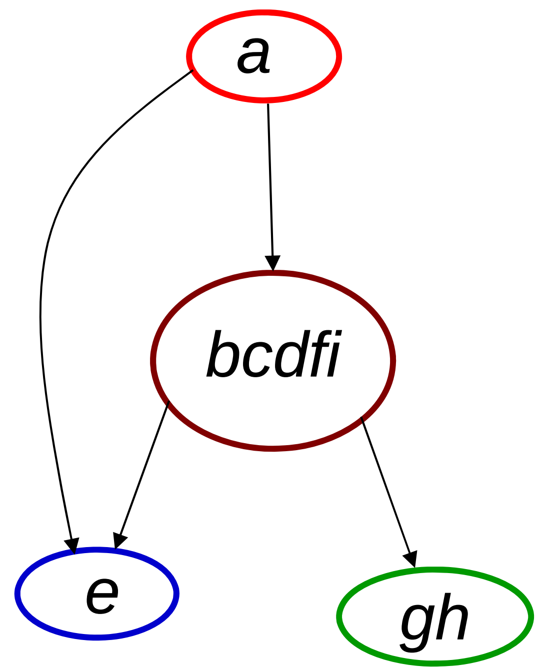
Example



SCC's: {a}, {b, c, d, f, i}, {e}, {g, h }

Example





DAG of SCC's

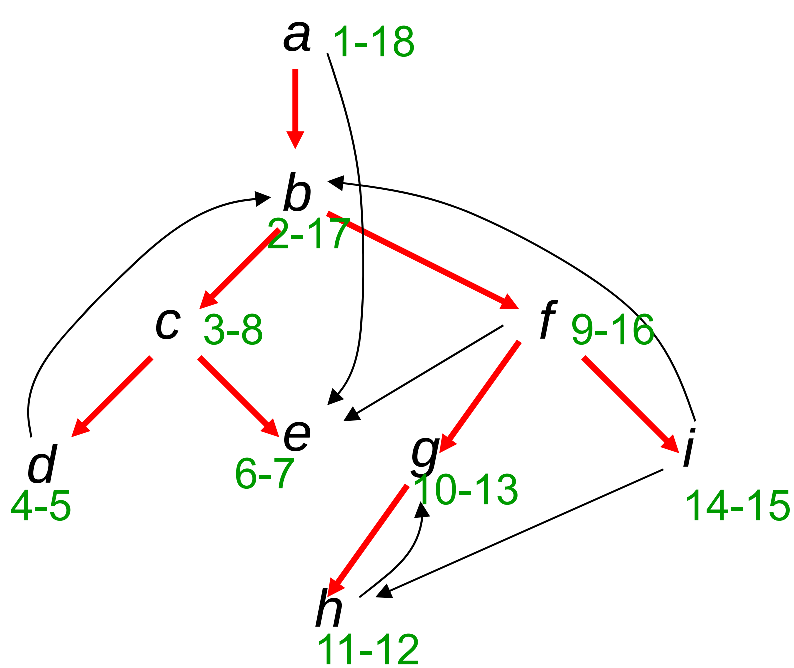
SCC Algorithm

- Call DFS(G) to compute f[u] for all nodes u
- Reverse the edges → digraph G_r
- DFS(G_r) with DFS(u) calls initiated in order of decreasing f[u]
- Nodes of DFS trees of second DFS = strongly connected components

Time Complexity: O(n+e)

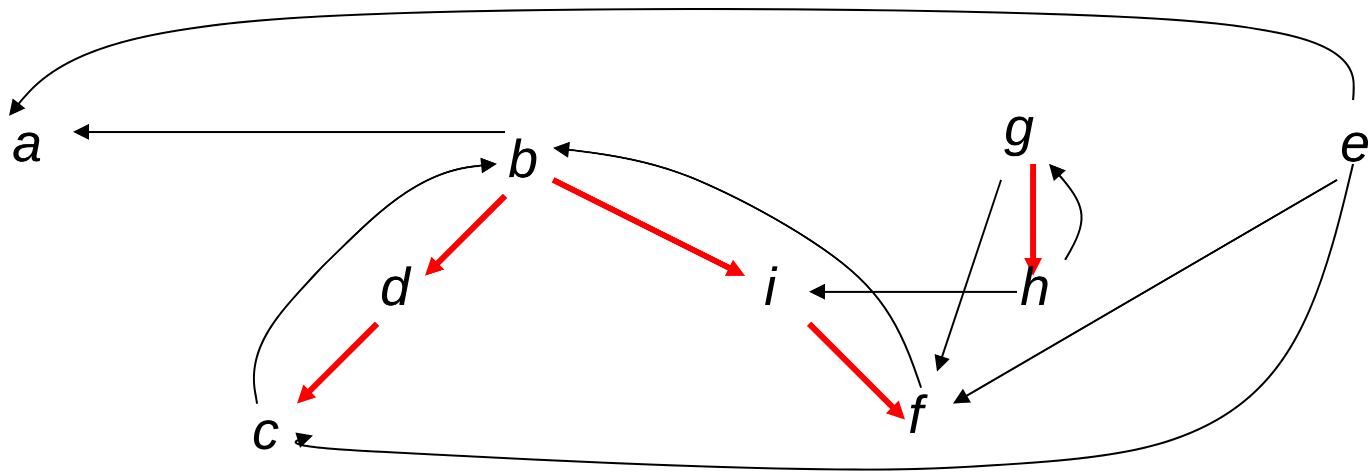
- Once we have computed the strongly connected components, we can “shrink” them and construct the DAG of the scc’s in O(n+e) time

Example



1st DFS, on original graph G

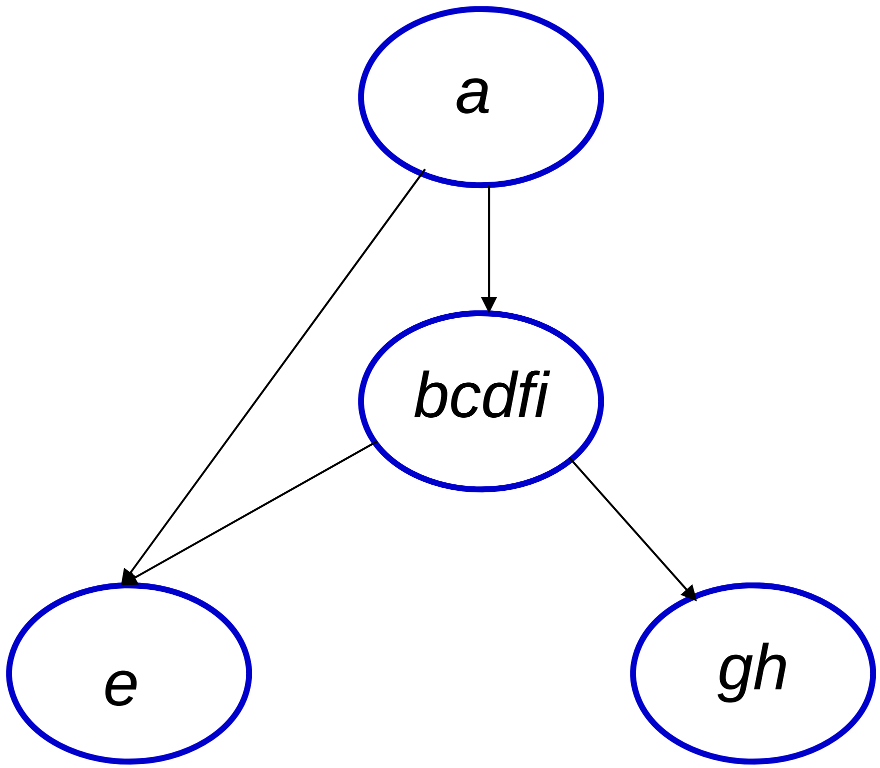
Depth First Search of Gr



Strongly connected components:

{a}, {b,c,d,f,i}, {g,h}, {e}

DAG of SCC's



Correctness Proof

- Must show: for all pairs of nodes u,v
  u,v mutually reachable in G $\Leftrightarrow$ u,v in same DFS tree of Gr

1. Nodes u,v mutually reachable in G $\Rightarrow$

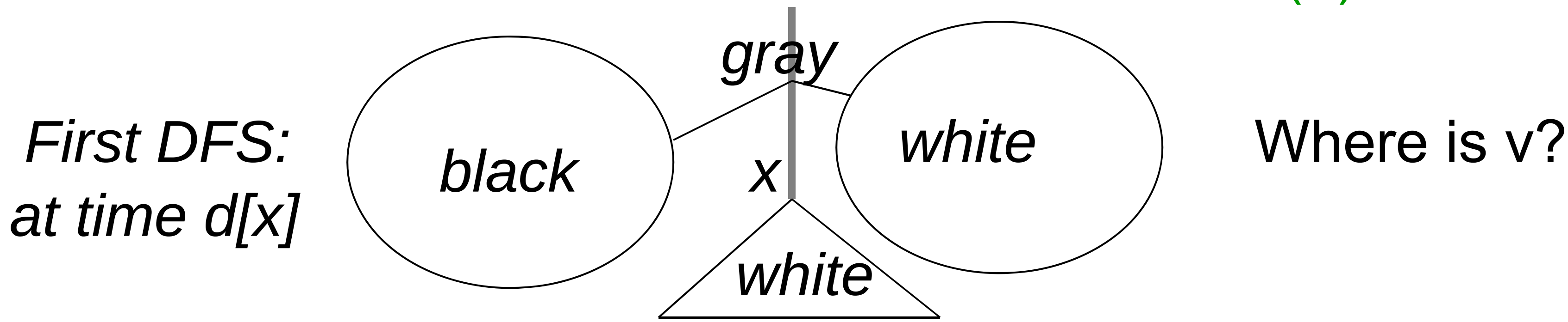
- also mutually reachable in Gr $\Rightarrow$
- if one of them is in a DFS tree then also the other
  $\Rightarrow$ u,v in same DFS tree of Gr

Correctness Proof ctd.

2. Suppose u,v in same DFS tree of Gr with root x

$\Rightarrow$ f[x] > f[u],f[v]

- x can reach v in Gr $\Rightarrow$ v can reach x in G (1)



- All nodes y that are above or right of x have f[y] > f[x] $\Rightarrow$
- in 2nd DFS they are all done before x $\Rightarrow$
- same for all their reachable nodes in Gr, i.e. that can reach them in G $\Rightarrow$
- v--->x path does not go through a gray node $\Rightarrow$
- v is in the subtree of x $\Rightarrow$ x can reach v in G (2)

- (1),(2) $\Rightarrow$ x,v mutually reachable
- Similarly, x,u mutually reachable

$\Rightarrow$ u,v mutually reachable